# Practical File
# of
# Source Code Management
# (22CS003)

## *Submitted in*

***partial fulfillment for the award of the degree of***
**BACHELOR OF ENGINEERING**

***in***

COMPUTER SCIENCE & ENGINEERING


**CHITKARA UNIVERSITY INSTITUTE OF ENGINEERING & TECHNOLOGY**
**CHITKARA UNIVERSITY**

**CHANDIGARH-PATIALA NATIONAL HIGHWAY**
**RAJPURA (PATIALA) PUNJAB-140401 (INDIA)**

**Submitted To:**
Faculty name: Dr Kalpna Guleria
Designation: Professor (Research)
Chitkara University, Punjab

**Submitted By:**
Student Name: Parampreet Singh
Roll No: 23109911
Sem: 2nd Batch:2023

Session: 2023-2024

# List of practicals

| S-No. | Practical | Signature |
|---|---|---|
| 1. | Introducing Version Control – Git client (CLI, GUI), Linux environment Emulation, Advantages of VCS, Installing git CLI and git GUI | |
| 2. | Setting up of the Git-hub Account and linking Git-hub account with git-bash | |
| 3. | Creating a repository, Adding files, checking status, staged files/ untraced files and performing Commits | |
| 4. | Generating logs: Visualization of various git logs | |
| 5. | Git lifecycle description Git status, add, commit, stage – Life cycle of a file in Git managed in Repos. | |
| 6. | Git branching and Merging: Visualization of git Branch and HEAD, Git branches management, Create a new branch, Commit changes in the new branch, Explore commit in the new branch, Merging the branches | |
| 7. | Pushing your local repository on GitHub Repository: Creating a local repository, creating remote origin, performing Git push operation to GitHub account | |
| 8. | To Add collaborators on Git-hub repository And Git Cloning-Cloning repository, Explore contents of the cloned repository, Explore cloned repository in GitHub Desktop, Commit changes in the cloned repository Git | |
| 9. | Fork and commit on our repository - Fork is a copy of a repository that you manage After creating a fork, we can make any change like adding collaborators, rename files | |
| 10. | Merge and Resolve conflicts - Resolution of merge conflicts, GitHub and remote repositories, Cloning remote repository, resolving conflicts | |
| 11. | Git Reset (soft reset and hard reset), Git revert - Performing Soft reset and hard reset, Git revert | |

| S-No. | Title | Signature |
|---|---|---|
| 12. | Version control with Git | |
| 13. | Problem Statement | |
| 14. | Objective | |
| 15. | Resources Requirements – Frontend / Backend | |
| 16. | Concepts and commands | |
| 17. | Workflow and Discussion | |
| 18. | Reference | |

CHITKARA UNIVERSITY

Name – Parampreet Roll no.-2310991088

**Experiment 1:**

**Aim:** Introducing Version Control – Git client (CLI, GUI), Linux environment Emulation, Advantages of VCS, Installing git CLI and git GUI

**Theory:**

**Git:** Git is a free and open-source version control system used to handle small to very large projects efficiently. This is also used for tracking changes in any set of files and usually helps in coordinating work among members of a team. Hence, enables multiple developers to work together on non-linear development.

**Version Control Systems (VCS)** like Git are essential tools for managing and tracking changes in software development projects. Here's a guide to introducing Git, including CLI and GUI tools, emulation in Linux environments, the advantages of VCS, and how to install Git CLI and Git GUI.

**1. Introduction to Version Control Systems (VCS)**
Version Control Systems help developers manage changes to source code over time. They provide the ability to track modifications, revert to previous versions, and collaborate efficiently.

**2. Git CLI and GUI**
CLI (Command Line Interface)

- Git CLI is the command-line interface for Git, offering powerful functionality for managing repositories and version control operations.
- It provides commands for tasks such as initializing repositories, adding files, committing changes, branching, merging, and more.

GUI (Graphical User Interface)

- Git GUI tools offer a more visual and user-friendly way to interact with Git repositories.
- GUIs typically provide features like visual diff tools, branch management, and commit history visualization.

**3. Emulating Linux Environment**
Linux Environment Emulation

- Git is natively supported on Linux, but it can also be used in emulation environments like WSL (Windows Subsystem for Linux) or through virtual machines.
- Emulating a Linux environment allows users on non-Linux platforms to access Git and other Linux tools seamlessly.

**4. Advantages of Version Control Systems**
Advantages:

1. **Collaboration:** Multiple developers can work on the same project simultaneously, with changes tracked and merged seamlessly.
2. **Versioning:** Every change made to the project is tracked, allowing for easy rollback to previous versions if needed.
3. **Backup**: VCS serves as a backup mechanism, ensuring that code is not lost even if local copies are corrupted or deleted.

4. **Branching and Merging:** Developers can work on features in isolation through branching and merge changes back into the main codebase when ready.
5. **History Tracking**: Detailed history logs provide insights into who made what changes and when, aiding in debugging and accountability.

**5. Installing Git CLI and Git GUI**

Installation Steps:

1. **Git CLI:**

- On Linux: Git is often available in the default package repositories. Install it using your distribution's package manager (e.g., apt, yum, dnf).
- On Windows: Download and run the Git installer from the official Git website (https://git-scm.com/).
- On macOS: Git comes pre-installed on macOS, but you can also install it via package managers like Homebrew.

2. **Git GUI:**

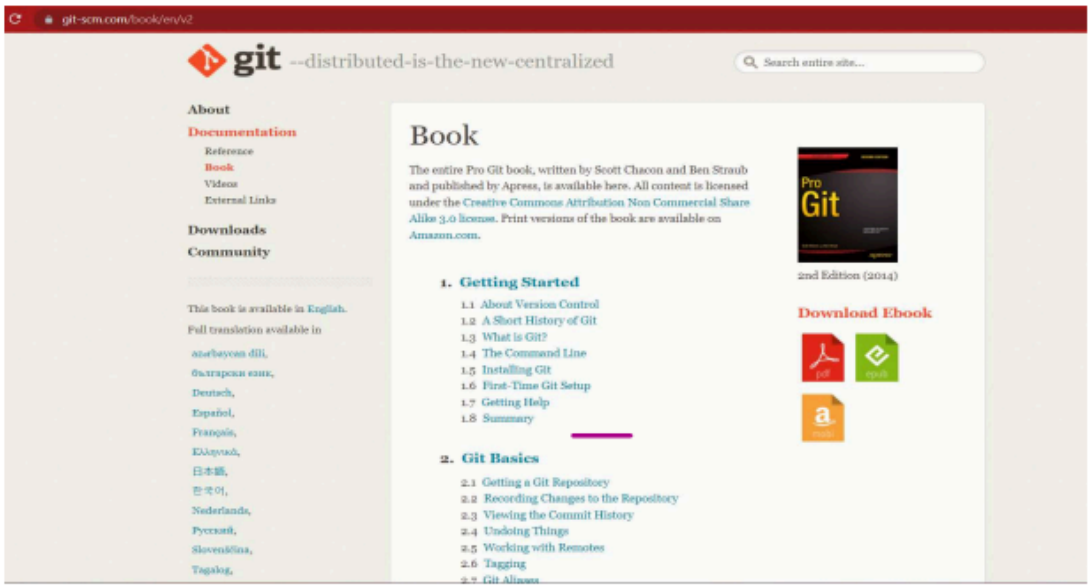
- There are several Git GUI tools available, including GitHub Desktop, GitKraken, Sourcetree, etc.
- Download and install your preferred Git GUI tool from its official website.
- Follow the setup instructions provided by the GUI tool.

**Conclusion**

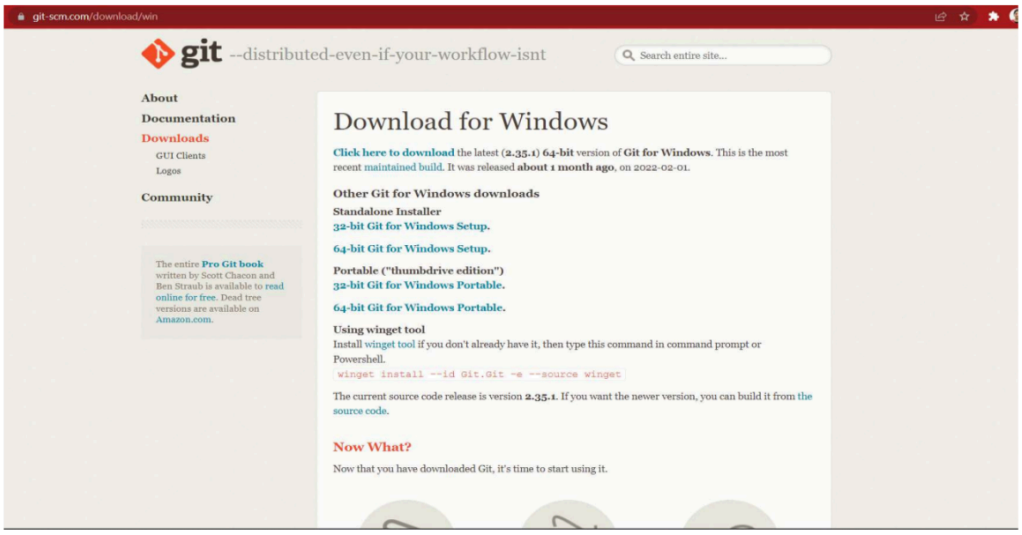
Version Control Systems like Git are indispensable tools for modern software development, offering efficient collaboration, versioning, backup, and history tracking capabilities. By installing Git CLI and GUI tools, developers can harness the power of version control to streamline their workflows and improve productivity.Top of Form

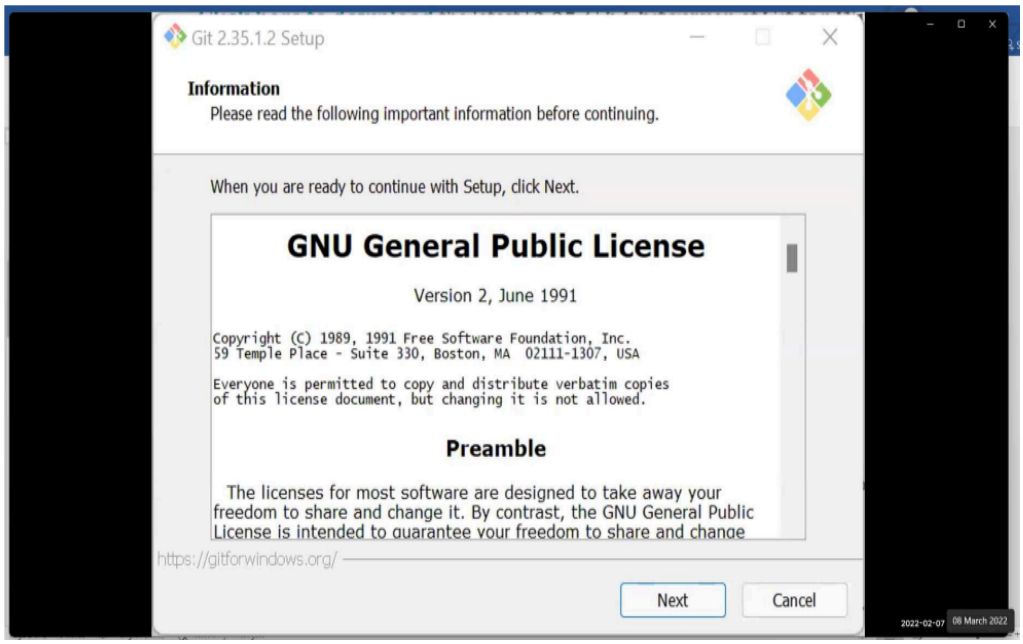
**Installing GIT on Windows :**

There are many ways to install Git on Windows. The most official build is available for download on the Git website. Go to https://gitscm.com/download/win and after a few settings the download will start automatically.

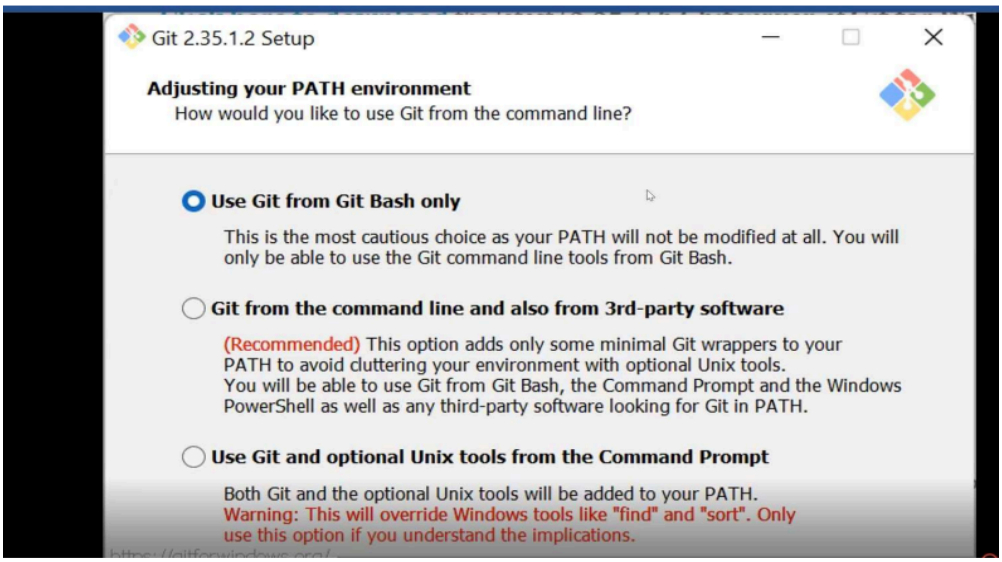


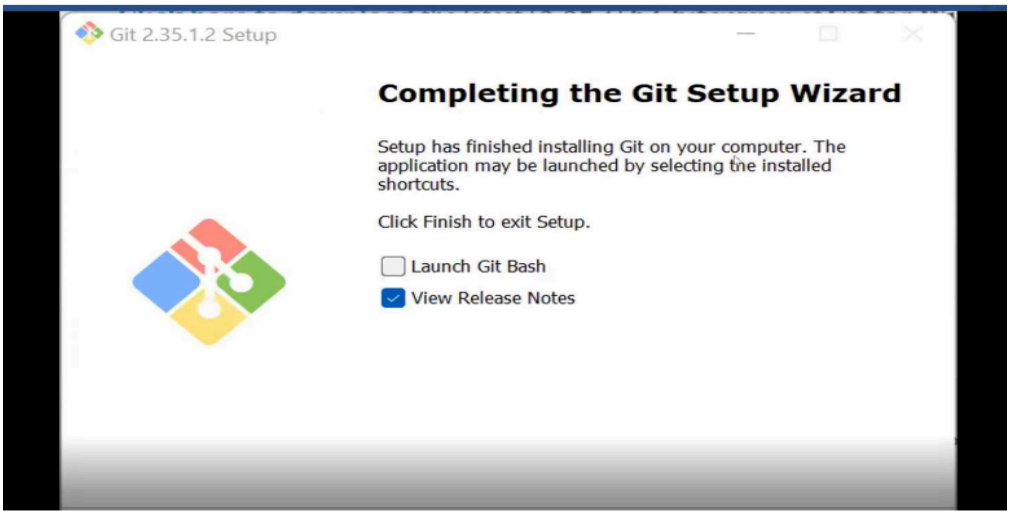

Then click on Installation Git and click on whatever system you want, available are three- Windows, Apple and Linux.






After some more simple and easy settings and choosing your favourable environment and doing some SSH settings, it finally starts exporting the files in system and completes the Git hub wizard.






Git bash got installed in system and seemed and opened on clicking seems of like:

CHITKARA UNIVERSITY

Name – Parampreet Roll no.-2310991088

**Experiment 2:**

**Aim:** Setting up of the Github Account and linking Github account with gitbash

**Theory:**

**GitHub :** GitHub is a code hosting platform for version control and collaboration. GitHub is a development platform inspired by the way you work. From open source to business, we can host and review code, manage projects, and build software alongside 36 million developers.

**Advantages:**

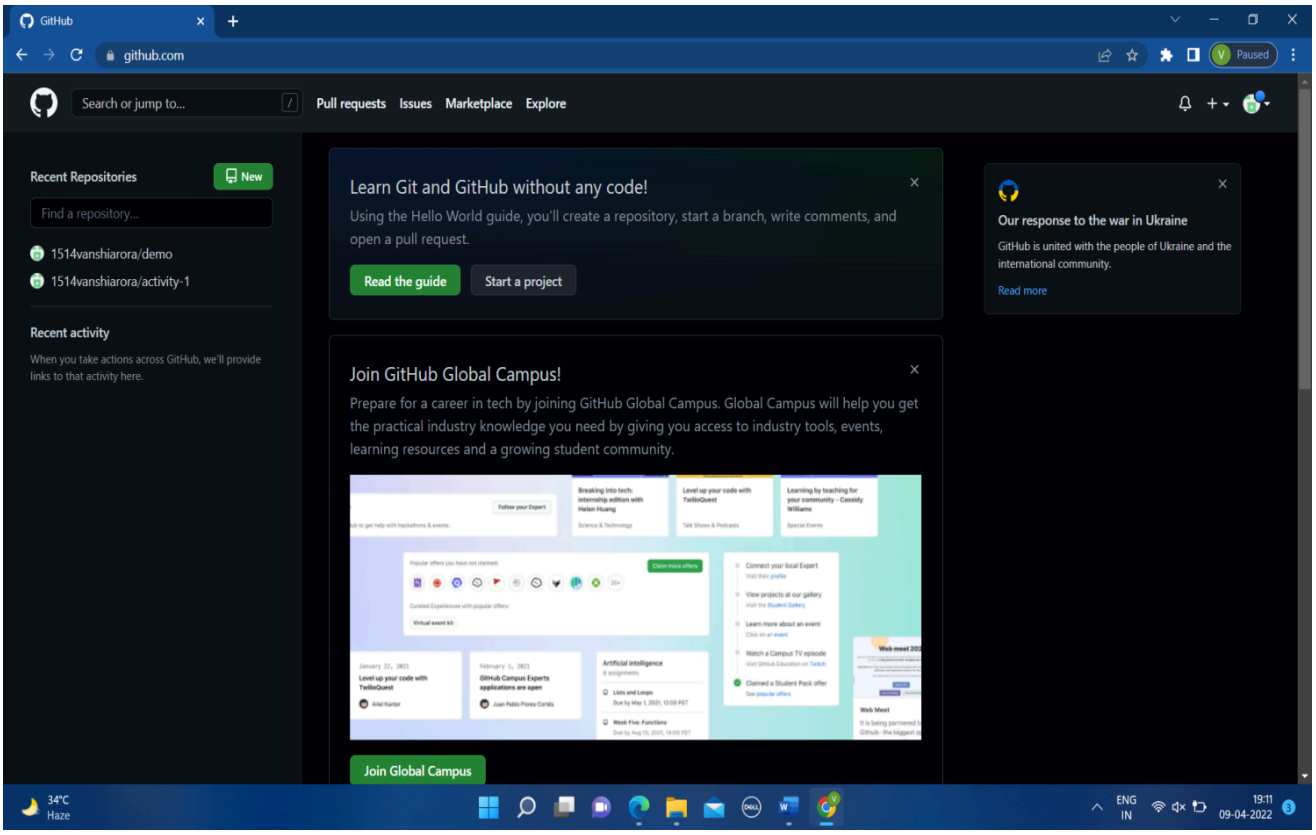
1.Documentation.
2.Showcase your work.
3.Markdown.
4.GitHub is a repository.
5.Track changes in your code across versions.
6.Integration options.

**Procedure:**

Search about GitHub: https://github.com/signup

Name – Parampreet Roll no.-2310991088

By signing up for git you must remember your email and pass phases or password. For a new user, you must add your email and click on Sign up for GitHub. Otherwise click on Sign In at the top right corner.




For linking Git Hub with Git Bash:

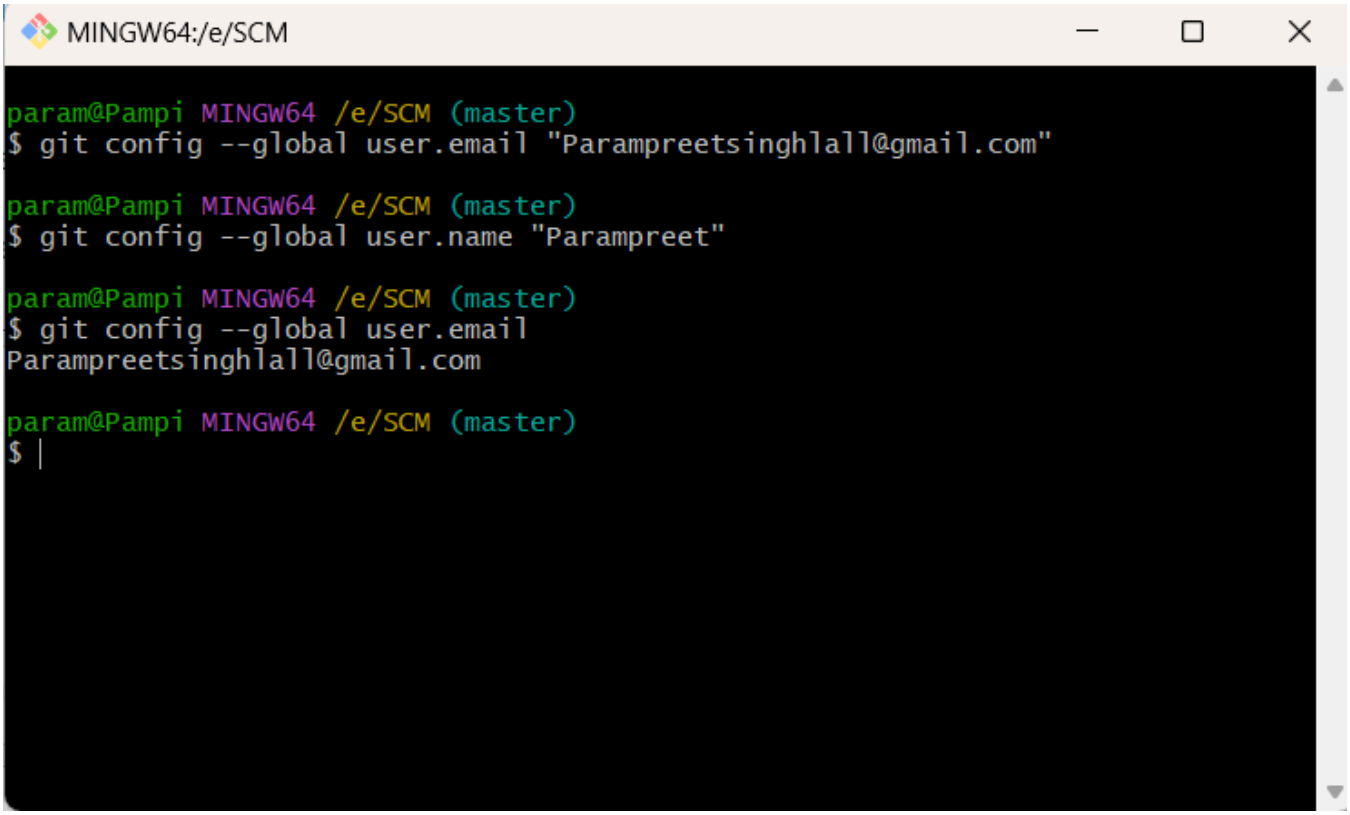
Usernamegit config --global user.name “username in github”

Emailgit config --global user.email “your email in github”

Check Username & Email: git config user.name , git config user.email

CHITKARA UNIVERSITY

Name – Parampreet Roll no.-2310991088



**Experiment 3:**

**Aim: Creating a repository, Adding files, checking status , staged files/ untraced files and performing Commits**

**Theory:**

Repositories in Git contain a collection of files of various different versions of a Project. These files are imported from the repository into the local server of the user for further updations and modifications in the content of the file. The git init command is used to initialize the git into the existing project. By using this command we can convert the normal project folder into a git repository.

**1. Initialize a Repository**

Open Git Bash and navigate to the directory where you want to create your repository.

