CHITKARA
UNIVERSITY

Subject Name: **Source Code Management**

Subject Code: CS181

Cluster: Beta

Department: DCSE

Submitted By:

Name: Gunand Sharma

Roll no: 2110990527

Submitted To:

Dr. Monit Kapoor

Index

CHITKARA
UNIVERSITY

Subject Name: **Source Code Management**

Subject Code: CS181

Cluster: Beta

Department: DCSE

Submitted By:	Submitted To:
Name: Gunand Sharma	Dr. Monit Kapoor
Roll no: 2110990527	

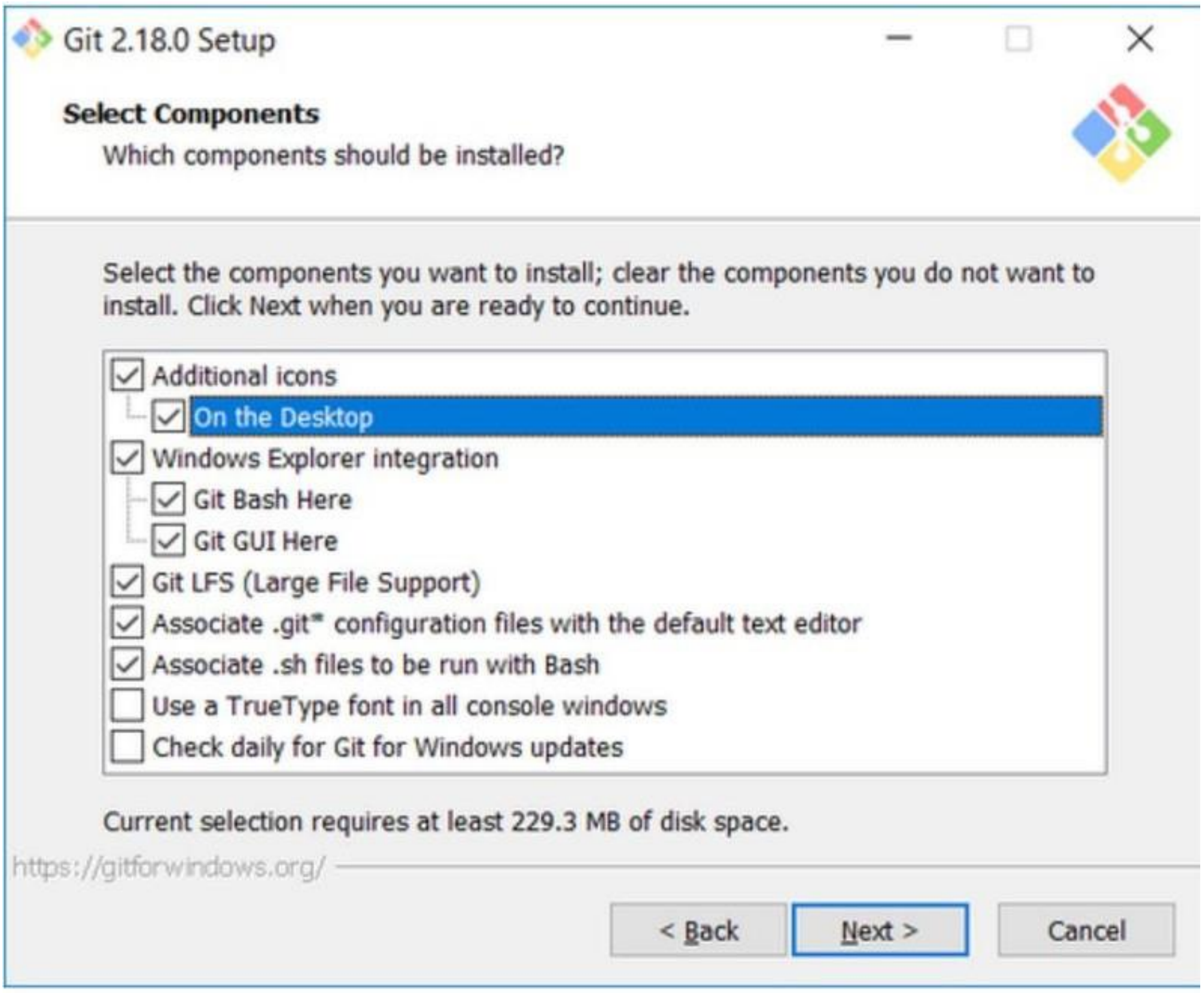
Index

Experiment No. 01

Aim: Setting up the git client.

Git Installation: Download the Git installation program (Windows, Mac, or Linux) from Git - Downloads (git-scm.com).
When running the installer, various screens appear (Windows screens shown). Generally, you can accept the default selections, except in the screens below where you do not want the default selections:

In the Select Components screen, make sure Windows Explorer Integration is selected as shown:


Git 2.18.0 Setup

Select Components

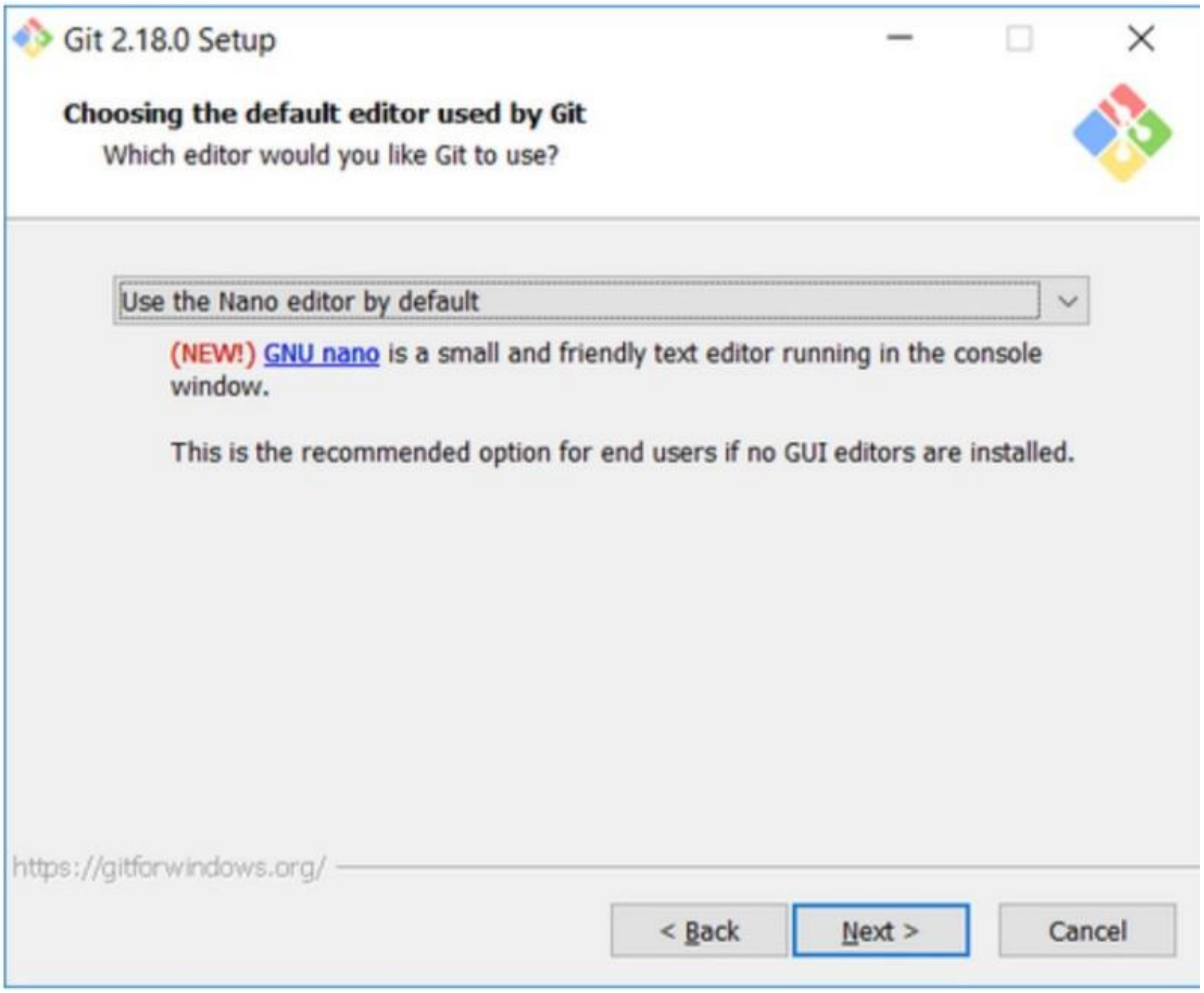
Which components should be installed?

Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

- [✓] Additional icons
  - [✓] On the Desktop
- [✓] Windows Explorer integration
  - [✓] Git Bash Here
  - [✓] Git GUI Here
- [✓] Git LFS (Large File Support)
- [✓] Associate .git* configuration files with the default text editor
- [✓] Associate .sh files to be run with Bash
- [ ] Use a TrueType font in all console windows
- [ ] Check daily for Git for Windows updates

Current selection requires at least 229.3 MB of disk space.

In the choosing the default editor is used by Git dialog, it is strongly recommended that you DO NOT select default VIM editor- it is challenging to learn how to use it, and there are better modern editors available. Instead, choose Notepad++ or Nano – either of those is much easier to use. It is strongly recommended that you select Notepad++.



Choosing the default editor used by Git

Which editor would you like Git to use?

Use the Nano editor by default

(NEW!) GNU nano is a small and friendly text editor running in the console window.

This is the recommended option for end users if no GUI editors are installed.

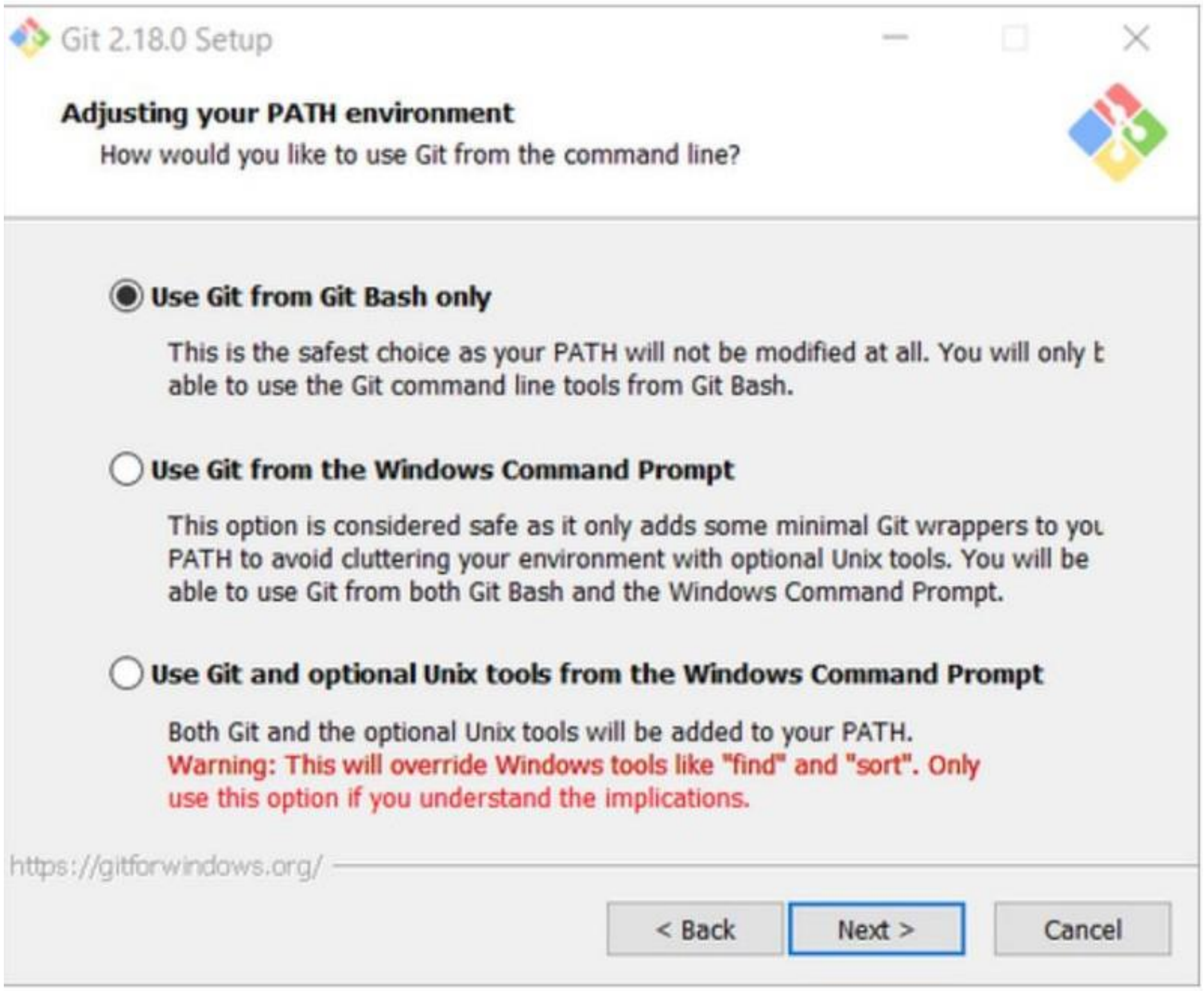
https://gitforwindows.org/

In the Adjusting your PATH screen, all three options are acceptable:

1. Use Git from Git Bash only: no integration, and no extra command in your command path.

2. Use Git from the windows Command Prompt: add flexibility – you can simply run git from a windows command prompt, and is often the setting for people in industry – but this does add some extra commands.

3. Use Git and optional Unix tools from the Windows Command Prompt: this is also a robust choice and useful if you like to use Unix like commands like grep.


Git 2.18.0 Setup

Adjusting your PATH environment

How would you like to use Git from the command line?

● **Use Git from Git Bash only**

This is the safest choice as your PATH will not be modified at all. You will only b
able to use the Git command line tools from Git Bash.

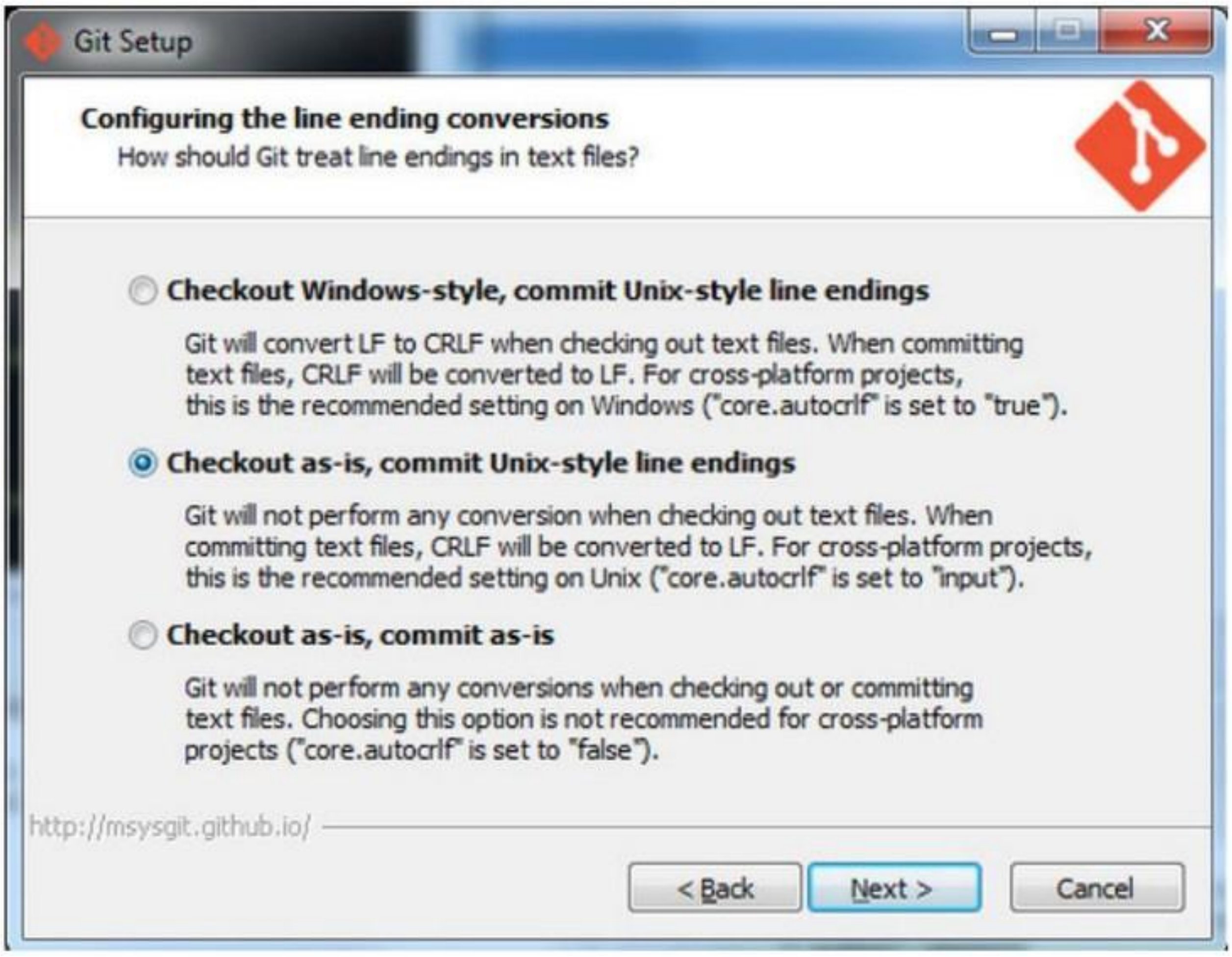
○ **Use Git from the Windows Command Prompt**

This option is considered safe as it only adds some minimal Git wrappers to you
PATH to avoid cluttering your environment with optional Unix tools. You will be
able to use Git from both Git Bash and the Windows Command Prompt.

○ **Use Git and optional Unix tools from the Windows Command Prompt**

Both Git and the optional Unix tools will be added to your PATH.
Warning: This will override Windows tools like "find" and "sort". Only
use this option if you understand the implications.

In the Configuring the line ending screen, select the middle option (Checkout-as-is, commit Unix-style line endings) as shown. This helps migrate files towards the Unix-style (LF) terminators that most modern IDE’s and editors support. The Windows convention (CR-LF line termination) is only important for Notepad.


Git Setup

Configuring the line ending conversions

How should Git treat line endings in text files?

Checkout Windows-style, commit Unix-style line endings

Git will convert LF to CRLF when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Windows ("core.autocrlf" is set to "true").

[✓ Checkout as-is, commit Unix-style line endings]

Git will not perform any conversion when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Unix ("core.autocrlf" is set to "input").

Checkout as-is, commit as-is

Git will not perform any conversions when checking out or committing text files. Choosing this option is not recommended for cross-platform projects ("core.autocrlf" is set to "false").

Configuring Git to ignore certain files:

This part is extra important and required so that your repository does not get cluttered with garbage files.

By default, Git tracks allfiles in a project. Typically, this is notwhat you want; rather, you want Git to ignore certain files such as .bakfiles created by an editor or .classfiles created by the Java compiler. To have Git automatically ignore particular files, create a file named .gitignore (note that the filename begins with a dot) in the C:\users\namefolder (where name is your MSOE login name).

NOTE: The .gitignore file must NOT have any file extension (e.g. .txt). Windows normally tries to place a file extension (.txt) on a file you create from File Explorer - and then it (by default) HIDES the file extension. To avoid this, create the file from within a useful editor (e.g. Notepad++ or UltraEdit) and save the file without a file extension).

Edit this file and add the lines below (just copy/paste them from this screen); these are patterns for files to be ignored (taken from examples provided at https://github.com/github/gitignore.)

```
#Lines (like this one) that begin with # are comments; all other lines are rules


# common build products to be ignored at MSOE
*.o
*.obj
*.class
*.exe


# common IDE-generated files and folders to ignore
workspace.xml
bin
/
out
/
.classpath
# uncomment following for courses in which Eclipse .project files are not checked
in # .project


#ignore automatically generated files created by some common applications, operating
systems
*.bak
*.log
*.ldb
~*
.DS_Store*
._*
Thumbs.d
b


# Any files you do not want to ignore must be specified starting with ! # For example, if you
didn't want to ignore .classpath, you'd uncomment the following rule: #
!.classpath
```

Note: You can always edit this file and add additional patterns for other types of files you might want to ignore. Note that you can also have a .gitignore files in any folder naming additional files to ignore. This is useful for project-specific build products.

Once Git is installed, there is some remaining custom configuration you must do. Follow the steps below:

a. From within File Explorer, right-click on any folder. A context menu appears containing the commands " Git Bash here" and "Git GUI here". These commands permit you to launch either Git client. For now, select Git Bash
here.

b. Enter the command (replacing name as appropriate) git config --
global core.excludesfile c:/users/name/. gitignore

This tells Git to use the. gitignore file you created in step 2

NOTE: TO avoid typing errors, copy and paste the commands shown here into the Git Bash window, using the arrow keys to edit the red text to match your information.

git config --global user.Email "name@msoe.edu"

c.Enter the command

This links your Git activity to your email address. Without this, your commits will often show up as "unknown login". Replace name with your own MSOE email name.

d Enter the command git config --global user.name "Your Name"

Git uses this to log your activity. Replace "Your Name" by your actual first and last name.

e. Enter the command git config --global push.default simple

This ensures that all pushes go back to the branch from which they were
pulled. Otherwise pushes will go to the master branch, forcing a merge.

Experiment No. 02

Aim
Setting up GitHub Account

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

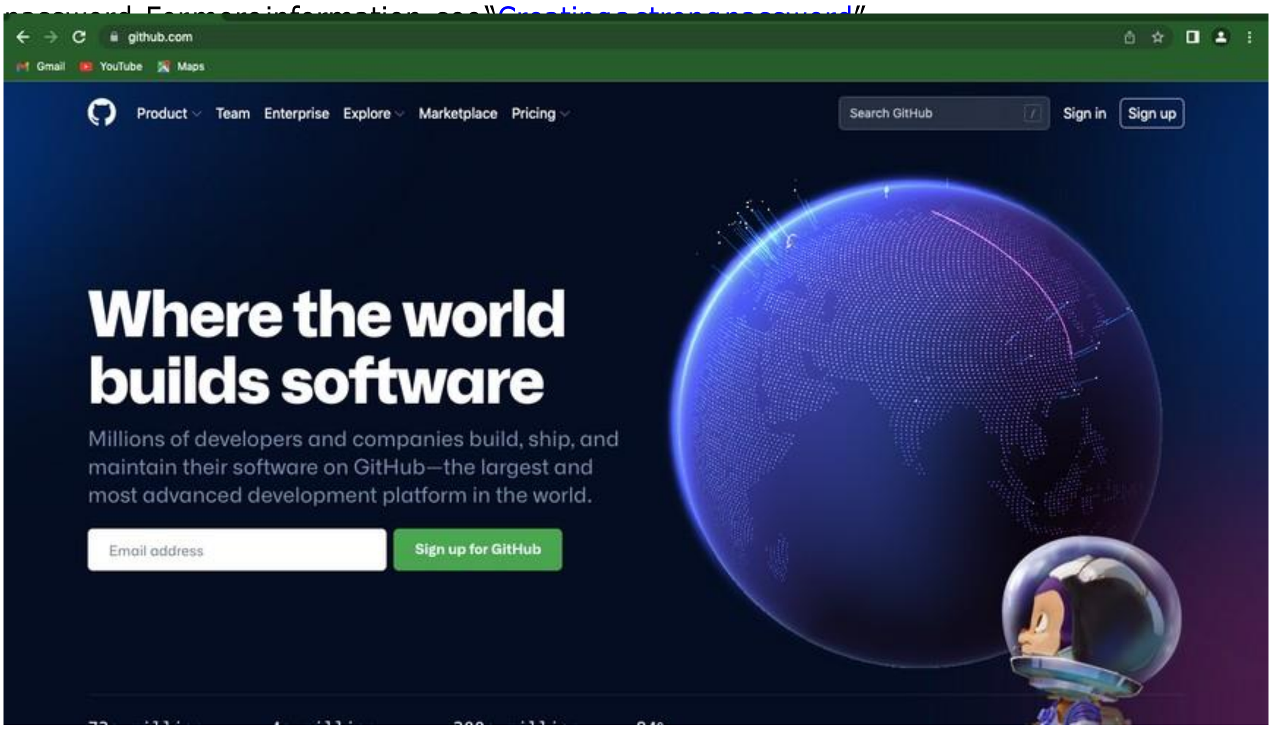
There are several types of accounts on GitHub. Every person who uses GitHub has their own user account, which can be part of multiple organisations and teams. Your user account is your identity on GitHub.com and represents you as an individual.

1. **Creating an account:** To sign up for an account on GitHub.com, navigate to

https://github.com/

and follow the prompts. To keep your GitHub account secure you should use a strong and unique

password. For more information, see “Creating a strong password”.