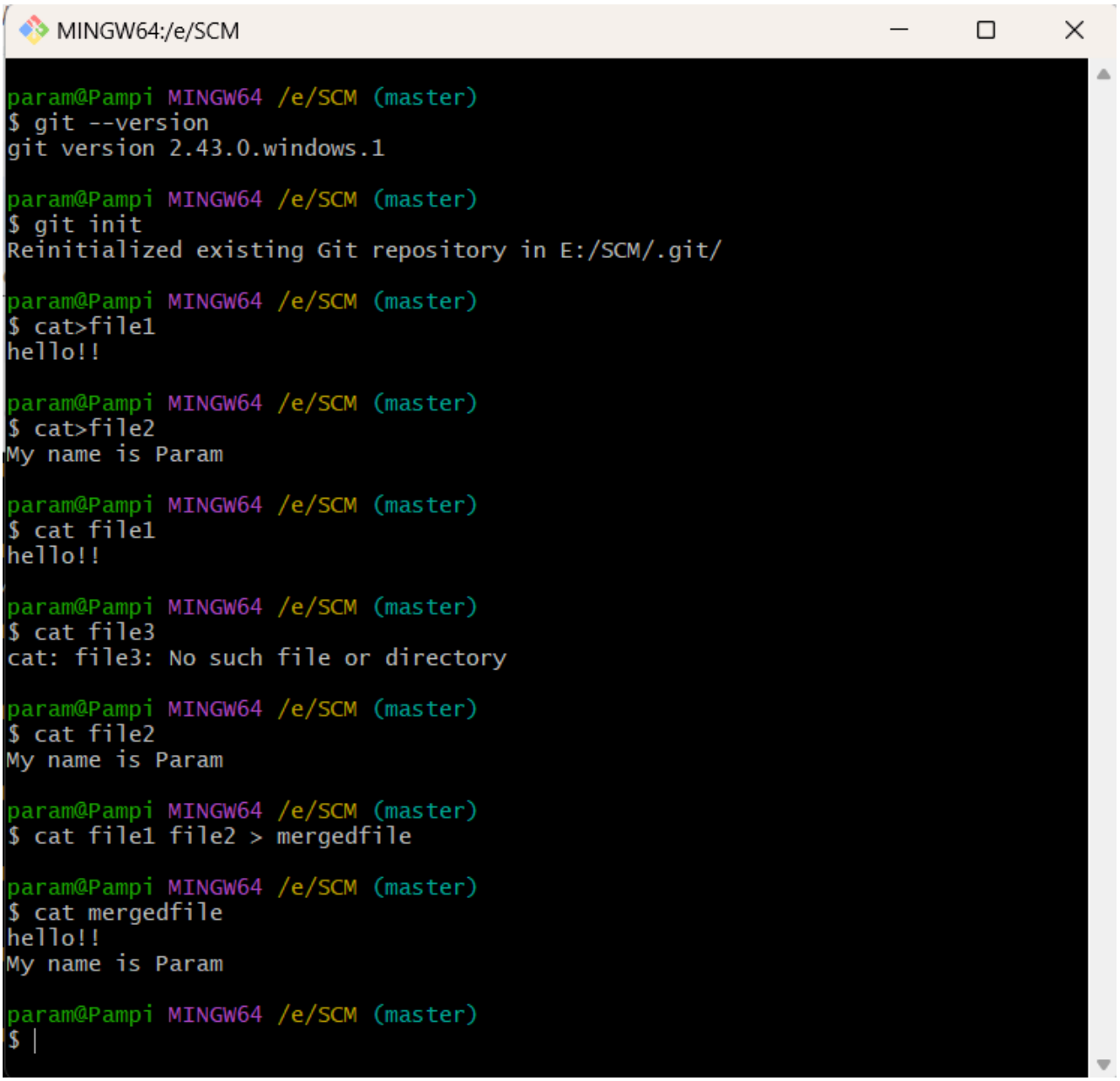
```
param@Pampi MINGW64 /e/SCM (master)
$ cd
```

Initialize a new Git repository.

```
param@Pampi MINGW64 ~
$ git init
Initialized empty Git repository in C:/Users/param/.git/
```

**2. Add Files**

Create or copy files into your repository directory. Add files to the staging area. You can add multiple files at once. Or add all files in the directory.




### 3. Check Status

The git status command is a powerful tool that provides a comprehensive overview of the current state of your Git repository. By executing this command, you gain valuable insights into which files are staged for commit, which ones have been modified but not yet staged, and which files are untracked.

Staging files is a crucial step in the Git workflow. When you make changes to your project, Git recognizes these modifications as either unstaged or staged. Unstaged changes are modifications that you've made but have not yet told Git to include in the next commit. On the other hand, staged changes are those that you've marked as ready to be committed.

Upon running git status, Git will present you with a clear and detailed report. It will display the current branch, inform you about any changes in your working directory, and list the files that are staged or modified. Untracked files, which are files that Git is not currently managing, will also be highlighted.

This information is crucial for maintaining a well-organized version control system. You can use the output from git status to decide which changes to include in your next commit. If you have new files that are untracked, you may want to add them to the staging area using git add so that they can be included in the upcoming commit.

```
param@Pampi MINGW64 /e/SCM (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   branch1.c
        modified:   hello.c
        new file:   practice.c
        new file:   practice.exe
        new file:   repo

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1
        file2
        mergedfile


param@Pampi MINGW64 /e/SCM (master)
$
```

From this screenshot, you see we have two sections: "Changes not staged for commit" and "Untracked files"

**Tracked files:**

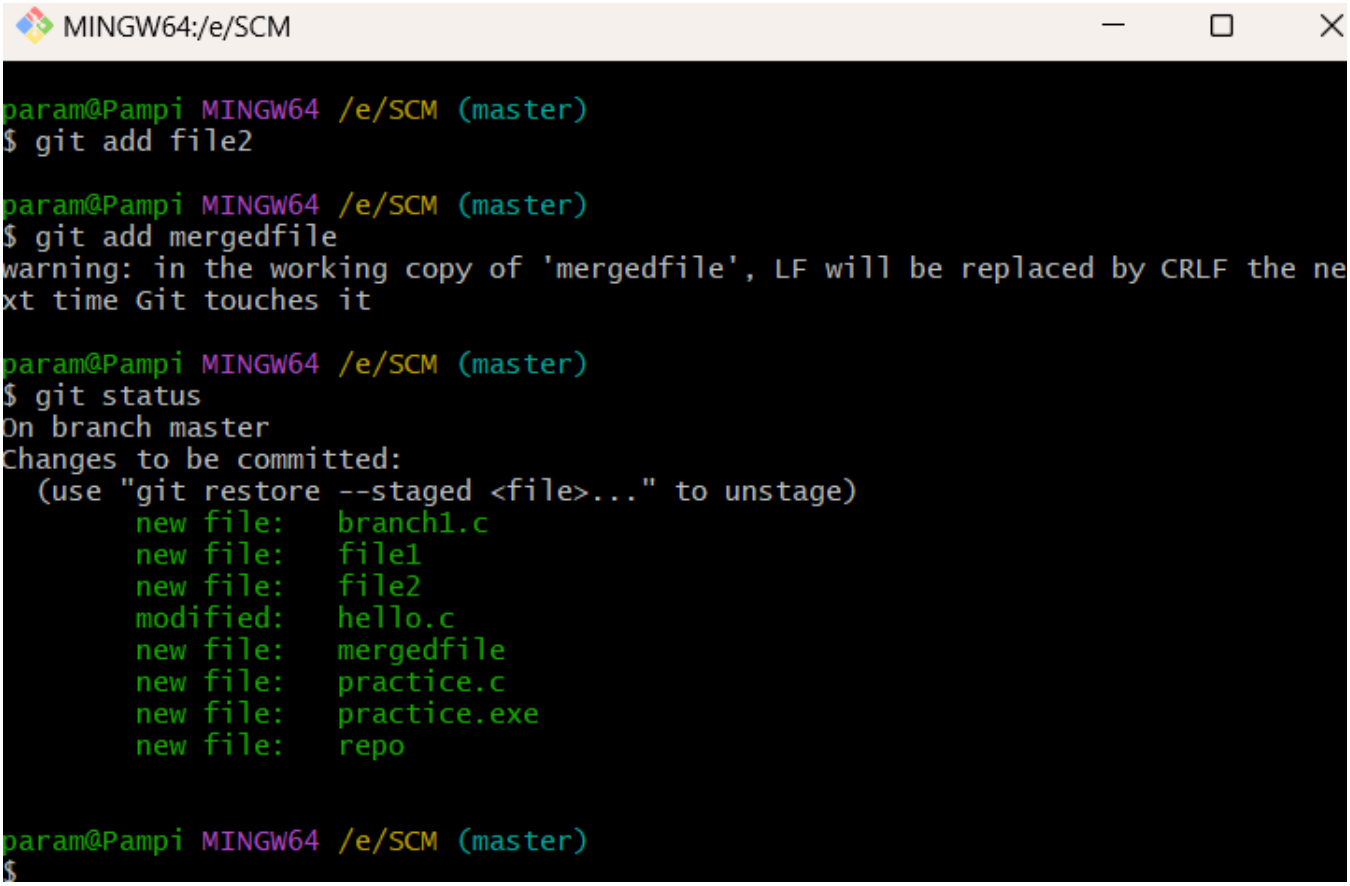
We have the first section which is Changes not staged for commit. These are tracked files--files Git has recorded a history of. So Git knows that these files have changed since the last time they were committed. A change can be that the file was modified, renamed, or deleted. Tracked files are files that Git is aware of. This way, Git can keep track of the changes or commits related to this file, and you can revert to previous versions of the file.

**Untracked files :**

The second section is Untracked files. This section shows the files that Git does not track currently. This means it will not keep a history of the changes in this file and the subsequent changes. The only way to inform Git to keep a record of this file's history is to add the file to the list of tracked files.

**Tracking a file in Git:**

We can run a command , git add which adds it to the staging area, and then you commit the change.




**4. Perform Commits**

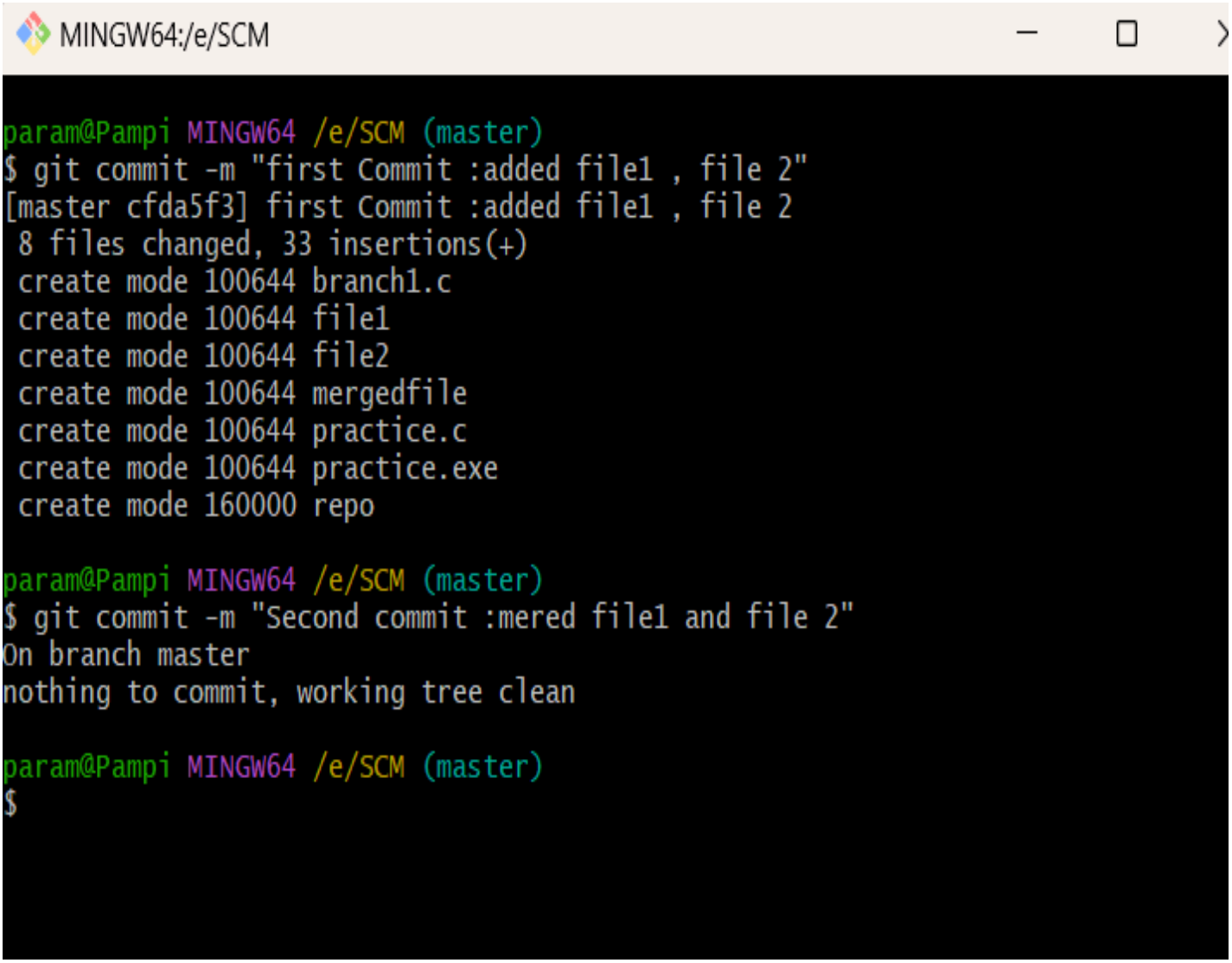
The git commit command is a fundamental aspect of the Git version control system, playing a pivotal role in documenting the progress and changes made to a project. Each commit serves as a snapshot, or a "save point," that allows you to revisit specific points in the project's history. When making a commit, it's essential to provide a descriptive and informative message using the -m flag to outline the purpose and nature of the changes.

Executing a commit is akin to creating a milestone in the project timeline, and the commit message serves as the label for that milestone. A well-crafted commit message is not just for the present but also

for the future, providing clarity to yourself and your collaborators about the intentions behind the changes made.

For instance, when initiating a new project, the first commit is often referred to as the "Initial commit." However, it's advisable to make this commit message more detailed and informative. You might want to include information about the project's structure, the technologies used, or any significant decisions made at the project's outset. A more extensive commit message can be beneficial for onboarding new contributors or for anyone reviewing the project's history.

git commit -m "Initial commit"



**Summary of Commands:**

- git init: Initialize a new Git repository

  The git init command is the inaugural step in creating a new Git repository. By executing this command within your project directory, you establish a version control system, enabling Git to track changes in your files.

- git add <file>: Add a specific file to the staging area

  When you want to include specific changes in your next commit, use git add <file> to add a particular file to the staging area. Staging is a crucial step that marks files for inclusion in the upcoming commit.

- git add .: Add all files in the current directory to the staging area

  The git add . command is a convenient way to add all modified or new files in the current directory to the staging area. It streamlines the process when you have multiple changes across various files.

- git status: Check the status of the repository

  Executing git status provides a comprehensive overview of the current state of your Git repository. It details which files are modified, staged, or untracked. This information is essential for making informed decisions before committing changes.

- git commit -m "message": Commit the staged changes with a descriptive message

  The git commit -m "message" command finalizes the process by creating a commit with the changes staged in the previous step. Including a descriptive message is crucial for documenting the purpose of the commit, aiding future understanding and collaboration.

In summary, these Git commands collectively lay the foundation for effective version control. The git init command initializes a new repository, while git add stages changes for commit. git status keeps you informed about the repository's status, and finally, git commit solidifies your changes with a clear message. Mastering these commands empowers you to efficiently manage your project's development history and collaborate seamlessly with others.

**Experiment 4:**

**Aim: Generating logs: Visualization of various git logs**

**Theory:**

**Git Logs:** The git log command shows a list of all the commits made to a repository. You can see the hash of each Git commit, the message associated with each commit, and more metadata. This command is basically used for displaying the history of a repository.

**Why do we need logs?**

Git log is a utility tool to review and read a history of everything that happens to a repository. Anything we change at what time, by which log, everything is getting recorded in git logs.

Logs in version control systems like Git serve several important purposes:

- **History Tracking:** Logs provide a detailed history of changes made to the codebase over time. Each commit in the log represents a snapshot of the project at a specific point in time, including who made the changes, when they were made, and what changes were made.

- **Debugging and Troubleshooting**: When issues arise in the code, logs can help developers trace back the changes that led to the problem. By examining the commit history, developers can identify which changes introduced the bug and take appropriate action to fix it.

- **Code Review and Collaboration**: Logs facilitate code review by allowing reviewers to see the changes made in each commit. Reviewers can provide feedback on specific changes, suggest improvements, and ensure that coding standards and best practices are followed.

- **Understanding Project Evolution:** Logs provide insights into how the project has evolved over time. By analyzing the commit history, developers can understand the progression of features, the rationale behind design decisions, and the evolution of coding patterns and conventions.

- **Auditing and Compliance**: Logs serve as an audit trail, documenting who made changes to the codebase and when. This information is valuable for compliance purposes, such as ensuring that regulatory requirements are met and tracking changes made by team members.

- **Reproducibility and Rollback:** Logs enable developers to reproduce past states of the codebase by checking out specific commits. This is useful for testing, debugging, and rolling back to a previous version in case of emergencies or regressions.

Overall, logs play a crucial role in version control by providing a comprehensive record of changes, facilitating collaboration and communication among team members, and ensuring the integrity and reliability of the codebase over time. Top of Form

```
param@Pampi MINGW64 /e/SCM (master)
$ git log
commit cfda5f3ee53bff2b8cb62aa9968ac1f12b36d245 (HEAD -> master)
Author: Parampreet <Parampreetsinghlall@gmail.com>
Date:   Mon Feb 12 13:05:14 2024 +0530

    first Commit :added file1 , file 2
```

```
param@Pampi MINGW64 /e/SCM (master)
$ nano file2

param@Pampi MINGW64 /e/SCM (master)
$ git status
On branch 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:   file2

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

param@Pampi MINGW64 /e/SCM (master)
$ git add file2
```

```
param@Pampi MINGW64 /e/SCM (master)
$ git commit -m "third commit : modified file2"
[master 459558d] third commit : modified file2
 1 file changed, 1 insertion(+), 1 deletion(-)

param@Pampi MINGW64 /e/SCM (master)
$ git log
commit 459558d0af45401406ed0f0f9e335766b9a73101 (HEAD -> master)
Author: Parampreet <Parampreetsinghlall@gmail.com>
Date:   Mon Feb 12 13:13:03 2024 +0530

    third commit : modified file2
```

**Experiment 5:**

**Aim: Git lifecycle description Git status, add, commit, stage – Life cycle of a file in Git managed in Repos.**

**Theory:**

**Stages in GIT Life Cycle:** Files in a Git project have various stages like Creation, Modification, Refactoring, and Deletion and so on. Irrespective of whether this project is tracked by Git or not, these phases are still prevalent. However, when a project is under Git version control system, they are present in three major Git states in addition to these basic ones. Here are the three Git states:

- Working directory
- Staging area
- Git directory

**Working Directory:**

When a project is residing in our local system we don't know whether the project is tracked by Git or not. In any of the case, this project directory is called our Working directory.

**Staging Area:**

The staging area is like a rough draft space, it's where you can git add the version of a file or multiple files that you want to save in your next commit (in other words in the next version of your project)

**Git Directory:**

The git folder contains all information that is necessary for the project and all information relating commits, remote repository address, etc. It also contains a log that stores the commit history. This log can help you to roll back to the desired version of the code

Remote Repository: Remote repositories are hosted on a server that is accessible for all team members - most likely on the internet or on a local network. Assessable and reachable by all.

In Git, the lifecycle of a file refers to the stages it goes through from being untracked to being committed to the repository. The main stages in the lifecycle of a file in Git are:

**Untracked:** When a file is newly created or added to a directory that is not being tracked by Git, it is considered untracked. Git is unaware of the existence of this file and will not include it in version control until it is explicitly added.

**Staged:** After creating or modifying a file, you can stage it to prepare it for a commit. Staging a file means marking it as ready to be included in the next commit. Staged files are tracked by Git and will be included in the snapshot of the repository when you make a commit.

**Committed:** Once you have staged your changes, you can commit them to the repository. A commit creates a snapshot of the current state of the staged files and adds it to the repository's history. Committed files are permanently stored in the repository and can be accessed or reverted to at any time.

Here's a more detailed description of each stage and the corresponding Git commands:

**Untracked:**

- Files in this stage have not been added to the Git repository yet.
- Command to check the status of untracked files: git status.
- Command to start tracking a new file: git add <file>.

**Staged:**

- Files in this stage are ready to be included in the next commit.
- Command to stage changes: git add <file> or git add . (to stage all changes).
- Command to unstage changes: git reset HEAD <file>.

**Committed:**

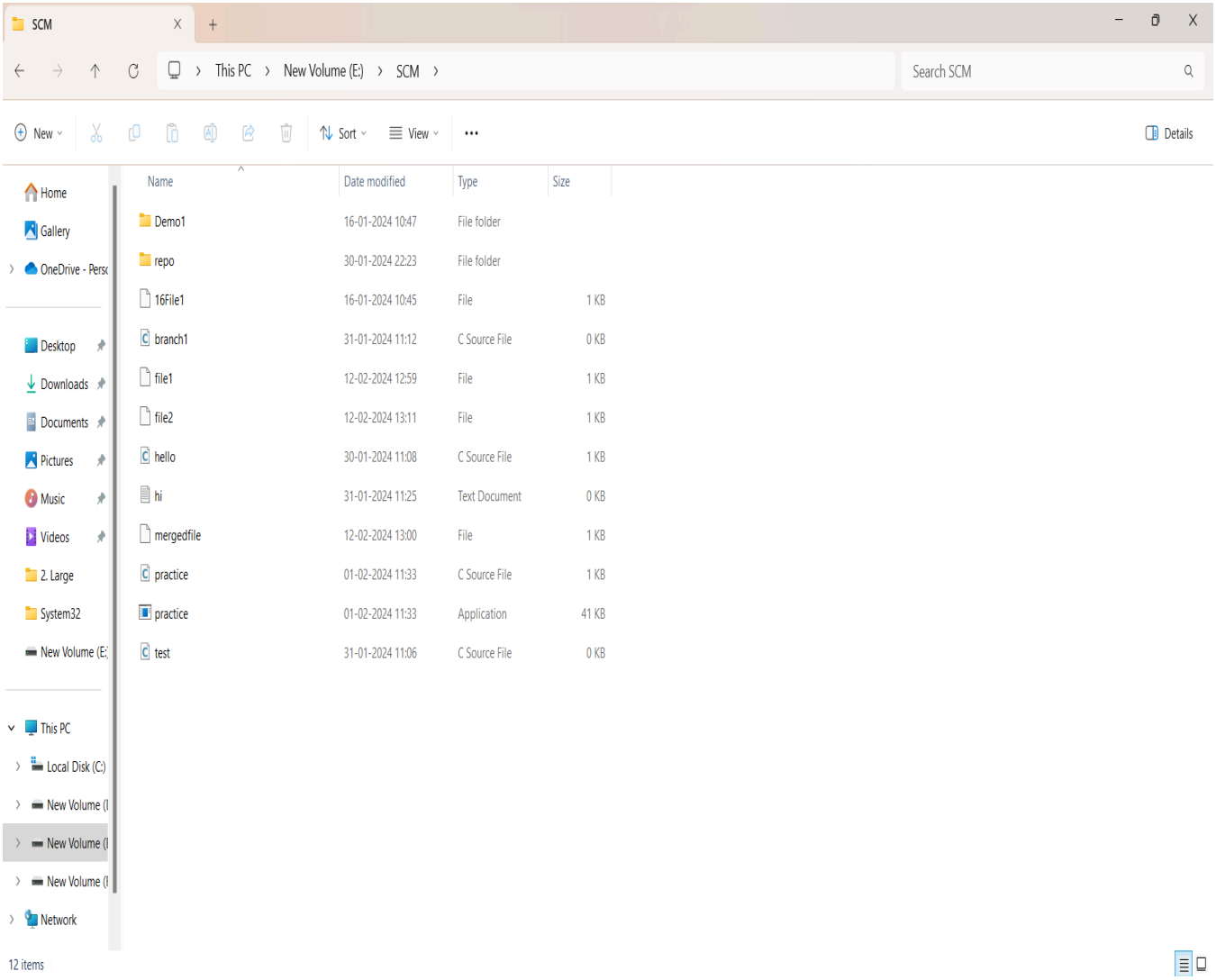
- Files in this stage have been permanently saved in the Git repository.
- Command to commit changes: git commit -m "Commit message".
- Command to view commit history: git log.
- The typical workflow for managing files in a Git repository follows these stages:

1. Edit/Create Files: Modify existing files or create new ones.
2. Check Status: Use git status to see which files are modified or untracked.
3. Stage Changes: Use git add to stage the changes you want to include in the next commit.
4. Commit Changes: Use git commit to commit the staged changes to the repository.

By following this lifecycle, you can effectively manage the version control of your files in a Git repository, ensuring that changes are tracked, documented, and safely stored.

CHITKARA UNIVERSITY

Name – Parampreet Roll no.-2310991088



```
param@Pampi MINGW64 /e/SCM (master)
$ git branch sigma

param@Pampi MINGW64 /e/SCM (master)
$ pwd
/e/SCM

param@Pampi MINGW64 /e/SCM (master)
$ mkdir git-repo

param@Pampi MINGW64 /e/SCM (master)
$ cd git-repo

param@Pampi MINGW64 /e/SCM/git-repo (master)
$ git init
Initialized empty Git repository in E:/SCM/git-repo/.git/

param@Pampi MINGW64 /e/SCM/git-repo (master)
$ touch python.py

param@Pampi MINGW64 /e/SCM/git-repo (master)
$ vi python.py
```

```
param@Pampi MINGW64 /e/SCM/git-repo (master)
$ git add python.py

param@Pampi MINGW64 /e/SCM/git-repo (master)
$ git commit -m "Python file created"
[master (root-commit) 9b4d087] Python file created
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 python.py
```

```
param@Pampi MINGW64 /e/SCM/git-repo (master)
$ git checkout sigma
Switched to branch 'sigma'

param@Pampi MINGW64 /e/SCM/git-repo (sigma)
$ git branch
  master
* sigma
```

**Experiment 6:**

**Aim: Git branching and Merging: Visualization of git Branch and HEAD, Git branches management, Create a new branch, Commit changes in the new branch, Explore commit in the new branch, Merging the branches**

**Theory:**

**How to create branches?**

Creating branches in Git is a straightforward process. Here's how you can create a new branch:

1. Check Current Branch

Before creating a new branch, it's a good practice to check which branch you're currently on. You can do this by running:

The branch with an asterisk (*) next to it is the currently active branch.

```
param@Pampi MINGW64 /e/SCM (master)
$ git branch
  branch1
  branch2
  branchA
* master
  pampi
  sigma
```

2. Create a New Branch

To create a new branch, use the following command:

git branch <branch_name>

Replace <branch_name> with the desired name for your new branch.

For example, to create a branch named feature-branch, you would run:

```
param@Pampi MINGW64 /e/SCM (master)
$ git branch name
```

3. Switch to the New Branch

After creating the new branch, you may want to switch to it to start working on it immediately. You can do this with the checkout command:

git checkout <branch_name>

```
param@Pampi MINGW64 /e/SCM (master)
$ git checkout branchA
Switched to branch 'branchA'

param@Pampi MINGW64 /e/SCM (branchA)
$ git checkout master
Switched to branch 'master'
```

Alternatively, you can combine branch creation and switching into one command using -b option:

git checkout -b <branch_name>

For example:

git checkout -b feature-branch

This command creates a new branch named feature-branch and switches to it in a single step.

**Summary:**

- To create a new branch: git branch <branch_name>.
- To switch to the new branch: git checkout <branch_name> or git checkout -b <branch_name>.
- After creating and switching to your new branch, you can start making changes to your codebase. Remember to commit your changes regularly as you work on your new branch. Once you're finished with your changes and ready to merge them back into the main branch, you can follow the steps for merging branches. Top of Form
- The main branch in which we are working is master branch. you can use the “git branch” command with the branch name and the commit SHA for the new.
- For creating a new branch: git branch “name of the branch”.

```
param@Pampi MINGW64 /e/SCM (master)
$ git branch branchA

param@Pampi MINGW64 /e/SCM (master)
$ git branch branchB
```

To check how many branches we have , use command git branch :

```
param@Pampi MINGW64 /e/SCM (master)
$ git branch
  branch1
  branch2
  branchA
* master
  pampi
  sigma
```

As you can see here three branches are showing- branch1, branch2 and master.

When you're ready to switch your current working context to a different branch, utilize the git checkout command followed by the name of the desired branch. For example, to transition to a branch named "feature-branch," execute git checkout feature-branch. This straightforward command not only changes your active branch but also aligns your working directory with the specific branch's content.

Now, suppose you need to return to the master branch to consolidate or manage changes from various branches. In that case, the command git checkout master becomes your go-to directive. This command efficiently reorients your working environment, placing you back in the master branch.

The act of switching between branches using git checkout is akin to navigating between different aspects of your project's development. Each branch represents a unique line of work or a distinct set of features. This flexibility allows developers to compartmentalize changes and collaborate on different aspects of the project simultaneously.

So, to recap, whenever you wish to alter your current working branch, employ the git checkout command with the relevant branch name. And when it's time to return to the master branch for overarching coordination or integration, the command git checkout master seamlessly guides you back to the central directory, ready to manage and consolidate the diverse contributions from various branches.

```
param@Pampi MINGW64 /e/SCM (master)
$ git checkout branchA
Switched to branch 'branchA'

param@Pampi MINGW64 /e/SCM (branchA)
$ git checkout master
Switched to branch 'master'
```

Here, you see by using checkout command we can switch branches and from branches to even master branch too.

**Visualizing branches:**

- Visualizing branches in Git helps you understand the relationship between different branches and their commit history. One way to visualize branches is by using the git log command with additional options to display a graphical representation of the commit history.
- To visualize branches, use the following command:
- For visualizing, we have to create a new file in the branch that we made “branch1” instead of the master branch. After this we have to do three step architecture that is working directory, staging area and git repository
- Firstly change the branch from master to branch1 that then check git status. Now add text in file and use git add “file_name”.
- Then use git commit -m “key_name” command for the changes that has been made.
- In this example:
- HEAD points to the latest commit on the master branch.
- master and origin/master (remote tracking branch) both point to the same commit.
- feature-branch points to a commit that diverged from the master branch. It was later merged back into master.
- This visualization provides a clear picture of the branching and merging history in your repository, helping you understand how different branches relate to each other and how changes flow through the repository over time.

MINGW64:/e/SCM

```
param@Pampi MINGW64 /e/SCM (master)
$ git checkout branchA
Switched to branch 'branchA'

param@Pampi MINGW64 /e/SCM (branchA)
$ cat>sample1
Source Code Management

param@Pampi MINGW64 /e/SCM (branchA)
$ git status
On branch branchA
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        sample1

nothing added to commit but untracked files present (use "git add" to track)

param@Pampi MINGW64 /e/SCM (branchA)
$ git add sample1
warning: in the working copy of 'sample1', LF will be replaced by CRLF the next
time Git touches it

param@Pampi MINGW64 /e/SCM (branchA)
$ git commit -m "Initial commit : added sample1 to branch1"
[branchA fa37399] Initial commit : added sample1 to branch1
 1 file changed, 1 insertion(+)
 create mode 100644 sample1

param@Pampi MINGW64 /e/SCM (branchA)
$ nano sample1

param@Pampi MINGW64 /e/SCM (branchA)
$
```

**Merging the branches:**

To merge branches in Git, you typically use the git merge command. Here's how you can merge one branch into another:

**1. Checkout the Target Branch**

Before initiating the merge process in Git, it's imperative to ensure that you are currently on the branch where you intend to integrate changes. For instance, if the goal is to merge modifications from a feature branch into the master branch, use the git checkout master command. This command explicitly switches your working directory to the master branch, creating the appropriate context for the upcoming merge. By doing this, you establish the foundation for a seamless integration, aligning your current state with

the branch into which you aim to incorporate changes. This meticulous step ensures that the subsequent merge operation unfolds in the desired branch, facilitating a smooth and well-organized version control workflow.

```
param@Pampi MINGW64 /e/SCM (master)
$ git checkout master
Already on 'master'
```

**2. Merge the Branch**

Following the branch switch, execute the crucial step of integrating changes by deploying the git merge command. In this instance, merging alterations from the source branch (feature-branch) into the target branch (master) is accomplished with the straightforward command git merge feature-branch. This command initiates the process of combining divergent code changes, harmonizing the contents of the source and target branches. The git merge operation is pivotal, as it consolidates the modifications made in the feature branch into the master branch, creating a unified codebase. This strategic integration fosters collaboration and ensures that the developments from the feature branch seamlessly become part of the overarching project in the master branch.

```
param@Pampi MINGW64 /e/SCM (master)
$ git merge branchname
```

This command will merge the changes from feature-branch into master. If there are no conflicts, Git will automatically perform the merge and create a new commit.

**Verify the Merge**

After completing the merge, you can verify that it was successful by checking the commit history and examining the state of your files.

git log

This command will display the commit history, including the merge commit that was created.

```
param@Pampi MINGW64 /e/SCM (master)
$ git log
commit 719c921509dba9772c82d8bd698e7238407af1a5 (HEAD -> master, sigma, name)
Merge: 459558d efb77bf
Author: Parampreet <Parampreetsinghlall@gmail.com>
Date:   Tue Feb 13 09:50:49 2024 +0530
```

**Summary:**

- Checkout the target branch: git checkout <target_branch>.
- Merge the source branch into the target branch: git merge <source_branch>.
- Resolve any merge conflicts if they occur.
- Stage and commit the resolved changes to complete the merge.
- Verify the merge using git log.
- By following these steps, you can merge changes from one branch into another, integrating new features or fixes into your main codebase:

1. Merging takes your branch changes and implements them into the main branch.
2. To merge two branches , use the following command:
3. Git merge branchname

```
param@Pampi MINGW64 /e/SCM (branch1)
$ git branch
* branch1
  branch2
  branchA
  master
  pampi

param@Pampi MINGW64 /e/SCM (branch1)
$ git checkout master
Switched to branch 'master'

param@Pampi MINGW64 /e/SCM (master)
$ git merge branch1
Merge made by the 'ort' strategy.
 sample1 | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 sample1
```

At last, check the activities with the help of git log command.

```
param@Pampi MINGW64 /e/SCM (master)
$ git log
commit 719c921509dba9772c82d8bd698e7238407af1a5 (HEAD -> master)
Merge: 459558d efb77bf
Author: Parampreet <Parampreetsinghlall@gmail.com>
Date:   Tue Feb 13 09:50:49 2024 +0530

    Merge branch 'branch1'
    abortsndakjdakj
```

**Experiment 1:**

**Aim:**Pushing your local repository on Github Repository:
Creating a local repository , creating remote origin, performing Git push operation to Github account

**Theory:**
Pushing a local repository to GitHub involves several steps, including creating a local repository, connecting it to a GitHub repository (remote origin), and then pushing your changes. Here's a step-by-step guide to help you through the process:

# 1. Create a Local Repository

If you haven't already created a local Git repository, you can do so by following these steps:

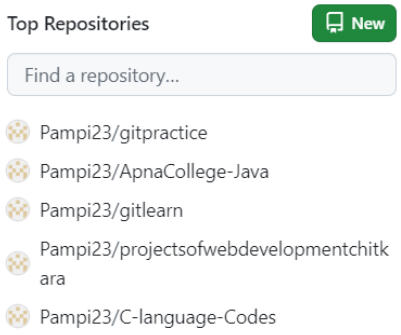
- Open your terminal or command prompt and navigate to the directory where you want to create your project.
- Create a new directory for your project if it doesn't exist, using mkdir project-name, and navigate into it with cd project-name.
- Initialize a Git repository by running:git init

```
param@Pampi MINGW64 /e/SCM (master)
$ git init
Reinitialized existing Git repository in E:/SCM/.git/
```

# 2. Create a GitHub Repository

Go to GitHub and log in with your account. Follow these steps to create a new repository:

- Click on the "+" icon in the top-right corner and select "New repository".
- Name your repository and add a description if you want.
- Choose whether the repository will be public or private.
- Click "Create repository".



## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?
Import a repository.

*Required fields are marked with an asterisk (*).*

Owner * Repository name *

Pampi23 / NewRepo

NewRepo is available.

Great repository names are short and memorable. Need inspiration? How about **silver-octo-memory** ?

Description (optional)

- Public
  Anyone on the internet can see this repository. You choose who can commit.
- Private
  You choose who can see and commit to this repository.

Initialize this repository with:

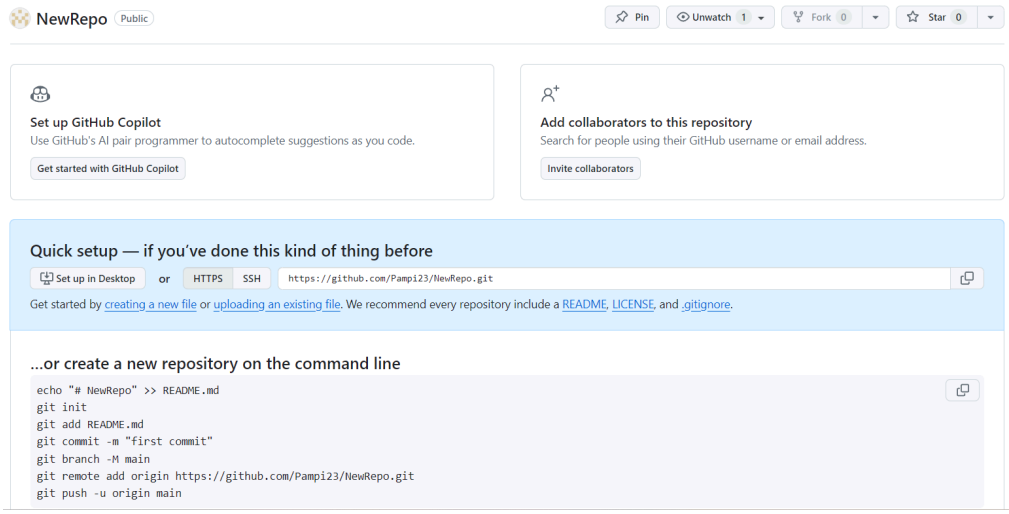
- [ ] Add a README file
  This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore

.gitignore template: None

Choose which files not to track from a list of templates. Learn more about ignoring files.

Choose a license



## 3. Connect Your Local Repository to GitHub

After creating a new repository on GitHub, you'll be taken to a page with a quick setup section that includes a URL for your new repository. Copy that URL to use in the next steps.

In your terminal, inside your local project directory, link your local repository to the GitHub repository (remote origin) using the following command. Replace https://github.com/username/repository.git with the URL of your new GitHub repository:

git remote add origin https://github.com/username/repository.git

```
param@Pampi MINGW64 /e/SCM (master)
$ git remote add origin https://github.com/Pampi23/NewRepo.git
```

## 4. Push Your Local Repository to GitHub

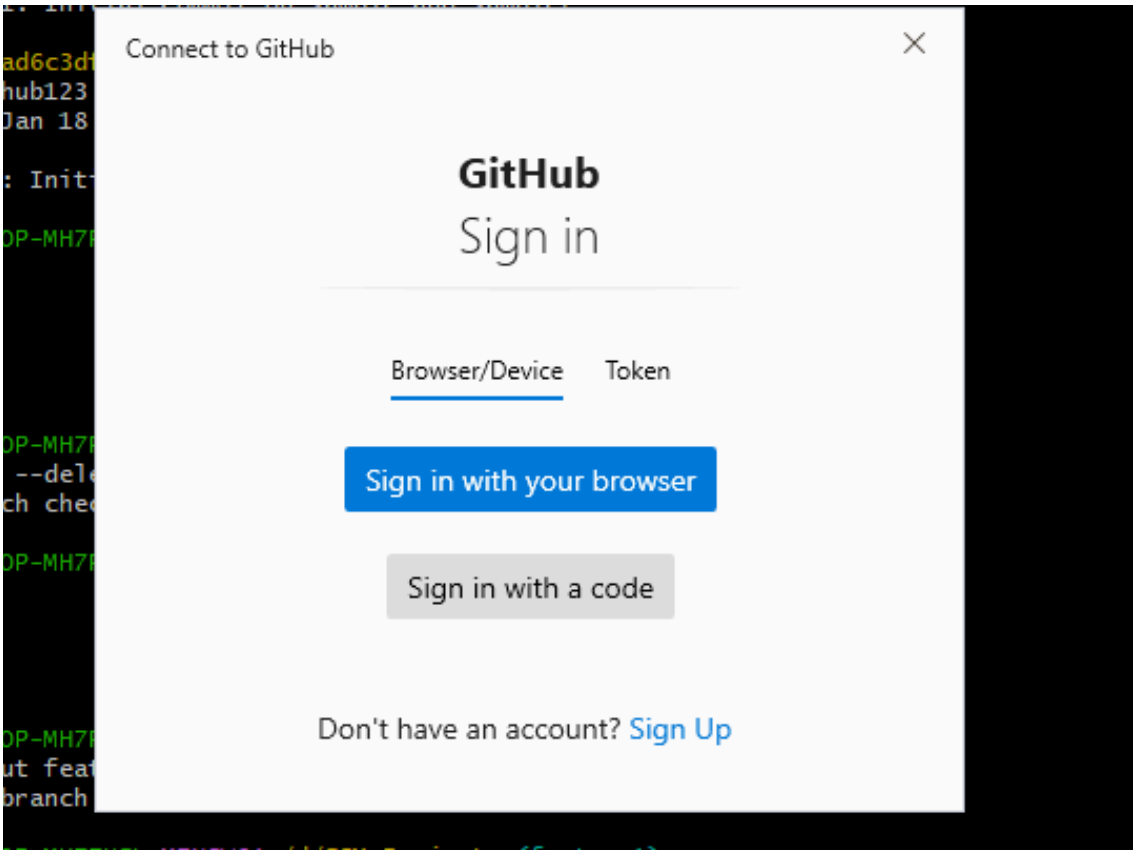
Before pushing, it's a good practice to pull any changes from the remote repository, ensuring that your local repository is up to date:

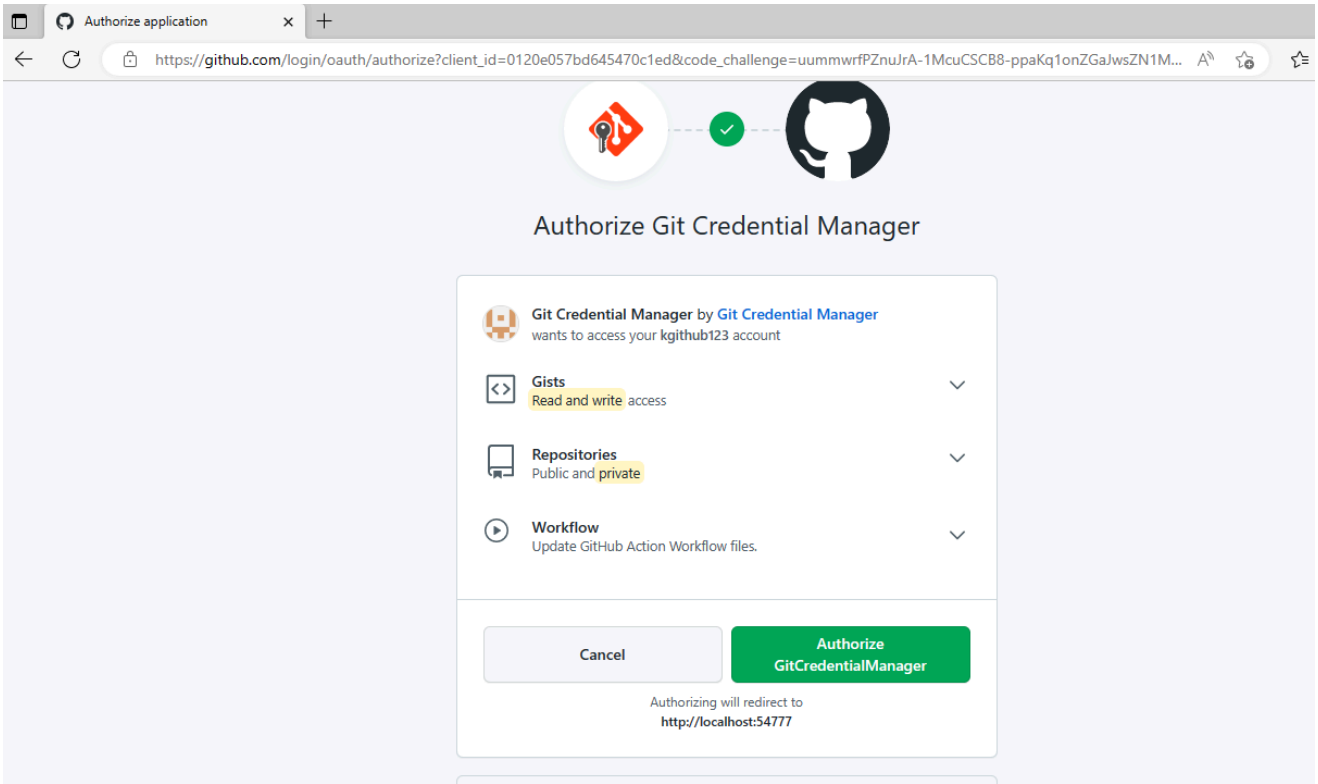
```
param@Pampi MINGW64 /e/SCM (master)
$ git checkout master
Already on 'master'
```

Then, push your local commits to GitHub with:

git push -u origin main

```
param@Pampi MINGW64 /e/SCM (master)
$ git push -u origin master
Enumerating objects: 33, done.
Counting objects: 100% (33/33), done.
Delta compression using up to 12 threads
Compressing objects: 100% (22/22), done.
Writing objects: 100% (33/33), 19.22 KiB | 1.37 MiB/s, done.
Total 33 (delta 9), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (9/9), done.
To https://github.com/Pampi23/NewRepo.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```


Connect to GitHub
GitHub
Sign in
Browser/Device
Token
Sign in with your browser
Sign in with a code
Don't have an account? Sign Up


Authorize application
https://github.com/login/oauth/authorize?client_id=0120e057bd645470c1ed&code_challenge=uummwrfPZnuJrA-1McuCSCB8-ppaKq1onZGaJwsZN1M...
Authorize Git Credential Manager
Git Credential Manager by Git Credential Manager
wants to access your kgithub123 account
Gists
Read and write access
Repositories
Public and private
Workflow
Update GitHub Action Workflow files.
Cancel
Authorize GitCredentialManager
Authorizing will redirect to
http://localhost:54777

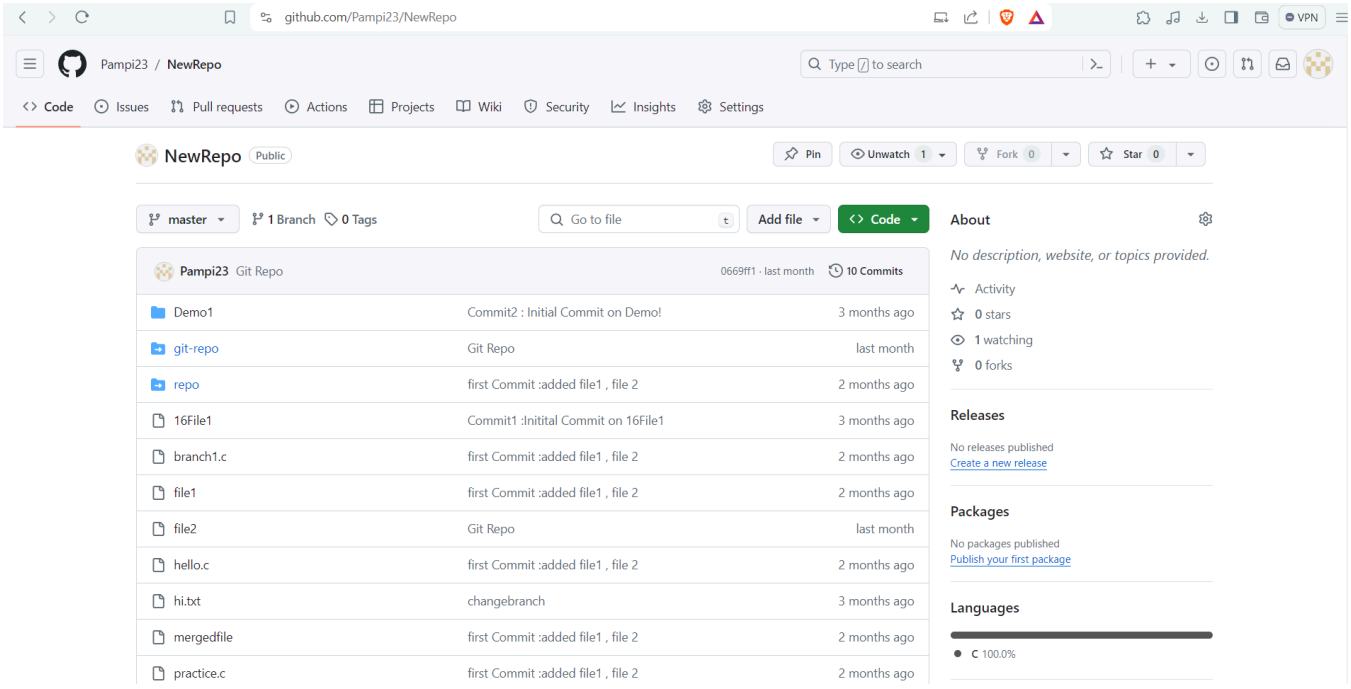
**Experiment 2:**

**Aim:**To Add collaborators on Github repository And Git Cloning- Cloning repository, Exploring contents of the cloned repository, Exploring cloned repository in GitHub Desktop, Commit changes in the cloned repository Git

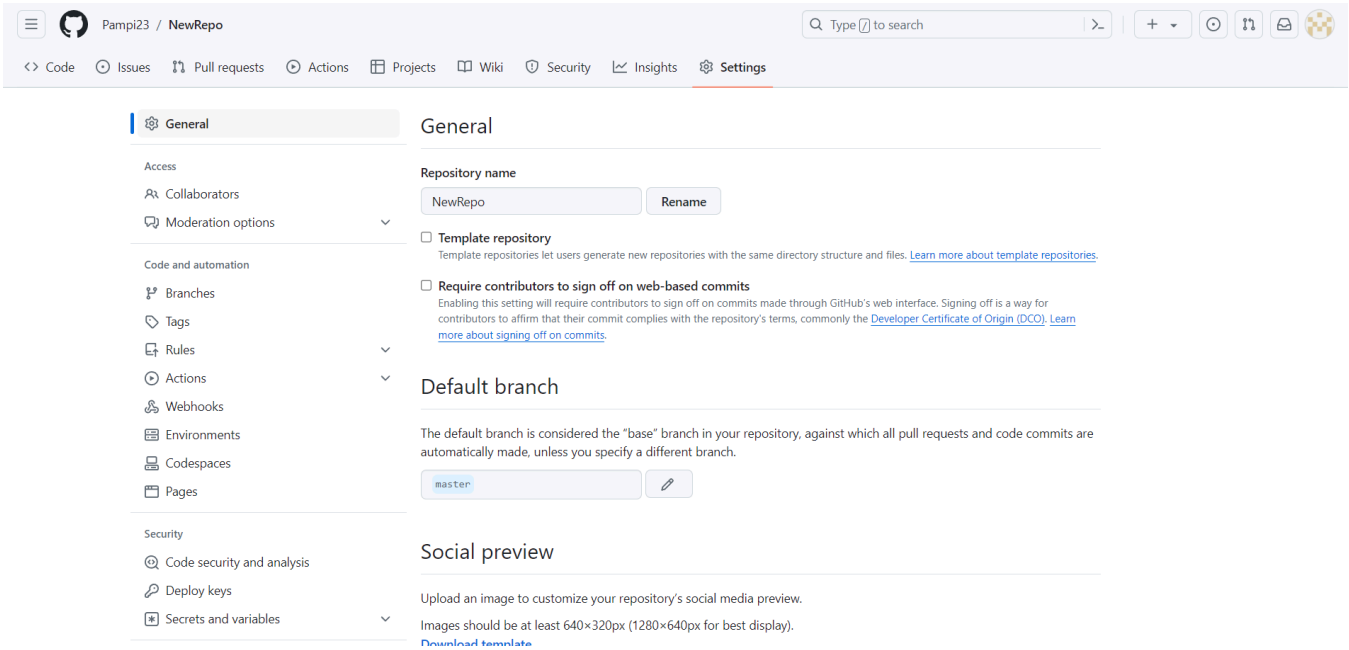
**Theory:**

# Adding Collaborators on GitHub Repository

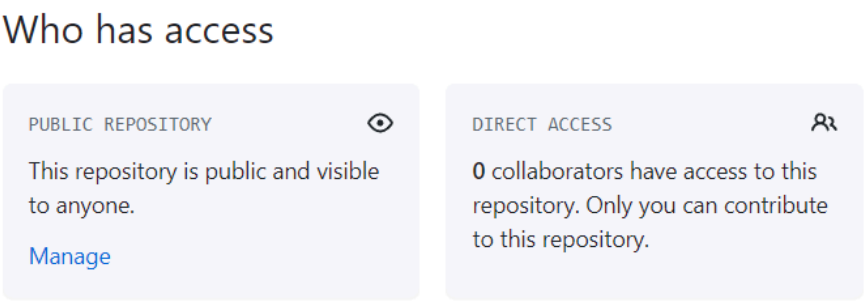
Navigate to Your Repository: Log in to your GitHub account and go to the repository where you want to add collaborators.

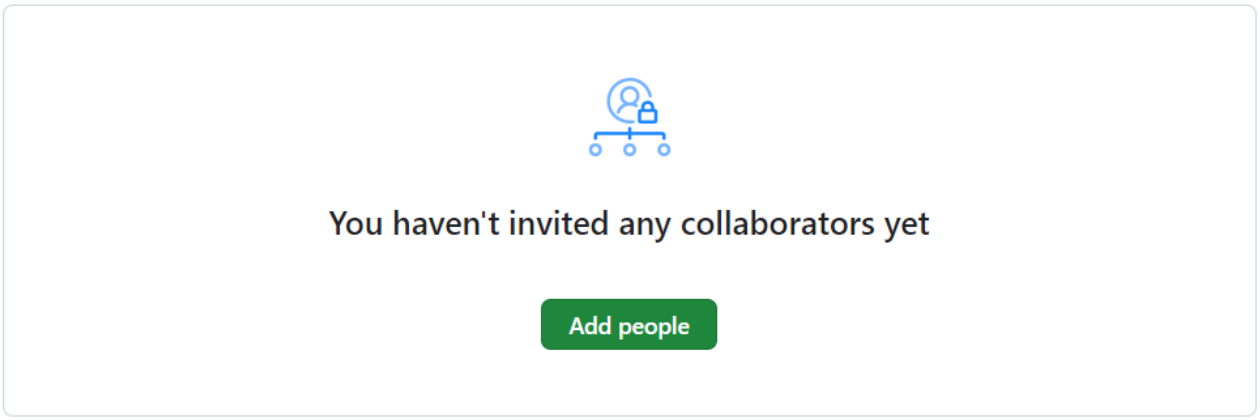



Go to Settings: Find the "Settings" tab in the repository menu.

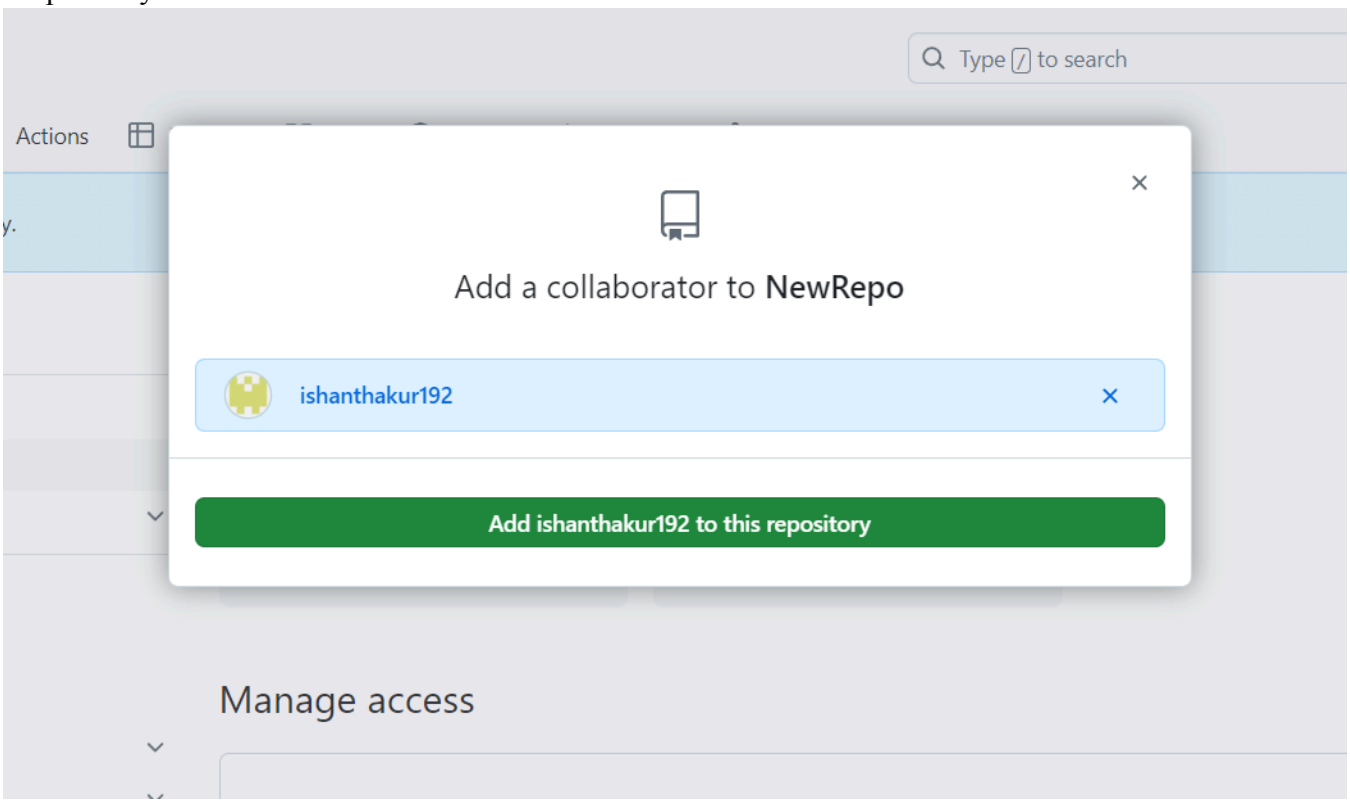



Access Manage Access: On the left sidebar, click on "Manage access". You might need to verify your password.

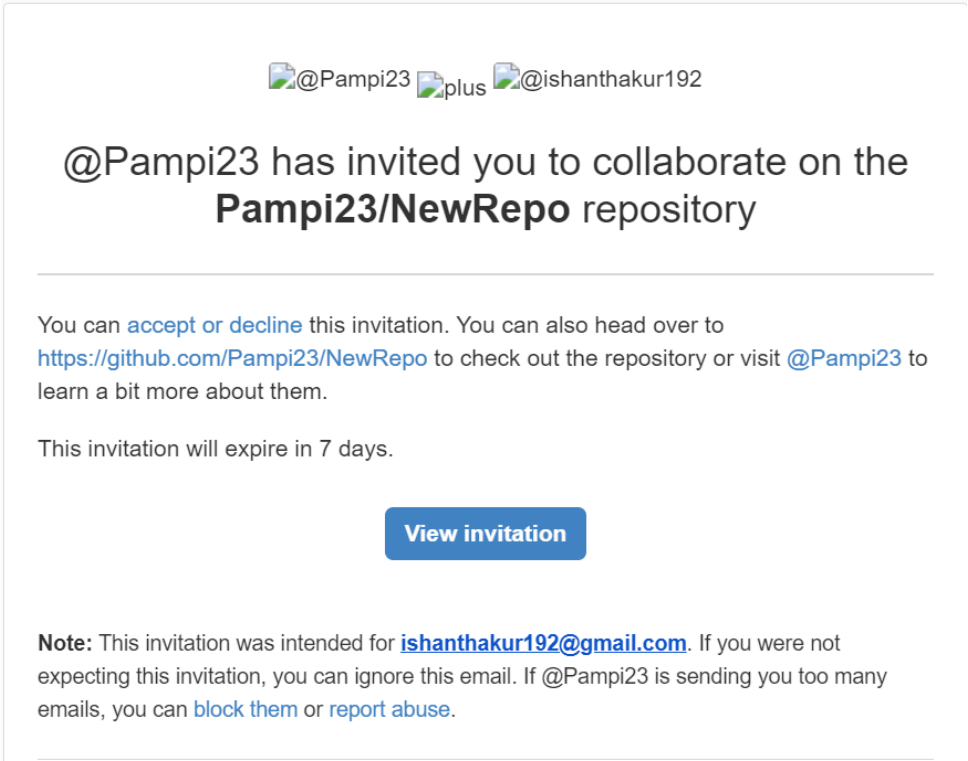


Invite a Collaborator: Click on the "Invite a collaborator" button. Enter the GitHub username of the person you want to add and select them from the list.



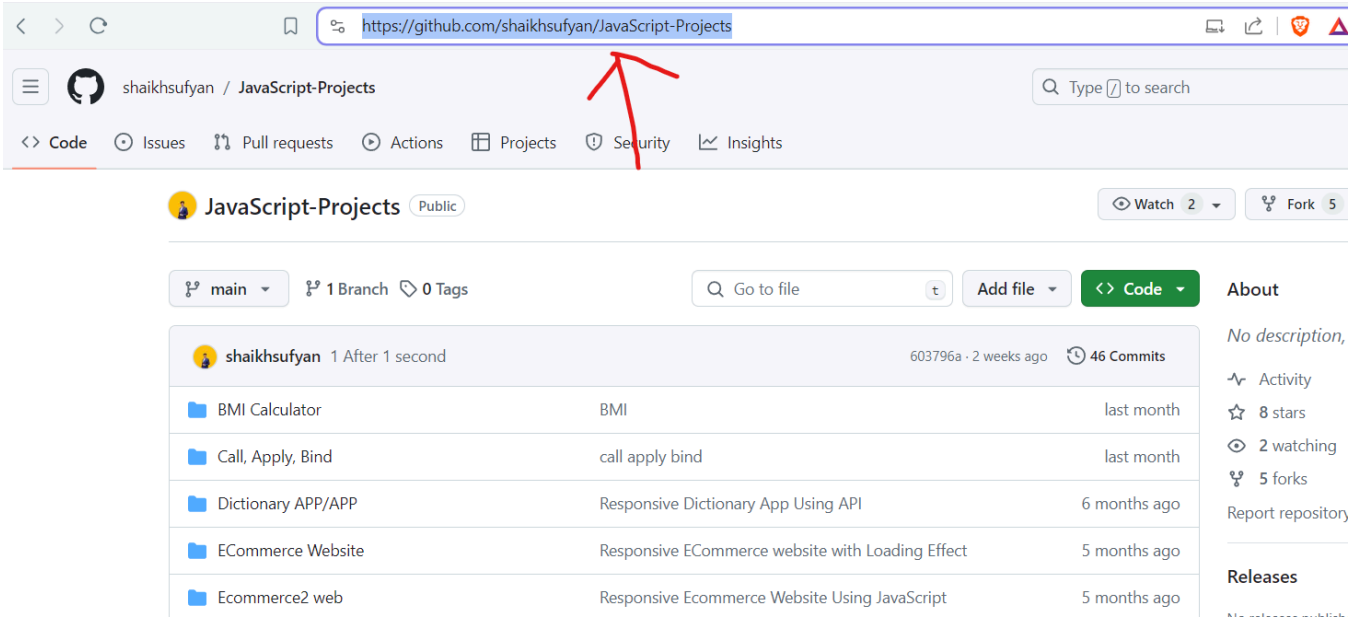

Send Invite: Once you've selected the collaborator, click the "Add collaborator" button to send an invite. The user will receive an email invitation to join the repository as a collaborator.



# Cloning a Repository

To clone a repository, you'll need its URL from GitHub:

Find the Repository URL: On GitHub, navigate to the repository you want to clone. Click the "Code" button and copy the URL under "Clone with HTTPS".




Clone the Repository: Open your terminal or command prompt, navigate to the directory where you want the cloned repository to reside, and run:
git clone https://github.com/username/repository.git

```
param@Pampi MINGW64 /e/SCM (master)
$ git clone https://github.com/shaikhsufyan/JavaScript-Projects.git
Cloning into 'JavaScript-Projects'...
remote: Enumerating objects: 407, done.
remote: Counting objects: 100% (165/165), done.
remote: Compressing objects: 100% (123/123), done.
remote: Total 407 (delta 66), reused 135 (delta 41), pack-reused 242
Receiving objects: 100% (407/407), 137.12 MiB | 5.10 MiB/s, done.
Resolving deltas: 100% (138/138), done.
Updating files: 100% (254/254), done.
```

## Exploring the Contents of the Cloned Repository

After cloning, you can navigate into the repository's directory using:

```
param@Pampi MINGW64 /e/SCM (master)
$ cd JavaScript-Projects
```

Then, you can list the files and directories within using:
ls - la

```
param@Pampi MINGW64 /e/SCM/JavaScript-Projects (main)
$ ls -la
total 64
drwxr-xr-x 1 param 197609 0 Apr  1 21:19  ./
drwxr-xr-x 1 param 197609 0 Apr  1 21:19  ../
drwxr-xr-x 1 param 197609 0 Apr  1 21:19  .git/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'BMI Calculator'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Call, Apply, Bind'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Dictionary APP'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'ECommerce Website'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Ecommerce2 web'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Ecommerce3 Website'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Fitness Website'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Food Website'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Food Website2'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Greeting Website'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Gym Web 2'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Hospital Website'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Hospital Website2'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'How JS Execute'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Notes App'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Portfolio Website'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Resp Food Website'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'Typing Speed'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'print after 1 sec'/
drwxr-xr-x 1 param 197609 0 Apr  1 21:19 'share popup js'/
```

## Commit Changes in the Cloned Repository

To commit changes:

Make Changes: Edit, add, or delete files in your repository directory as needed.

```
param@Pampi MINGW64 /e/SCM/JavaScript-Projects (main)
$ touch Demo.txt

param@Pampi MINGW64 /e/SCM/JavaScript-Projects (main)
$ vim Demo.txt
```

Stage the Changes: Open your terminal or command prompt, navigate to your repository, and run:
git add .