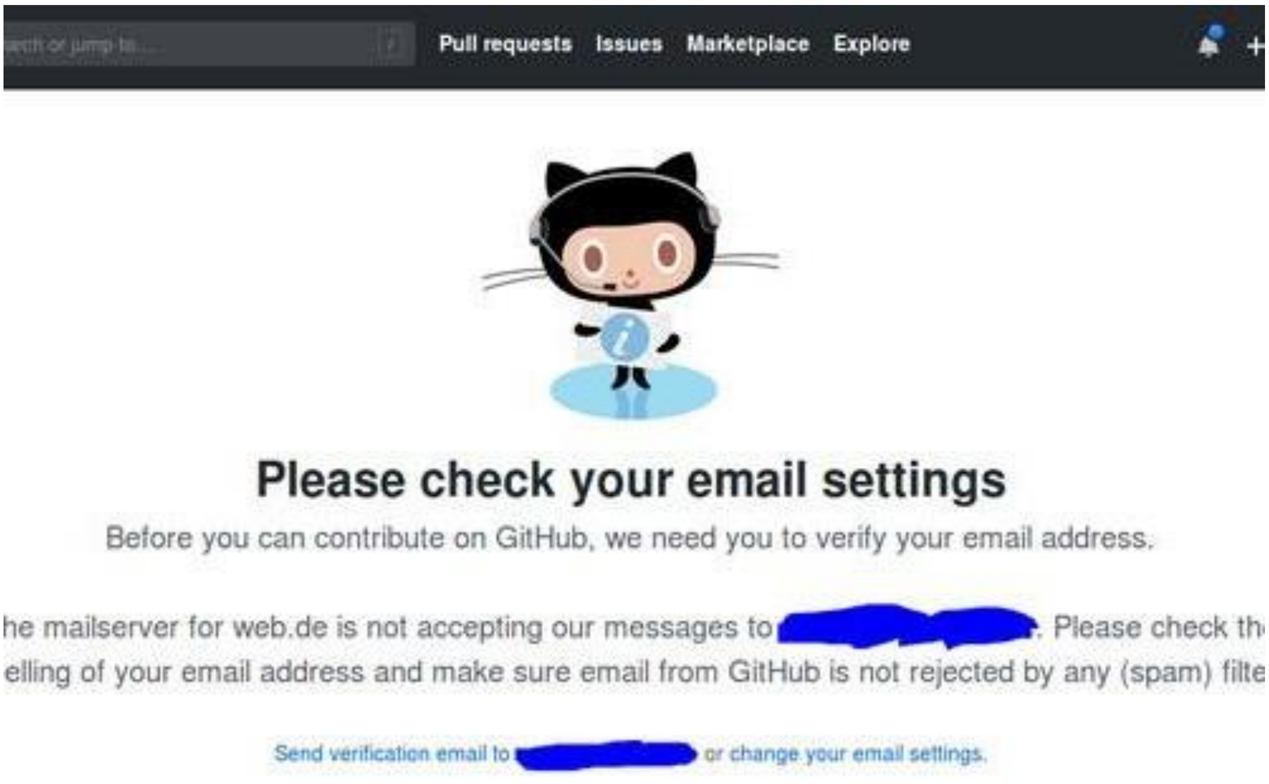
Where the world builds software

Millions of developers and companies build, ship, and maintain their software on GitHub—the largest and most advanced development platform in the world.

2. **Choosing your GitHub product:** You can choose GitHub Free or GitHub Pro to get access to different features for your personal account. You can upgrade at any time if you are unsure at first which product you want.

For more information on all GitHub’s plans, see “GitHub's products”.

3.Verifying your email address: To ensure you can use all the features in your GitHub plan, verify your email address after signing up for a new account. For more information, see "Verifying your email address".



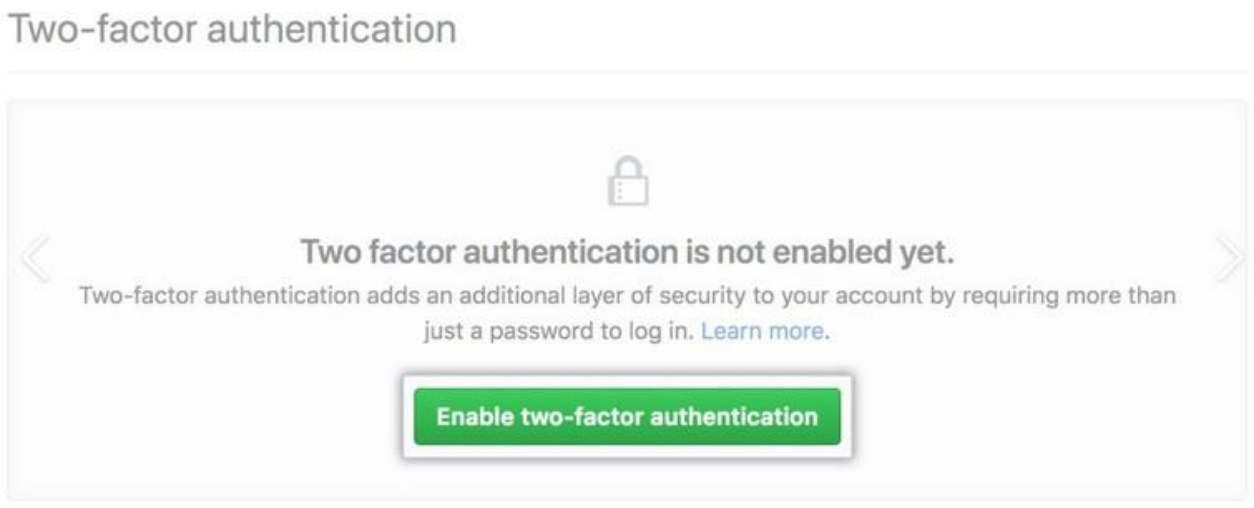
Please check your email settings

Before you can contribute on GitHub, we need you to verify your email address.

he mailserver for web.de is not accepting our messages to [redacted]. Please check th
elling of your email address and make sure email from GitHub is not rejected by any (spam) filte

Send verification email to [redacted] or change your email settings.

4. Configuring two-factor authentication: Two-factor authentication, or 2FA, is an extra layer of security used when logging into websites or apps. We strongly urge you to configure 2FA for safety of your account. For more information, see "About two-factor authentication."

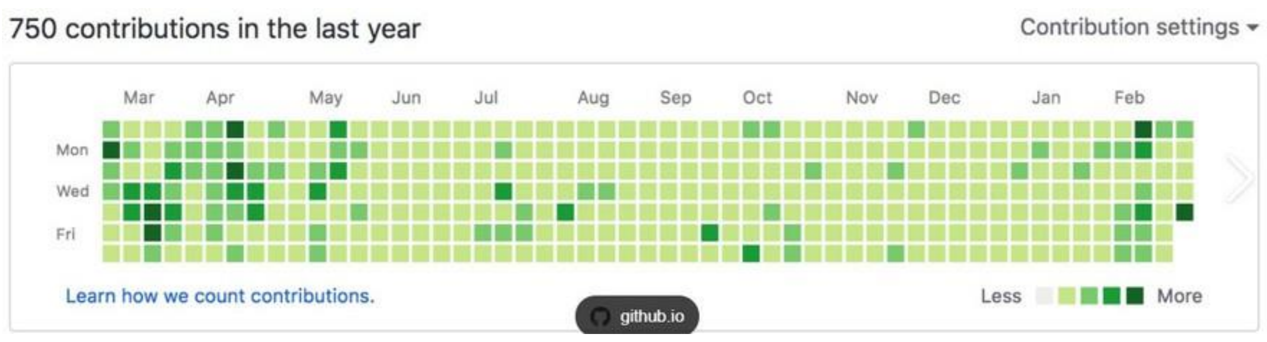


Two-factor authentication

Two factor authentication is not enabled yet.

Two-factor authentication adds an additional layer of security to your account by requiring more than just a password to log in. Learn more.

5. Viewing your GitHub profile and contribution graph: Your GitHub profile tells people the story of your work through the repositories and gists you've pinned, the organisation memberships you've chosen to publicise, the contributions you've made, and the projects you've created. For more information, see "About your profile" and "Viewing contributions on your profile."



Experiment No. 03

Aim: Program to generate logs

Basic Gitinit

Gitinit command creates a new Git repository. It can be used to convert an existing, undersigned project to a Git repository or initialize a new, empty repository. Most other Git commands are not available outside of an initialize repository, so this is usually the first command you'll run in a new project.

Basic Gitstatus

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git. Status output does not show you any information regarding the committed project history.

Basic Gitcommit

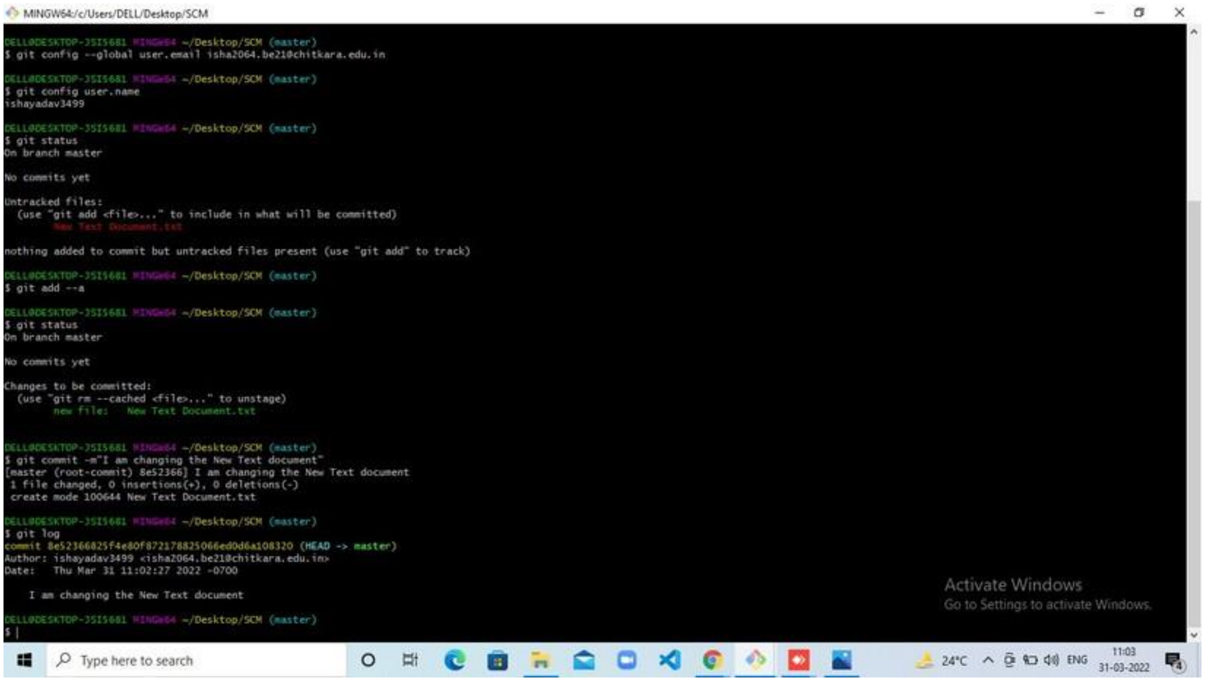
The git commit command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as "safe" versions of a project—Git will never change them unless you explicitly ask it to. Prior to the execution of git commit, The git add command is used to promote or 'stage' changes to the project that will be stored in a commit. These two commands git commit and git add are two of the most frequently used

Basic Git add command

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run git commit

Basic Git log

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



```
MINGW64:/c/Users/DELL/Desktop/SCM

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (master)
$ git config --global user.email isha2064.be21@chitkara.edu.in

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (master)
$ git config user.name
ishayadav3499

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        New Text Document.txt

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

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (master)
$ git add --a

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   New Text Document.txt


DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (master)
$ git commit -m"I am changing the New Text document"
[master (root-commit) 8e52366] I am changing the New Text document
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 New Text Document.txt

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (master)
$ git log
commit 8e52366825f4e80f87217882506 6ed0d6a108320 (HEAD -> master)
Author: ishayadav3499 <isha2064.be21@chitkara.edu.in>
Date:   Thu Mar 31 11:02:27 2022 -0700

    I am changing the New Text document

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (master)
$ |
```

Experiment No. 04

Aim: Create and visualize branches in Git

How to create branches?

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

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

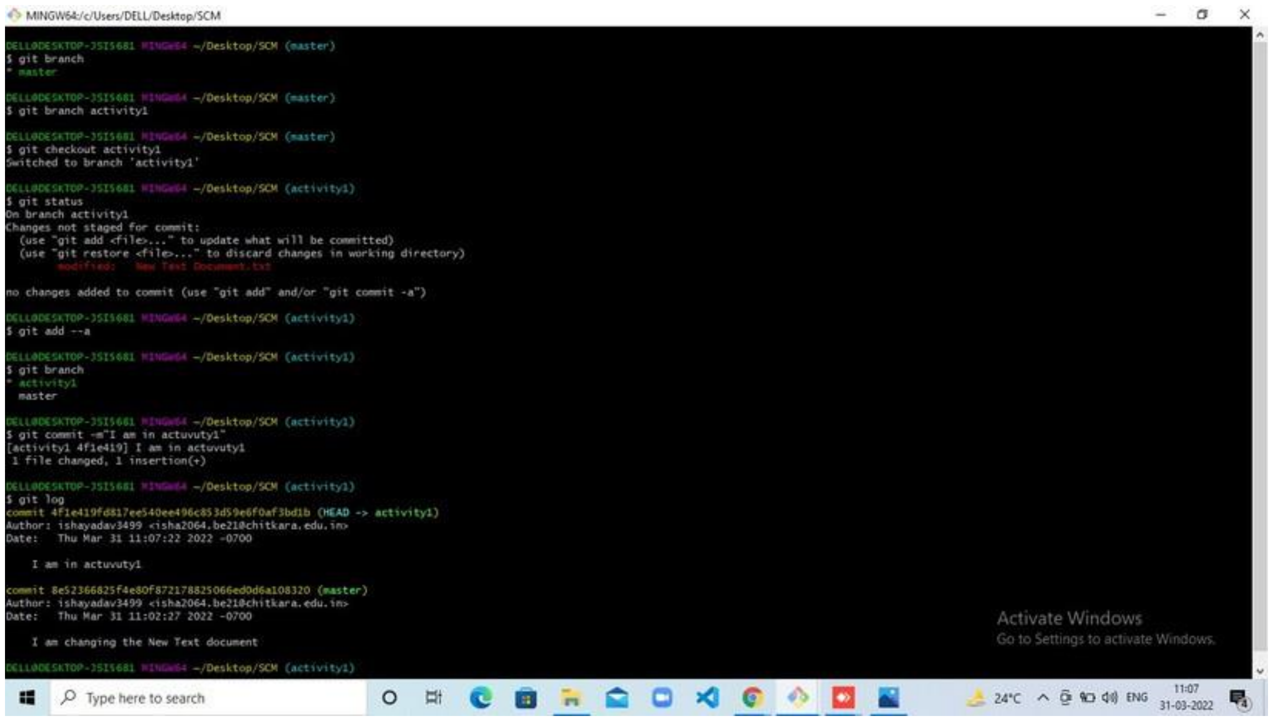
Visualizing Branches:

To visualize, we have to create a new file in the new branch "activity1" instead of the master branch. After this we have to do three step architecture i.e. working directory, staging area and git repository.

After this I have done the 3 Step architecture which is tracking the file, send it to stagging area and finally we can rollback to any previously saved version of this file.

After this we will change the branch from activity1 to master, but when we switch back to master branch the file we created i.e "hello" will not be there. Hence the new file will not be shown in the master branch. In this way we can create and change different branches. We can also merge the branches by using the git merge command.

In this way we can create and change different branches. We can also merge the branches by using git merge command.



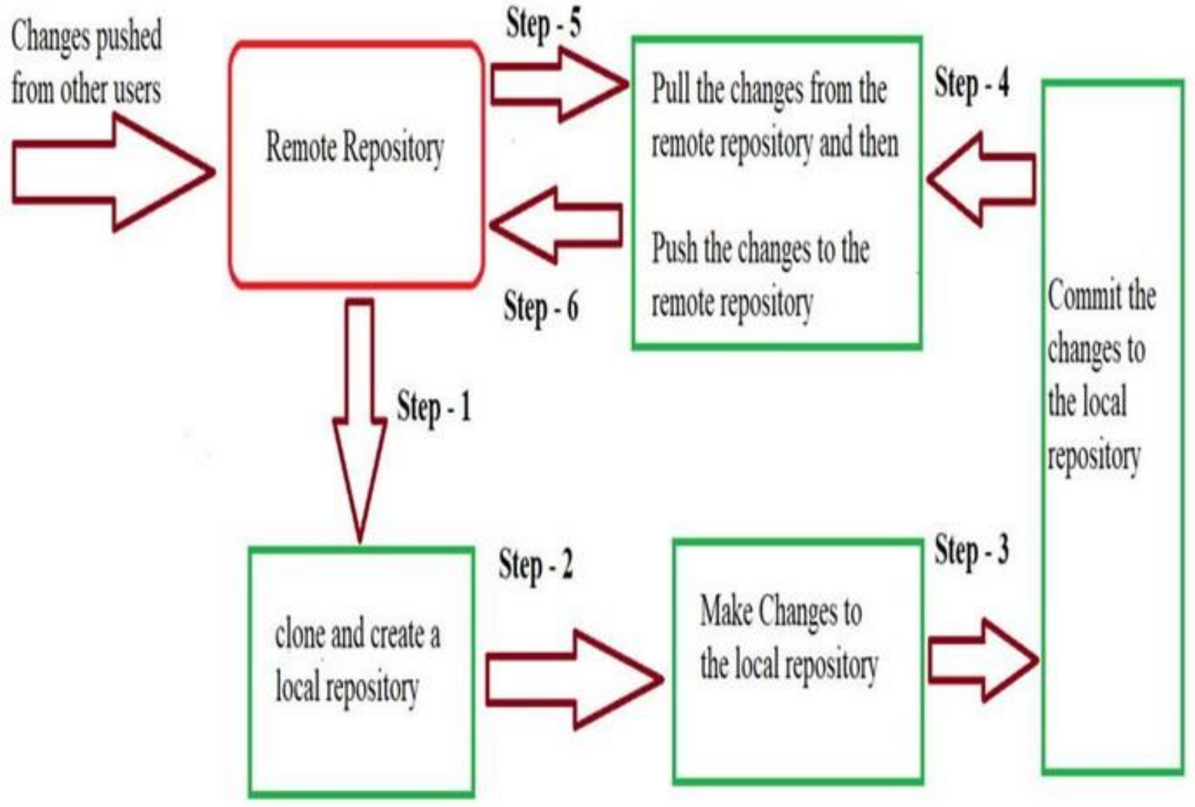
MINGW64:/c/Users/DELL/Desktop/SCM

```
DELL@DESKTOP-JSIS681 MINGW64 ~/Desktop/SCM (master)
$ git branch
* master

DELL@DESKTOP-JSIS681 MINGW64 ~/Desktop/SCM (master)
$ git branch activity1

DELL@DESKTOP-JSIS681 MINGW64 ~/Desktop/SCM (master)
$ git checkout activity1
Switched to branch 'activity1'

DELL@DESKTOP-JSIS681 MINGW64 ~/Desktop/SCM (activity1)
$ git status
On branch activity1
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:   New Text Document.txt

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

DELL@DESKTOP-JSIS681 MINGW64 ~/Desktop/SCM (activity1)
$ git add --a

DELL@DESKTOP-JSIS681 MINGW64 ~/Desktop/SCM (activity1)
$ git branch
* activity1
  master

DELL@DESKTOP-JSIS681 MINGW64 ~/Desktop/SCM (activity1)
$ git commit -m"I am in actuvuty1"
[activity1 4f1e419] I am in actuvuty1
 1 file changed, 1 insertion(+)

DELL@DESKTOP-JSIS681 MINGW64 ~/Desktop/SCM (activity1)
$ git log
commit 4f1e419fd817ee540ee496c853d59e6f0af3bd1b (HEAD -> activity1)
Author: ishayadav3499 <isha2064.be21@chitkara.edu.in>
Date:   Thu Mar 31 11:07:22 2022 -0700

    I am in actuvuty1

commit 8e52366825f4e80f87217882506 6ed0d6a108320 (master)
Author: ishayadav3499 <isha2064.be21@chitkara.edu.in>
Date:   Thu Mar 31 11:02:27 2022 -0700

    I am changing the New Text document

DELL@DESKTOP-JSIS681 MINGW64 ~/Desktop/SCM (activity1)
```

Experiment No. 05

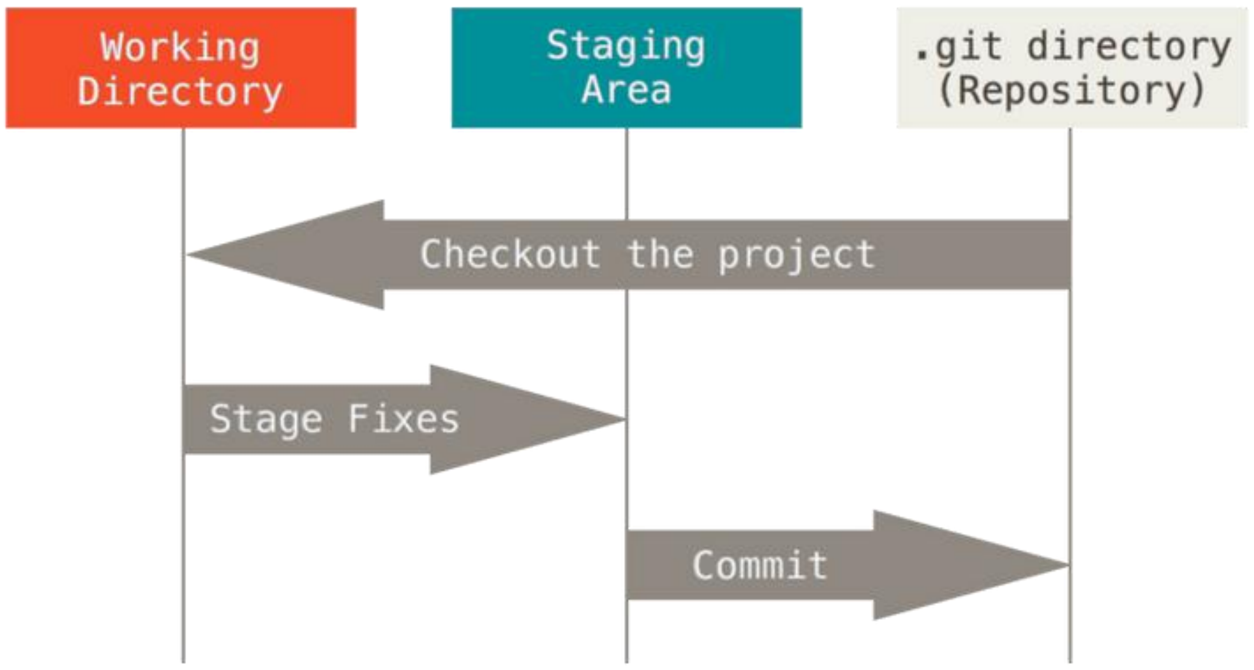
Aim: Git lifecycle description

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


Changes pushed from other users
Remote Repository
Step - 5
Pull the changes from the remote repository and then
Push the changes to the remote repository
Step - 6
Step - 4
Commit the changes to the local repository
Step - 1
Step - 2
clone and create a local repository
Make Changes to the local repository
Step - 3

- **Step 1-** We first clone any of the code residing in the remote repository to make our won local repository.
- **Step 2-** We edit the files that we have cloned in our local repository and make the necessary changes in it.
- **Step 3-** We commit our changes by first adding them to our staging area and committing them with a commit message.
- **Step 4 and Step 5-** We first check whether there are any of the changes done in the remote repository by some other users and we first pull that changes.
- **Step 6-** If there are no changes we push our changes to the remote repository and we are done with our work.

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



This Photo by Unknown Author is licensed under CC BY-SA

1. Working Directory

Whenever we want to initialize aur local project directory to make a Git repository, we use the git init command. After this command, git becomes aware of the files in the project although it does not track the files yet. The files are further tracked in the staging area.

2. Staging Area

Now, to track files the different versions of our files we use the command git add. We can term a staging area as a place where different versions of our files are stored. git add command copies the version of your file from your working directory to the staging area. We can, however, choose which files we need to add to the staging area because in our working directory there are some files that we don't want to get tracked, examples include node modules, temporary files, etc. Indexing in Git is the one that helps Git in understanding which files need to be added or sent. You can find your staging area in the .git folder inside the index file. git add<filename> git add.

3. Git Directory

Now since we have all the files that are to be tracked and are ready in the staging area, we are ready to commit aur files using the git commit command. Commit helps us in keeping the track of the metadata of the files in our staging area. We specify every commit with a message which tells what the commit is about. Git preserves the information or the metadata of the files that were committed in a Git Directory which helps Git in tracking files basically it preserves the photocopy of the committed files. Commit also stores the name of the author who did the commit, files that are committed, and the date at which they are committed along with the commit message. git commit -m <Message>

CHITKARA
UNIVERSITY

Subject Name: **Source Code Management**

Subject Code: CS181

Cluster: Beta

Department: DCSE

Submitted By:

Name: Gautam Sharma

Roll no: 2110990508

Submitted To:

Dr. Monit Kapoor

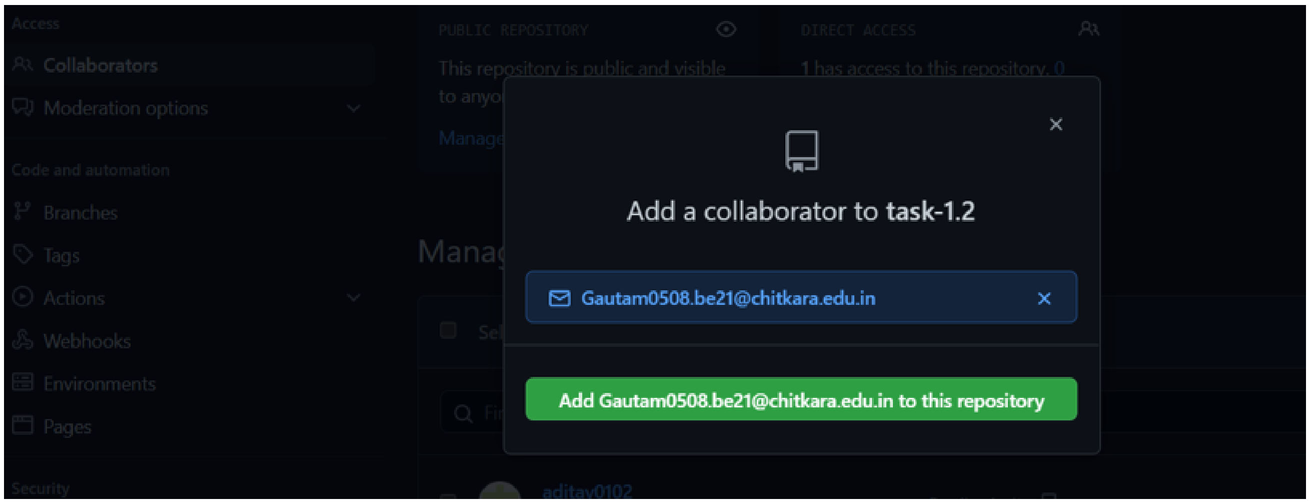
Index

Experiment No. 01

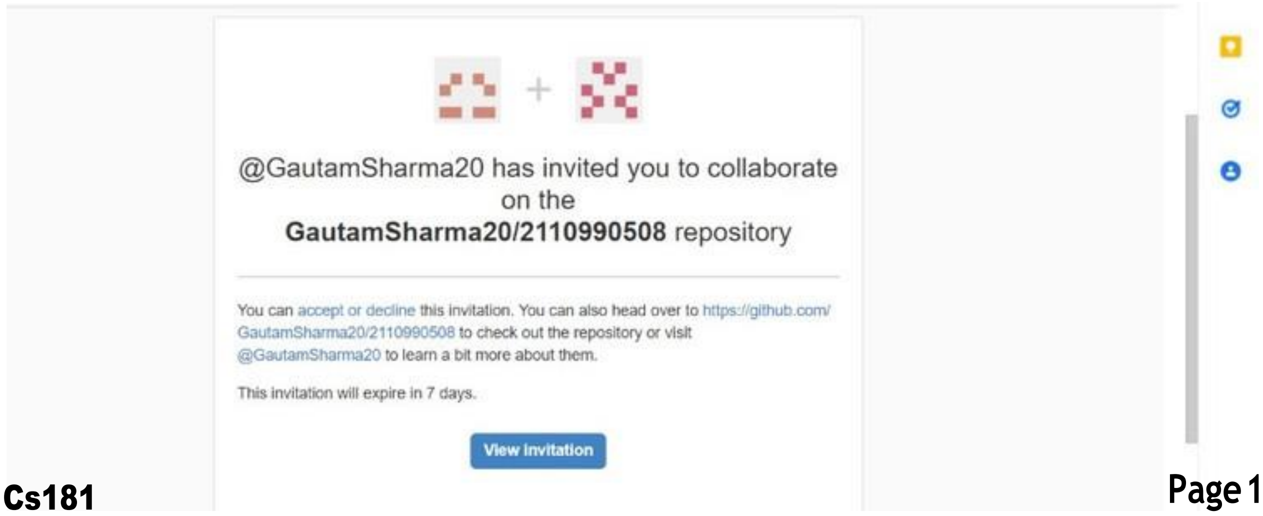
Aim: Add collaborators on Github Repo

To accept access to the Owner's repo, the Collaborator needs to go to https://github.com/notifications or check for email notification. Once there she can accept access to the Owner's repo.

Next, the Collaborator needs to download a copy of the Owner's repository to her machine. This is called "cloning a repo".The Collaborator doesn't want to overwrite her own version of planets.git, so needs to clone the Owner's repository to a different location than her own repository with the same name.To clone the Owner's repo into her Desktop folder, the Collaborator enters:



The Collaborator can now make a change in her clone of the Owner's repository, exactly the same way as we've been doing before:

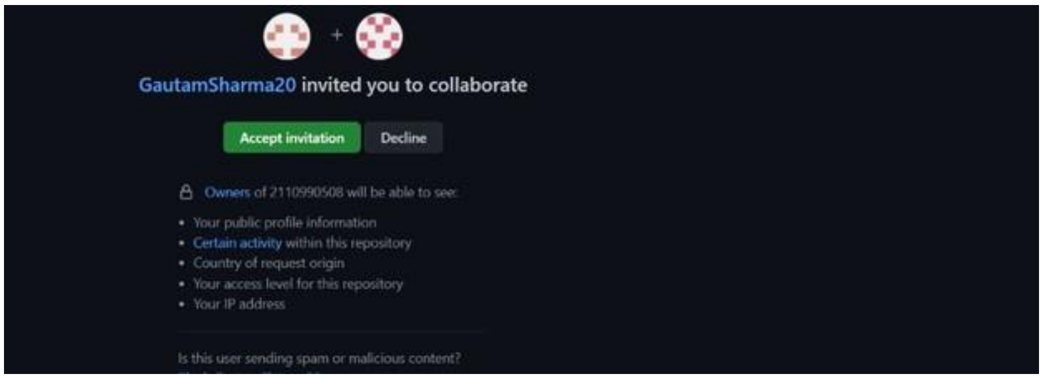


@GautamSharma20 has invited you to collaborate on the **GautamSharma20/2110990508** repository

You can accept or decline this invitation. You can also head over to https://github.com/GautamSharma20/2110990508 to check out the repository or visit @GautamSharma20 to learn a bit more about them.

This invitation will expire in 7 days.

Then push the change to the *Owner's repository* on GitHub:


GautamSharma20 invited you to collaborate

Accept invitation | Decline

Owners of 2110990508 will be able to see:

- Your public profile information
- Certain activity within this repository
- Country of request origin
- Your access level for this repository
- Your IP address

Is this user sending spam or malicious content?

Note that we didn't have to create a remote called origin: Git uses this name by default when we clone a repository. (This is why origin was a sensible choice earlier when we were setting up remotes by hand. Take a look at the Owner's repository on GitHub again, and you should be able to see the new commit made by the Collaborator. You may need to refresh your browser to see the new commit.

Some more about remotes

In this episode and the previous one, our local repository has had a single "remote", called origin. A remote is a copy of the repository that is hosted somewhere else, that we can push to and pull from, and there's no reason that you have to work with only one. For example, on some large projects you might have your own copy in your own GitHub account (you'd probably call this origin) and also the main "upstream" project repository (let's call this upstream for the sake of examples). You would pull from upstream from time to time to get the latest updates that other people have committed. Remember that the name you give to a remote only exists locally. It's an alias that you choose - whether origin, or upstream, or fred - and not something intrinstic to the remote repository.

The git remote family of commands is used to set up and alter the remotes associated with a repository.

Here are some of the most useful ones:

git remote -v lists all the remotes that are configured (we already used this in the last episode)

git remote add [name] [url] is used to add a new remote

git remote remove [name] removes a remote. Note that it doesn't affect the remote repository at all - it just removes the link to it from the local repo.

git remote set-url [name] [newurl] changes the URL that is associated with the remote. This is useful if it has moved, e.g. to a different GitHub account, or from GitHub to a different hosting service. Or, if we made a typo when adding it!

git remote rename [oldname] [newname] changes the local alias by which a remote is known - its name. For example, one could use this to change upstream tofred.

To download the Collaborator's changes from GitHub, the Owner now enters:

Now the three repositories (Owner's local, Collaborator's local, and Owner's on GitHub) are back in sync.

A Basic Collaborative Workflow

In practice, it is good to be sure that you have an updated version of the repository you are collaborating on, so you should git pull before making our changes. The basic collaborative workflow would be:

update your local repo with git pull origin main,
make your changes and stage them with git add,
commit your changes with git commit -m, and
upload the changes to GitHub with git push origin main
It is better to make many commits with smaller changes rather than of one commit with massive

changes:
small commits are easier to read and review.

Switch Roles and Repeat-

Switch roles and repeat the whole process.

Review Changes-

The Owner pushed commits to the repository without giving any information to the Collaborator. How can the Collaborator find out what has changed with command line? And on GitHub?

Comment Changes in GitHub-

The Collaborator has some questions about one line change made by the Owner and has some suggestions to propose. With GitHub, it is possible to comment the diff of a commit. Over the line of code to comment, a blue comment icon appears to open a comment window. The Collaborator posts its comments and suggestions using GitHub interface.

~~Version History, Backup, and~~ Version Control-

Some backup software can keep a history of the versions of your files. They also allows you to recover specific versions. How is this functionality different from version control? What are some of the benefits of using version control, Git and GitHub?

Aim: Fork and Commit

About forks

Most commonly, forks are used to either propose changes to someone else's project or to use someone else's project as a starting point for your own idea. You can fork a repository to create a copy of the repository and make changes without affecting the upstream repository. For more information, see "Working with forks."

Propose changes to someone else's project

For example, you can use forks to propose changes related to fixing a bug. Rather than logging an issue for a bug you've found, you can:

Fork the repository.

Make the fix.

Submit a pull request to the project owner.

Use someone else's project as a starting point for your own idea

Open source software is based on the idea that by sharing code, we can make better, more reliable software. For more information, see the "About the Open Source Initiative" on the Open Source Initiative.
For more information about applying open source principles to your organization's development work on
GitHub.com, see GitHub's white paper "An introduction to innersource."

When creating your public repository from a fork of someone's project, make sure to
include a license file that determines how you want your project to be shared with others.
For more information, see "Choose an open source license" at choosealicense.com.

For more information on open source, specifically how to create and grow an open source
project, we've created Open Source Guides that will help you foster a healthy open source
community by recommending best practices for creating and maintaining repositories for
your open source project. You can also take a free GitHub Learning Lab course on
maintaining open source communities.

Prerequisites

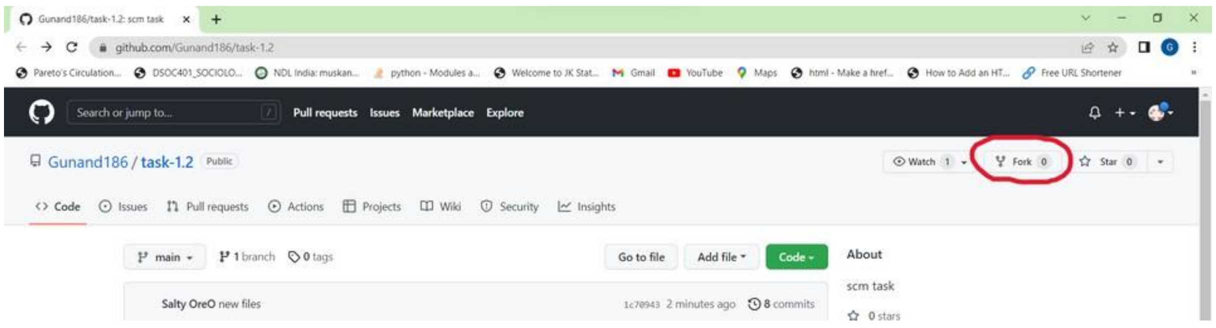
If you haven't yet, you should first set up Git. Don't forget to set up authentication to GitHub.com from Git as well.

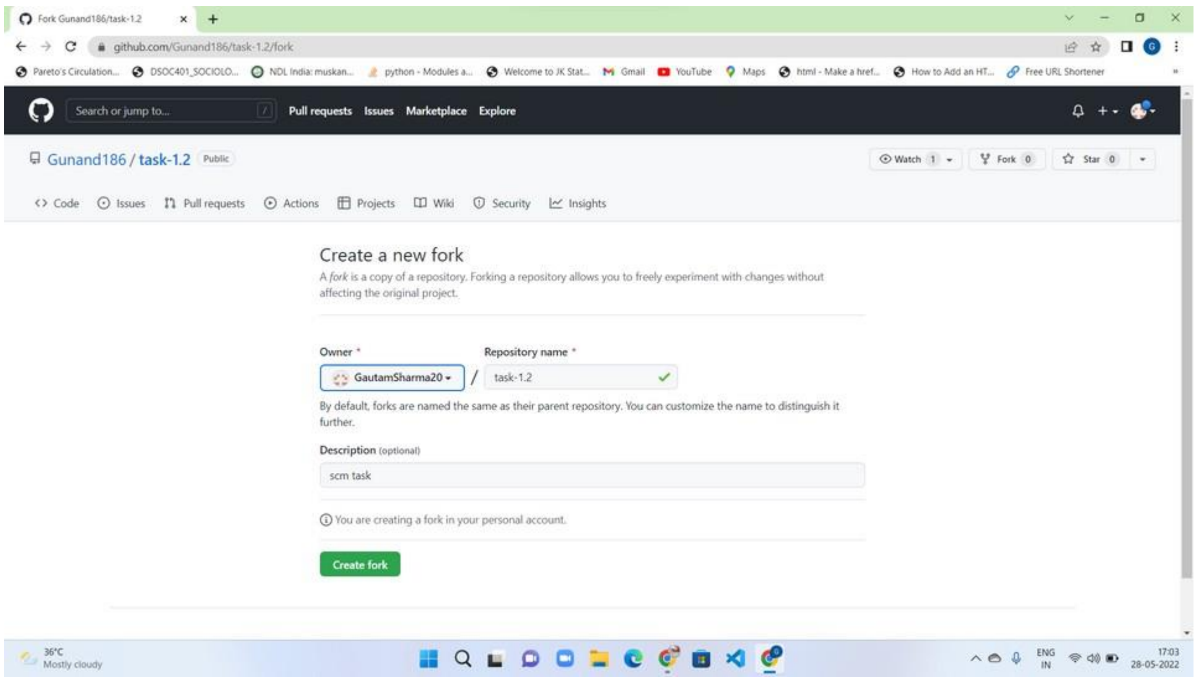
Forking a repository

You might fork a project to propose changes to the upstream, or original, repository. In this case, it's good practice to regularly sync your fork with the upstream repository. To do this, you'll need to use Git on the command line. You can practice setting the upstream repository using the same octocat/Spoon-Knife repository you just forked.

On GitHub.com, navigate to the octocat/Spoon-Knife repository.

In the top-right corner of the page, click Fork.




github.com/Gunand186/task-1.2/fork
Gunand186 / task-1.2 Public
Watch 1
Fork 0
Star 0

Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Owner * GautamSharma20 / Repository name * task-1.2

By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

Description (optional)

scm task

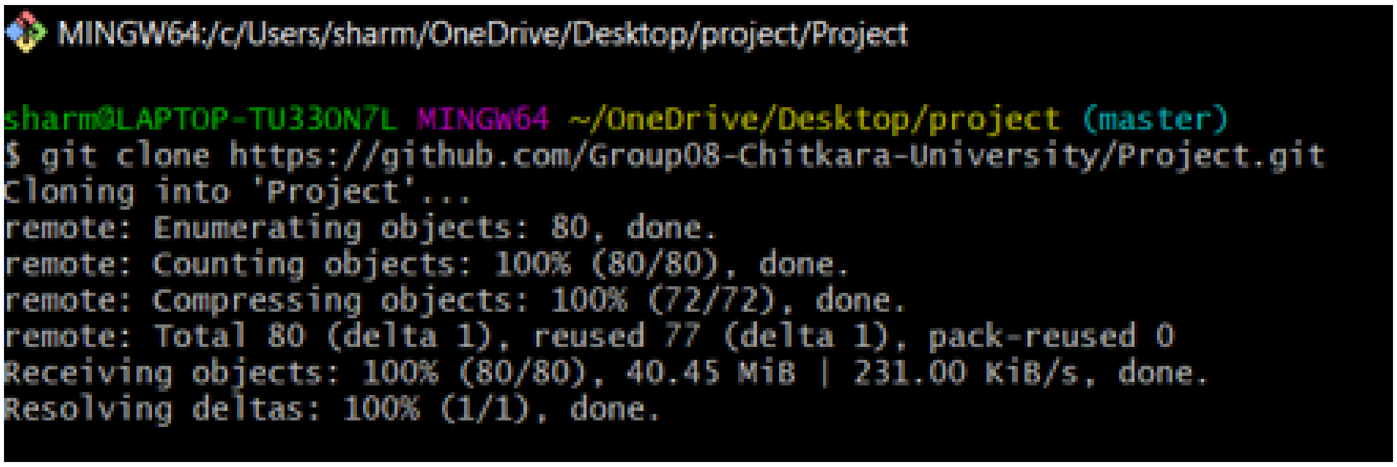
You are creating a fork in your personal account.

Cloning your forked repository

Right now, you have a fork of the Spoon-Knife repository, but you don't have the files in that repository locally on your computer.

On GitHub.com, navigate to your fork of the Spoon-Knife repository.

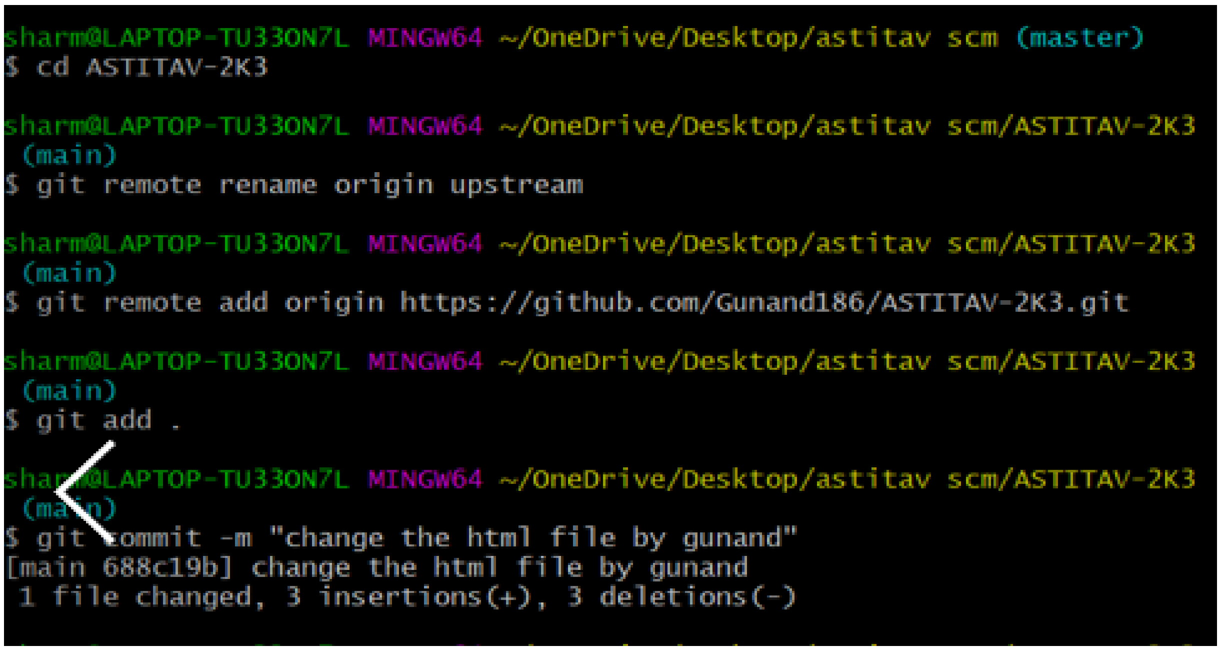
Above the list of files, click Code



```
MINGW64:/c/Users/sharm/OneDrive/Desktop/project/Project

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/project (master)
$ git clone https://github.com/Group08-Chitkara-University/Project.git
Cloning into 'Project'...
remote: Enumerating objects: 80, done.
remote: Counting objects: 100% (80/80), done.
remote: Compressing objects: 100% (72/72), done.
remote: Total 80 (delta 1), reused 77 (delta 1), pack-reused 0
Receiving objects: 100% (80/80), 40.45 MiB | 231.00 KiB/s, done.
Resolving deltas: 100% (1/1), done.
```

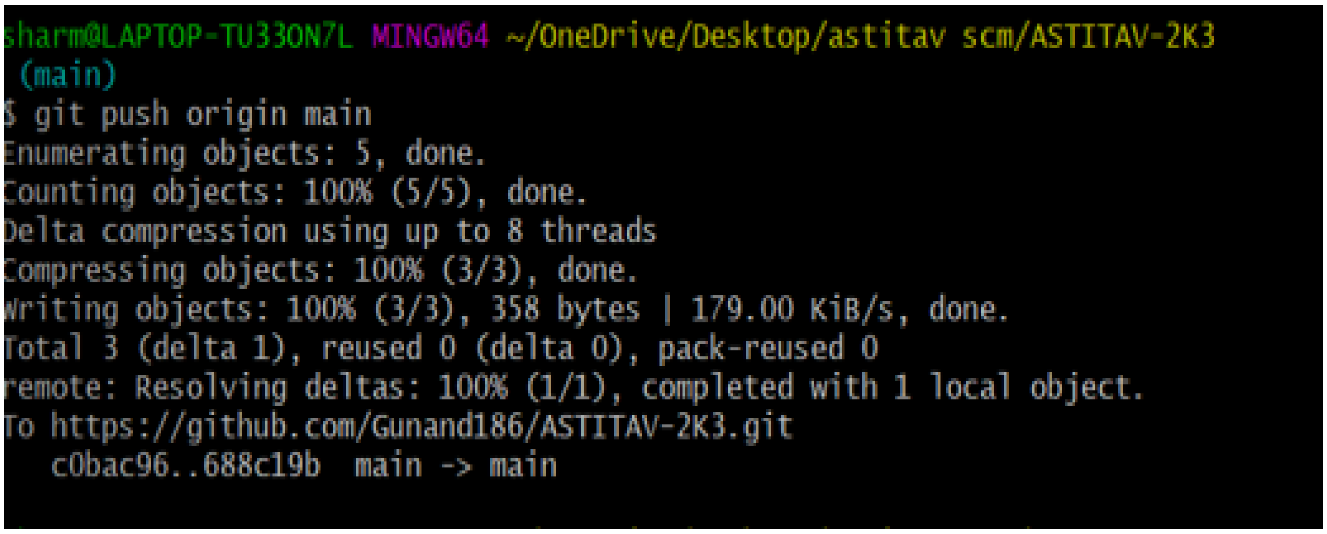
To clone the repository using HTTPS, under "Clone with HTTPS", click . To clone the repository using an SSH key, including a certificate issued by your organization's SSH certificate authority, click **Use SSH**, then click . To clone a repository using GitHub CLI, click **Use GitHub CLI**, then click .

The Collaborator can now make a change in her clone of the Owner’s repository, exactly the same way as we’ve been doing before:



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/astitav scm (master)
$ cd ASTITAV-2K3

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/astitav scm/ASTITAV-2K3
 (main)
$ git remote rename origin upstream

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/astitav scm/ASTITAV-2K3
 (main)
$ git remote add origin https://github.com/Gunand186/ASTITAV-2K3.git

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/astitav scm/ASTITAV-2K3
 (main)
$ git add .

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/astitav scm/ASTITAV-2K3
 (main)
$ git commit -m "change the html file by gunand"
[main 688c19b] change the html file by gunand
 1 file changed, 3 insertions(+), 3 deletions(-)
```

Then push the change to the *Owner’s repository* on GitHub:



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/astitav scm/ASTITAV-2K3
 (main)
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 358 bytes | 179.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/Gunand186/ASTITAV-2K3.git
   c0bac96..688c19b  main -> main
```

Open Git Bash.

Change the current working directory to the location where you want the cloned directory.

Type git clone, and then paste the URL you copied earlier. It will look like this, with your GitHub username instead of YOUR-USERNAME:

$ git clone https://github.com/*YOUR-USERNAME*/Spoon-Knife

Press Enter. Your local clone will be created.

$ git clone https://github.com/*YOUR-USERNAME*/Spoon-Knife

> Cloning into `Spoon-Knife`...

> remote: Counting objects: 10, done.

> remote: Compressing objects: 100% (8/8), done.

> remove: Total 10 (delta 1), reused 10 (delta 1)

Configuring Git to sync your fork with the original repository

When you fork a project in order to propose changes to the original repository, you can configure Git to pull changes from the original, or upstream, repository into the local clone of your fork.