```
param@Pampi MINGW64 /e/SCM/JavaScript-Projects (main)
$ git add .
warning: in the working copy of 'Demo.txt', LF will be replaced by CRLF the next time Git touches it
```

Commit the Changes: Commit your changes with:

# git commit -m "Your commit message"

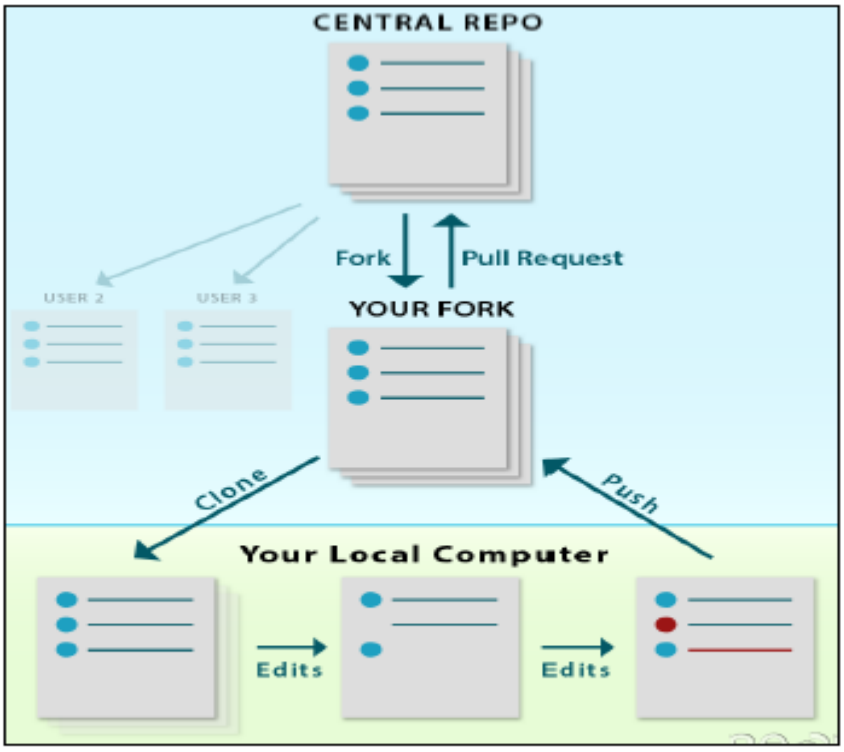
```
param@Pampi MINGW64 /e/SCM/JavaScript-Projects (main)
$ git commit -m "Demo file commit"
[main f0a8ffd] Demo file commit
 2 files changed, 1 insertion(+)
 create mode 100644 Demo.txt
 create mode 100644 demofile.txt
```

**Experiment 3:**

**Aim:**Fork and commit on our repository - Fork is a copy of a repository that you manage After creating a fork, we can make any change like adding collaborators, rename files

**Theory:**Forking a repository on GitHub allows you to freely experiment with changes without affecting the original project. It's particularly useful for proposing changes to someone else's project (via pull requests) or using someone else's project as a starting point for your own idea. Here's how to fork a repository and make changes to your fork, such as adding collaborators, renaming files, or committing changes.

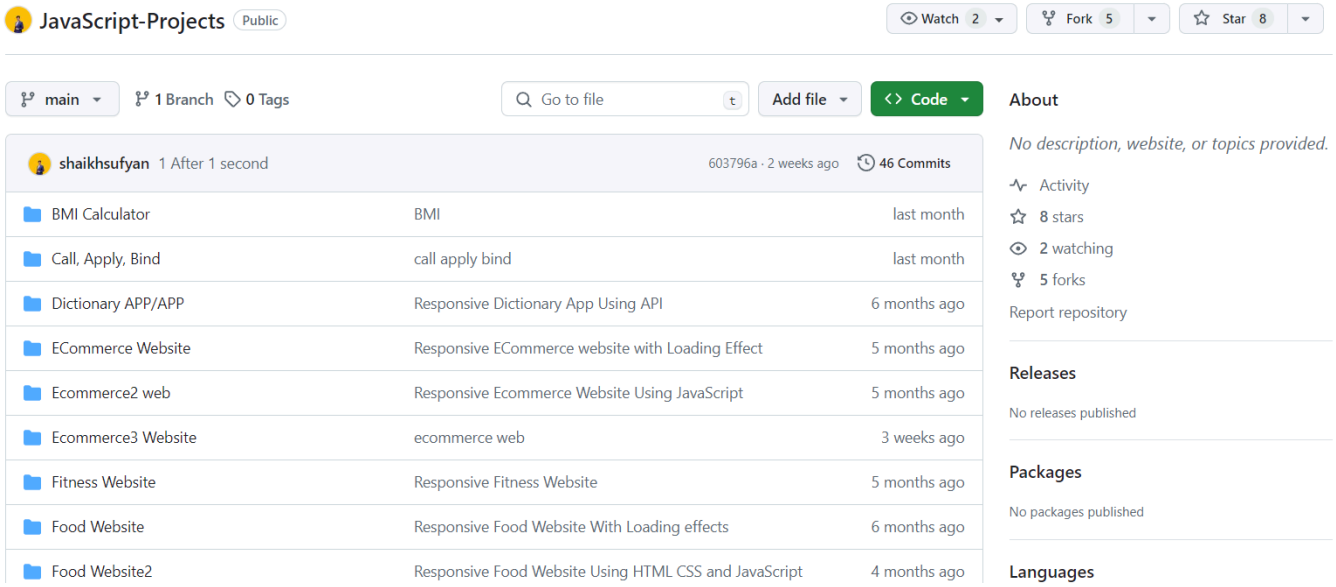
- A fork is a copy of a repository that you manage.
- It allows us to freely experiment with the data.
- After creating a fork, we can make any desired change like adding collaborators, rename files, generate GitHub pages but all these changes won't be reflected in the original repository

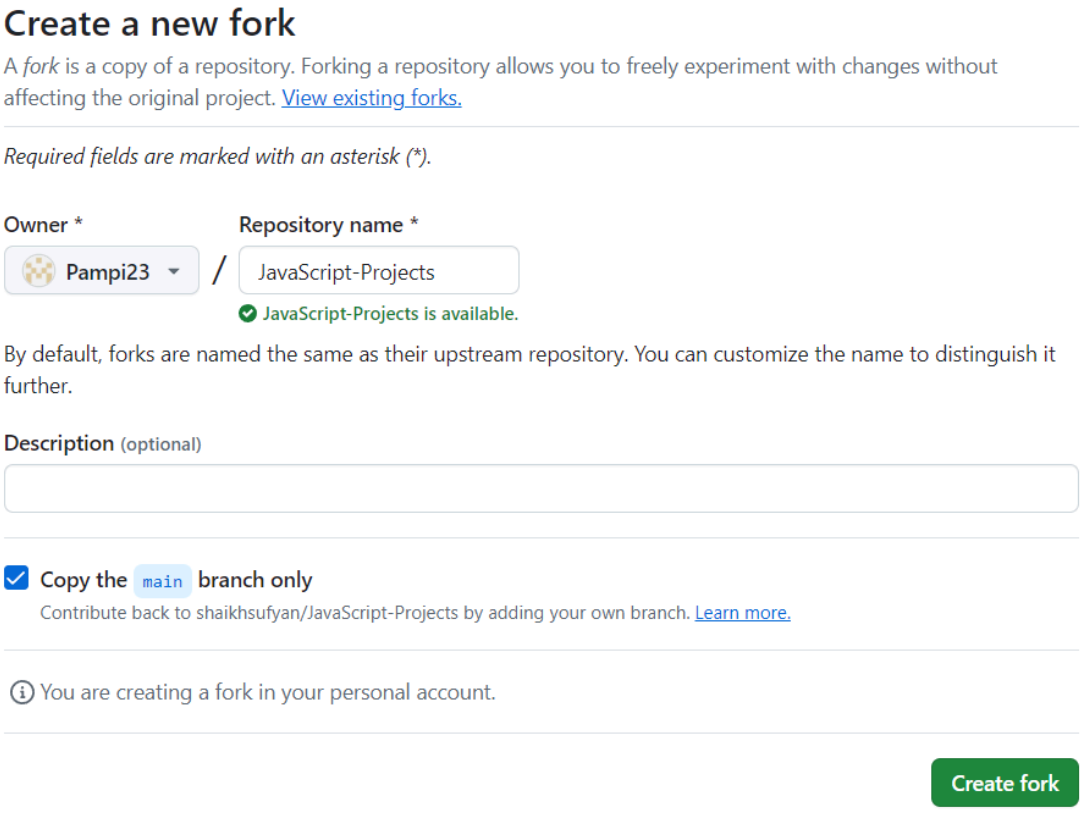



reference for picture: https://www.earthdatascience.org

# How to Fork a Repository

Navigate to the Repository: Go to the GitHub page of the repository you wish to fork.
Fork the Repository: Click on the "Fork" button, usually located at the top right of the page.
GitHub will create a copy of the repository in your account.



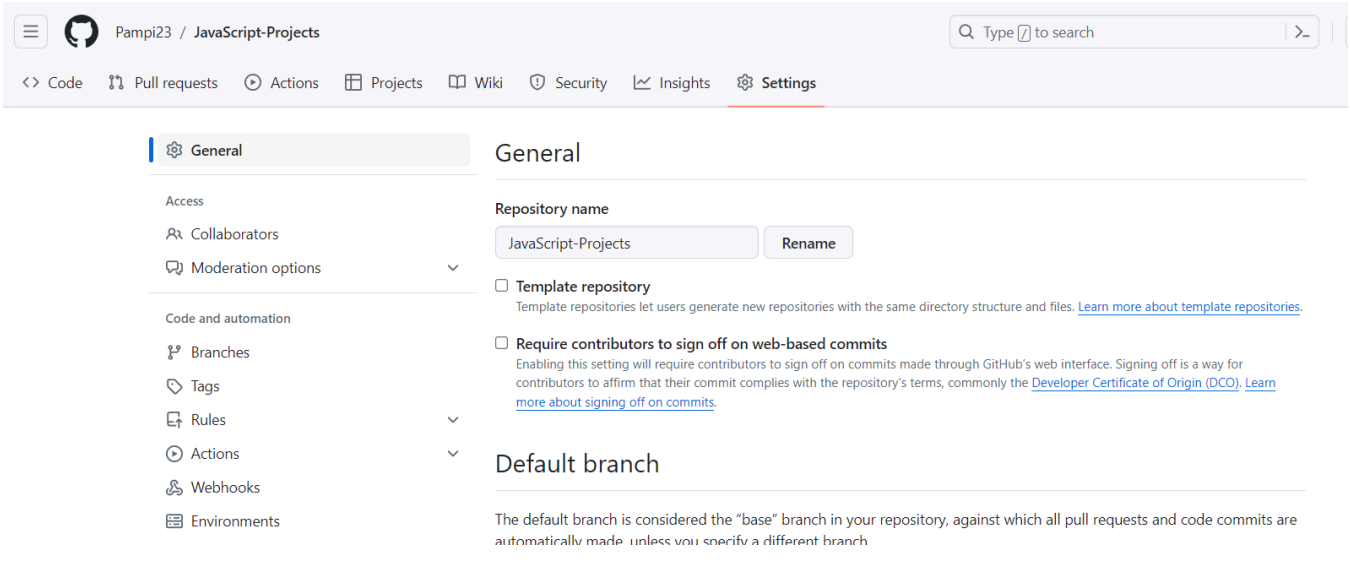



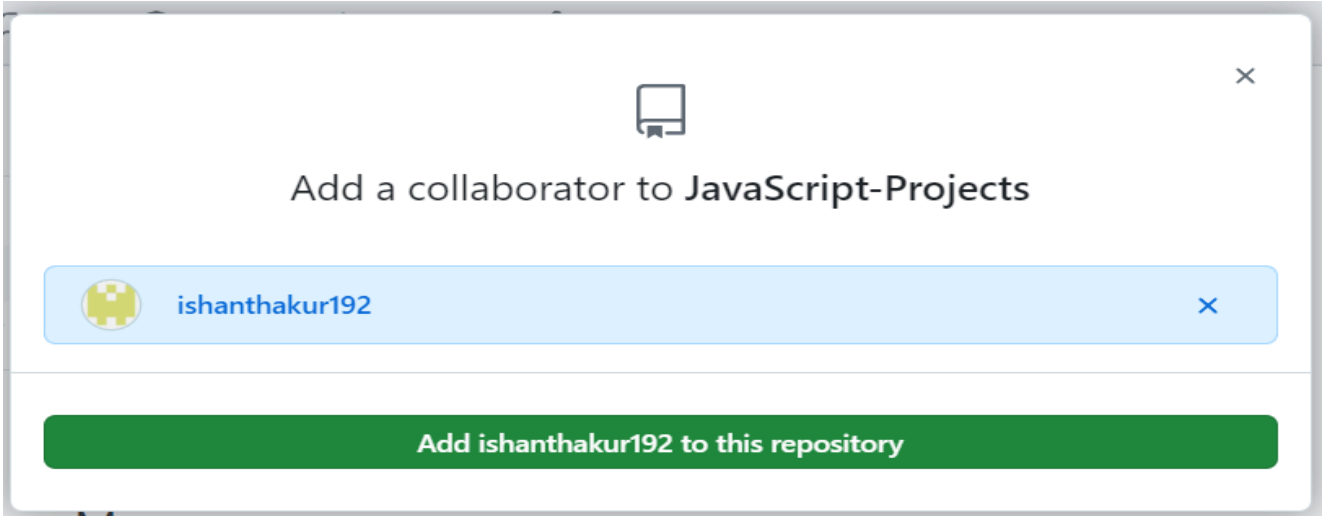
# Making Changes to Your Fork

After forking, you can make changes like adding collaborators, renaming files, or any other modifications as you see fit. Here's how to perform some common tasks:

Adding Collaborators to Your Fork

Go to Your Fork: Navigate to the forked repository on your GitHub account.
Access Settings: Click on the "Settings" tab, then select "Manage access" from the left sidebar.
Invite Collaborators: Click on "Invite a collaborator", search for their GitHub username, and then send an invite.



# @Pampi23 has invited you to collaborate on the **Pampi23/JavaScript-Projects** repository

You can accept or decline this invitation. You can also head over to https://github.com/Pampi23/JavaScript-Projects to check out the repository or visit @Pampi23 to learn a bit more about them.

This invitation will expire in 7 days.

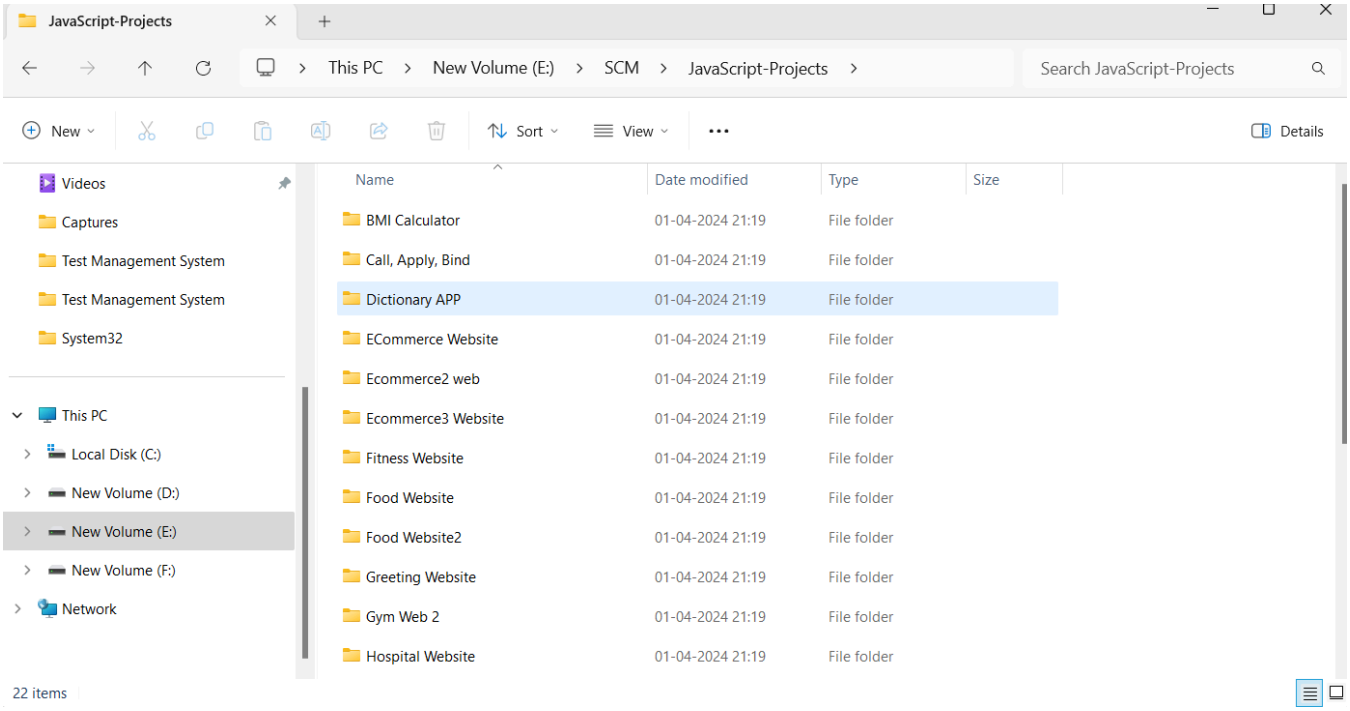
**View invitation**

**Note:** This invitation was intended for **ishanthakur192@gmail.com**. If you were not expecting this invitation, you can ignore this email. If @Pampi23 is sending you too many emails, you can block them or report abuse.

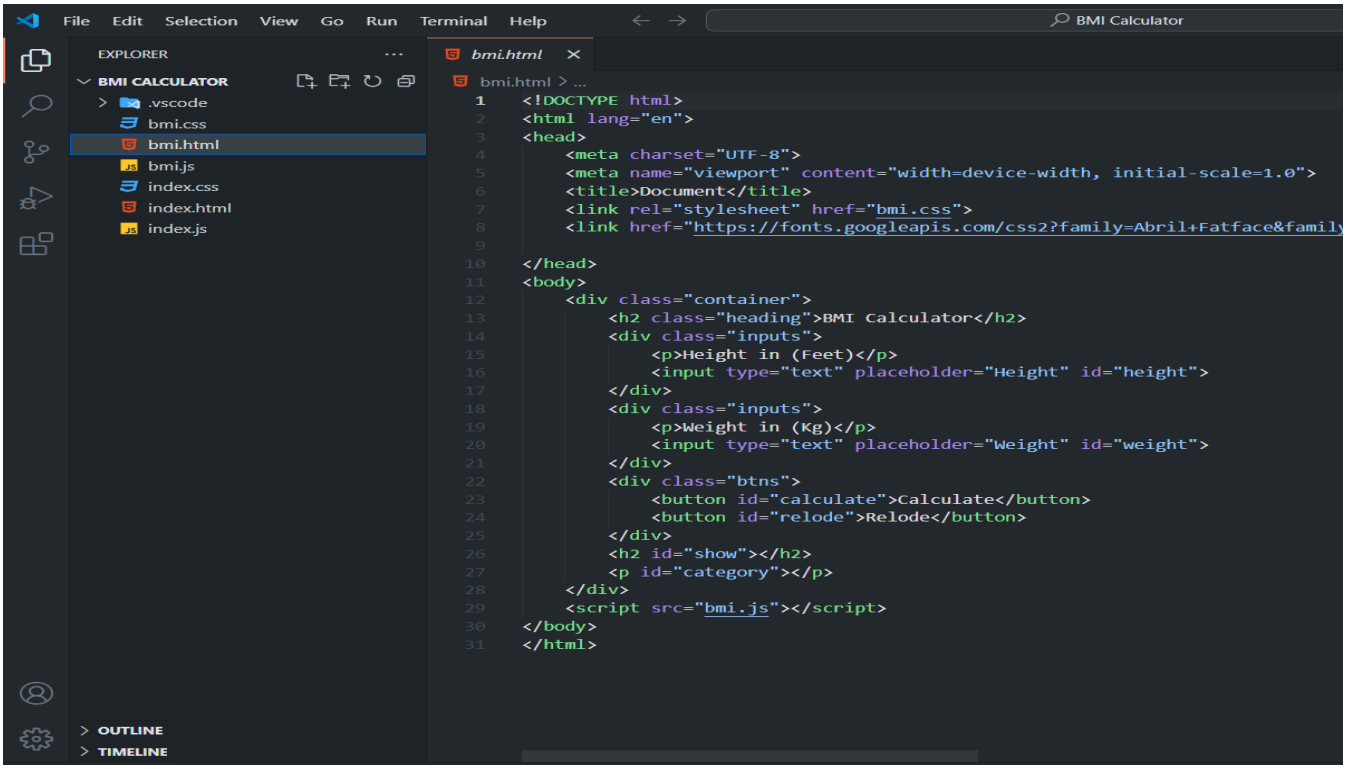
**Getting a 404 error?** Make sure you're signed in as **ishanthakur192**.

**Button not working?** Copy and paste this link into your browser:
https://github.com/Pampi23/JavaScript-Projects/invitations

Forked repo in GUI




Open a file from the forked repo



Update a file :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <link rel="stylesheet" href="bmi.css">
    <link href="https://fonts.googleapis.com/css2?family=Abril+Fatface&family=Catamaran:wght@200&family=Courgette&family=Dancing+Scr

</head>
<body>
    <div class="container">
        <h2 class="heading">BMI Calculator</h2>
        <div class="inputs">
            <p>height in (Feet)</p>
            <inp
        </div>
        <div cla
            <p>W
            <inp
        </div>
        <div cla
            <but
            <but
        </div>
        <h2 id="
        <p id="c
    </div>
    <script src="bmi.js"></script>
</body>
</html>
```

h1 Emmet Abbreviation
h2
h3
h4
h5
h6
hdr
head
header
hgroup
hr
html

Check the origin created earlier to push the changes to Github

```
param@Pampi MINGW64 /e/SCM/JavaScript-Projects (main)
$ git remote -v
origin  https://github.com/shaikhsufyan/JavaScript-Projects.git (fetch)
origin  https://github.com/shaikhsufyan/JavaScript-Projects.git (push)
```

Push the changes to Github

```
$ git push -u origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 309 bytes | 309.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/kgithub123/AWD-Project
   4823bff..74a5e98  main -> main
branch 'main' set up to track 'origin/main'.
```

**Experiment 4:**

**Aim:**Merge and Resolve conflicts - Resolution of merge conflicts, GitHub and remote repositories, Cloning remote repository, resolving conflicts

**Theory:**Merge conflicts occur when Git is unable to automatically resolve differences in code between two commits. If two contributors make changes to the same line in a file or if one deletes a file while the other is modifying it, Git will not be able to merge these changes automatically. Resolving conflicts is a critical skill in software development, especially when collaborating with others. Here's how you can handle merge conflicts, both locally and when working with GitHub and remote repositories.

Create a file in master branch and commit it.

```
param@Pampi MINGW64 /e/SCM (master)
$ mkdir scm_merge

param@Pampi MINGW64 /e/SCM (master)
$ cd scm_merge

param@Pampi MINGW64 /e/SCM/scm_merge (master)
$ git init
Initialized empty Git repository in E:/SCM/scm_merge/.git/

param@Pampi MINGW64 /e/SCM/scm_merge (master)
$ cat >> file1.txt
this is new file
```

```
param@Pampi MINGW64 /e/SCM/scm_merge (master)
$ git add .
warning: in the working copy of 'file1.txt', LF will be replaced by CRLF the nex
t time Git touches it

param@Pampi MINGW64 /e/SCM/scm_merge (master)
$ git commit -m "First commit"
[master (root-commit) 89ba778] First commit
 1 file changed, 1 insertion(+)
 create mode 100644 file1.txt
```

Create a branch and checkout in it

```
param@Pampi MINGW64 /e/SCM/scm_merge (master)
$ git branch branch1

param@Pampi MINGW64 /e/SCM/scm_merge (master)
$ git checkout branch1
Switched to branch 'branch1'
M       file1.txt
```

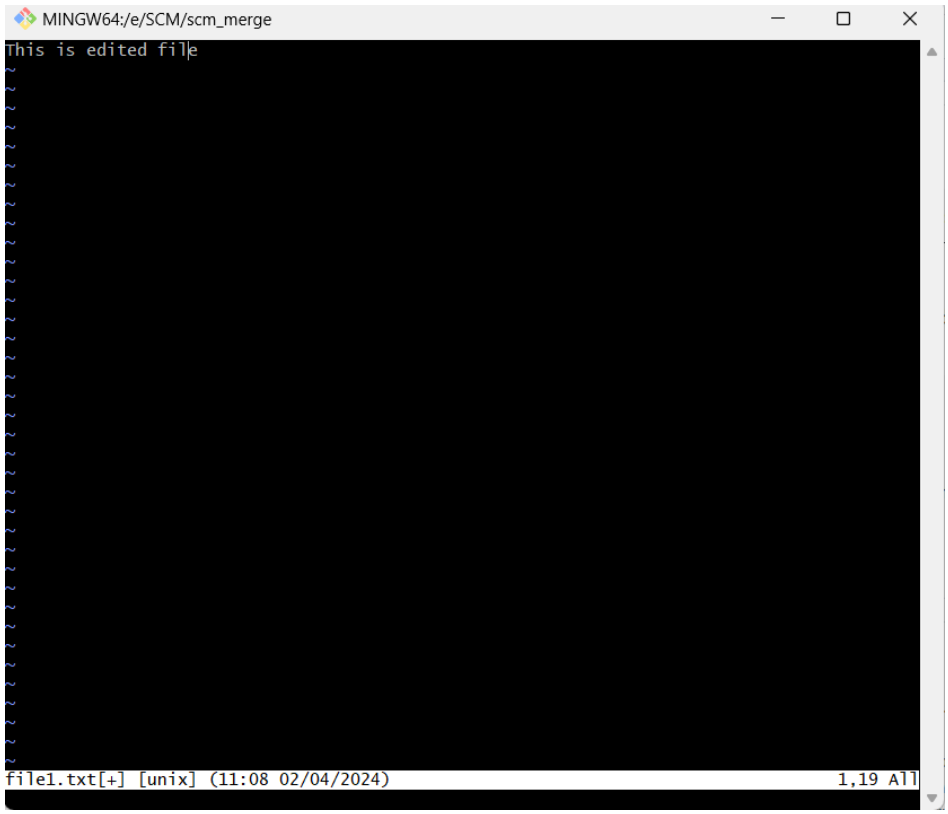
Open vi editor and add a line in the same file

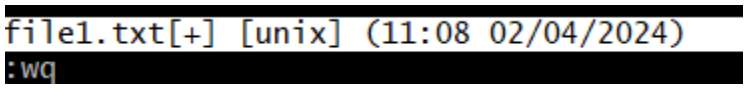
- Open vi editor by command

vi filename

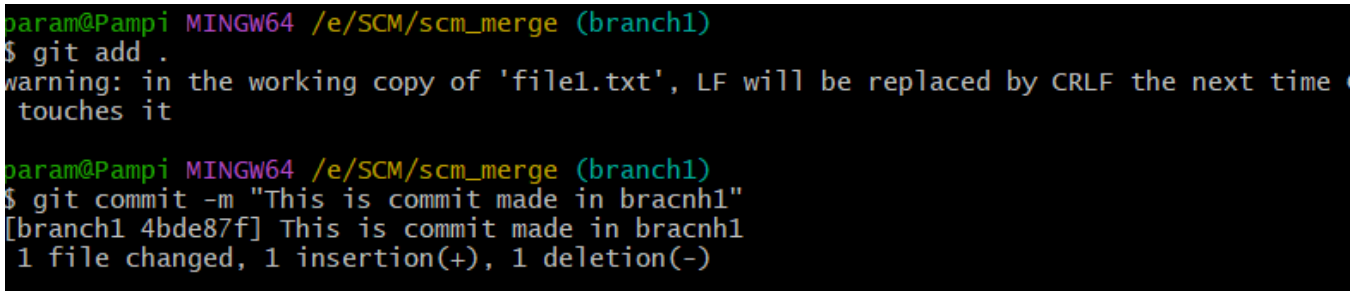
- I for insert mode
- Type text you want to add
- Esc:wq to save and exit vi editor.

```
param@Pampi MINGW64 /e/SCM/scm_merge (branch1)
$ vi file1.txt
```




Commit the changes made in branch_1



Checkout to master and see the contents of file1 , the changes made in branch 1 are not reflected in master

```
param@Pampi MINGW64 /e/SCM/scm_merge (branch1)
$ git checkout master
Switched to branch 'master'

param@Pampi MINGW64 /e/SCM/scm_merge (master)
$ cat file1.txt
this is new file
```

Merge the branch_1 with master and view the changes being reflected in the file

```
param@Pampi MINGW64 /e/SCM/scm_merge (master)
$ git merge branch1
Updating 89ba778..4bde87f
Fast-forward
 file1.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

param@Pampi MINGW64 /e/SCM/scm_merge (master)
$ cat file1.txt
This is edited file1.txt
```

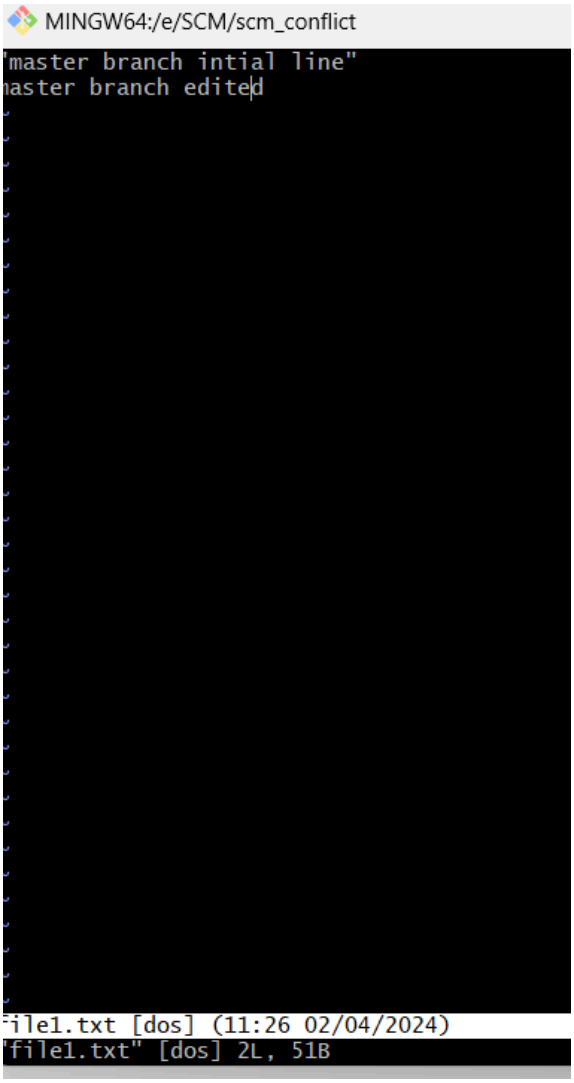
Resolve conflict:

```
param@Pampi MINGW64 /e/SCM/scm_merge (master)
$ cd ..

param@Pampi MINGW64 /e/SCM (master)
$ mkdir scm_conflict

param@Pampi MINGW64 /e/SCM (master)
$ cd scm_conflict

param@Pampi MINGW64 /e/SCM/scm_conflict (master)
$ cat >> file1.txt
master branch intial line
param@Pampi MINGW64 /e/SCM/scm_conflict (master)
$ git init
Initialized empty Git repository in E:/SCM/scm_conflict/.git/

param@Pampi MINGW64 /e/SCM/scm_conflict (master)
$ git add .

param@Pampi MINGW64 /e/SCM/scm_conflict (master)
$ git commit -m "first commit in master"
[master (root-commit) 4e190f4] first commit in master
 1 file changed, 1 insertion(+)
 create mode 100644 file1.txt
```

```
param@Pampi MINGW64 /e/SCM/scm_conflict (master)
$ git branch branch1
fatal: a branch named 'branch1' already exists

param@Pampi MINGW64 /e/SCM/scm_conflict (master)
$ git checkout branch1
Switched to branch 'branch1'

param@Pampi MINGW64 /e/SCM/scm_conflict (branch1)
$ vi file1.txt

param@Pampi MINGW64 /e/SCM/scm_conflict (branch1)
$ git add .
warning: in the working copy of 'file1.txt', LF will be replaced by CRLF
s it

param@Pampi MINGW64 /e/SCM/scm_conflict (branch1)
$ giy commit -m "Initial commit in branch 1"
bash: giy: command not found

param@Pampi MINGW64 /e/SCM/scm_conflict (branch1)
$ git commit -m "inital commit in branch 1"
[branch1 070f0cc] inital commit in branch 1
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Return to master branch and change the file and commit

```
param@Pampi MINGW64 /e/SCM/scm_conflict (branch1)
$ git checkout master
Switched to branch 'master'

param@Pampi MINGW64 /e/SCM/scm_conflict (master)
$ vi file1.txt
```



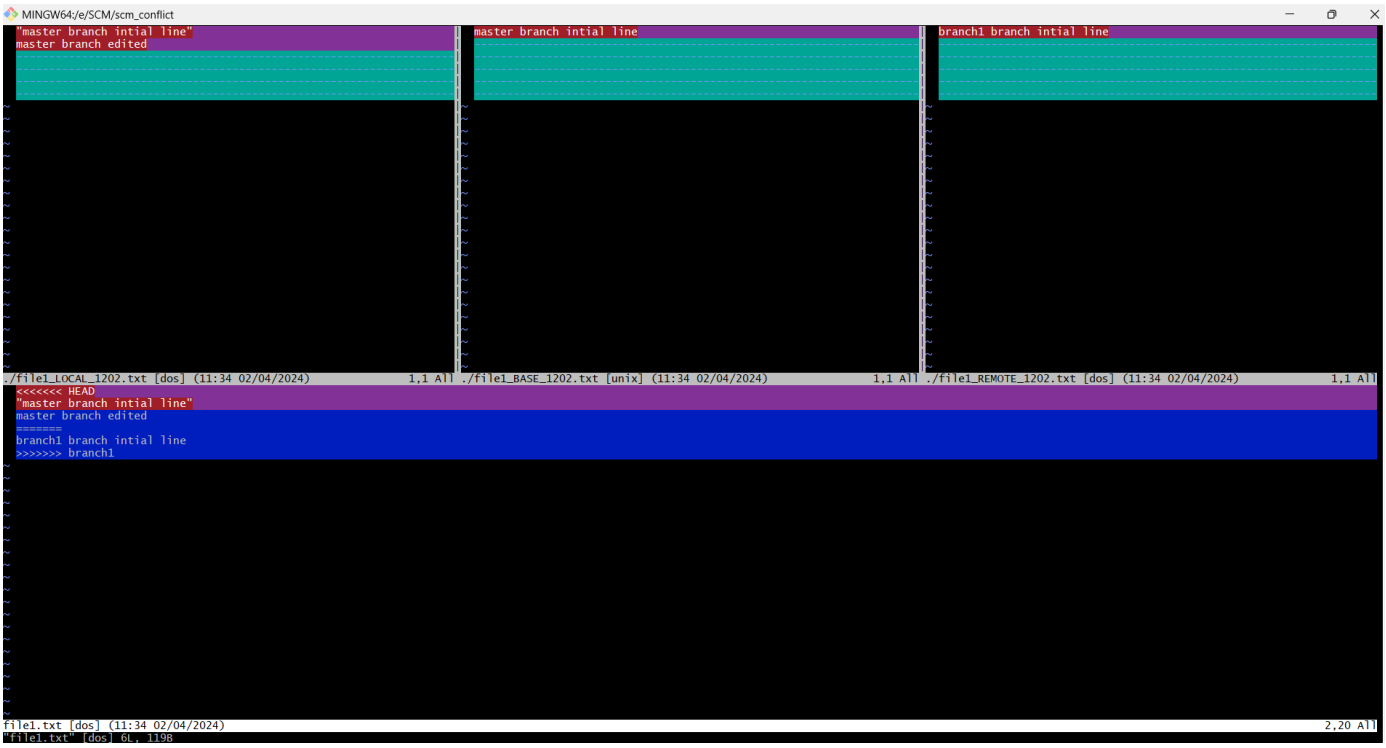
Merge branch_1 with master and view the conflict occurred

```
param@Pampi MINGW64 /e/SCM/scm_conflict (master)
$ git merge branch1
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
Automatic merge failed; fix conflicts and then commit the result.

param@Pampi MINGW64 /e/SCM/scm_conflict (master|MERGING)
$
```

Use merge tool to resolve conflict:

```
param@Pampi MINGW64 /e/SCM/scm_conflict (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:
file1.txt

Normal merge conflict for 'file1.txt':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (vimdiff):
```

Mergetool opened:




Make the desired changes in the last window

Exit mergetool esc:wq and view the final file in which the conflicts have been resolved:

```
param@Pampi MINGW64 /e/SCM/scm_conflict (master|MERGING)
$ cat file1.txt
<<<<<<< HEAD
"master"
master branch edited
=======
branch1 branch intial line
>>>>>>> branch1
```

**Experiment 5:**

**Aim:**Git Reset (soft reset and hard reset), Git revert - Performing Soft reset and hard reset , Git revert

**Theory:**Understanding the differences between git reset and git revert, and when to use each, is crucial for managing your project's history effectively. Both commands are used to undo changes, but they do it in very different ways.

In Git, reset, soft reset, hard reset, and revert are all commands that help manage changes to a project's history, but they serve different purposes and have different consequences. It's crucial to understand these differences, especially when collaborating with others, to avoid potential issues with your project's history.

**Git Reset**

git reset moves the current branch backward to a specified commit, essentially "undoing" commits that came after the target commit. The command has three main modes: soft, mixed (the default), and hard.

Soft Reset

A soft reset (git reset --soft <commit>) moves the HEAD to the specified commit, but it does not change the working directory or the staging area (index). This means that the changes from the "undone" commits are preserved as unstaged changes in your working directory. You can then re-commit these changes as a new commit. This is useful for combining several commits into a single commit.

```
param@Pampi MINGW64 /e/SCM (master)
$ mkdir scm_reset

param@Pampi MINGW64 /e/SCM (master)
$ cd scm_reset

param@Pampi MINGW64 /e/SCM/scm_reset (master)
$ git init
Initialized empty Git repository in E:/SCM/scm_reset/.git/

param@Pampi MINGW64 /e/SCM/scm_reset (master)
$ cat >> file1.txt
how are you?
param@Pampi MINGW64 /e/SCM/scm_reset (master)
$ git add .

param@Pampi MINGW64 /e/SCM/scm_reset (master)
$ git commit -m "first commit"
[master (root-commit) 67f3f5e] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 file1.txt
```

```
param@Pampi MINGW64 /e/SCM/scm_reset (master)
$ vi file1.txt

param@Pampi MINGW64 /e/SCM/scm_reset (master)
$ git add .
warning: in the working copy of 'file1.txt', LF will be replaced by CRLF the nex
t time Git touches it

param@Pampi MINGW64 /e/SCM/scm_reset (master)
$ git commit -m "second commit "
[master 85aa592] second commit
 1 file changed, 1 insertion(+), 1 deletion(-)
```

```
$ git log --oneline
85aa592 (HEAD -> master) second commit
67f3f5e first commit

param@Pampi MINGW64 /e/SCM/scm_reset (master)
$ git reset --soft HEAD~1

param@Pampi MINGW64 /e/SCM/scm_reset (master)
$ git statu
git: 'statu' is not a git command. See 'git --help'.