1. On GitHub.com, navigate to the octocat/Spoon-Knife repository.
2. Above the list of files, click Code

3. To clone the repository using HTTPS, under "Clone with HTTPS", click . To clone the repository using an SSH key, including a certificate issued by your organization's SSH certificate authority, click **Use SSH**, then click . To clone a

   repository using GitHub CLI, click **Use GitHub CLI**, then click.

4. Open Git Bash.
5. Change directories to the location of the fork you cloned.

oTo go to your home directory, type just cd with no other text.

oTo list the files and folders in your current directory, type ls.

oTo go into one of your listed directories, type cd your_listed_directory.

oTo go up one directory, type cd ...

6. Type git remote -v and press Enter. You'll see the current configured remote repository for your fork.

7. $ git remote -v

8. > origin https://github.com/YOUR_USERNAME/YOUR_FORK.git (fetch)

> origin https://github.com/YOUR_USERNAME/YOUR_FORK.git (push)

9. Type git remote add upstream, and then paste the URL you copied in Step 2 and press Enter. It will look like this:
$ git remote add upstream https://github.com/octocat/Spoon-Knife.git

10. To verify the new upstream repository you've specified for your fork, type git remote -v again.
You should see the URL for your fork as origin, and the URL for the original repository as upstream.

```
11. $ git remote -v
12. >originhttps://github.com/YOUR_USERNAME/YOUR_FORK.git (fetch)
13. >originhttps://github.com/YOUR_USERNAME/YOUR_FORK.git (push)
14. > upstream https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git (fetch)
> upstream https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git (push)
```

Now, you can keep your fork synced with the upstream repository with a few Git commands. For more information, see " Syncing a fork ."

Next steps

You can make any changes to a fork, including:

Creating branches: *Branches* allow you to build new features or test out ideas without putting your main project at risk.

Opening pull requests: If you are hoping to contribute back to the original repository, you can send a request to the original author to pull your fork into their repository by submitting a ~~pull~~ request.

Find another repository to fork

Fork a repository to start contributing to a project. You can fork a repository to your user account or any organization where you have repository creation permissions. For more information, see " Roles in an organization ." If you have access to a private repository and the owner permits forking, you can fork the repository to your user account or any organization on GitHub Team where you have repository creation permissions. You cannot fork a private repository to an organization using GitHub Free. For more information, see " GitHub's products ."

You can browse Ex~~plore to~~ find projects and start contributing to open source repositories. For more information, see "~~Finding ways to contribute to open source on GitHub~~."

Celebrate

You have now forked a repository, practiced cloning your fork, and configured an upstream repository. For more information about cloning the fork and syncing the changes in a forked repository from your computer see "Set up Git." You can also create a new repository where you can put all your projects and share the code on GitHub. For more information see, "Create a repository."

Each repository in GitHub is owned by a person or an organization. You can interact with the people, repositories, and organizations by connecting and following them on GitHub. For more information see "Be social." GitHub has a great support community where you can ask for help and talk to people from around the world. Join the conversation on Github Support Community.

```
11. $ git remote -v
12. >originhttps://github.com/YOUR_USERNAME/YOUR_FORK.git (fetch)
13. >originhttps://github.com/YOUR_USERNAME/YOUR_FORK.git (push)
14. > upstream https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git (fetch)
> upstream https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git (push)
```

Now, you can keep your fork synced with the upstream repository with a few Git commands. For more information, see " Syncing a fork ."

Next steps

You can make any changes to a fork, including:

Creating branches: *Branches* allow you to build new features or test out ideas without putting your main project at risk.

Opening pull requests: If you are hoping to contribute back to the original repository, you can send a request to the original author to pull your fork into their repository by submitting a ~~pul~~l request.

Find another repository to fork

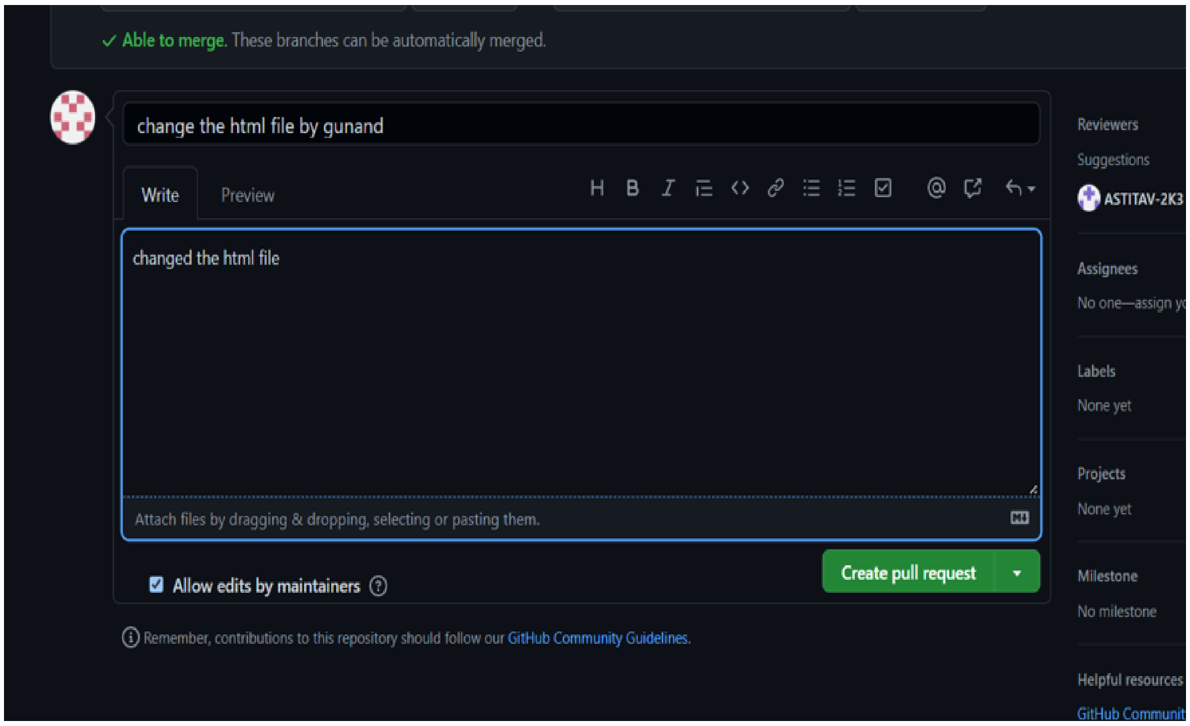
Fork a repository to start contributing to a project. You can fork a repository to your user account or any organization where you have repository creation permissions. For more information, see " Roles in an organization ." If you have access to a private repository and the owner permits forking, you can fork the repository to your user account or any organization on GitHub Team where you have repository creation permissions. You cannot fork a private repository to an organization using GitHub Free. For more information, see " GitHub's products ."

You can browse Ex~~plore to~~ find projects and start contributing to open source repositories. For more information, see "~~Finding ways to contribute to open source on GitHu~~b."

Celebrate

You have now forked a repository, practiced cloning your fork, and configured an upstream repository. For more information about cloning the fork and syncing the changes in a forked repository from your computer see "Set up Git." You can also create a new repository where you can put all your projects and share the code on GitHub. For more information see, "Create a repository."

Each repository in GitHub is owned by a person or an organization. You can interact with the people, repositories, and organizations by connecting and following them on GitHub. For more information see "Be social." GitHub has a great support community where you can ask for help and talk to people from around the world. Join the conversation on Github Support Community.



✓ Able to merge. These branches can be automatically merged.

change the html file by gunand

Write | Preview

changed the html file

Attach files by dragging & dropping, selecting or pasting them.

☑ Allow edits by maintainers

Create pull request

Remember, contributions to this repository should follow our GitHub Community Guidelines.

Reviewers
Suggestions
ASTITAV-2K3

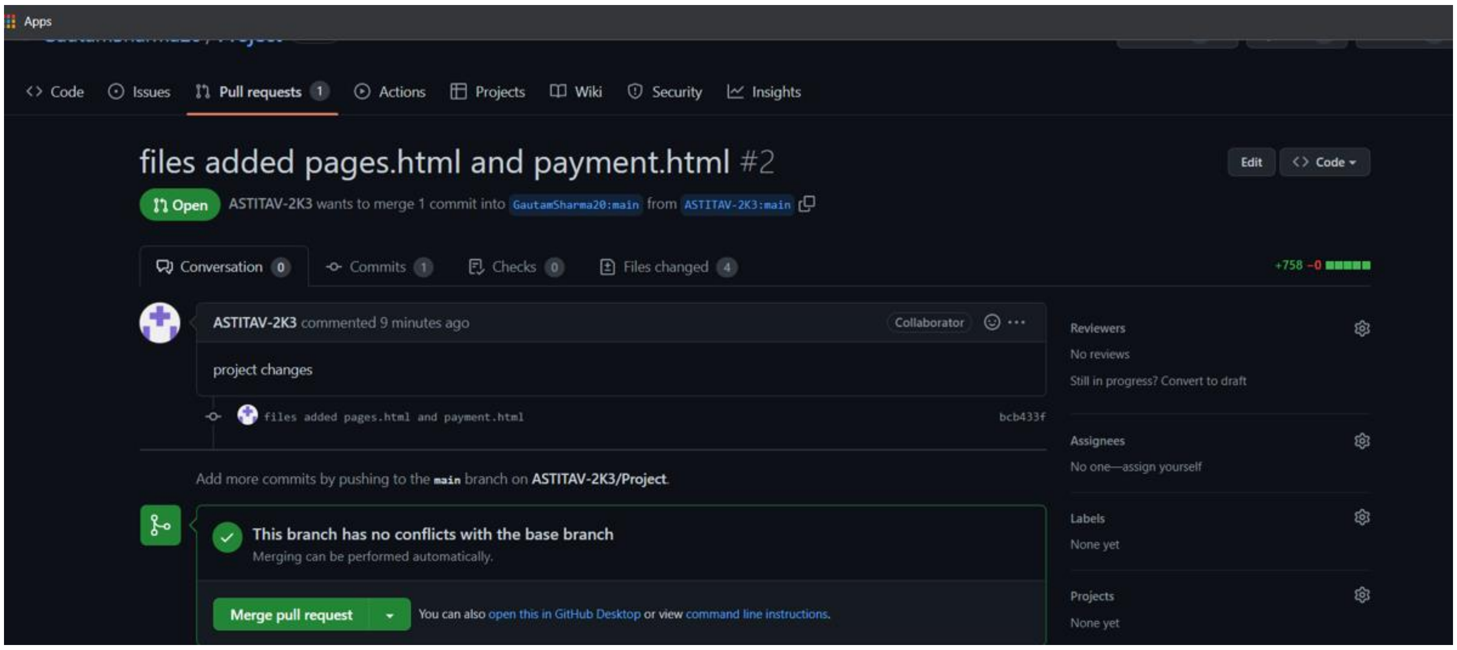
Assignees
No one—assign y

Labels
None yet

Projects
None yet

Milestone
No milestone

Helpful resources
GitHub Communit



files added pages.html and payment.html #2

Open ASTITAV-2K3 wants to merge 1 commit into GautamSharma20:main from ASTITAV-2K3:main

Conversation 0 | Commits 1 | Checks 0 | Files changed 4 | +758 −0

ASTITAV-2K3 commented 9 minutes ago — Collaborator

project changes

files added pages.html and payment.html — bcb433f

Add more commits by pushing to the main branch on ASTITAV-2K3/Project.

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request — You can also open this in GitHub Desktop or view command line instructions.

Reviewers
No reviews
Still in progress? Convert to draft

Assignees
No one—assign yourself

Labels
None yet

Projects
None yet

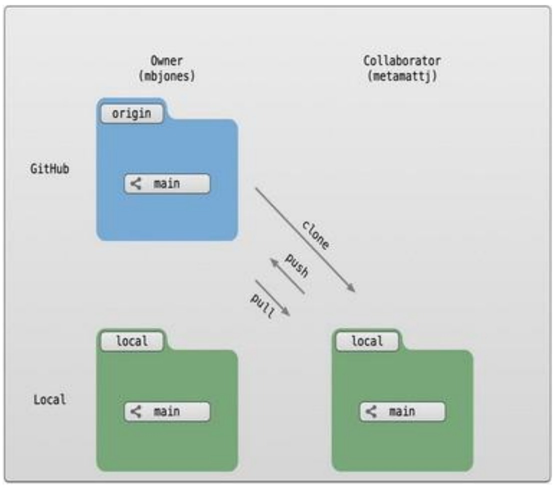
Experiment No. 03

Aim: Merge and Resolve conflicts created due to own activity and collaborators activity

Git is a great tool for working on your own, but even better for working with friends and colleagues. Git allows you to work with confidence on your own local copy of files with the confidence that you will be able to successfully synchronize your changes with the changes made by others.

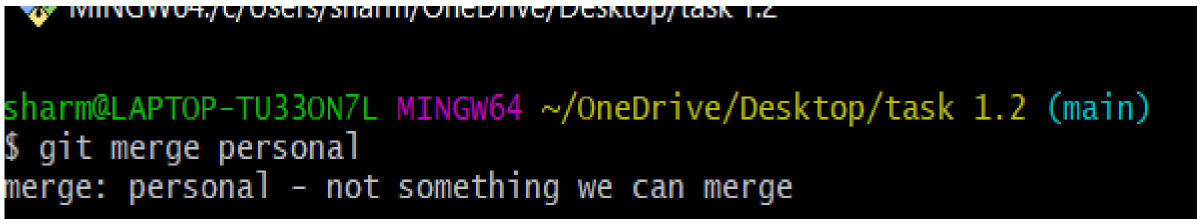
The simplest way to collaborate with Git is to use a shared repository on a hosting service such as GitHub , and use this shared repository as the mechanism to move changes from one collaborator to another. While there are other more advanced ways to sync git repositories, this "hub and spoke" model works really well due to its simplicity.

In this model, the collaborator will clone a copy of the owner's repository from GitHub, and the owner will grant them collaborator status, enabling the collaborator to directly pull and push from the owner's GitHub repository.



Collaborati n g with a trusted colleague without conflicts

We start by enabling collaboration with a trusted colleague. We will designate the Owner as the person who owns the shared repository, and the Collaborator as the person that they wish to grant the ability to make changes to their reposity. We start by giving that person access to our GitHub repository.



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/task 1.2 (main)
$ git merge personal
merge: personal - not something we can merge
```

We will start by having the collaborator make some changes and share those with the Owner without generating any conflicts, In an ideal world, this would be the normal workflow. Here are the typical steps.

Step 1: Collaborator clone

To be able to contribute to a repository, the collaborator must clone the repository from the Owner's github account. To do this, the Collaborator should visit the github page for the Owner's repository, and then copy the clone URL. In R Studio, the Collaborator will create a new project from version control by pasting this clone URL into the appropriate dialog (see the earlier chapter introducing GitHub).

Step 2: Collaborator Edits

With a clone copied locally, the Collaborator can now make changes to the index.Rmd file in the repository, adding a line or statment somewhere noticeable near the top. Save your changes.

Step 3: Collaborator commit and push

To sync changes, the collaborator will need to add, commit, and push their changes to the Owner's repository. But before doing so, its good practice to pull immediately before committing to ensure you have the most recent changes from the owner. So, in R Studio's Git tab, first click the "Diff" button to open the git window, and then press the green "Pull" down arrow button. This will fetch any recent changes from the origin repository and merge them. Next, add the changed index.Rmd file to be committed by cicking the checkbox next to it, type in a commit message, and click 'Commit.' Once that finishes, then the collaborator can immediately click 'Push' to send the commits to the Owner's GitHub repository.

Step 4: Owner pull

Now, the owner can open their local working copy of the code in RStudio, and pull those changes down to their local copy. Congrats, the owner now has your changes!

Step 5: Owner edits, commit, and push

Next, the owner should do the same. Make changes to a file in the repository, save it, pull to make sure no new changes have been made while editing, and then add, commit, and push the Owner changes to GitHub.

Step 6: Collaborator pull

The collaborator can now pull down those owner changes, and all copies are once again fully synced. And you're off to collaborating.

Challenge

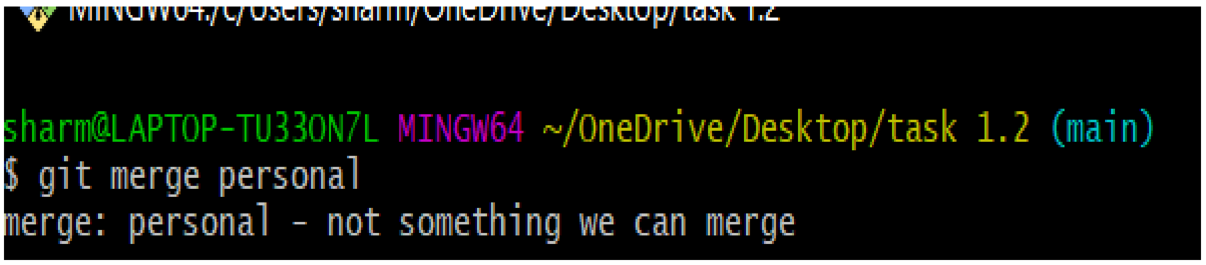
Now that the instructors have demonstrated this conflict-free process, break into pairs and try the same with your partner. Start by designating one person as the Owner and one as the Collaborator, and then repeat the steps described above:

Step 0: Setup permissions for your collaborator
Step 1: Collaborator clones the Owner repository
Step 2: Collaborator Edits the README file
Step 3: Collaborator commits and pushes the file to GitHub
Step 4: Owner pulls the changes that the Collaborator made
Step 5: Owner edits, commits, and pushes some new changes
Step 6: Collaborator pulls the owners changes from GitHub

Merge conflicts

So things can go wrong, which usually starts with a merge conflict, due to both collaborators making incompatible changes to a file. While the error messages from merge conflicts can be daunting, getting things back to a normal state can be straightforward once you've got an idea where the problem lies.

A merge conflict occurs when both the owner and collaborator change the same lines in the same file without first pulling the changes that the other has made. This is most easily avoided by good communication about who is working on various sections of each file, and trying to avoid overlaps. But sometimes it happens, and git is there to warn you about potential problems. And git will not allow you to overwrite one person's changes to a file with another's changes to the same file if they were based on the same version.



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/task 1.2 (main)
$ git merge personal
merge: personal - not something we can merge
```

The main problem with merge conflicts is that, when the Owner and Collaborator both make changes to the same line of a file, git doesn't know whose changes take precedence. You have to tell git whose changes to use for that line.

How to resolve a conflict

Abort, abort, abort...

Sometimes you just made a mistake. When you get a merge conflict, the repository is placed in a 'Merging' state until you resolve it. There's a commandline command to abort doing the merge altogether:

```
git merge --abort
```

Of course, after doing that you stull haven't synced with your collaborator's changes, so things are still unresolved. But at least your repository is now usable on your local machine.

Checkout

The simplest way to resolve a conflict, given that you know whose version of the file you want to keep, is to use the commandline git program to tell git to use either your changes (the person doing the merge), or their changes (the other collaborator).

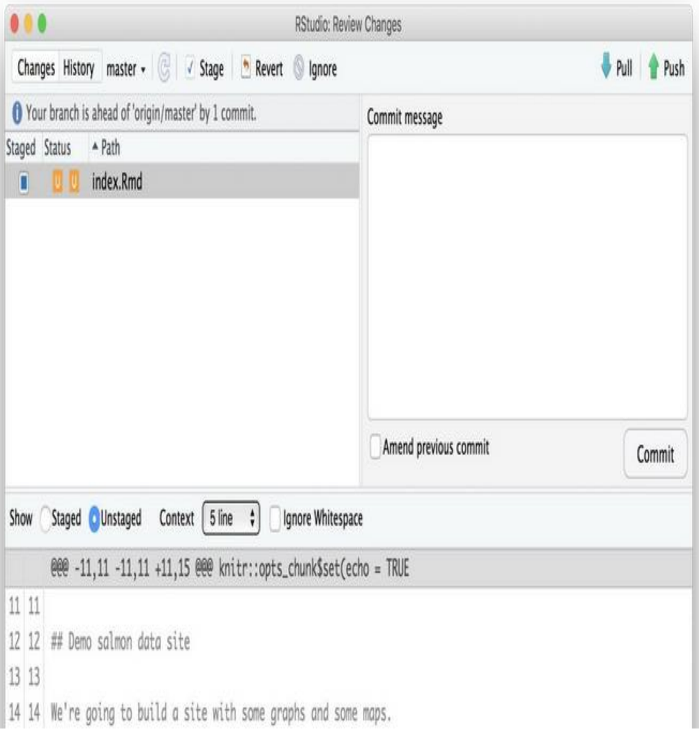
keep your collaborators file: git checkout --theirs conflicted_file.Rmd

keep your own file: git checkout --ours conflicted_file.Rmd

Once you have run that command, then run add,commit , and push the changes as normal.

Pull and edit the file

But that requires the commandline. If you want to resolve from RStudio, or if you want to pick and choose some of your changes and some of your collaborator's, then instead you can manually edit and fix the file. When you pulled the file with a conflict, git notices that there is a conflict and modifies the file to show both your own changes and your collaborator's changes in the file. It also shows the file in the Git tab with an orange U icon, which indicates that the file is Unmerged, and therefore awaiting you help to resolve the conflict. It delimits these blocks with a series of less than and greater than signs, so they are easy to find:



Your branch is ahead of 'origin/master' by 1 commit.

index.Rmd

```
@@@ -11,11 -11,11 +11,15 @@@ knitr::opts_chunk$set(echo = TRUE

## Demo salmon data site

We're going to build a site with some graphs and some maps.
```

To resolve the conficts, simply find all of these blocks, and edit them so that the file looks how you want (either pick your lines, your collaborators lines, some combination, or something altogether new), and save. Be sure you removed the delimiter lines that started with <<<<<<<, =======, and >>>>>>>.

Once you have made those changes, you simply add, commit, and push the files to resolve the conflict.

Producing and resolving merge conflicts

To illustrate this process, we're going to carefully create a merge conflict step by step, show how to resolve it, and show how to see the results of the successful merge after it is complete. First, we will walk through the exercise to demonstrate the issues.

Owner and collaborator ensure all changes are updated

First, start the exercise by ensuring that both the Owner and Collaborator have all of the changes synced to their local copies of the Owner's repositoriy in RStudio. This includes doing a git pull to ensure that you have all changes local, and make sure that the Git tab in RStudio doesn't show any changes needing to be committed.

Owner makes a change and commits

From that clean slate, the Owner first modifies and commits a small change inlcuding their name on a specific line of the README.md file (we will change line 4). Work to only change that one line, and add your username to the line in some form and commit the changes (but DO NOT push). We are now in the situation where the owner has unpushed changes that the collaborator can not yet see.

Collaborator makes a change and commits on the same line

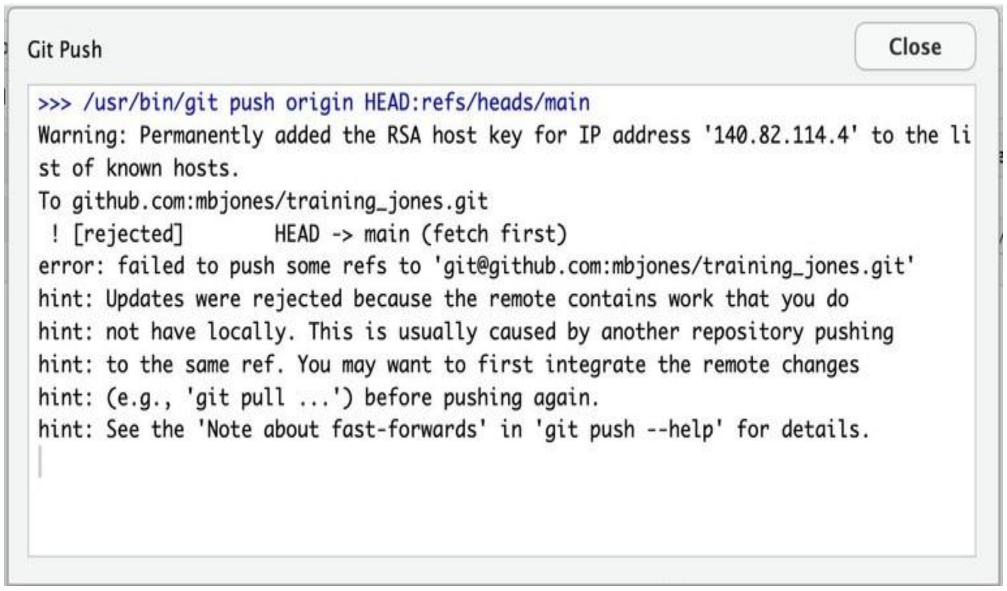
Now the collaborator also makes changes to the same (line 4) of the README.md file in their RStudio copy of the project, adding their name to the line. They then commit. At this point, both the owner and collaborator have committed changes based on their shared version of the README.md file, but neither has tried to share their changes via GitHub.

Collaborator pushes the file to GitHub

Sharing starts when the Collaborator pushes their changes to the GitHub repo, which updates GitHub to their version of the file. The owner is now one revision behind, but doesn't yet know it.

Owner pushes their changes and gets an error

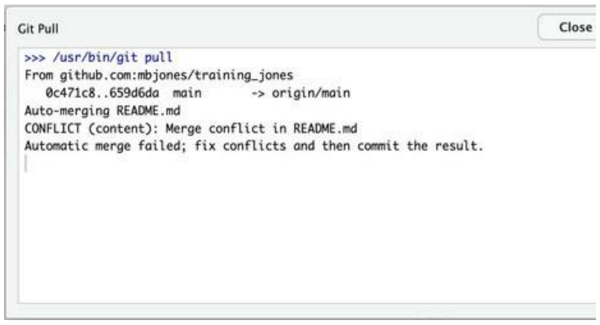
At this point, the owner tries to push their change to the repository, which triggers an error from GitHub. While the error message is long, it basically tells you everything needed (that the owner's repository doesn't reflect the changes on GitHub, and that they need to pull before they can push).


Git Push

```
>>> /usr/bin/git push origin HEAD:refs/heads/main
Warning: Permanently added the RSA host key for IP address '140.82.114.4' to the li
st of known hosts.
To github.com:mbjones/training_jones.git
 ! [rejected]        HEAD -> main (fetch first)
error: failed to push some refs to 'git@github.com:mbjones/training_jones.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

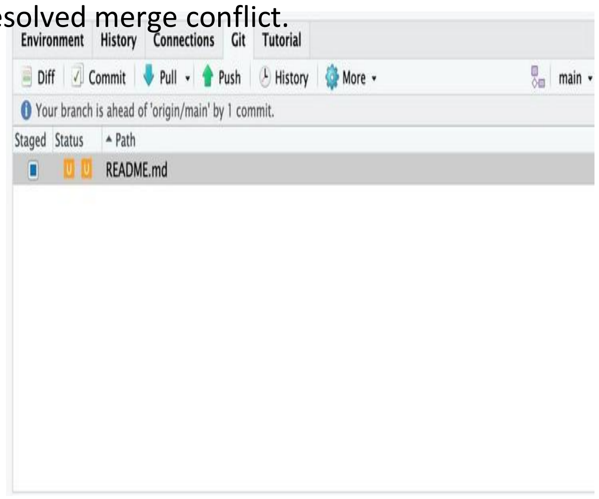
Owner pulls from GitHub to get Collaborator changes

Doing what the message says, the Owner pulls the changes from GitHub, and gets another, different error message. In this case, it indicates that there is a merge conflict because of the conflicting lines.


Git Pull

```
>>> /usr/bin/git pull
From github.com:mbjones/training_jones
   0c471c8..659d6da  main       -> origin/main
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

In the Git pane of RStudio, the file is also flagged with an orange 'U', which stands for an unresolved merge conflict.

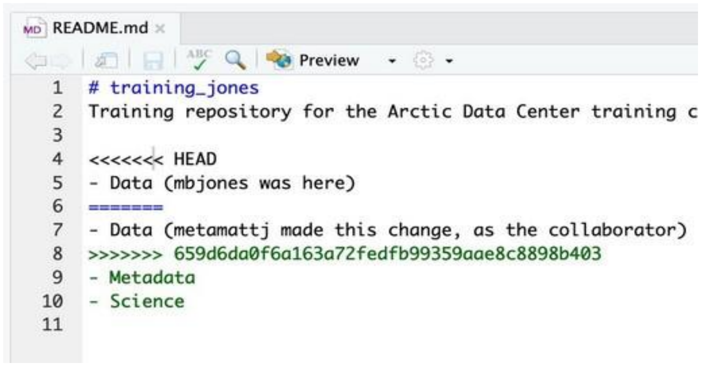


Your branch is ahead of 'origin/main' by 1 commit.

Staged	Status	Path
	U U	README.md

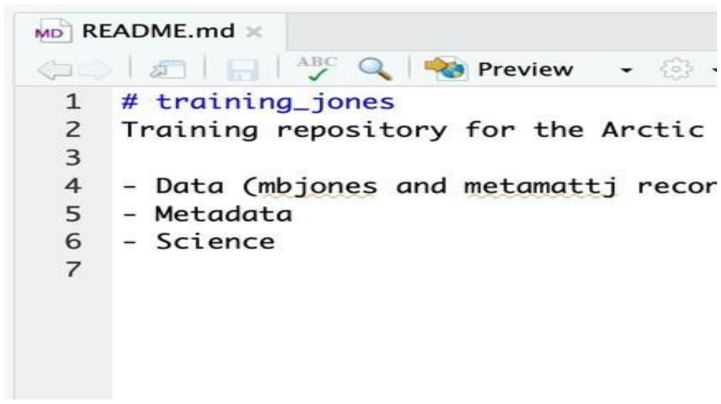
Owner edits the file to resolve the conflict

To resolve the conflict, the Owner now needs to edit the file. Again, as indicated above, git has flagged the locations in the file where a conflict occcurred with <<<<<<<, =======, and >>>>>>>. The Owner should edit the file, merging whatever changes are appropriate until the conflicting lines read how they should, and eliminate all of the marker lines with with <<<<<<<, =======, and >>>>>>>.



```
# training_jones
Training repository for the Arctic Data Center training c

<<<<<<< HEAD
- Data (mbjones was here)
=======
- Data (metamattj made this change, as the collaborator)
>>>>>>> 659d6da0f6a163a72fedfb99359aae8c8898b403
- Metadata
- Science

```

Of course, for scripts and programs, resolving the changes means more than just merging the text – whoever is doing the merging should make sure that the code runs properly and none of the logic of the program has been broken.



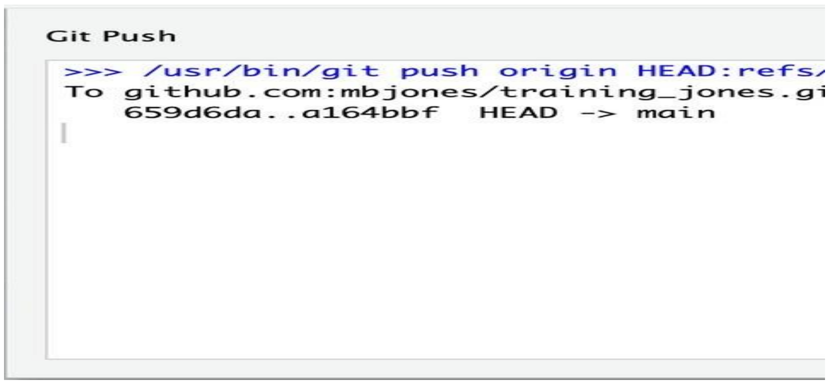
```
# training_jones
Training repository for the Arctic

- Data (mbjones and metamattj recor
- Metadata
- Science

```

Owner commits the resolved changes

From this point forward, things proceed as normal. The owner first ‘Adds’ the file changes to be made, which changes the orange U to a blue M for modified, and then commits the changes locally. The owner now has a resolved version of the file on their system.

Owner pushes the resolved changes to GitHub

Have the Owner push the changes, and it should replicate the changes to GitHub without error.



Git Push

```
>>> /usr/bin/git push origin HEAD:refs/
To github.com:mbjones/training_jones.gi
   659d6da..a164bbf  HEAD -> main
```

Collaborator pulls the resolved changes from GitHub

Finally, the Collaborator can pull from GitHub to get the changes the owner made.

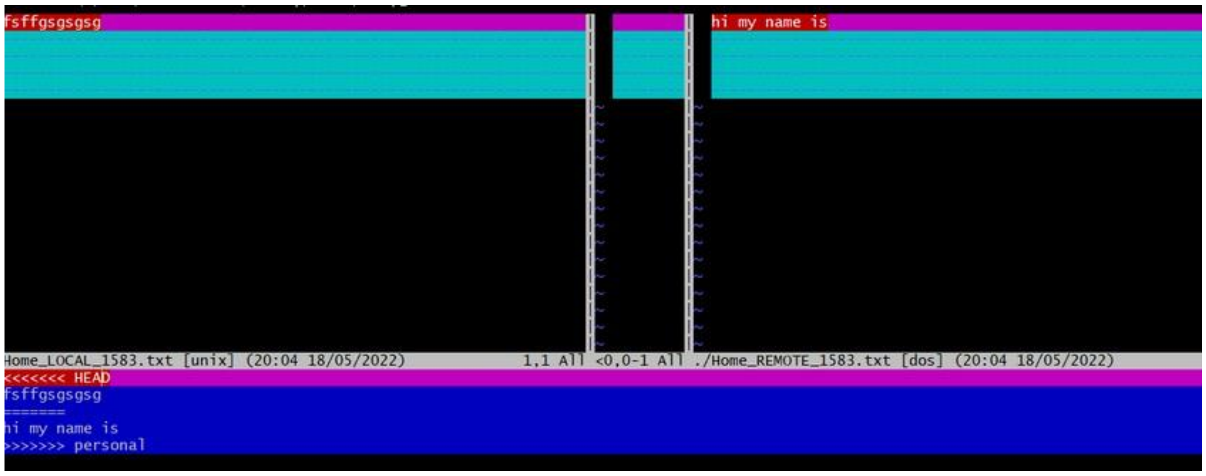
Both can view commit history

When either the Collaborator or the Owner view the history, the conflict, associated branch, and the merged changes are clearly visible in the history.

Merge Conflict Challenge

Now it's your turn. In pairs, intentionally create a merge conflict, and then go through the steps needed to resolve the issues and continue developing with the merged files. See the sections above for help with each of these steps:

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



```
fsffgsgsgsg                                                              hi my name is
Home_LOCAL_1583.txt [unix] (20:04 18/05/2022)          1,1 All <0,0-1 All ./Home_REMOTE_1583.txt [dos] (20:04 18/05/2022)
<<<<<<< HEAD
fsffgsgsgsg
=======
hi my name is
>>>>>>> personal
```

Workflows to avoid merge conflicts

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

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

A good workflow is encapsulated as follows:

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

Aim: Reset and Revert

One of the lesser understood (and appreciated) aspects of working with Git is how easy it is to get back to where you were before—that is, how easy it is to undo even major changes in a repository. In this article, we'll take a quick look at how to reset, revert, and completely return to previous states, all with the simplicity and elegance of individual Git commands.

How to reset a Git commit

Let's start with the Git command reset. Practically, you can think of it as a "rollback"—it points your local environment back to a previous commit. By "local environment," we mean your local repository, staging area, and working directory.

Take a look at Figure 1. Here we have a representation of a series of commits in Git. A branch

in

Git is simply a named, movable pointer to a specific commit. In this case, our branch master is a

pointer to the latest commit in the chain.

If we look at what's in our *master* branch now, we can see the chain of commits made so far.

```
$ git log --oneline b7
64
64
```

Programming and development

Red

Programming cheat sheets
Try for free: Red Hat Learning Subscription
eBook: An introduction to programming with Bash
Bash Shell Scripting Cheat Sheet
eBook: Modernizing Enterprise Java

What happens if we want to roll back to a previous commit. Simple—we can just move the branch pointer. Git supplies the reset command to do this for us. For example, if we want to reset *master* to point to the commit two back from the current commit, we could use either of the following methods:

$ git reset 9ef9173 (using an absolute commit SHA1 value 9ef9173) or

$ git reset current~2 (using a relative value -2 before the "current" tag)

Figure 2 shows the results of this operation. After this, if we execute a git log command on the current branch (*master*), we'll see just the one commit

```
$ git log --oneline 9ef9173
File with one line
```

The git reset command also includes options to update the other parts of your local environment with the contents of the commit where you end up. These options include: hard to reset the commit being pointed to in the repository, populate the working directory with the contents of the commit, and reset the staging area; soft to only reset the pointer in the repository; and mixed (the default) to reset the pointer and the staging area.

Using these options can be useful in targeted circumstances such as git reset --hard <commit sha1 | reference>. This overwrites any local changes you haven't committed. In effect, it resets (clears out) the staging area and overwrites content in the working directory with the content from the commit you reset to. Before you use the hard option, be sure that's what you really want to do, since the command overwrites any uncommitted changes.

How to revert a Git commit

The net effect of the git revert command is similar to reset, but its approach is different. Where the reset command moves the branch pointer back in the chain (typically) to "undo" changes, the revert command adds a new commit at the end of the chain to "cancel" changes. The effect is most easily seen by looking at Figure 1 again. If we add a line to a file in each commit in the chain, one way to get back to the version with only two lines is to reset to that commit, i.e., git reset HEAD~1.

Another way to end up with the two-line version is to add a new commit that has the third line removed—effectively canceling out that change. This can be done with a git revert command, such as:

$ git revert HEAD
Because this adds a new commit, Git will prompt for the commit message:

```
Revert "File with three lines"

This reverts commit
b764644bad524b80457
7684bf74e7bca3117f5
54.
```

```
# On branch master
# Changes to be committed:
#
```

Figure 3 (below) shows the result after the revert operation is completed.

If we do a git log now, we'll see a new commit that reflects the contents before the previous commit.

```
$ git log --oneline
11b7712 Revert "File with three lines"
b764644 File with three lines
7c709f0 File with two lines
9ef9173 File with one line
```

Here are the current contents of the file in the working directory:

```
$   cat   <filename>
Line 1
Line 2
```

Revert or reset?

Why would you choose to do a revert over a reset operation? If you have already pushed your chain of commits to the remote repository (where others may have pulled your code and started working with it), a revert is a nicer way to cancel out changes for them. This is because the Git workflow works well for picking up additional commits at the end of a branch, but it can be challenging if a set of commits is no longer seen in the chain when someone resets the branch pointer back.

This brings us to one of the fundamental rules when working with Git in this manner: Making these kinds of changes in your *local repository* to code you haven't pushed yet is fine. But avoid making changes that rewrite history if the commits have already been pushed to the remote repository and others may be working with them.

In short, if you rollback, undo, or rewrite the history of a commit chain that others are working

with, your colleagues may have a lot more work when they try to merge in changes based on the original chain they pulled. If you must make changes against code that has already been pushed and is being used by others, consider communicating before you make the changes and give people the chance to merge their changes first. Then they can pull a fresh copy after the infringing operation without needing to merge.

You may have noticed that the original chain of commits was still there after we did the reset. We

moved the pointer and reset the code back to a previous commit, but it did not delete any commits. This means that, as long as we know the original commit we were pointing to, we can "restore" back to the previous point by simply resetting back to the original head of the branch:

git reset <sha1 of commit>

A similar thing happens in most other operations we do in Git when commits are replaced. New commits are created, and the appropriate pointer is moved to the new chain. But the old chain of commits still exists.

Rebase

Now let's look at a branch rebase. Consider that we have two branches—

master and *feature*—with the chain of commits shown in Figure 4
below. *Master* has the chain C4->C2->C1->C0 and *feature* has the chain C5->C3-
>C2->C1->C0.

If we look at the log of commits in the branches, they might look like the following. (The C designators for the commit messages are used to make this easier to understand.)

```
$ git log --oneline master
6a92e7a C4

259bf36 C2

f33ae68 C1
5043e79 C0


$ git log --oneline feature
79768b8 C5
000f9ae C3
259bf36 C2
```

I tell people to think of a rebase as a "merge with history" in Git. Essentially what Git does is take each different commit in one branch and attempt to "replay" the differences onto the other branch.

So, we can rebase a feature onto master to pick up C4 (e.g., insert it into feature's chain).

Using the basic Git commands, it might look like this:

```
$ git checkout feature
$ git rebase master

First, rewinding head to replay your work on top of
it... Applying: C3
```

Afterward, our chain of commits would look like Figure 5.

Again, looking at the log of commits, we can see the changes.

```
$ git log --oneline master
6a92e7a C4

259bf36 C2
```

Notice that we have C3' and C5'—new commits created as a result of making the changes from the originals "on top of" the existing chain in master. But also notice that the "original" C3 and C5 are still there—they just don't have a branch pointing to them anymore.

If we did this rebase, then decided we didn't like the results and wanted to undo it, it would be as simple as:

$ git reset 79768b8

With this simple change, our branch would now point back to the same set of commits as before the rebase operation—effectively undoing it (Figure 6).

What happens if you can't recall what commit a branch pointed to before an operation?

Fortunately, Git again helps us out. For most operations that modify pointers in this way, Git remembers the original commit for you. In fact, it stores it in a special reference named ORIG_HEAD within the .git repository directory. That path is a file containing the most recent reference before it was modified. If we cat the file, we can see its contents.

```
$ cat .git/ORIG_HEAD
79768b891f47ce06f13456a7e222536ee47ad2fe
```

We could use the reset command, as before, to point back to the original chain. Then the log would show this:

```
$   git   log   --oneline   feature
79768b8 C5
```

```
000f9ae C3
259bf36 C2
f33ae68 C1
5043e79 C0
```

Another place to get this information is in the reflog. The reflog is a play-by- play listing of switches or changes to references in your local repository. To see it, you can use the git reflog command:

```
$ git reflog
79768b8 HEAD@{0}:reset: moving to79768b
c4533a5 HEAD@{1}: rebase finished: refs/heads/featurereturning to


c4533a5 HEAD@{2}:rebase: C5
64f2047
HEAD@{3}:rebase: C3
6a92e7a
HEAD@{4}:rebase: checkoutmaster
79768b8HEAD@{5}:checkout: movingfrom feature to
feature
79768b8HEAD@{6}:commit: C5
000f9ae HEAD@{7}:checkout: moving commit: from master to feature
6a92e7a
HEAD@{8}:C4
259bf36
HEAD@{9}:checkout: movingfrom feature to master
000f9ae HEAD@{10}: commit: C3
259bf36 HEAD@{11}: checkout: moving from master to feature
259bf36 HEAD@{12}: commit: C2 f33ae68
HEAD@{13}: commit: C1
5043e79 HEAD@{14}: commit (initial): C0
```

You can then reset to any of the items in that list using the special relative naming format you see in the log:

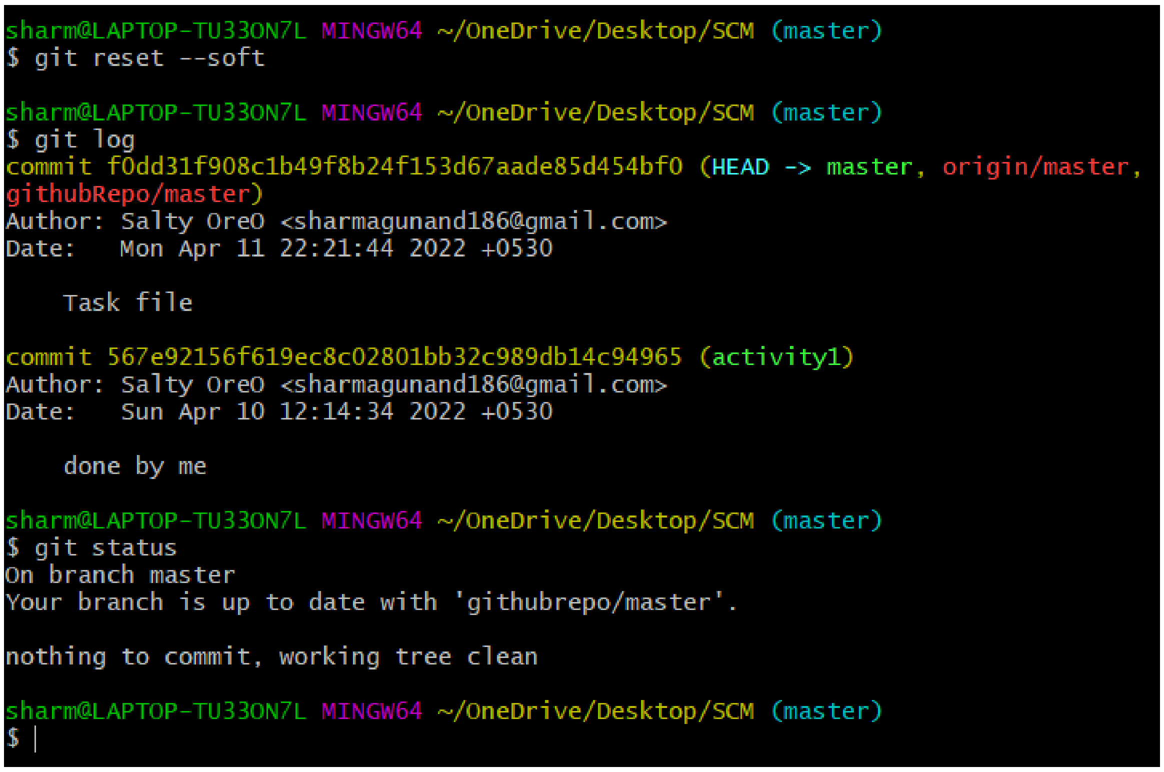
```
$ git reset HEAD@{1}
```

Once you understand that Git keeps the original chain of commits around when operations "modify" the chain, making changes in Git becomes much less scary. This is one of Git's core strengths: being able to quickly and easily try things out and undo them if they don't work.

Snapshots

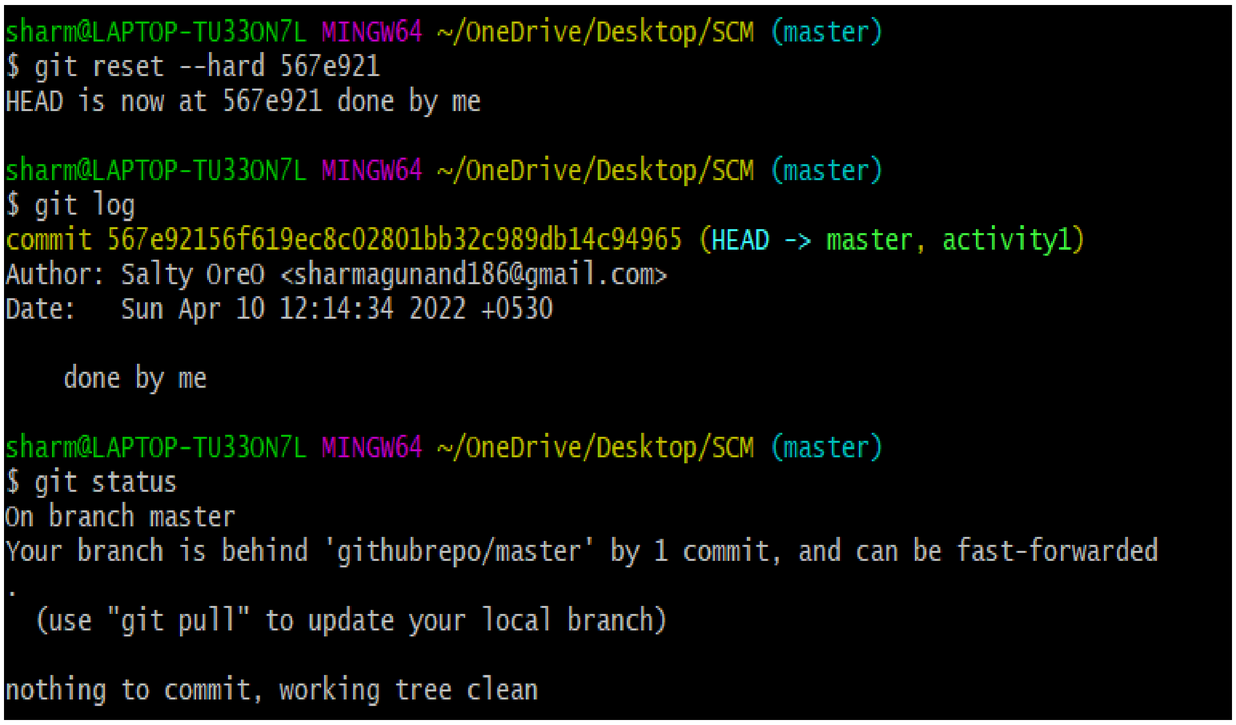
Git log for file before reset

git reset --soft



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git reset --soft

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git log
commit f0dd31f908c1b49f8b24f153d67aade85d454bf0 (HEAD -> master, origin/master,
githubRepo/master)
Author: Salty Ore0 <sharmagunand186@gmail.com>
Date:   Mon Apr 11 22:21:44 2022 +0530

    Task file

commit 567e92156f619ec8c02801bb32c989db14c94965 (activity1)
Author: Salty Ore0 <sharmagunand186@gmail.com>
Date:   Sun Apr 10 12:14:34 2022 +0530

    done by me

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git status
On branch master
Your branch is up to date with 'githubrepo/master'.