The most similar commands are
        status
        stage
        stash

param@Pampi MINGW64 /e/SCM/scm_reset (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt
```

# 1. Version control with Git

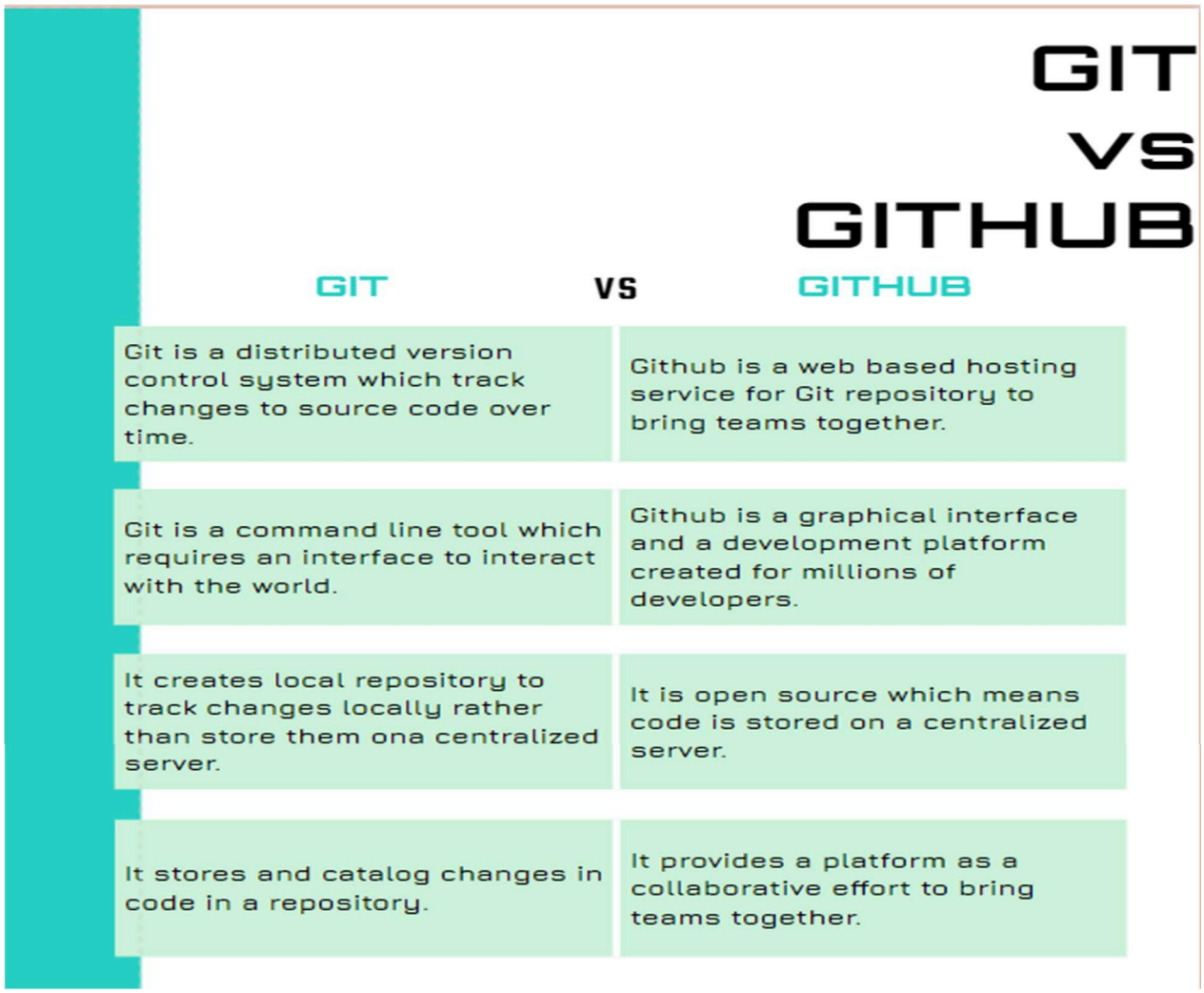
**What is GIT and why is it used?**

Git is a version control system that is widely used in the programming world. It is used for tracking changes in the source code during software development. It was developed in 2005 by Linus Torvalds, the creator of the Linux operating system kernel.
Git is a speedy and efficient distributed VCS tool that can handle projects of any size, from small to very large ones. Git provides cheap local branching, convenient staging areas, and multiple workflows. It is free, open-source software that lowers the cost because developers can use Git without paying money. It provides support for non- linear development. Git enables multiple developers or teams to work separately without having an impact on the work of others.
Git is an example of a distributed version control system (DVCS) (hence Distributed Version Control System).

**What is GITHUB?**

It is the world's largest open-source software developer community platform where the users upload their projects using the software Git.


**What is the difference between GIT and GITHUB?**




**What is Repository?**

A repository is a directory or storage space where your projects can live. Sometimes GitHub users shorten this to "repo." It can be local to a folder on your computer, or it can be a storage space on GitHub or another online host. You can keep code files, text files, image files, you name it, inside a repository.

**What is Version Control System (VCS)?**

A version control system is a tool that helps you manage "versions" of your code or changes to your code while working with a team over remote distances. Version control keeps track of every modification in a special kind of database that is accessible to the version control software. Version

control software (VCS) helps you revert back to an older version just in case a bug or issue is introduced to the system or fixing a mistake without disrupting the work of other team members.

**Types of VCS**

1. Local Version Control System
2. Centralized Version Control System
3. 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.

## 2. Problem Statement

**What is the problem. –**

1. In the digital age, secure authentication is critical for protecting user data and ensuring that only authorized individuals can access sensitive information. Many applications and websites face challenges in implementing robust and user-friendly authentication mechanisms that can defend against security threats such as password theft, session hijacking, and unauthorized access.
2. 
3. The goal of this project is to develop a secure, efficient, and user-friendly login page that utilizes modern authentication protocols and methods to safeguard user accounts. The project will address common security flaws associated with authentication such as weak password practices, lack of multifactor authentication, and vulnerability to brute force attacks.
4. 
5. The proposed login system will include features such as password hashing, session management, multifactor authentication, and user interface elements that guide users towards secure practices. It will be designed to integrate easily with existing web applications and provide a reliable framework for user authentication that meets current security standards.

**Reason for the problem's occurrence –**

**The reasons for the problems related to login page authentication typically stem from a variety of factors:**

1. Weak Password Policies: Many applications do not enforce strong password policies, allowing users to set simple, easily guessable passwords. This lack of enforcement makes accounts vulnerable to brute force attacks where attackers systematically check all possible passwords until the correct one is found.
2. Insufficient User Authentication Practices: Sole reliance on basic credentials like username and password without implementing additional security layers such as multifactor authentication (MFA) or biometrics leaves accounts more susceptible to unauthorized access, especially if credential information is stolen or leaked.
3. Poor Session Management: Inadequate session management can allow sessions to be hijacked, where attackers take over a user's session by stealing or manipulating their session tokens. Without proper mechanisms to secure and invalidate sessions, users remain logged in even after they have finished their activities, increasing the risk of unauthorized access.
4. 

**When the problem began –**

This problem got noticed between February or march of year 2020.

**How our project can help in this situation –**

The project to develop a secure and efficient login page authentication system can help address the issues outlined by implementing several key features and practices:

## 3. Objective

Enhance Security Measures: To implement robust security protocols that safeguard user data from unauthorized access, focusing on preventing common threats such as password theft, session hijacking, and brute force attacks.

Enforce Strong Password Policies: To incorporate strong password policies that require the use of complex and secure passwords, minimizing the risk of security breaches due to weak passwords.

Implement Multifactor Authentication: To integrate multifactor authentication (MFA) into the login process, adding an additional layer of security that ensures only authorized users can access their accounts even if one factor (like a password) is compromised.

**What is Git and why is it used?**

Git is a version control system that is widely used in the programming world. It is used for tracking changes in the source code during software development. It was developed in 2005 by Linus Torvalds, the creator of the Linux operating system kernel.

Git is a speedy and efficient distributed VCS tool that can handle projects of any size, from small to very large ones. Git provides cheap local branching, convenient staging areas, and multiple workflows. It is free, open-source software that lowers the cost becausedevelopers can use Git without paying money. It provides support for non-linear development. Git enables multiple developers or teams to work separately without having an impact on the work of others.

Git is an example of a **distributed version control system (DVCS)**.

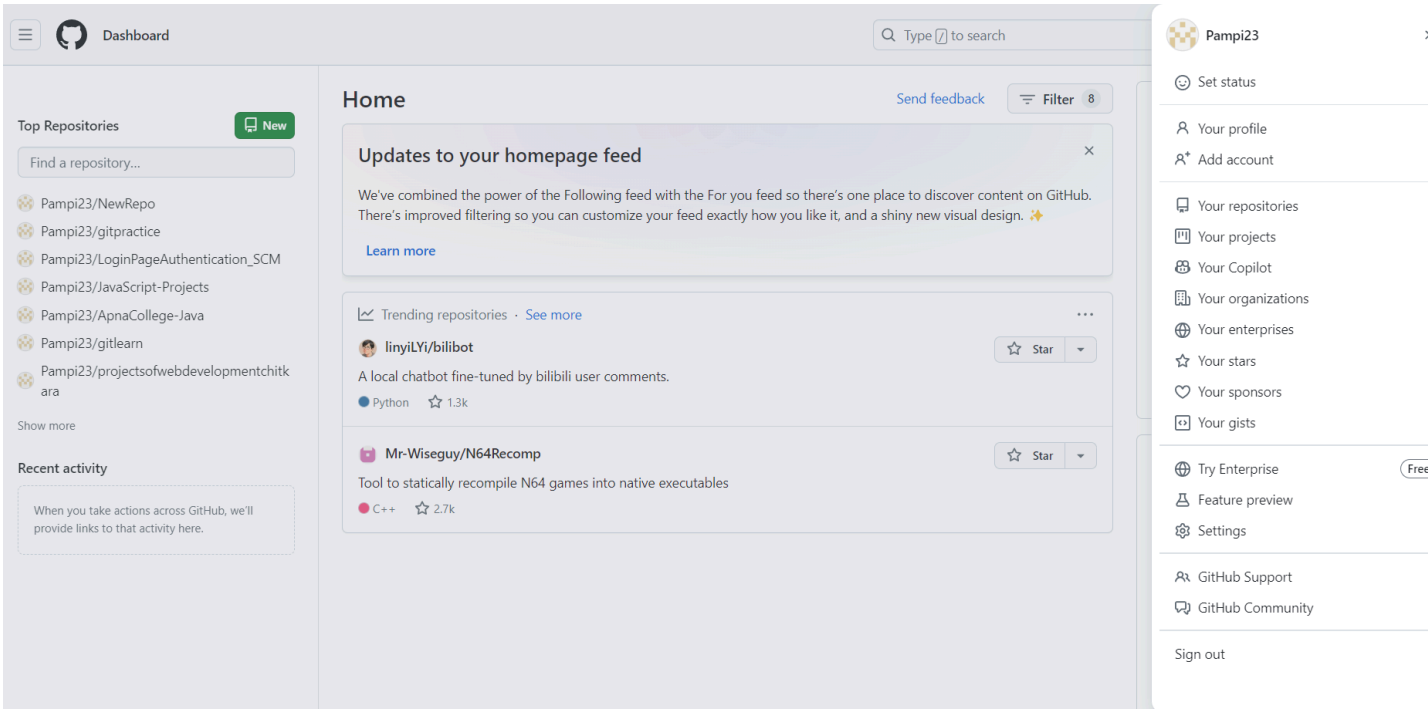
**So, what is GitHub, then?**

GitHub, Inc. is a provider of Internet hosting for software development and version control using Git. It offers the distributed version control and source code managementfunctionality of Git, plus its own features. With some chunks of code in Git, you can update your work on GitHub, making it more easy-for-use.

GitHub is a for-profit company that offers a cloud-based Git repository hosting service. Essentially, it makes it a lot easier for individuals and teams to use Git for version control and collaboration.

GitHub's interface is user-friendly enough so even novice coders can take advantage of

Git. Without GitHub, using Git generally requires a bit more technical savvy and use of the command line.



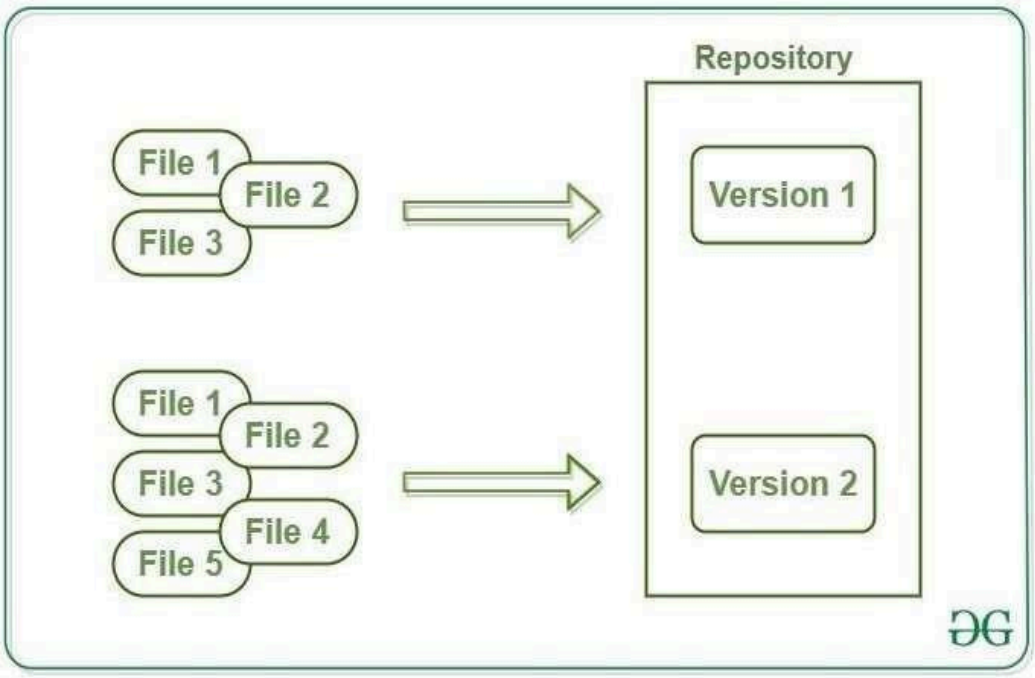

## Difference between Git and GitHub.

| S. No | Git | GitHub |
|---|---|---|
| 1. | Git is a software. | GitHub is a service. |
| 2. | Git is a command-line-tool | GitHub is a graphical user interface |
| 3. | Git is installed locally on the system | GitHub is hosted on web |
| 4. | Git has no user management feature. | GitHub has a built-in user management feature. |


**What is Repository?**

A repository is a directory or storage space where your projects can live. Sometimes GitHub users shorten this to “repo.” It can be local to a folder on your computer, or it canbe a storage space on GitHub or another online host. You can keep code files, text files, image files, you name it, inside a repository.

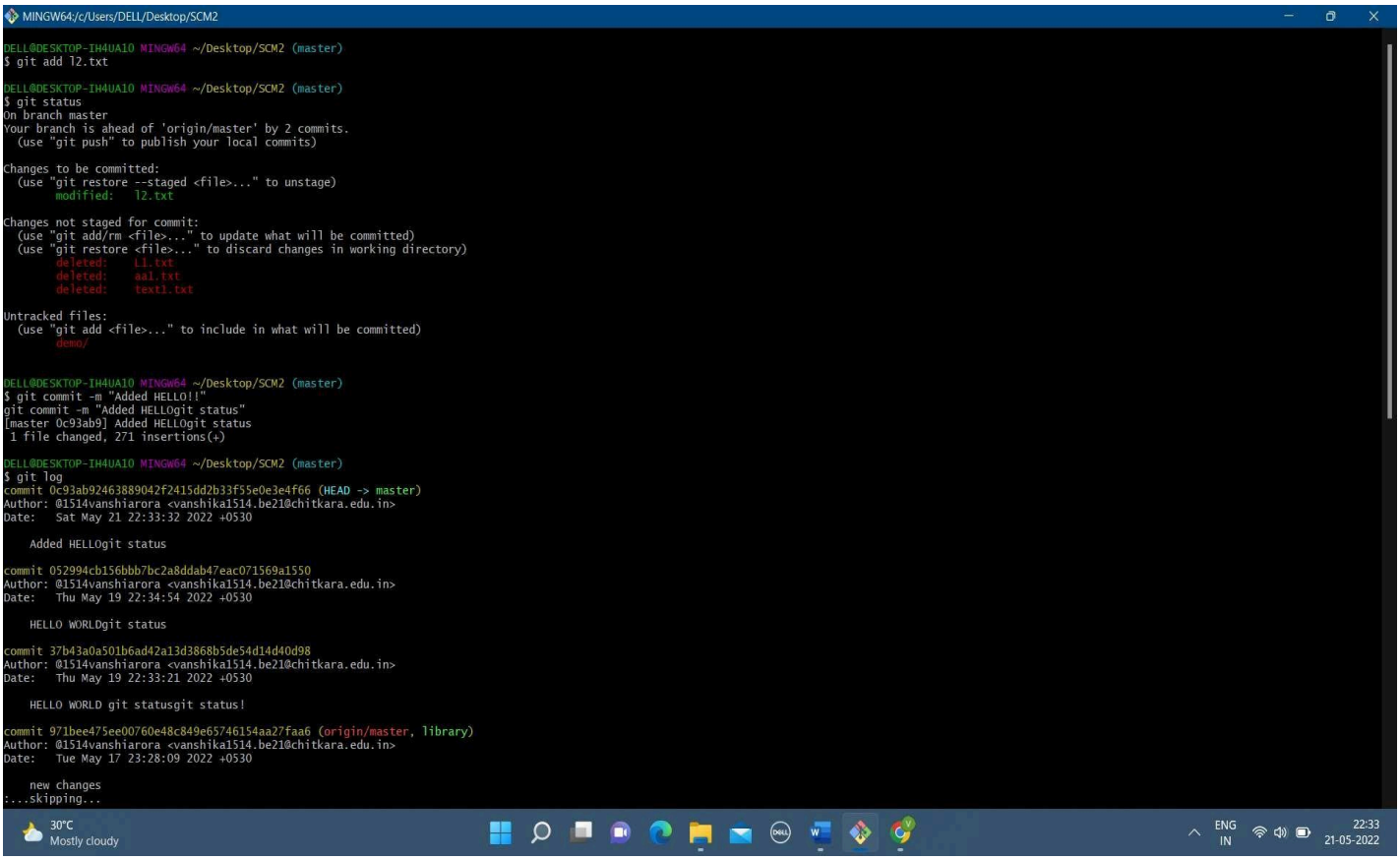
Repositories in GIT contain a collection of files of various different versions of a Project. These files are imported from the repository into the local server of the user for further updations and modifications in the content of the file. A VCS or the Version Control System is used to create these versions and store them in a specific place termed as a repository.


Repository
File 1
File 2
File 3
Version 1
File 1
File 2
File 3
File 4
File 5
Version 2

## 5. Concepts and commands

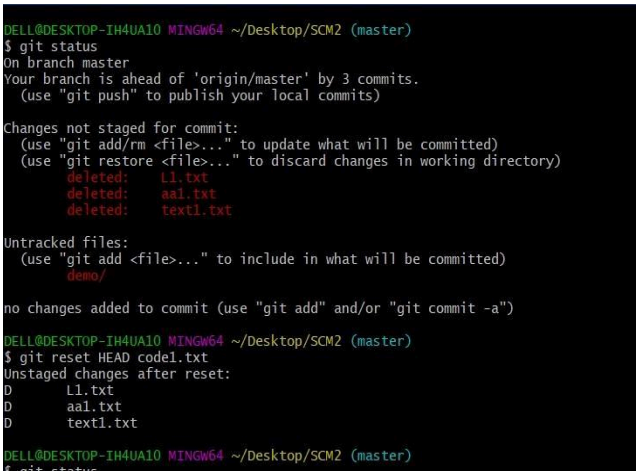
In the current era, open source contribution is on the peak. It's getting so much popular among students and there are also some open-source programs and competitions to learn so much about development. Open source is a great way to learn and make a better community. There are so many organizations where one can contribute and learn. For that, one have to know about git commands. (Here I make changes on a file named as l2(l2.txt)).

- · **git add :** Adds files in the to the staging area for Git. Before a file is available to commit to a repository, the file needs to be added to the Git index (staging area).
- · **git add - - all :** To add all files of current directory to staging area
- · **git add *.txt :** To add all text files of the current directory to staging area.
- · **git commit :** Record the changes made to the files to a local repository. For easy reference, each commit has a unique ID.
- · **git commit -a -m "Initial commit" :** To add any of our tracked files to staging area and commit them by providing a message to remember.
- · **git diff :** Diff command is used in git to track the difference between the changes made on a file.
- · **git diff –staged** : diff of what is staged but not yet committed
- · **git status :** 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 log :** To show the chronological commit history for a repository. This helps give context and history for a repository.
- · **git log –oneline :** Shows all the log and history of our commits in one line.
- · **git log –oneline – graph :** Shows a network graph of all our commits
- · **ls:** To see directories and files in the current directory




- · **git branch :** To see all the branches present and the current branch that we are working on
- · **git checkout <branchname> :** To switch to a branch example ‘act1’from master branch
- · **git checkout -b <branchname> :** This command creates a new branch and alsoswitches to it
- · **git merge <branchname>:** To merge a branch example ‘act1’ with master branch.
- · **git branch -D <branchname> :** To forcefully delete a branch without making commits.
- · **git branch –d <branchname>:** This is also used to delete a branch.The -d option will delete the branch only if it has already been pushed and merged with the remote branch.
- · **git remote add <address > :** To add new remotes to our local repository for a particular git address
- · **git push :** Sends local commits to the remote repository. git push requires two parameters: the remote repository and the branch that the push is for.
- · **git push – u origin master :** To push all the contents of our local repository that belong to master branch to the server(Global repository).

- · **git pull :** To get the latest version of a repository run git pull. This pulls the changes from the remote repository to the local computer.
- · **git config :** To set the basic configurations on github like your name and email.
- · **git config –user.name "xyz" :** Sets configuration values for your username on git
- · **git config –user.email xyz@gmail.com :** Sets configuration values for your user email on git.
- · **git init :** This command turns a directory into an empty Git repository.
- · **git clone <url>:** To clone or make a local copy of the global repository in your system
- · **git merge :** Integrate branches together .git merge combines the changes from one branch to another branch. For example, merge the changes made in a staging branch into the stable branch
- · **git remote :** To connect a local repository with a remote repository. A remote repository can have a name set to avoid having to remember the URL of the repository.
- · **git remote -v :** Shows URLs of remote repositories when listing your current remote connections.
- · **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.



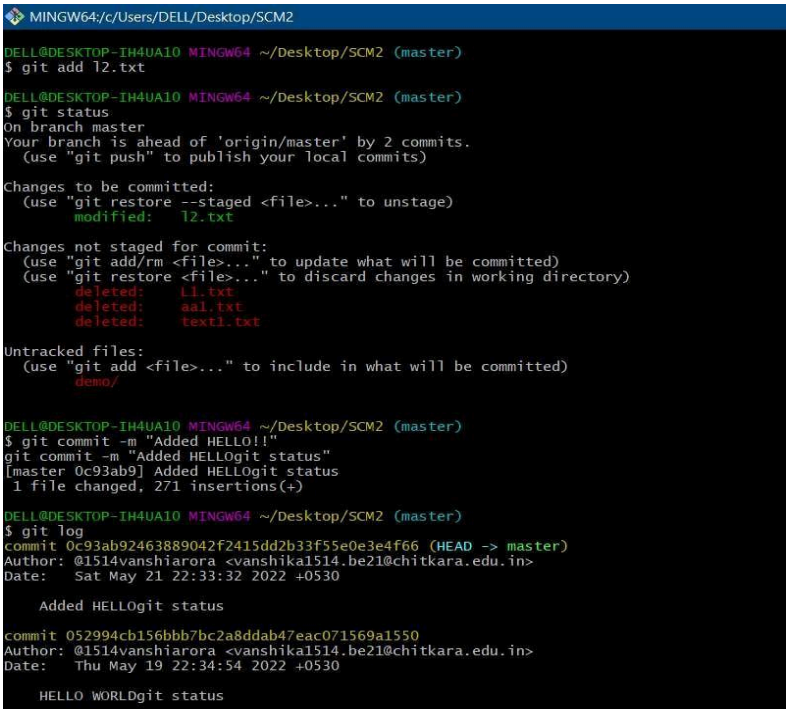

Git reset has 3 variants :

i. **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.

ii. **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.

iii. **git reset –hard HEAD~1 :** This command removes the commit as well as the changes from your working directory.

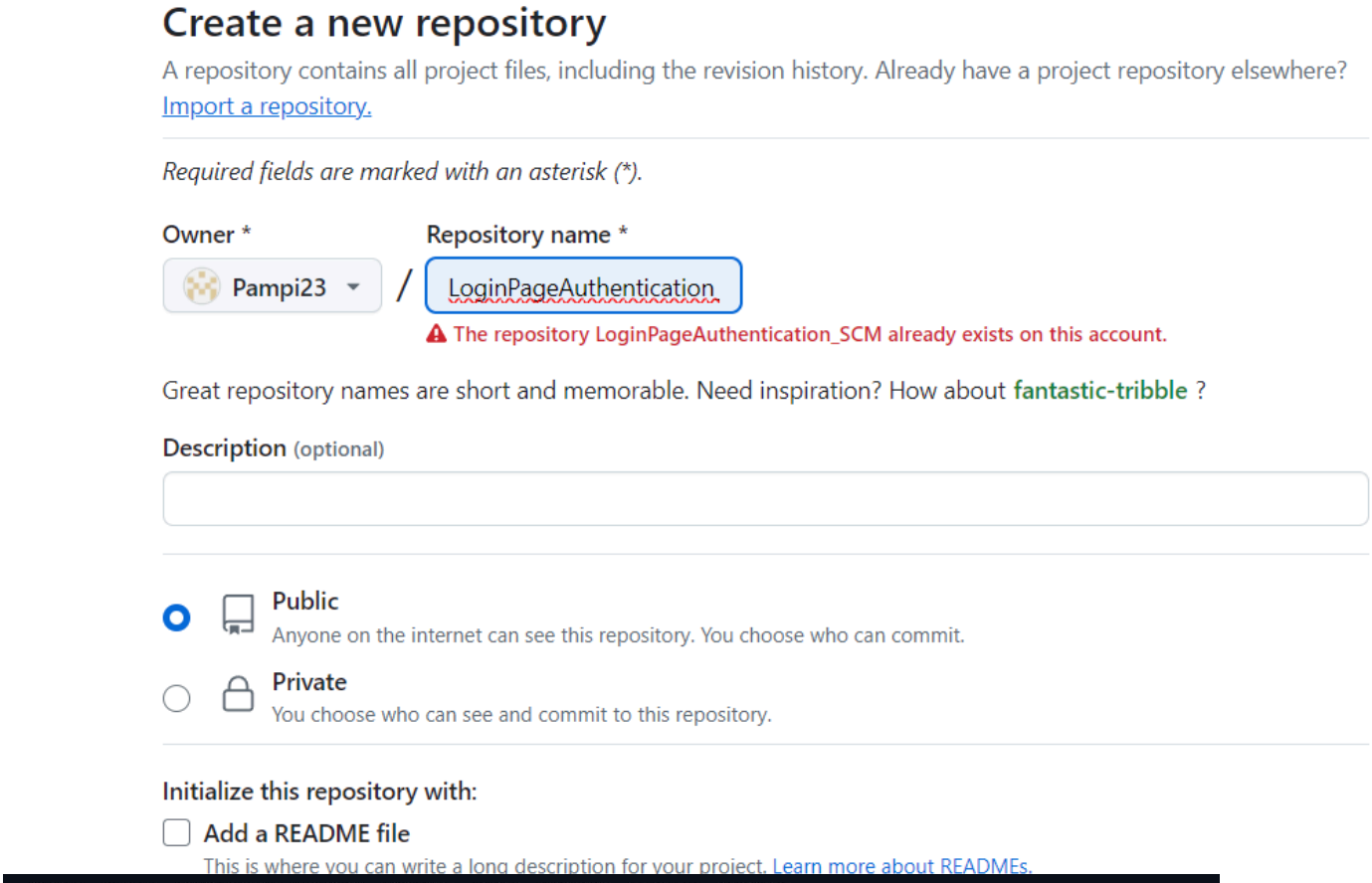
- **git revert :** This command is used to remove the commits from the remote repository. Since now our changes are in the working directory, let's add those changes to the staging area and commit them.
- **git ignore :** Creating a file with the name .gitignore .The files in that directory and deeper folders that match the patterns in the file will be ignored by gitignore.
- **git help :** Take help from github section for different commands and other errors
- **mkdir dir :** Create a directory with the name 'dir' if not created initially
- **cd dir :** To go inside the directory 'dir' and work upon its contents.
- **git commit -m "Initial commit" :** To commit our changes(taking a snapshot) and providing a message to remember for future reference.
- **git remove rm :** To remove a remote from our local repository.
- **git rm <filename> :** This command deletes the file from your working directory and stages the deletion.

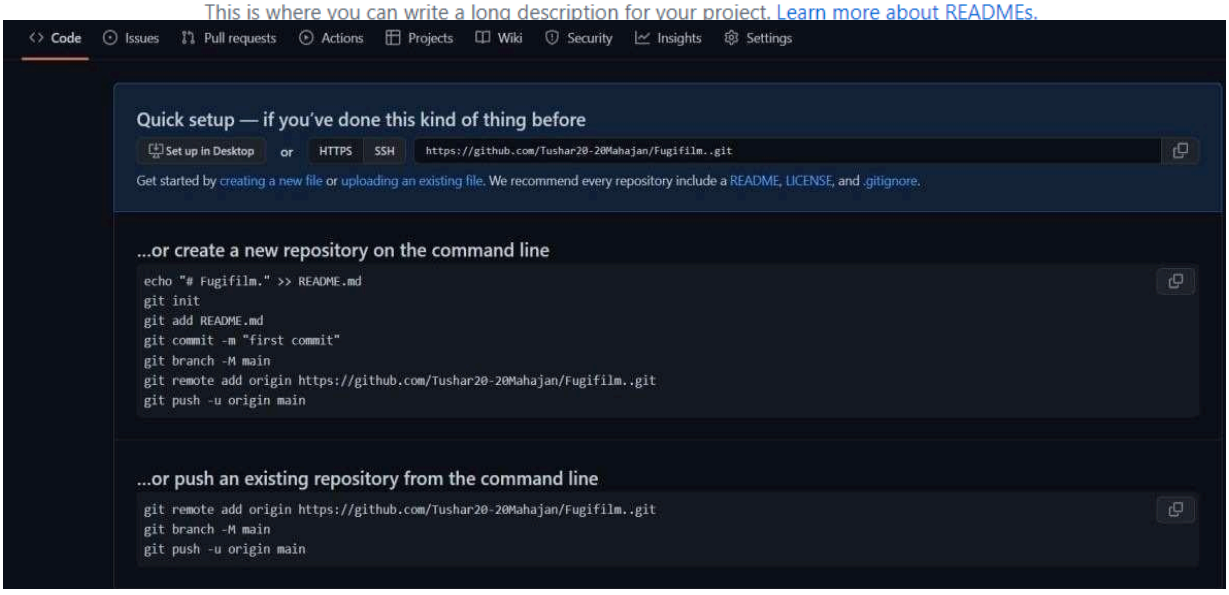


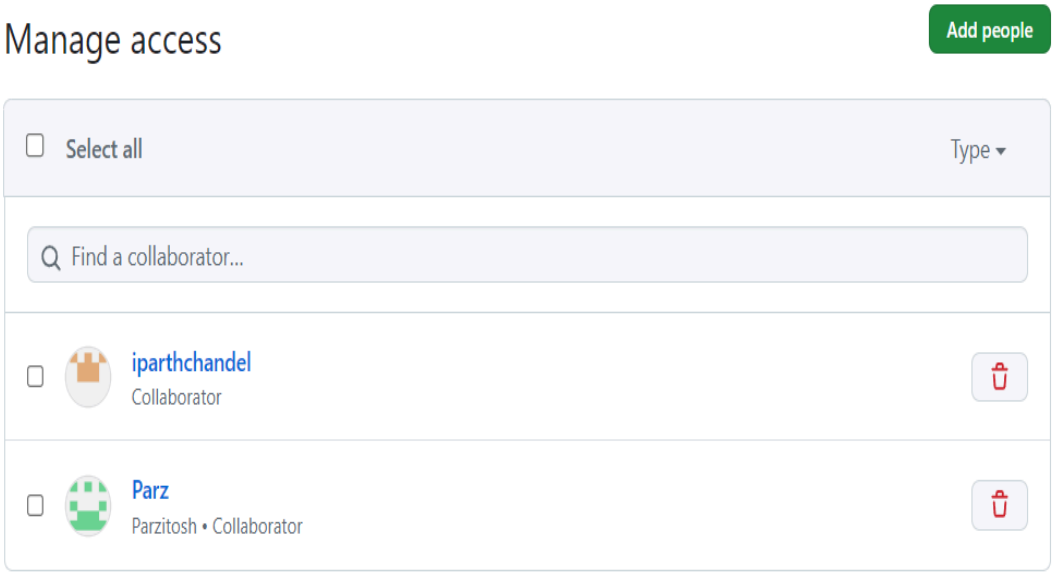
## 6. Workflow and Discussion

When we work on a development project as a collaborator, we have commit privileges to the main repository on GitHub. The development workflow for collaborators in a team takes advantage of some Git functionality that we don't often use on pair projects: **Branching** and **merging**. Below is a *general* outline of this workflow.

- Set Github Repo:-

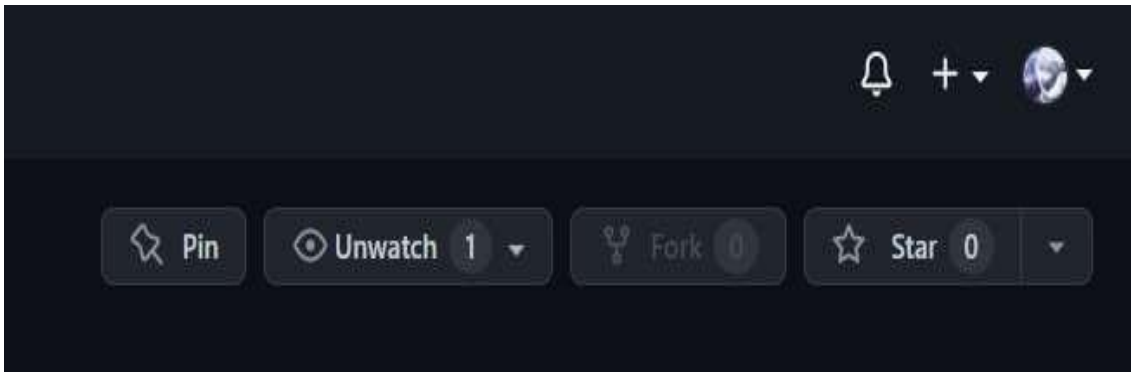





- Forking repository:-

A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. To fork, find the GitHub repository with which you'd like to work. Click the Fork button on the upper right hand side.

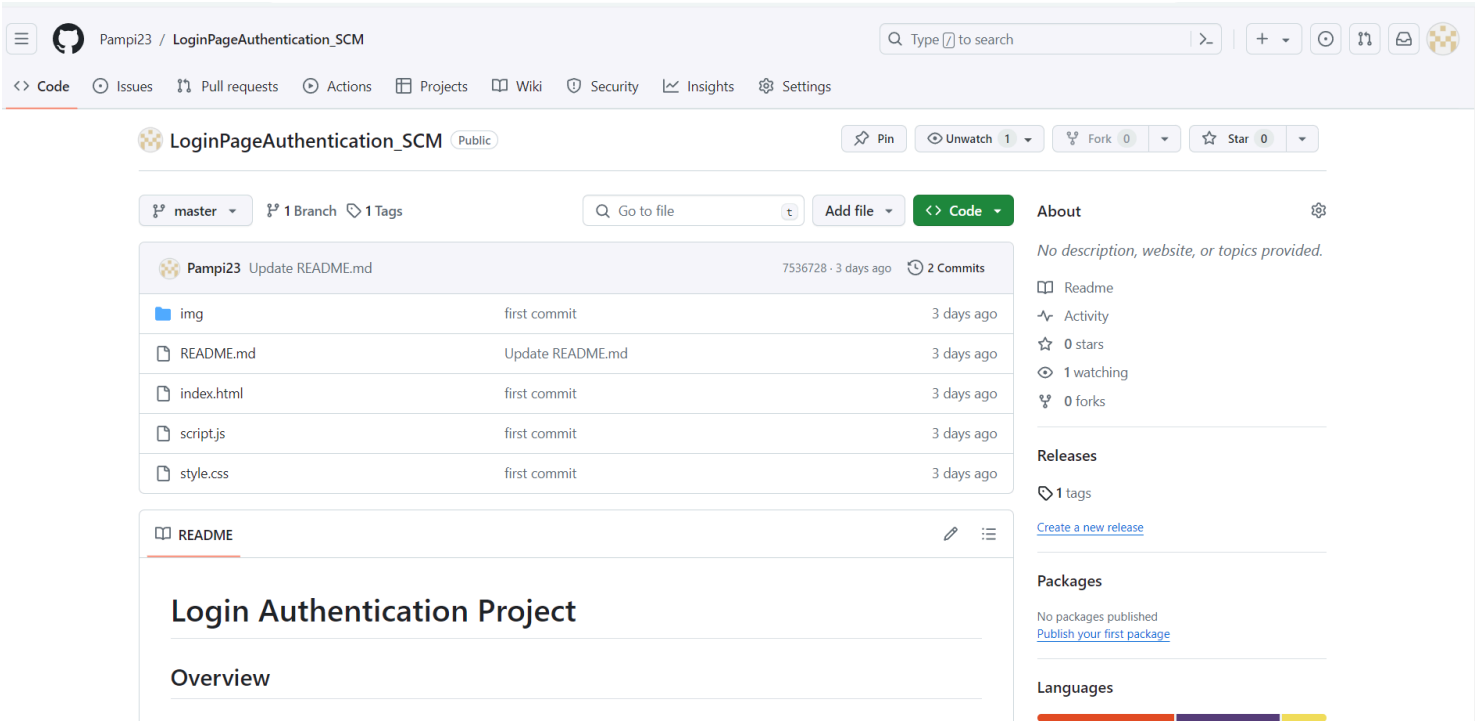



- Commit:-

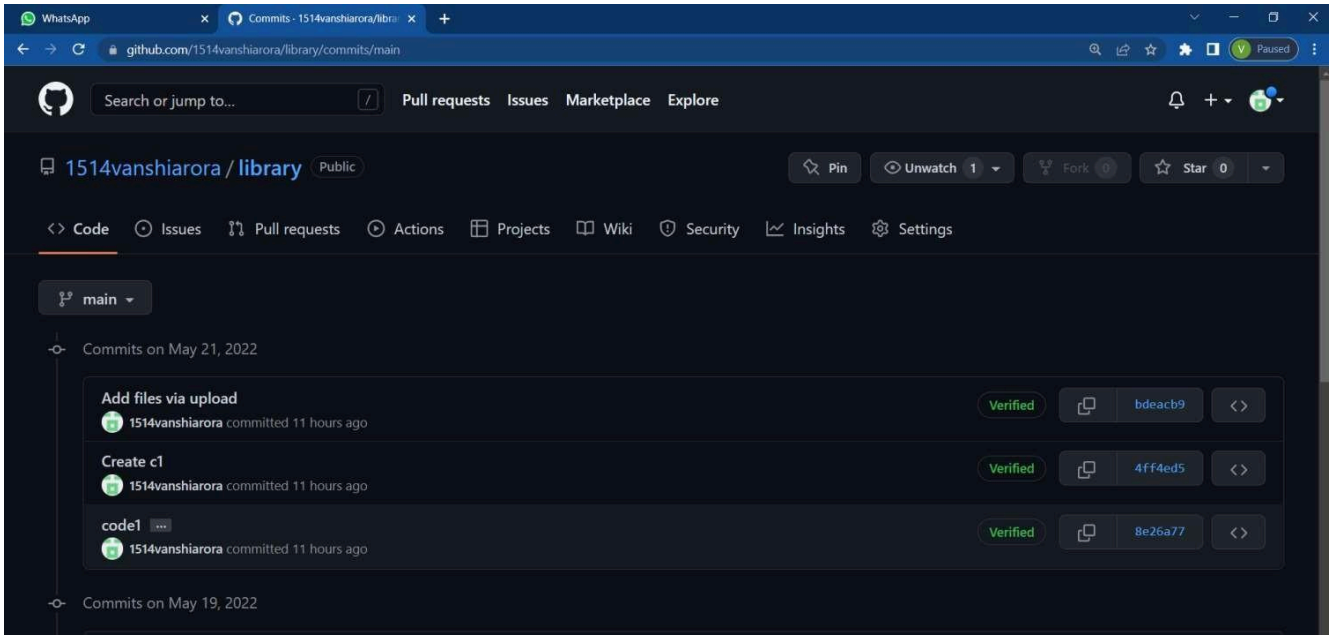
Commit and push your branch to remote so it is on the server and is visible to other people collaborating on the project. Track all files in the working directory with updated changes to be committed to the local repository. Add all committed files in the local repo branch to the remote repo branch, so all files and changes will be visible to anyone with access to the remote repository.

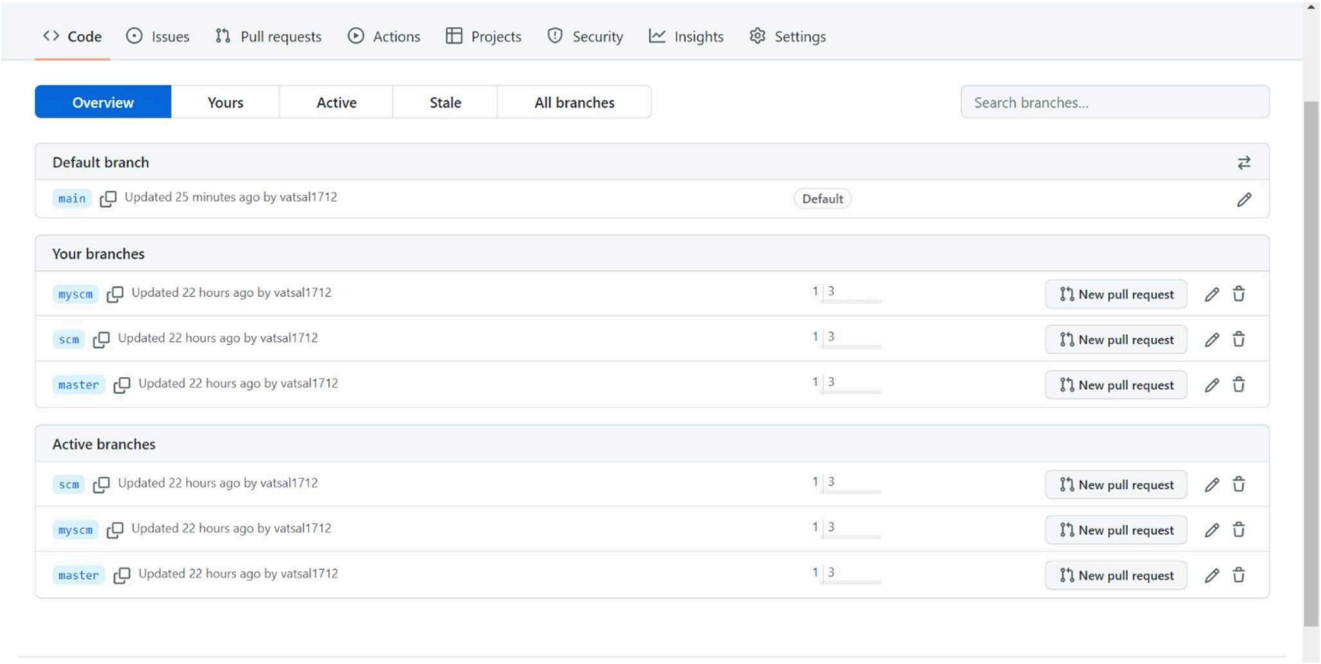
```
param@Pampi MINGW64 /e/Scm_
$ git init

Initialized empty Git repository in E:/Scm_/.git/

param@Pampi MINGW64 /e/Scm_ (master)
$ git remote rm origin
error: No such remote: 'origin'

param@Pampi MINGW64 /e/Scm_ (master)
$ git remote -v

param@Pampi MINGW64 /e/Scm_ (master)
$ git remote add origin https://github.com/Pampi23/LoginPageAuthentication_SCM.g
it

param@Pampi MINGW64 /e/Scm_ (master)
$ git remote -v
origin  https://github.com/Pampi23/LoginPageAuthentication_SCM.git (fetch)
origin  https://github.com/Pampi23/LoginPageAuthentication_SCM.git (push)
```