nothing to commit, working tree clean

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ |
```

git reset –hard



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git reset --hard 567e921
HEAD is now at 567e921 done by me

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git log
commit 567e92156f619ec8c02801bb32c989db14c94965 (HEAD -> master, activity1)
Author: Salty Ore0 <sharmagunand186@gmail.com>
Date:   Sun Apr 10 12:14:34 2022 +0530

    done by me

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git status
On branch master
Your branch is behind 'githubrepo/master' by 1 commit, and can be fast-forwarded
.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

CHITKARA
UNIVERSITY

Subject Name: Source Code Management

Subject Code: CS181

Cluster: Beta

Department: DCSE

Submitted By:

Name: Gunand Sharma

Roll no: 2110990527

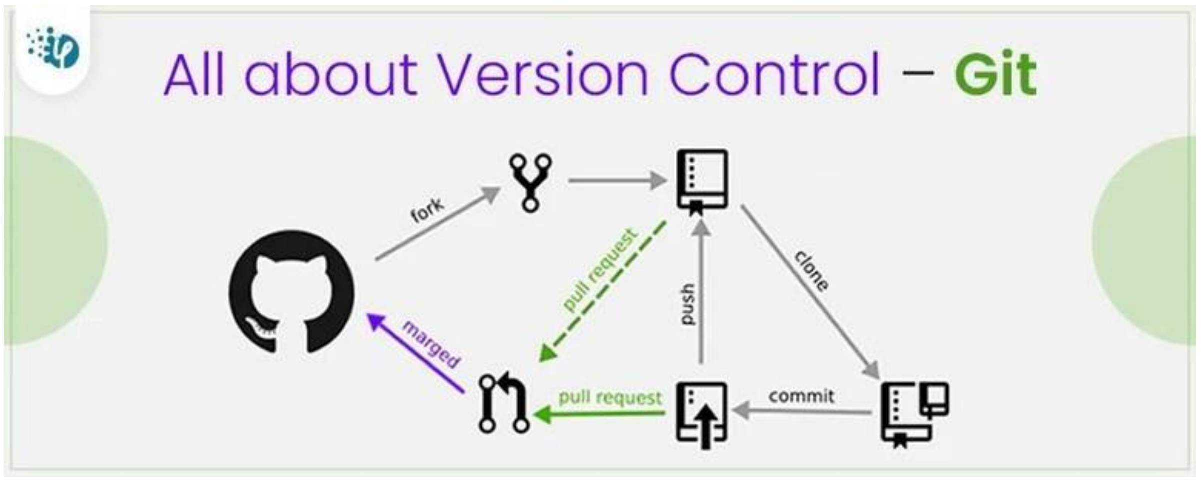
Submitted To:

Dr. Monit Kapoor

Index

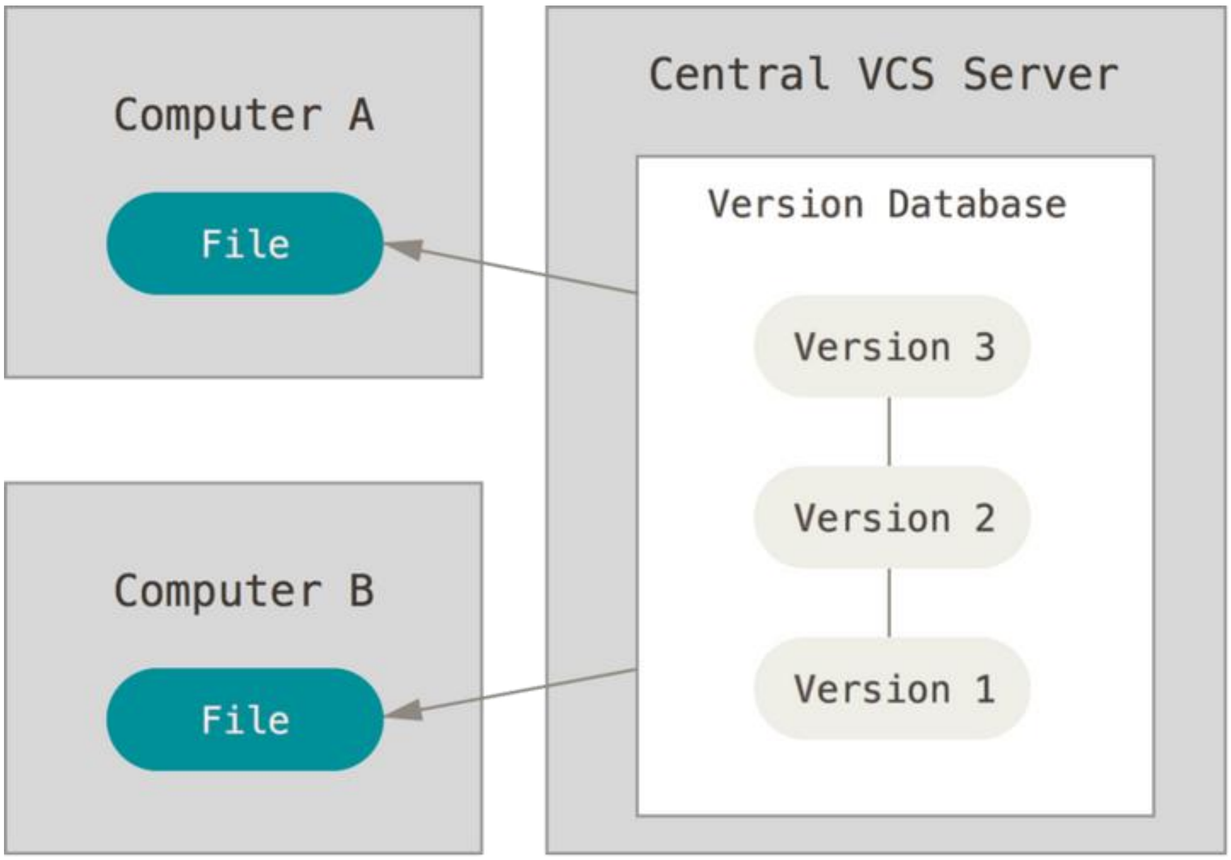
1. Version control with Git

What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book, you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer. If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.



Local Version Control Systems

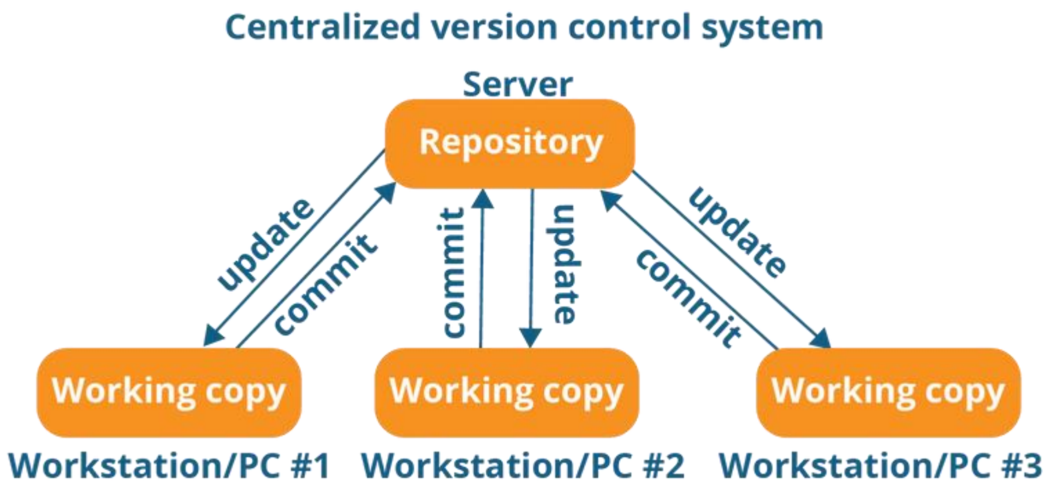
Many people’s version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they’re clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you’re in and accidentally write to the wrong file or copy over files you don’t mean to. To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.



One of the most popular VCS tools was a system called RCS, which is still distributed with many computers today. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.



This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local databases on every client. However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on. If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCSs suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything.

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out

the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

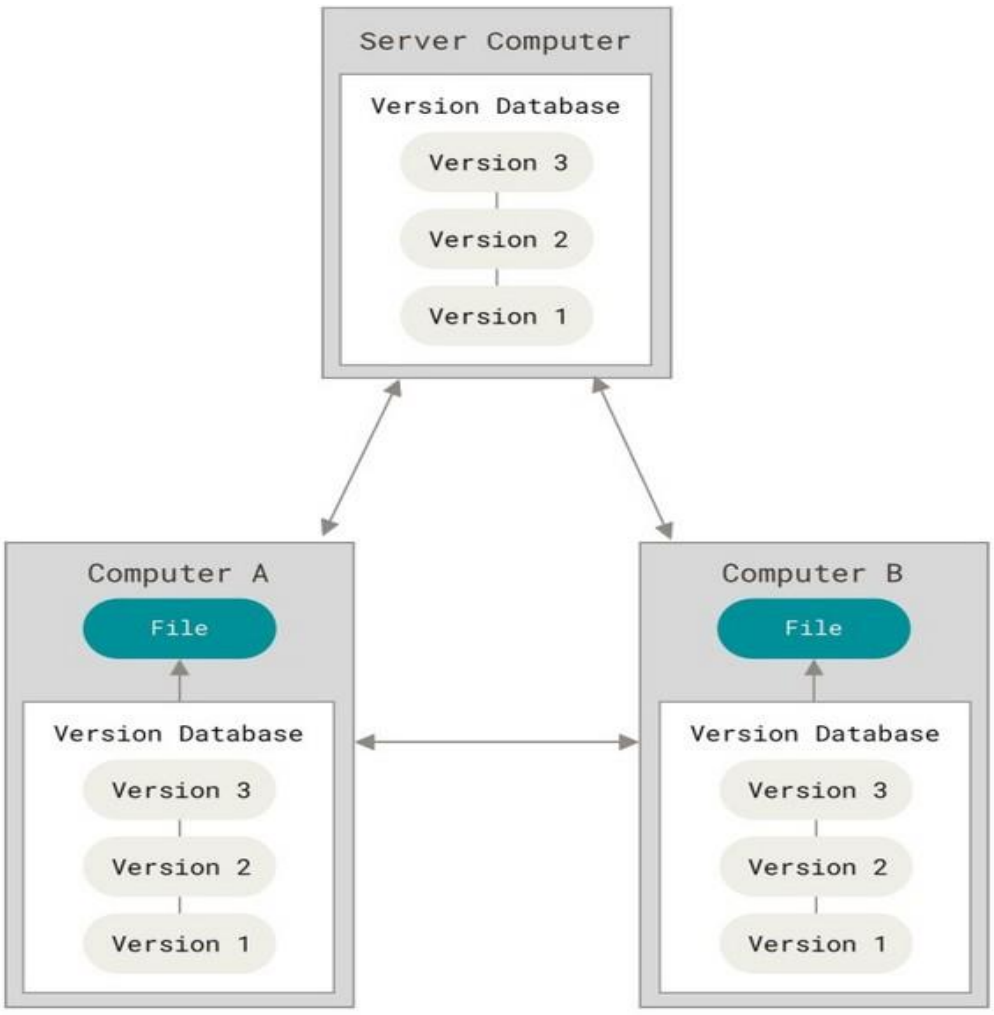


Figure 3. Distributed version control

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

2. Problem Statement

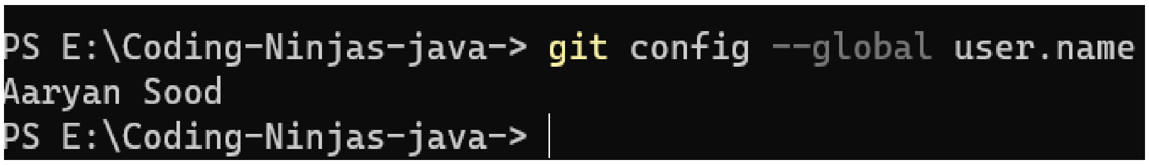
In the project The AWD clone, we created a clone of a non-profit website called help-u using HTML, CSS, and JavaScript. Many causes are listed under the ngos section of this website. The website's eye-catching design aids in attracting visitors. It is a day-to-day application that is useful in a person's daily life.
This website was created to help us gain confidence in coding and to better comprehend the fundamentals of HTML, CSS, and JavaScript.

3. Objective

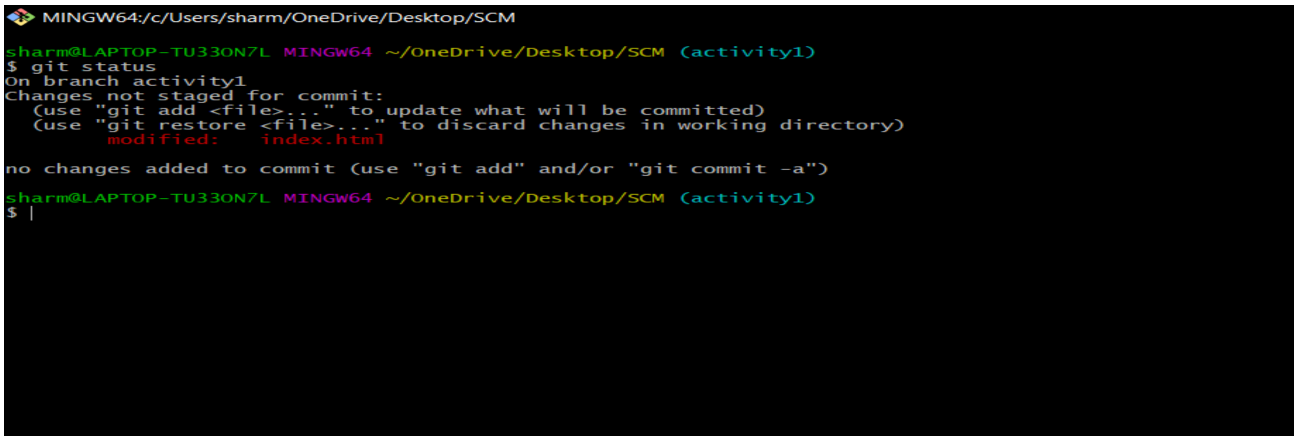
We learned the fundamentals of HTML and JavaScript through our project, a clone of an NGO website called 'help-u.in.' In JavaScript, we learned Developer Skills. We made a more authentic working for the project webpage with the help of DOM & Event Fundamentals. We also looked at how to manipulate CSS styles as well as how to handle click events. Dry had a fantastic time working with Class Object. The program's execution assisted us in identifying a common blunder made during such initiatives.The main goal of the website is to help generate the idea for a website that functions as a platform for listing verified issues from various non-governmental organisations so that those in need can receive assistance as soon as feasible.

4. Concepts and Commands

git config user.name
To verify linked mail
git config --global user.name
To link repo with GitHub username



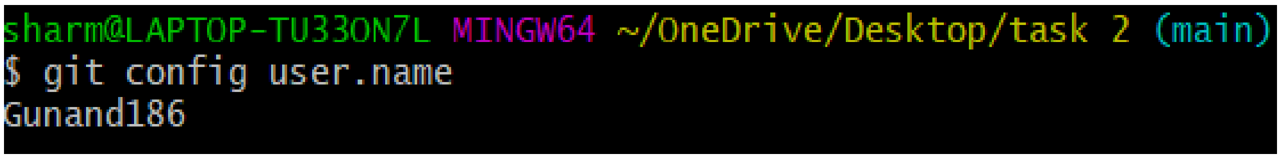
```
PS E:\Coding-Ninjas-java-> git config --global user.name
Aaryan Sood
PS E:\Coding-Ninjas-java->
```



```
MINGW64:/c/Users/sharm/OneDrive/Desktop/SCM

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (activity1)
$ git status
On branch activity1
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:   index.html

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

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (activity1)
$ |
```

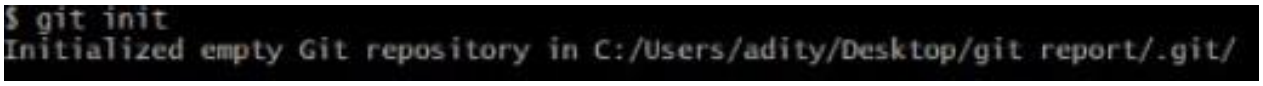
git config --global user.email To link repo with GitHub mail

git config user.email
To verify linked username



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/task 2 (main)
$ git config user.name
Gunand186
```

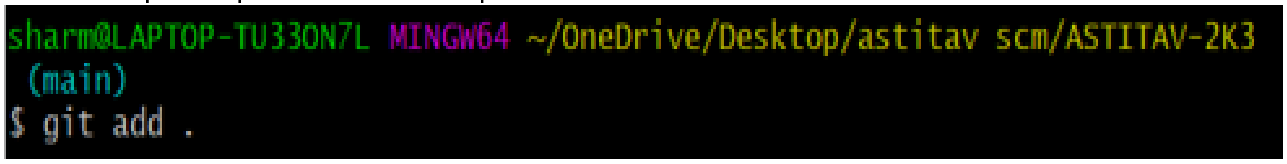
git init
To make folder git ready



```
$ git init
Initialized empty Git repository in C:/Users/adity/Desktop/git report/.git/
```

git add –a
To push all the files to repo

git add filename
To push a particular file to repo



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/astitav scm/ASTITAV-2K3
 (main)
$ git add .
```

git status

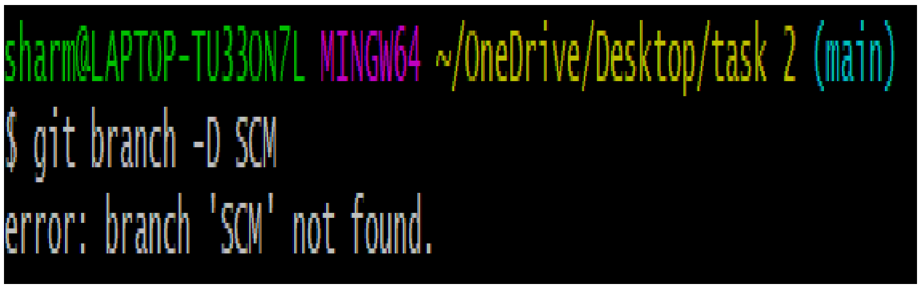
git branch name
To make a new branch

git checkout name
To switch between branch

git branch -d branch name
To delete branch (Soft delete because it ask to merge)

git branch -D branch name
To delete branch (**Hard delete** because it don’t ask to merge)

git branch
To see number of branches



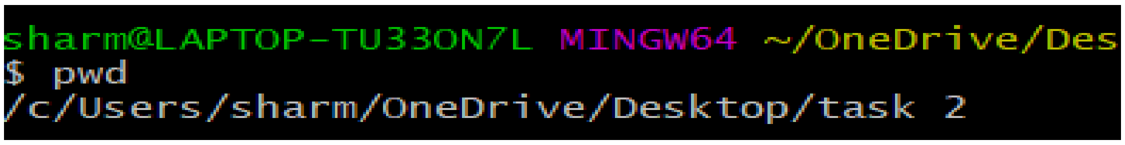
```
sharm@LAPTOP-TU330N7L MINGW64 ~/OneDrive/Desktop/task 2 (main)
$ git branch -D SCM
error: branch 'SCM' not found.
```

git branch -m new branch name
To rename a branch (we need to be in that branch)

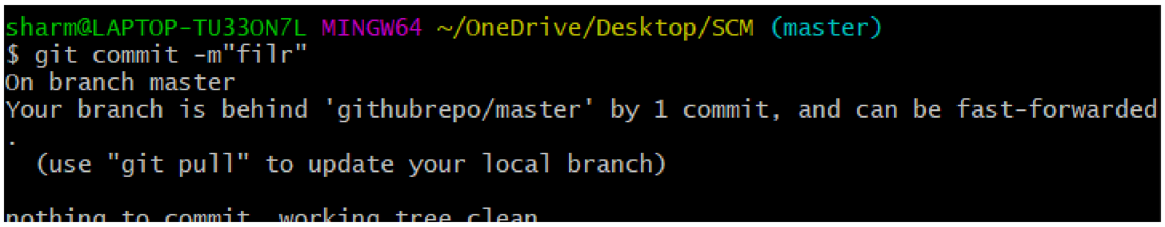
git branch -r
To see number of branches

git log
used to check the history of the work done also contains a checksum

pwd
Present working Directory

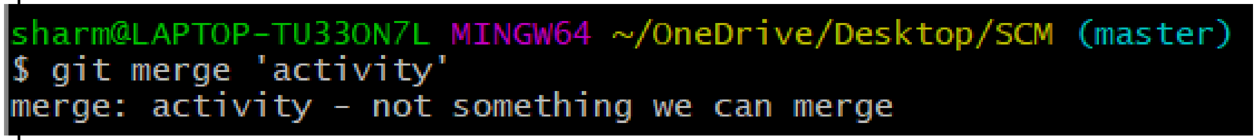


```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Des
$ pwd
/c/Users/sharm/OneDrive/Desktop/task 2
```



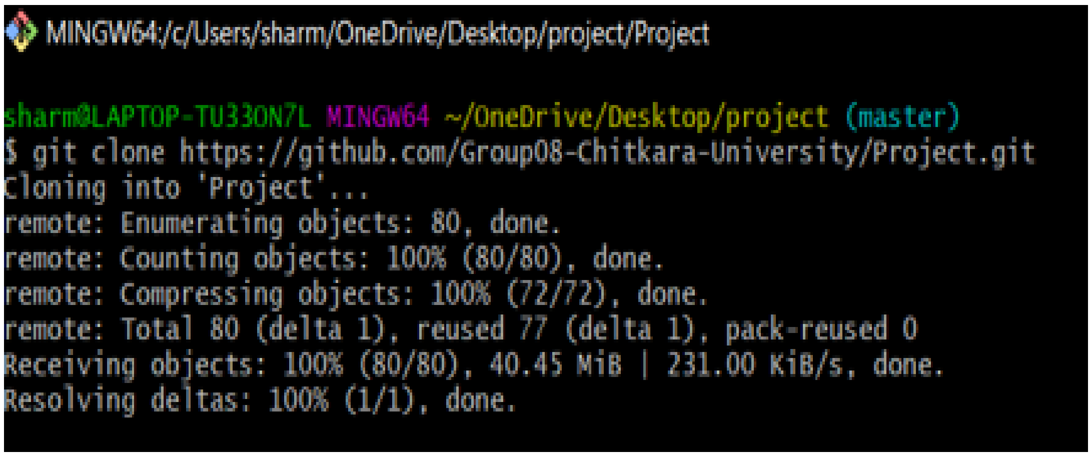
```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git commit -m"filr"
On branch master
Your branch is behind 'githubrepo/master' by 1 commit, and can be fast-forwarded
.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

git clone git hub link
To add repo from GitHub to personal system



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git merge 'activity'
merge: activity - not something we can merge
```

git merge branch name
To merge sub branch with master



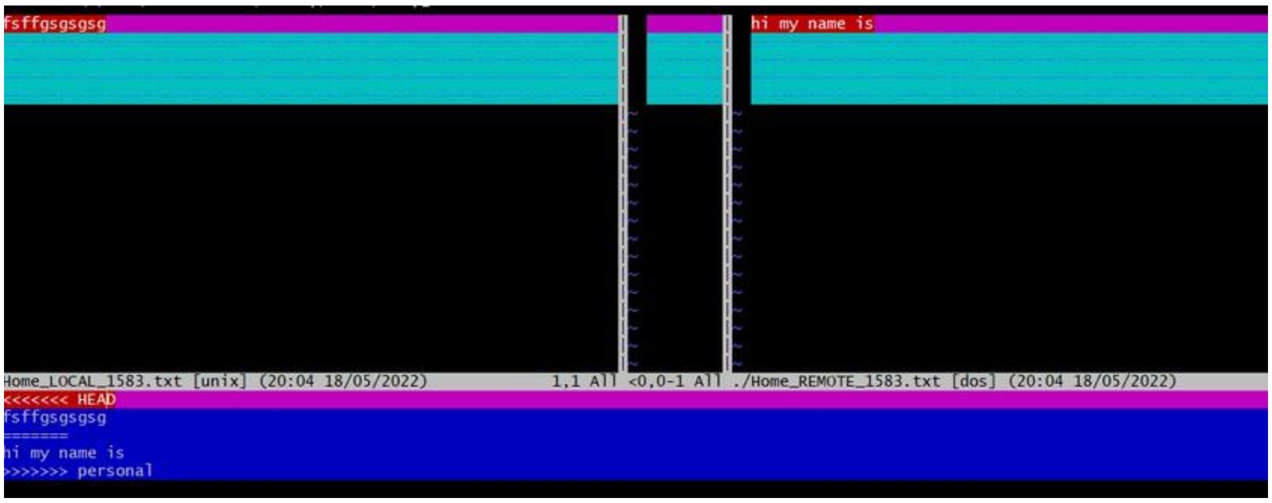
```
MINGW64:/c/Users/sharm/OneDrive/Desktop/project/Project

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/project (master)
$ git clone https://github.com/Group08-Chitkara-University/Project.git
Cloning into 'Project'...
remote: Enumerating objects: 80, done.
remote: Counting objects: 100% (80/80), done.
remote: Compressing objects: 100% (72/72), done.
remote: Total 80 (delta 1), reused 77 (delta 1), pack-reused 0
Receiving objects: 100% (80/80), 40.45 MiB | 231.00 KiB/s, done.
Resolving deltas: 100% (1/1), done.
```

mv old-file-name new-file-name
To rename a flolder (we have to do staging for this)
git mv old-file-name new-file-name
To rename a folder (no need for staging)

git restore --staged filed name
To reverse to to previous version
git restore filename
To go to previous command

git mergetool

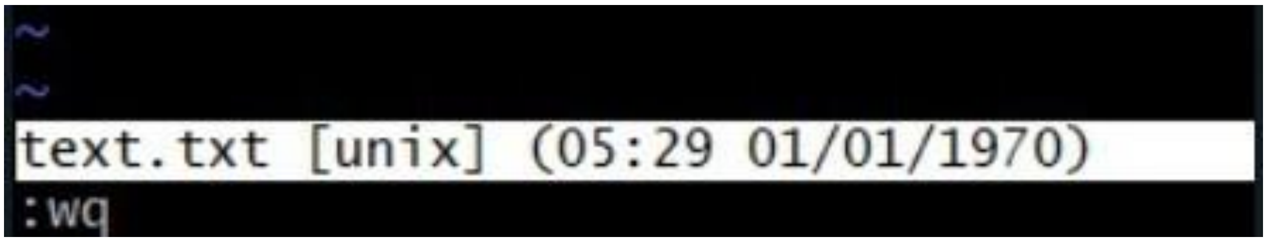
To remove merge conflict i.e. when content in master and branch is different.