```
param@Pampi MINGW64 /e/Scm_ (master)
$ ls
README.md  img/  index.html  script.js  style.css

param@Pampi MINGW64 /e/Scm_ (master)
$ git add .
warning: in the working copy of 'index.html', LF will be replaced by CRLF the ne
xt time Git touches it
warning: in the working copy of 'script.js', LF will be replaced by CRLF the nex
t time Git touches it
warning: in the working copy of 'style.css', LF will be replaced by CRLF the nex
t time Git touches it

param@Pampi MINGW64 /e/Scm_ (master)
$ git commit -m "first commit"
[master (root-commit) 5dd3e50] first commit
 6 files changed, 519 insertions(+)
 create mode 100644 README.md
 create mode 100644 img/checked1.png
 create mode 100644 img/log-img1.jpg
 create mode 100644 index.html
 create mode 100644 script.js
 create mode 100644 style.css

param@Pampi MINGW64 /e/Scm_ (master)
$ git push origin master
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 12 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 2.82 MiB | 1.55 MiB/s, done.
Total 9 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/Pampi23/LoginPageAuthentication_SCM.git
 * [new branch]      master -> master
```

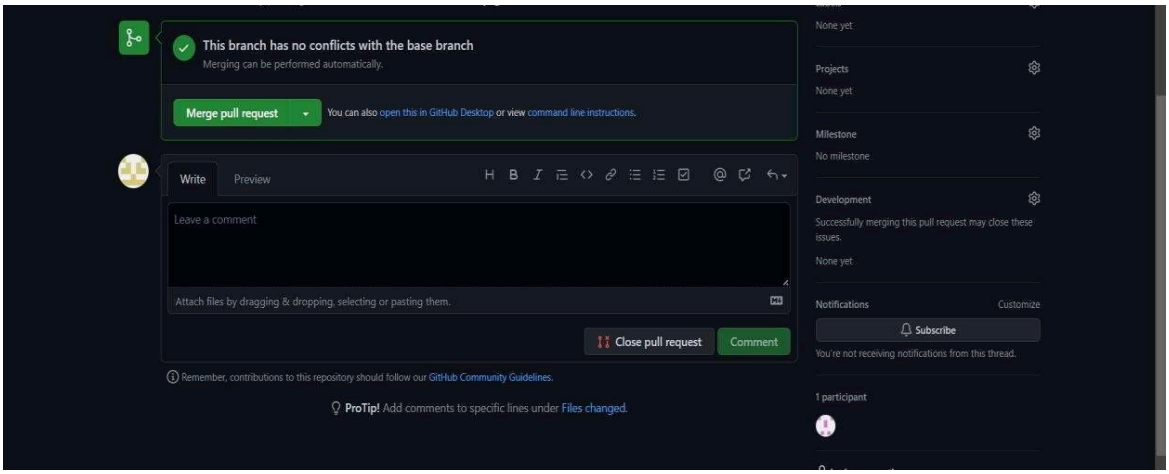

Pampi23 / LoginPageAuthentication_SCM
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
LoginPageAuthentication_SCM
Public
Pin
Unwatch 1
Fork 0
Star 0
master
1 Branch
1 Tags
Go to file
t
Add file
Code
Pampi23 Update README.md
7536728 · 3 days ago
2 Commits
img first commit 3 days ago
README.md Update README.md 3 days ago
index.html first commit 3 days ago
script.js first commit 3 days ago
style.css first commit 3 days ago
README
Login Authentication Project
Overview
About
No description, website, or topics provided.
Readme
Activity
0 stars
1 watching
0 forks
Releases
1 tags
Create a new release
Packages
No packages published
Publish your first package
Languages

CHITKARA UNIVERSITY



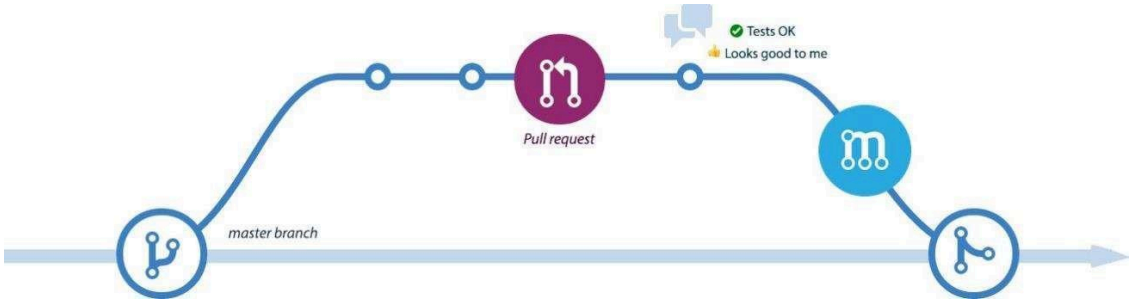
makes a change on the exact same line, a merge conflict occurs. To complete this type of merge, use your text editor to resolve the conflict, then add the file and commit it to complete the merge.



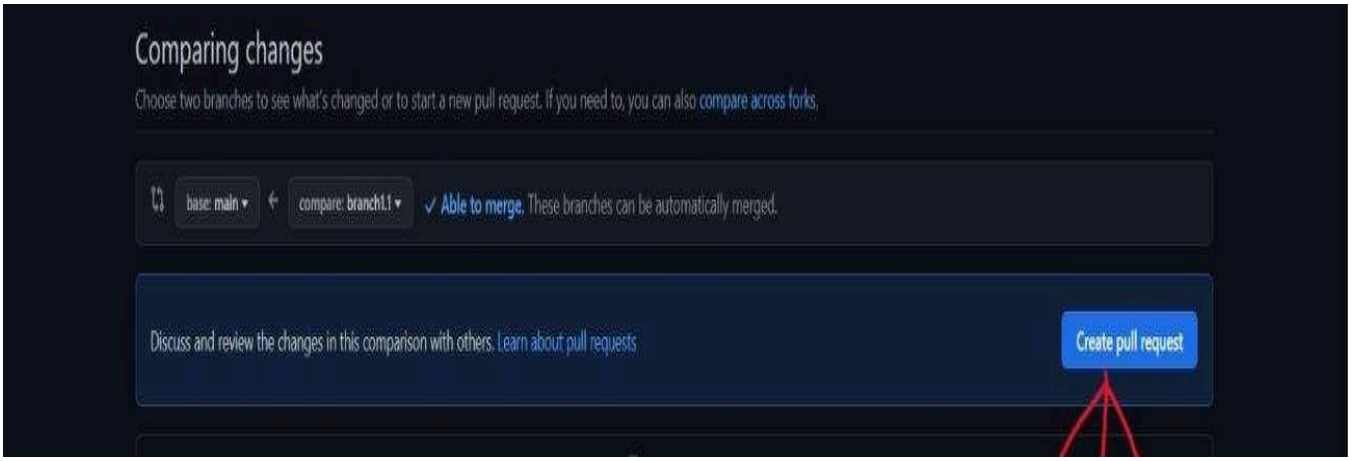

- Making a pull request:-

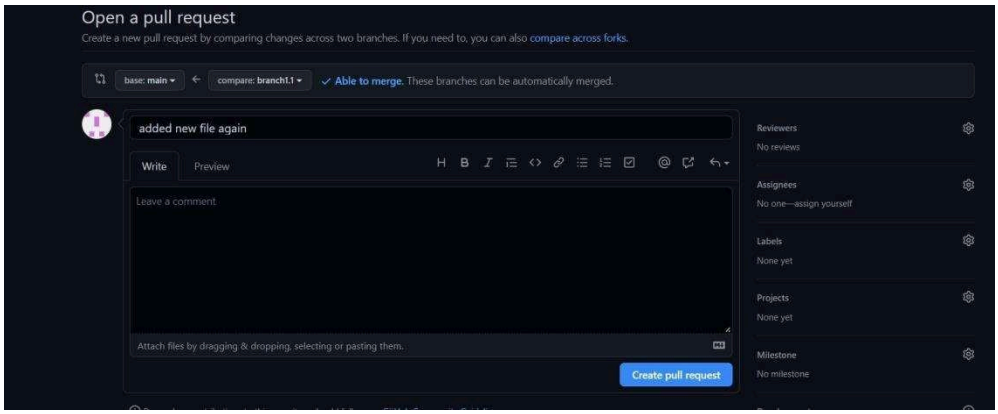
When a branch is ready to be merged, a pull request should be opened. Make sure your branch is up to data with master with the rebasing steps above.

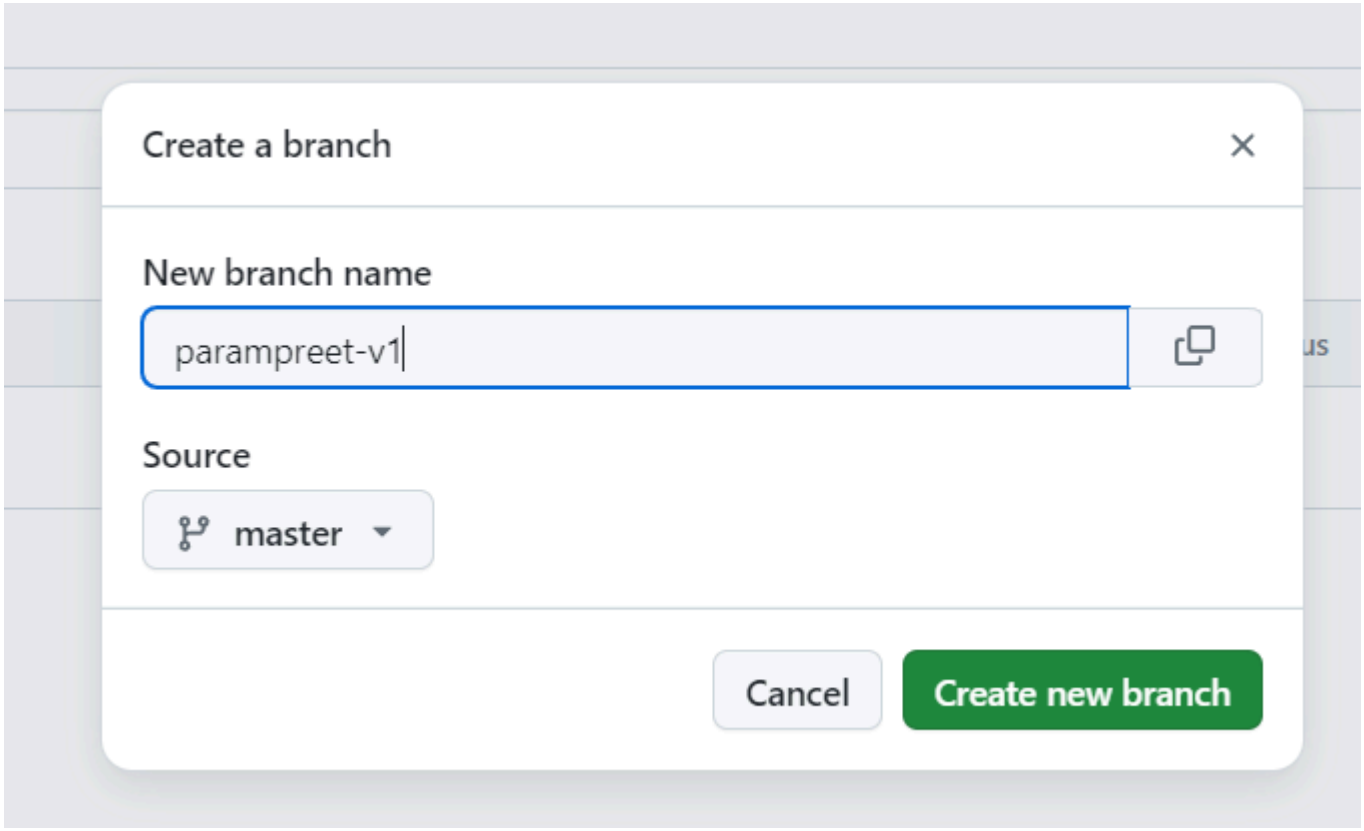
Click on the “Pull requests” tab on the main page of the GitHub repo and click the “Create Pull request” button.



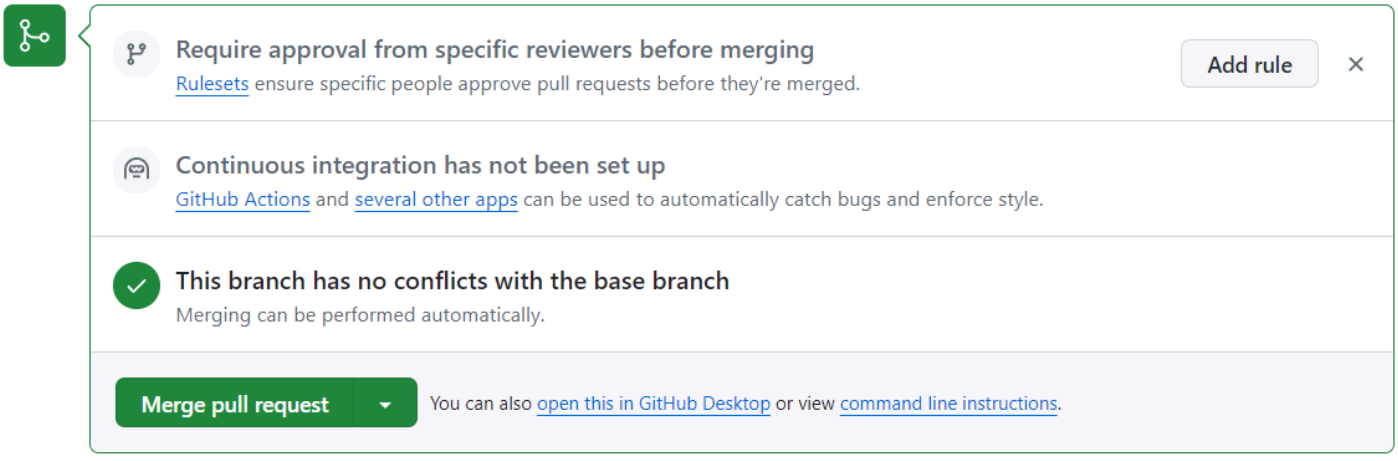

- To create your own pull request, click on pull request option.



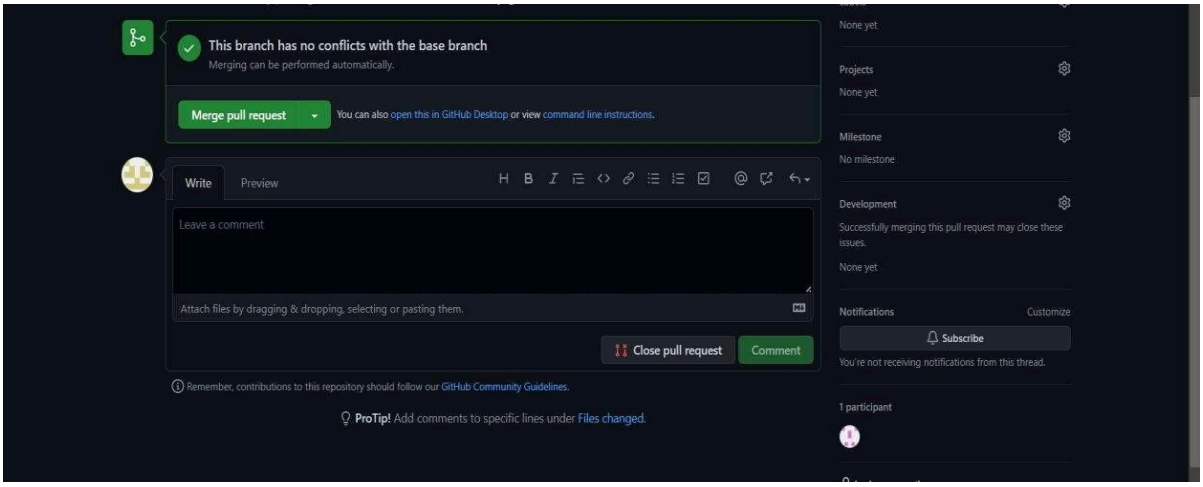


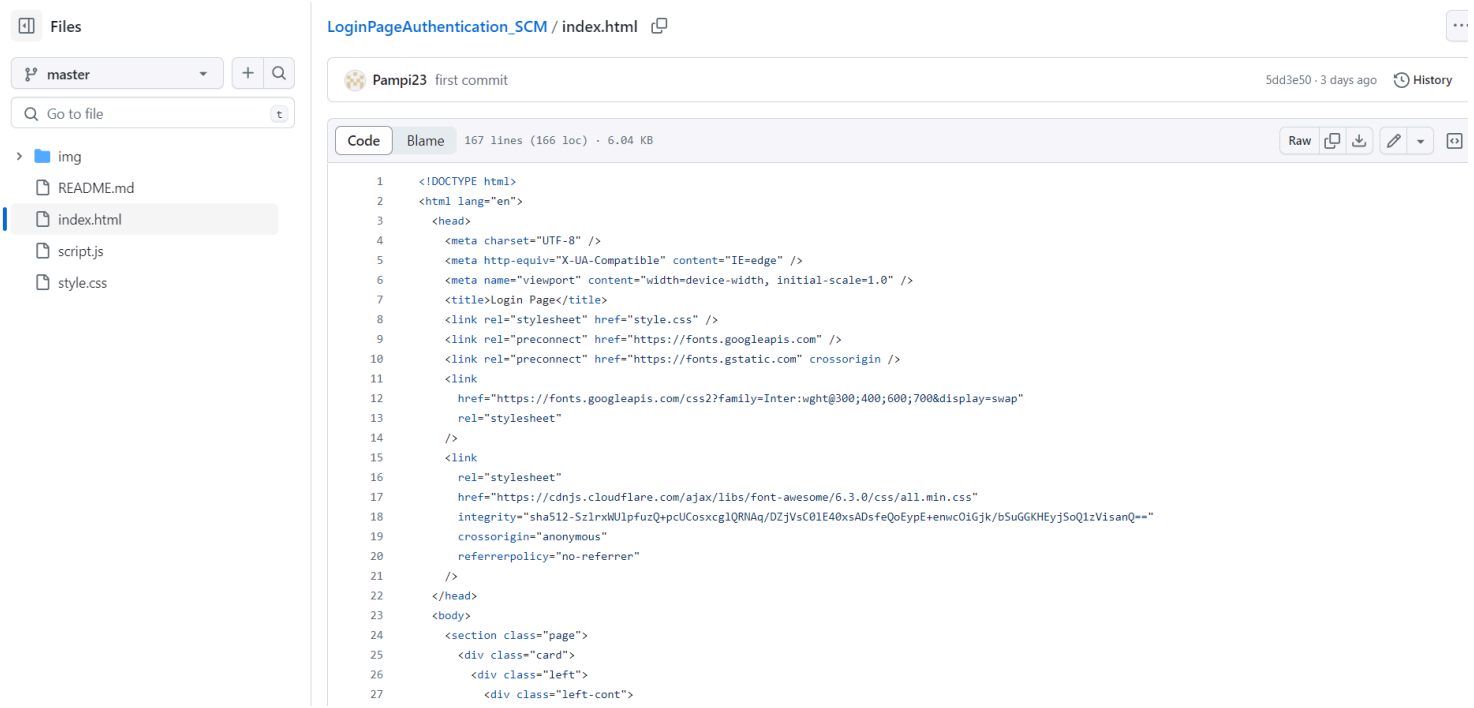

Add more commits by pushing to the `parampreet-v1` branch on Pampi23/LoginPageAuthentication_SCM.



After opening a pull request all the team members will be sent the requestif they want to merge or close the request.

Glimpse of Project:



CHITKARA UNIVERSITY

Files

Collapse file tree

master

Go to file t

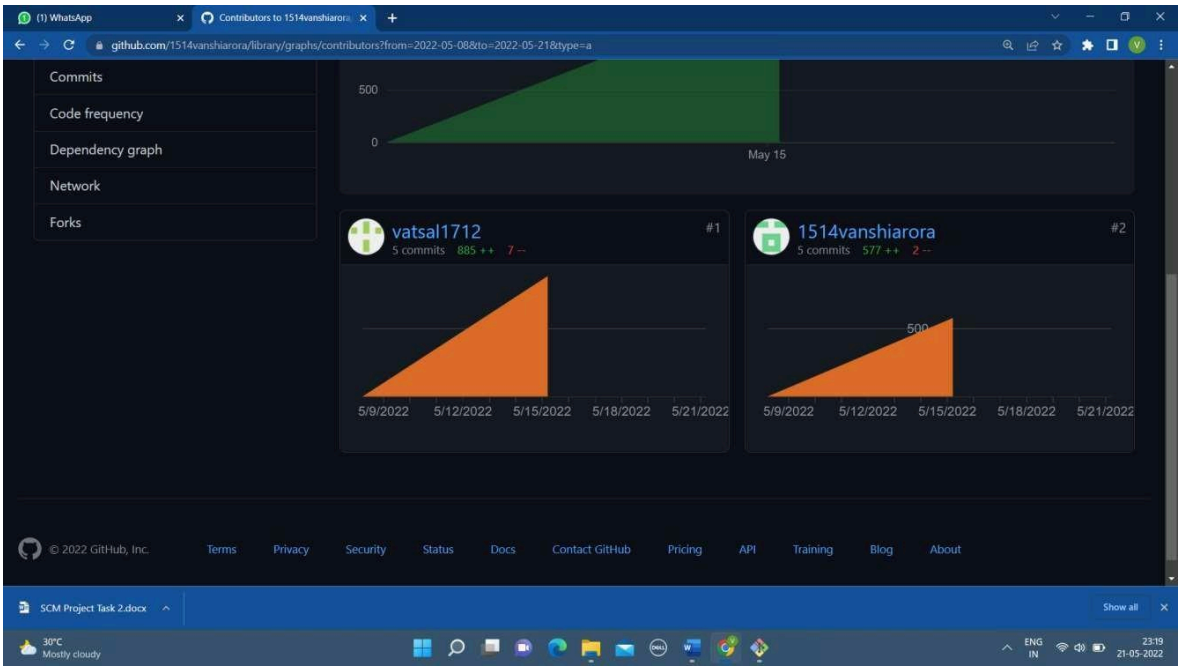
- img
- README.md
- index.html
- script.js
- style.css

LoginPageAuthentication_SCM / style.css

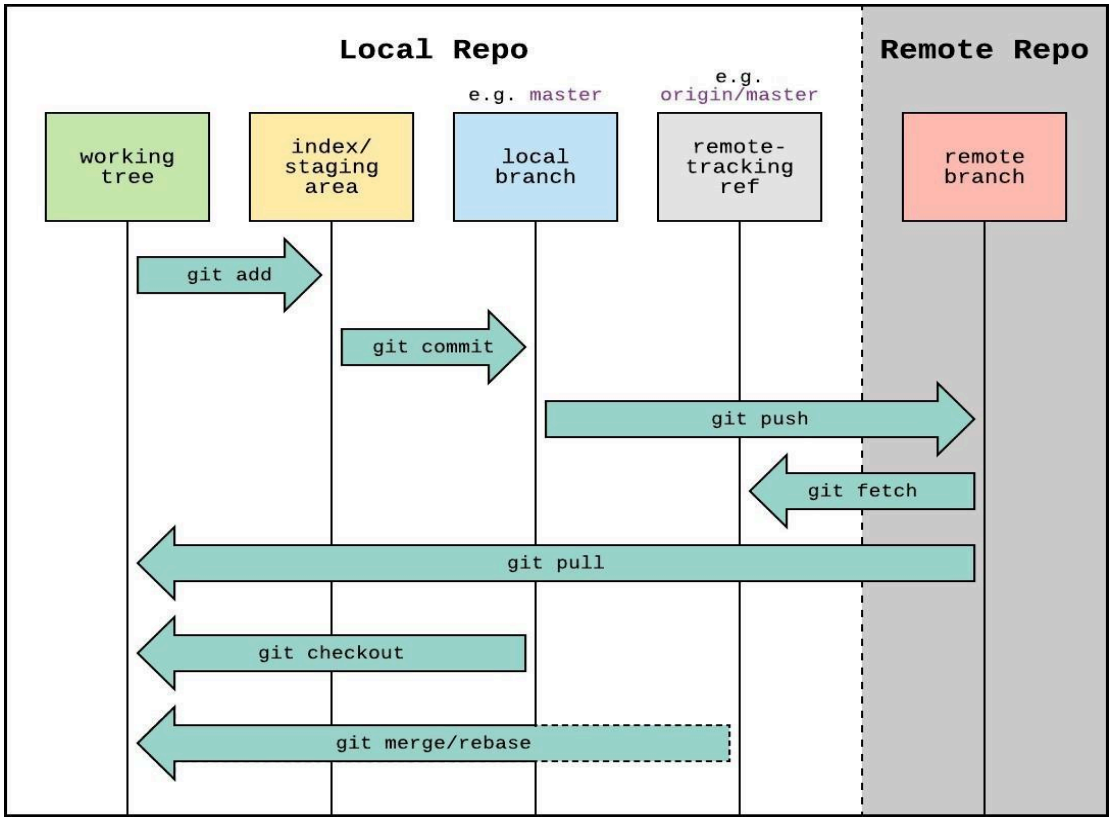
Pampi23 first commit 5dd3e50 · 3 days ago History

Code Blame 294 lines (254 loc) · 4.56 KB Raw

```
1   * {
2     padding: 0;
3     margin: 0;
4     font-family: "Inter", sans-serif;
5   }
6
7   .page {
8     height: 100vh;
9     position: relative;
10    display: flex;
11    justify-content: center;
12    align-items: center;
13    background: rgb(52, 81, 118);
14    background: linear-gradient(
15      151deg,
16      rgba(52, 81, 118, 1) 0%,
17      rgba(124, 145, 194, 1) 22%,
18      rgba(124, 145, 194, 1) 42%,
19      rgba(167, 182, 219, 1) 61%,
20      rgba(218, 215, 237, 1) 82%
21    );
22
23    /* background: rgb(52, 81, 118);
24    background: linear-gradient(
25      151deg,
26      rgba(52, 81, 118, 1) 0%,
27      rgba(124, 145, 194, 1) 21%,
```




Worfiflow Chart



## 7.References


https://guides.codepath.com/android

https://git-scm.com/

https://www.javatpoint.com/

From online sources.

Wikipedia.

Books available sites for downloading and having copywrite for example: Manybooks.


For frontend:

1. HTML
2. CSS
3. JAVASCRIPT
4. CREATIVE WEBSITES

For backend:

1. PYTHON
2. COMMUNICATION SKILLS
3. PROBLEM SOLVING SKILLS