```
fsffgsgsgsg                                                      hi my name is

Home_LOCAL_1583.txt [unix] (20:04 18/05/2022)        1,1 All <0,0-1 All ./Home_REMOTE_1583.txt [dos] (20:04 18/05/2022)
<<<<<<< HEAD
fsffgsgsgsg
=======
hi my name is
>>>>>>> personal
```

:wq

To quit from special screen



```
~
~
text.txt [unix] (05:29 01/01/1970)
:wq
```

rm -rf .git

To delete whole git folder

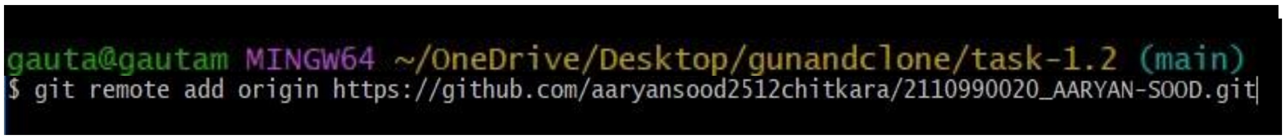
rm -rf filename

To delete a particular file

git remote add origin " link-of-repo-we-made-on-github

To make new remote

git push -u master **remote-nam** To make data visible on sscloud



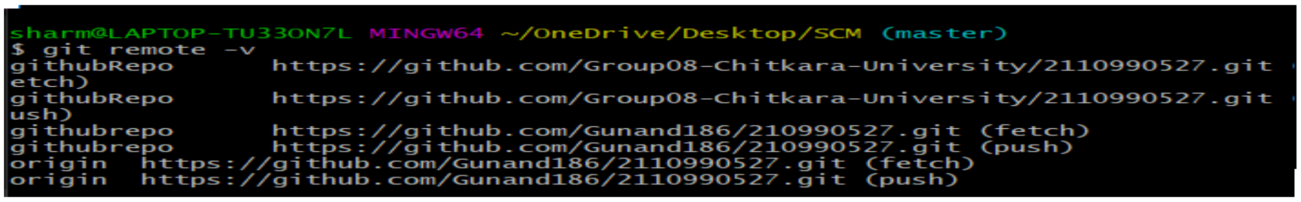
```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ git remote add origin https://github.com/aaryansood2512chitkara/2110990020_AARYAN-SOOD.git
```

git pull **link-of-repo-on-github**

To make changes done on cloud visible on the system.

git remote -v

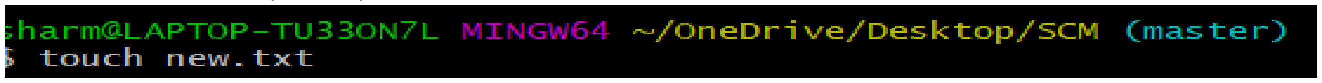
to see the location where remote is being stored



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git remote -v
githubRepo      https://github.com/Group08-Chitkara-University/2110990527.git
etch)
githubRepo      https://github.com/Group08-Chitkara-University/2110990527.git
ush)
githubrepo      https://github.com/Gunand186/210990527.git (fetch)
githubrepo      https://github.com/Gunand186/210990527.git (push)
origin  https://github.com/Gunand186/2110990527.git (fetch)
origin  https://github.com/Gunand186/2110990527.git (push)
```

touch **filename**

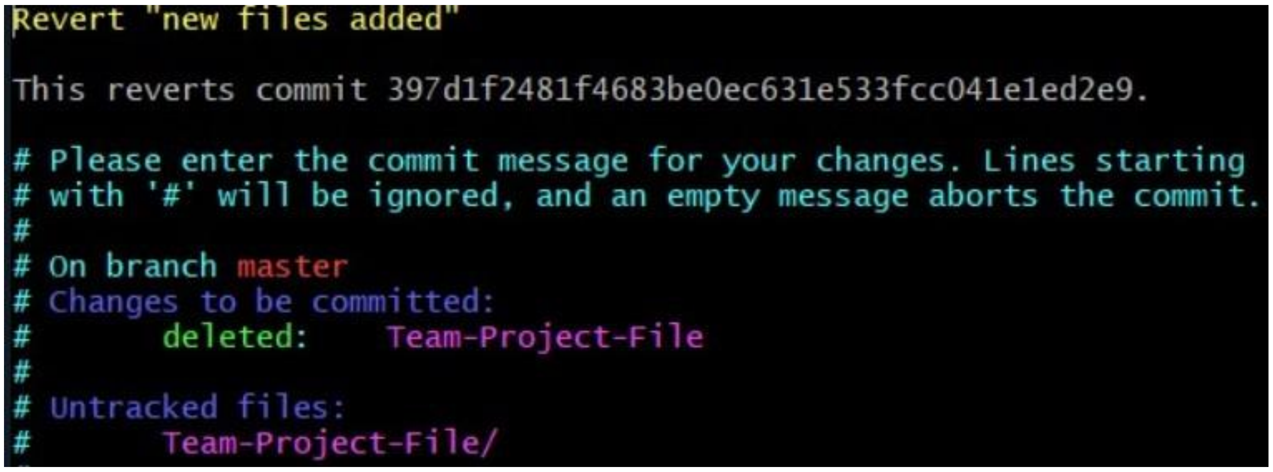
To make css/html/c++ files



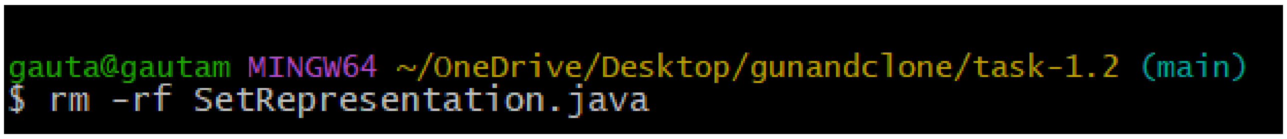
```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ touch new.txt
```

git revert **checksum**

It’s a forwar moving undo operation that offers a safe mode of undoing changes



```
Revert "new files added"

This reverts commit 397d1f2481f4683be0ec631e533fcc041e1ed2e9.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#       deleted:    Team-Project-File
#
# Untracked files:
#       Team-Project-File/
```



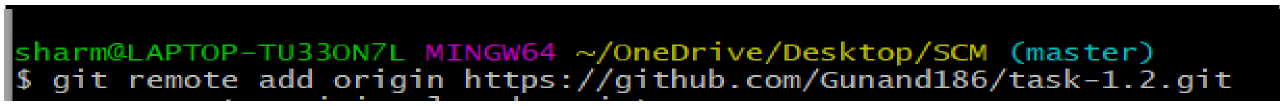
```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ rm -rf SetRepresentation.java
```

rm -rf .git

To delete whole git folder

rm -rf **filename**

To delete a particular file



```
sharm@LAPTOP-TU330N7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git remote add origin https://github.com/Gunand186/task-1.2.git
```

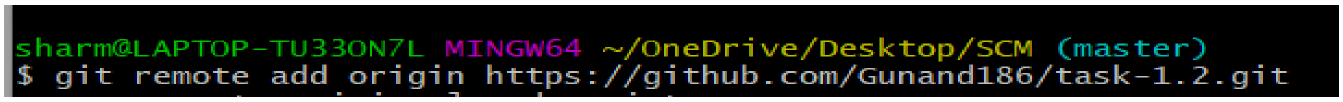
git remote add **origin** **“ link-of-repo-we-made-on-**

github

To make new remote

git push -u master **remote-nam**

To make data visible on sscloud



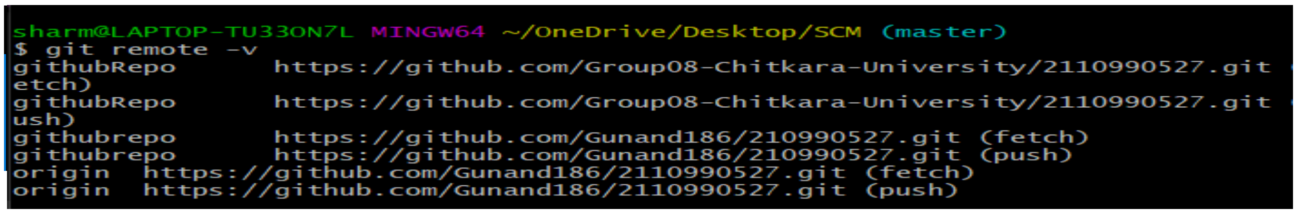
```
sharm@LAPTOP-TU330N7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git remote add origin https://github.com/Gunand186/task-1.2.git
```

git pull **link-of-repo-on-github**

To make changes done on cloud visible on the system.

git remote -v

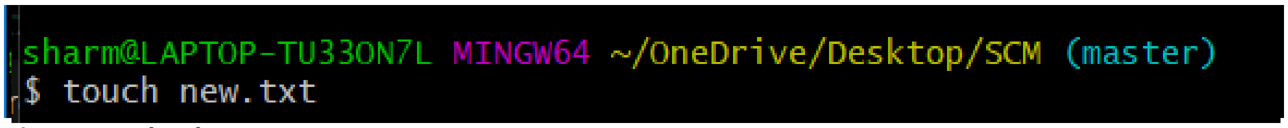
to see the location where remote is being stored



```
sharm@LAPTOP-TU330N7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git remote -v
githubRepo      https://github.com/Group08-Chitkara-University/2110990527.git
etch)
githubRepo      https://github.com/Group08-Chitkara-University/2110990527.git
ush)
githubrepo      https://github.com/Gunand186/210990527.git (fetch)
githubrepo      https://github.com/Gunand186/210990527.git (push)
origin  https://github.com/Gunand186/2110990527.git (fetch)
origin  https://github.com/Gunand186/2110990527.git (push)
```

touch **filename**

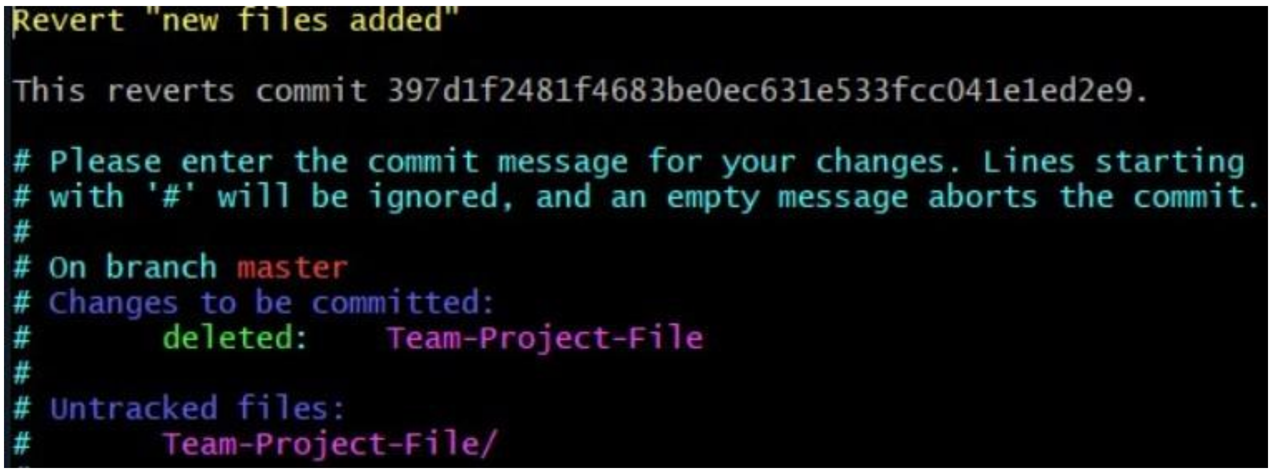
To make css/html/c++ files



```
sharm@LAPTOP-TU330N7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ touch new.txt
```

git revert **checksum**

It’s a forwar moving undo operation that offers a safe mode of undoing changes

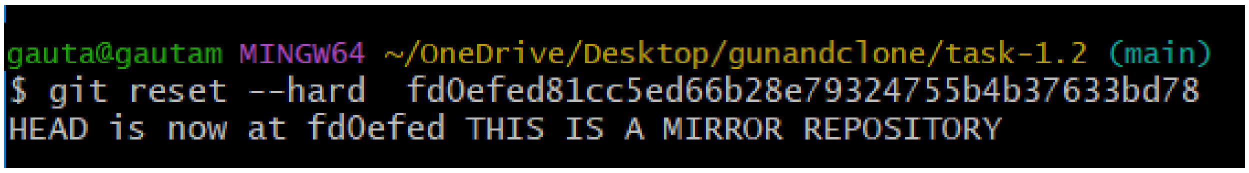


```
Revert "new files added"

This reverts commit 397d1f2481f4683be0ec631e533fcc041e1ed2e9.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#       deleted:    Team-Project-File
#
# Untracked files:
#       Team-Project-File/
```

touch .gitignore

git reset --hard checksum

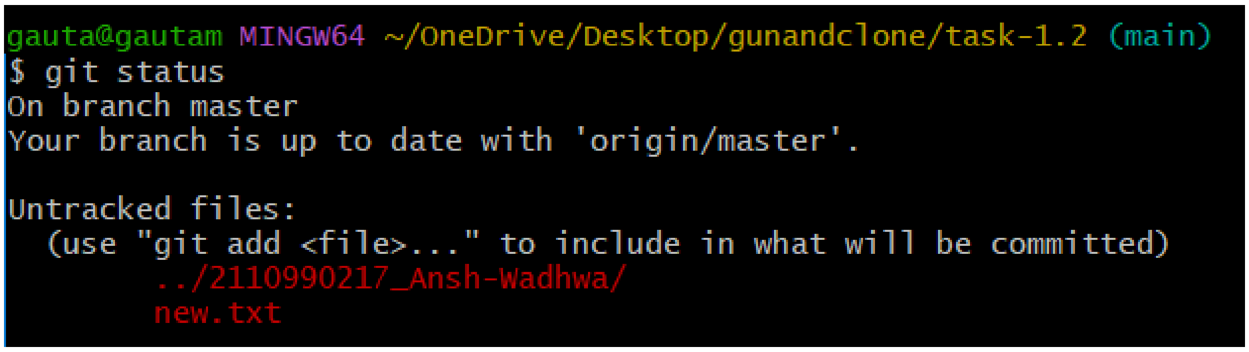
All the commits which was rested is deleted in the working directory along with the commit history.



```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ git reset --hard  fd0efed81cc5ed66b28e79324755b4b37633bd78
HEAD is now at fd0efed THIS IS A MIRROR REPOSITORY
```

git reset --mixed checksum

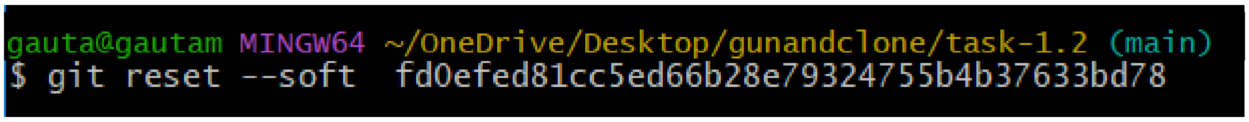
Reset commit files doesn't get deleted it goes untracked changes (red colour on git status)



```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        ../2110990217_Ansh-Wadhwa/
        new.txt
```

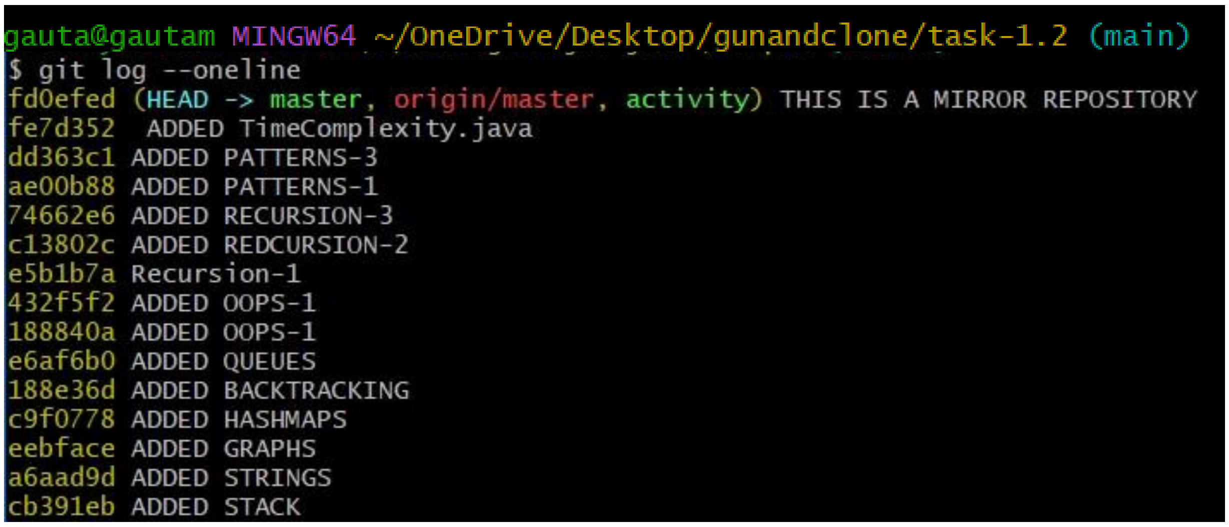
git reset --soft checksum

Resseted commit files doesn't get deleted it goes to staging area (green colour)

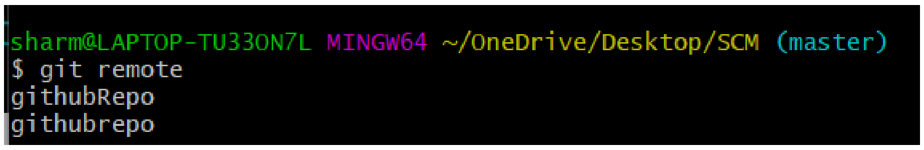


```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ git reset --soft  fd0efed81cc5ed66b28e79324755b4b37633bd78
```

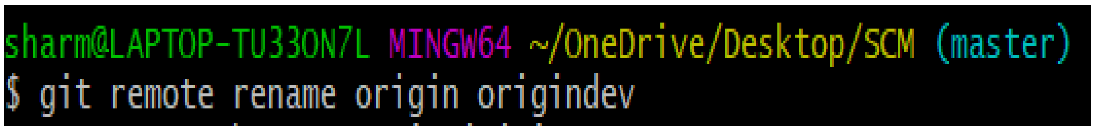
git reset -- any-type ~ n(no. of commits to be changed) To delete no. of commits


```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ git log --oneline
fd0efed (HEAD -> master, origin/master, activity) THIS IS A MIRROR REPOSITORY
fe7d352  ADDED TimeComplexity.java
dd363c1 ADDED PATTERNS-3
ae00b88 ADDED PATTERNS-1
74662e6 ADDED RECURSION-3
c13802c ADDED REDCURSION-2
e5b1b7a Recursion-1
432f5f2 ADDED OOPS-1
188840a ADDED OOPS-1
e6af6b0 ADDED QUEUES
188e36d ADDED BACKTRACKING
c9f0778 ADDED HASHMAPS
eebface ADDED GRAPHS
a6aad9d ADDED STRINGS
cb391eb ADDED STACK
```

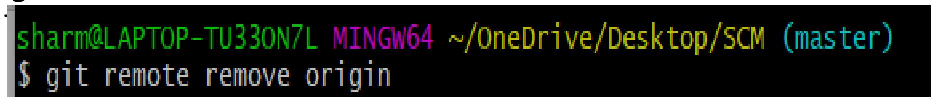
git remote
To see the name of origin formed


```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git remote
githubRepo
githubrepo
```

git remote rename old-file-name new-file-name To rename remote

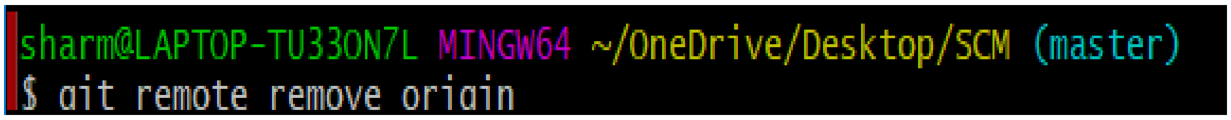

```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git remote rename origin origindev
```

git remote remove name


```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git remote remove origin
```

cat file-name

To see the data stored in file in git bash without opening the file

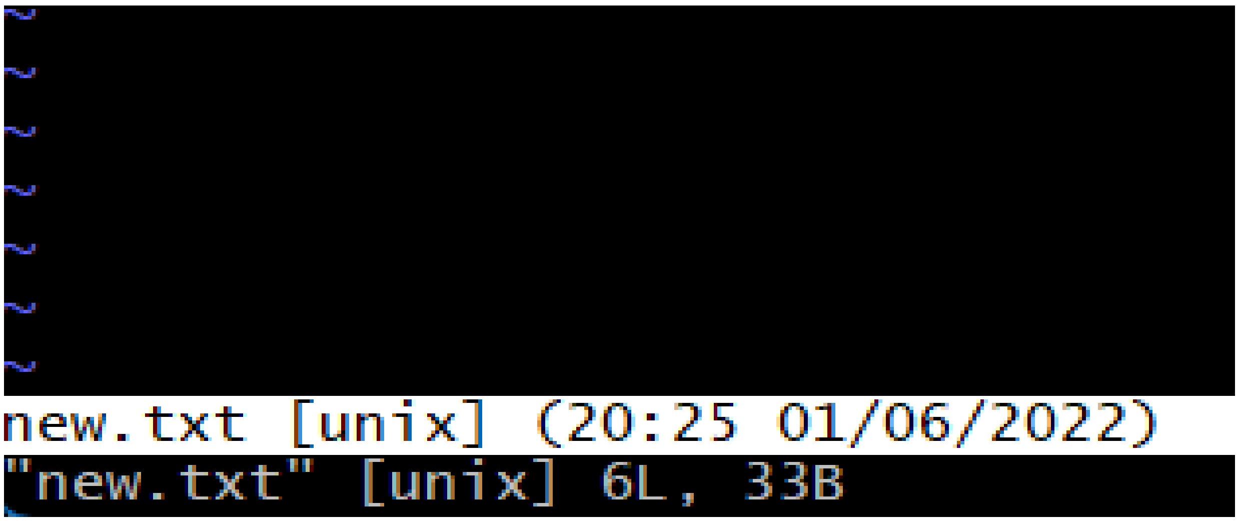

```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git remote remove origin
```

vi file-name

To edit content of file in git bash without opening the file

Cs181

5. Workflow and discussion



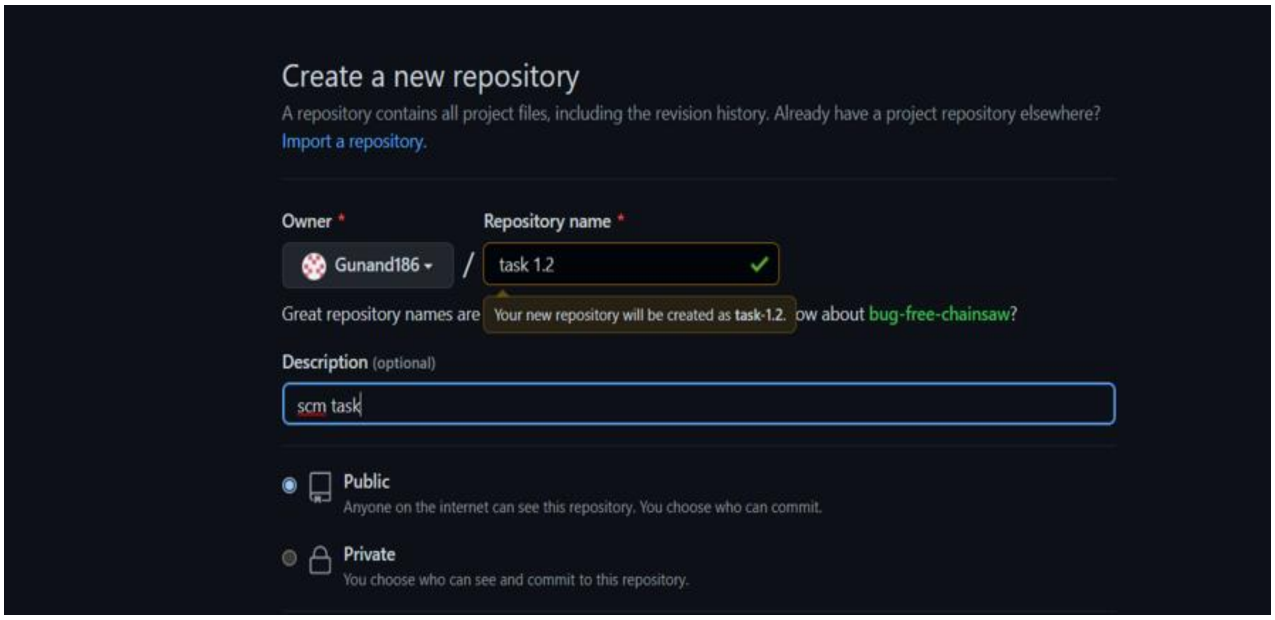
```
~
~
~
~
~
~
~
new.txt [unix] (20:25 01/06/2022)
"new.txt" [unix] 6L, 33B
```

Open source szoftware is based on the idea that by sharing code, we can make better, more reliable software. For more information, see the "About the Open Source Initiative" on the Open Source Initiative. For more information about applying open source principles to your organization's development work on GitHub.com, see GitHub's white paper "An introduction to innersource."

When creating your public repository from a fork of someone's project, make sure to include a license file that determines how you want your project to be shared with others. For more information, see "Choose an open source license" at choosealicense.com.

For more information on open source, specifically how to create and grow an open source project, we've created Open Source Guides that will help you foster a healthy open source community by recommending best practices for creating and maintaining repositories for your open source project. You can also take a free GitHub Learning Lab course on maintaining open source communities.

I made a new repository



Create a new repository

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

Owner * Gunand186

Repository name * task 1.2

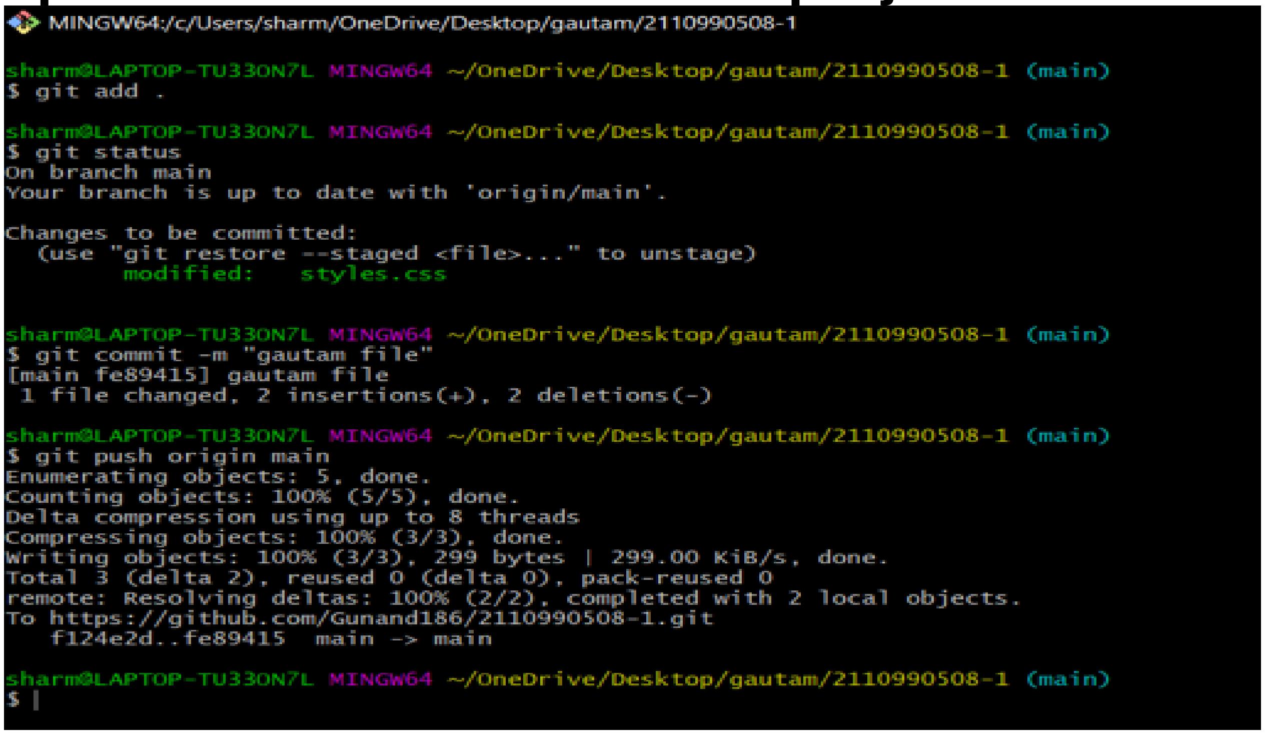
Great repository names are Your new repository will be created as task-1.2. ow about bug-free-chainsaw?

Description (optional) scm task

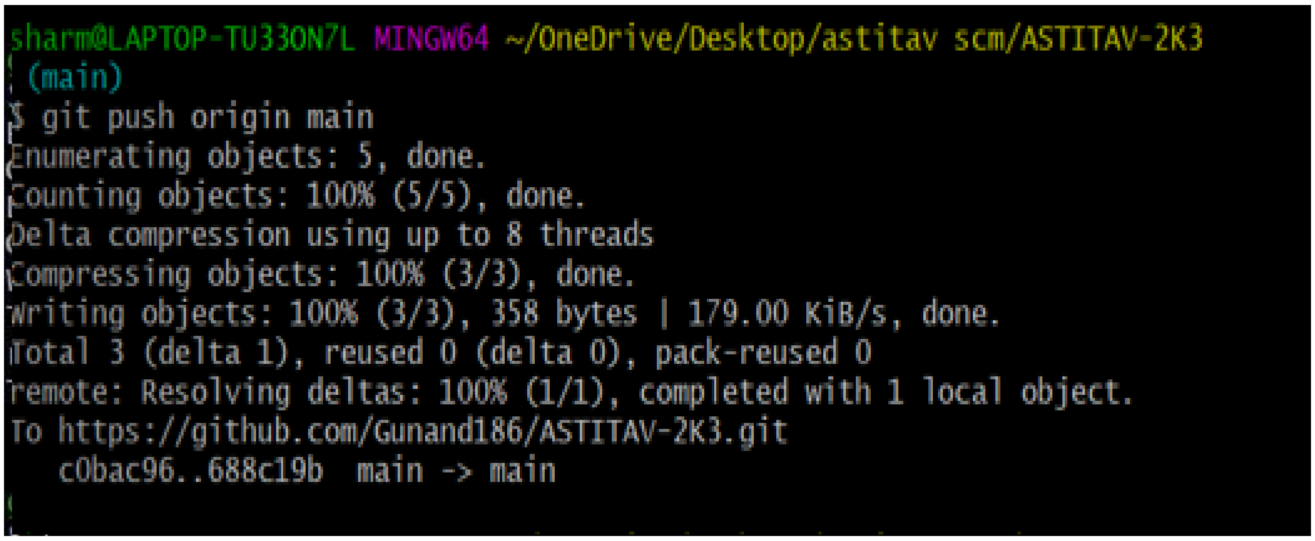
Public [✓]
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

I pushed some files into the project



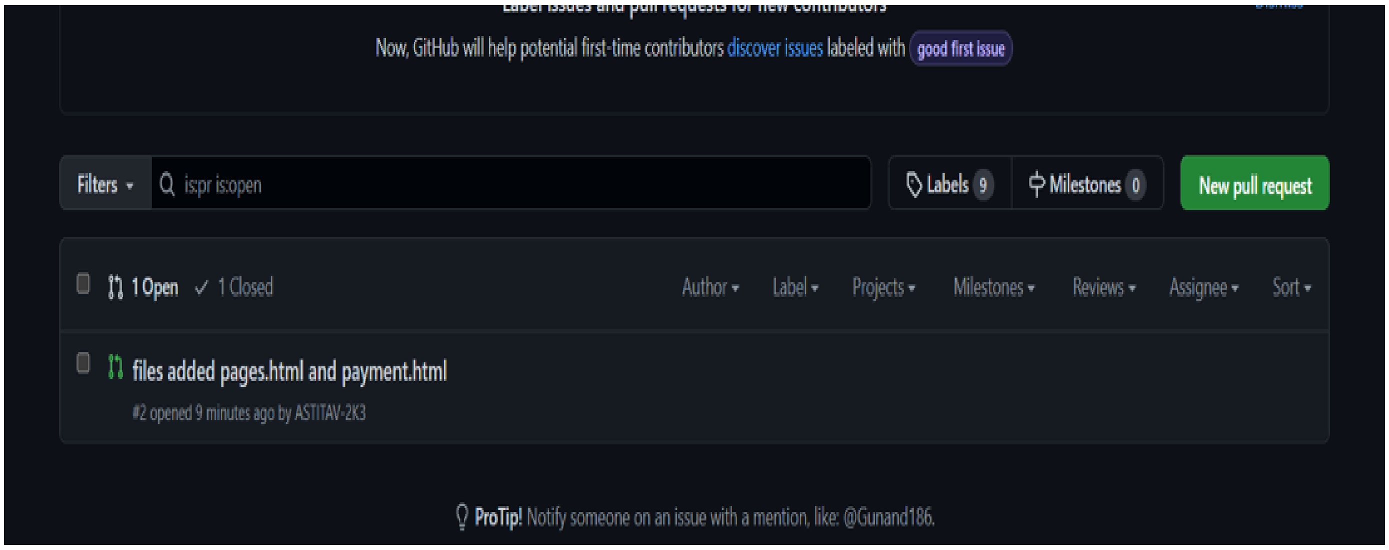
```
MINGW64:/c/Users/sharm/OneDrive/Desktop/gautam/2110990508-1

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/gautam/2110990508-1 (main)
$ git add .

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/gautam/2110990508-1 (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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


sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/gautam/2110990508-1 (main)
$ git commit -m "gautam file"
[main fe89415] gautam file
 1 file changed, 2 insertions(+), 2 deletions(-)

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/gautam/2110990508-1 (main)
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 299 bytes | 299.00 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/Gunand186/2110990508-1.git
   f124e2d..fe89415  main -> main

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/gautam/2110990508-1 (main)
$ |
```



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/astitav scm/ASTITAV-2K3
(main)
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 358 bytes | 179.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/Gunand186/ASTITAV-2K3.git
   c0bac96..688c19b  main -> main
```

All the collaborators made some changes to the files or added new files:

Gautam made some changes and sent the pull request and I merged it:



Now, GitHub will help potential first-time contributors discover issues labeled with good first issue

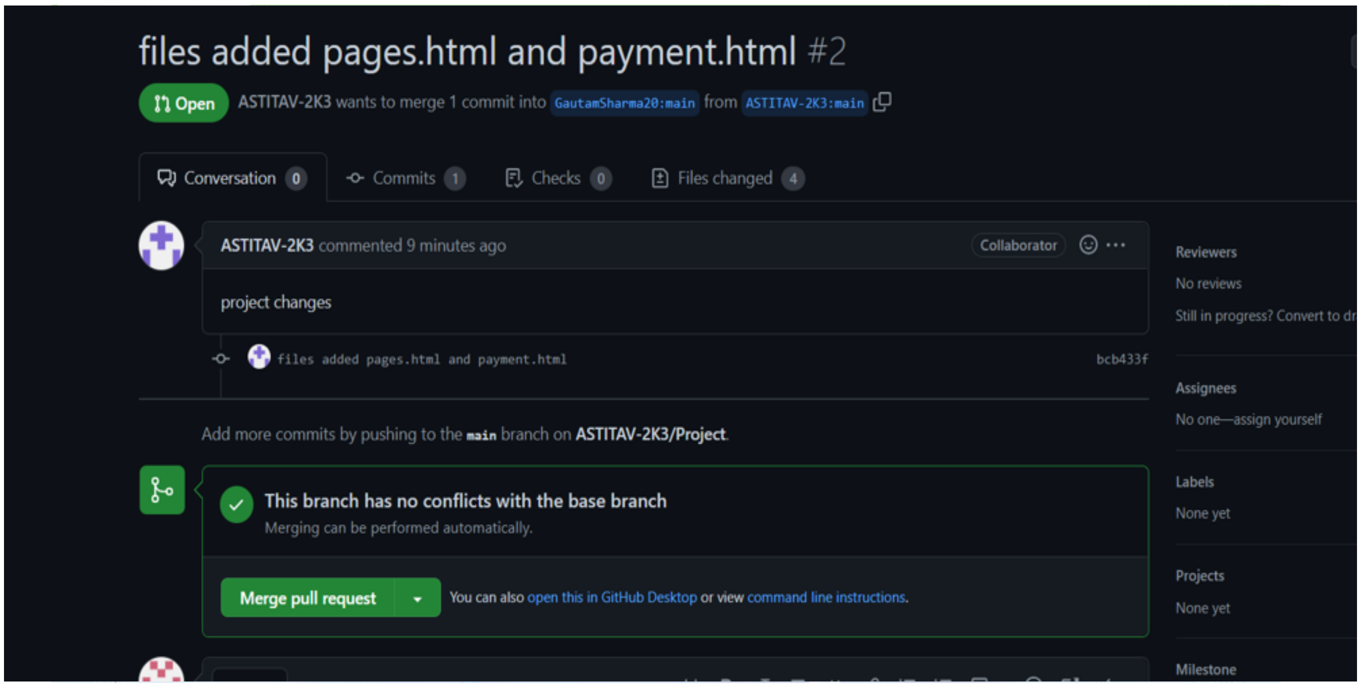
Filters ▾ is:pr is:open

Labels 9 Milestones 0 New pull request

1 Open ✓ 1 Closed Author ▾ Label ▾ Projects ▾ Milestones ▾ Reviews ▾ Assignee ▾ Sort ▾

files added pages.html and payment.html
#2 opened 9 minutes ago by ASTITAV-2K3

ProTip! Notify someone on an issue with a mention, like: @Gunand186.


files added pages.html and payment.html #2

Open ASTITAV-2K3 wants to merge 1 commit into `GautamSharma20:main` from `ASTITAV-2K3:main`

Conversation 0 · Commits 1 · Checks 0 · Files changed 4

ASTITAV-2K3 commented 9 minutes ago — Collaborator

project changes

`files added pages.html and payment.html` bcb433f

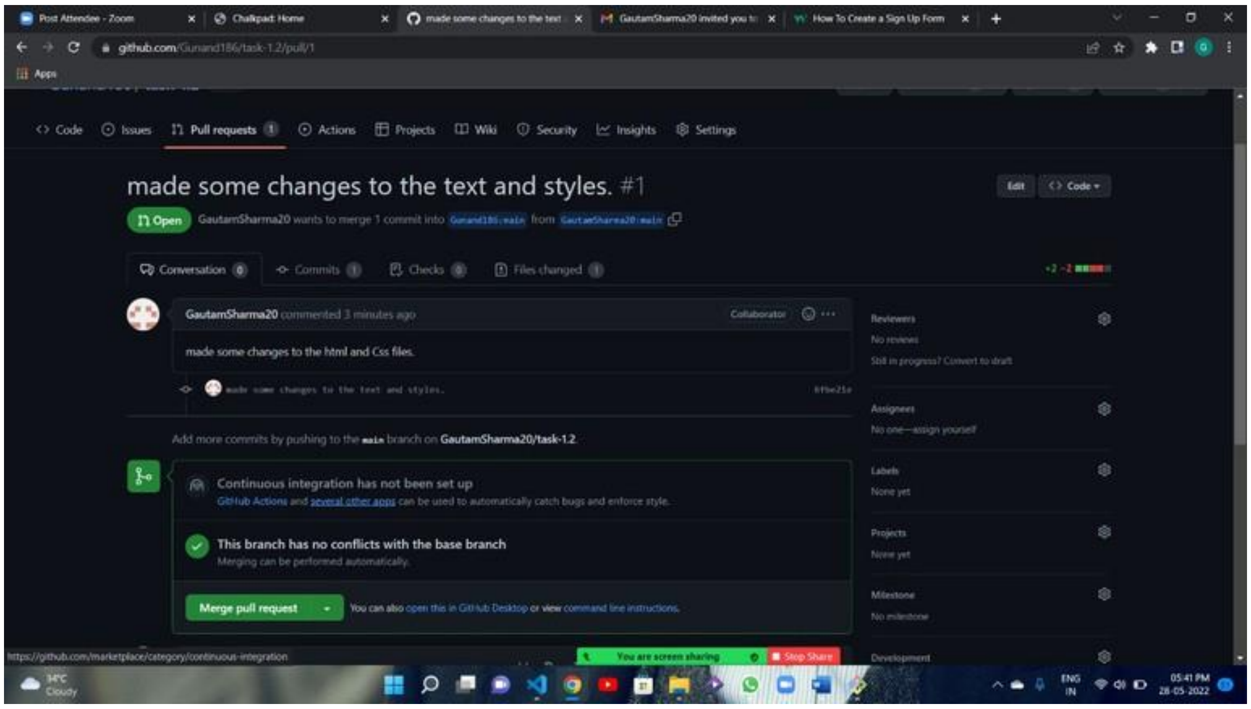
Add more commits by pushing to the main branch on ASTITAV-2K3/Project.

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request — You can also open this in GitHub Desktop or view command line instructions.

Reviewers: No reviews. Still in progress? Convert to dr
Assignees: No one—assign yourself
Labels: None yet
Projects: None yet
Milestone

I also made some changes and send the pull request to the collaborators and they merged it.


github.com/Gunand186/task-1.2/pull/1

made some changes to the text and styles. #1

Open GautamSharma20 wants to merge 1 commit into `Gunand186:main` from `GautamSharma20:main`

Conversation 0 · Commits 1 · Checks 0 · Files changed 1

GautamSharma20 commented 3 minutes ago — Collaborator

made some changes to the html and Css files.

`made some changes to the text and styles.` 6fbe21e

Add more commits by pushing to the main branch on GautamSharma20/task-1.2.

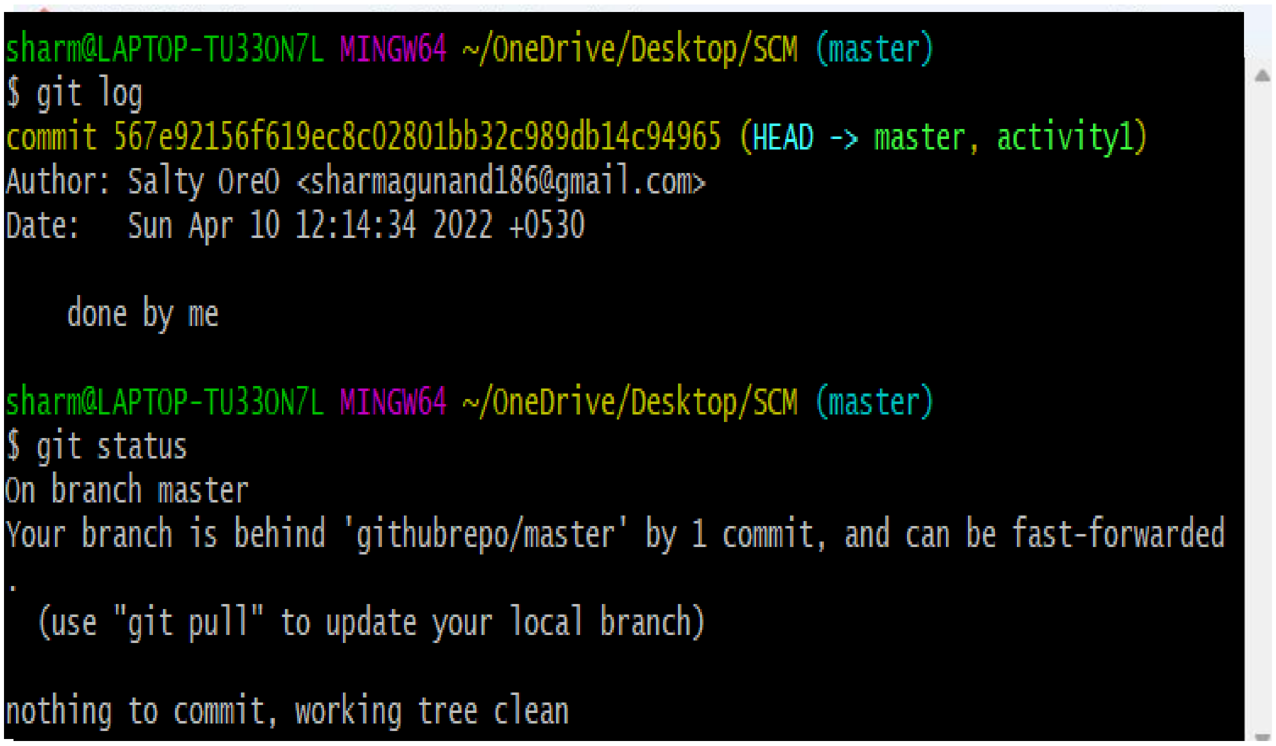
Continuous integration has not been set up
GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request — You can also open this in GitHub Desktop or view command line instructions.

Reviewers: No reviews. Still in progress? Convert to draft
Assignees: No one—assign yourself
Labels: None yet
Projects: None yet
Milestone: No milestone
Development

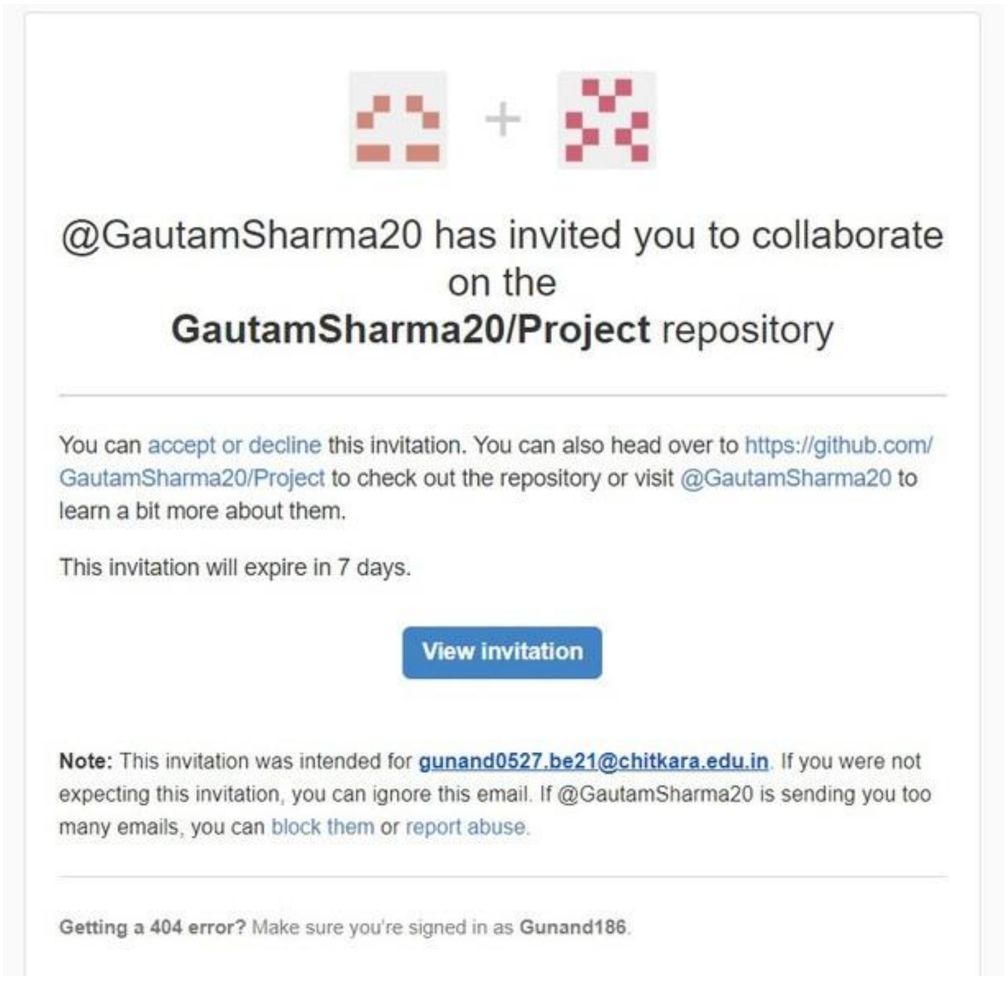
git log after forking and collaborating



```
sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git log
commit 567e92156f619ec8c02801bb32c989db14c94965 (HEAD -> master, activity1)
Author: Salty Ore0 <sharmagunand186@gmail.com>
Date:   Sun Apr 10 12:14:34 2022 +0530

    done by me

sharm@LAPTOP-TU33ON7L MINGW64 ~/OneDrive/Desktop/SCM (master)
$ git status
On branch master
Your branch is behind 'githubrepo/master' by 1 commit, and can be fast-forwarded
.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

Collab with -

This is the invitation mail sent to Gautam to add collaboration in the Project repo



@GautamSharma20 has invited you to collaborate on the **GautamSharma20/Project** repository

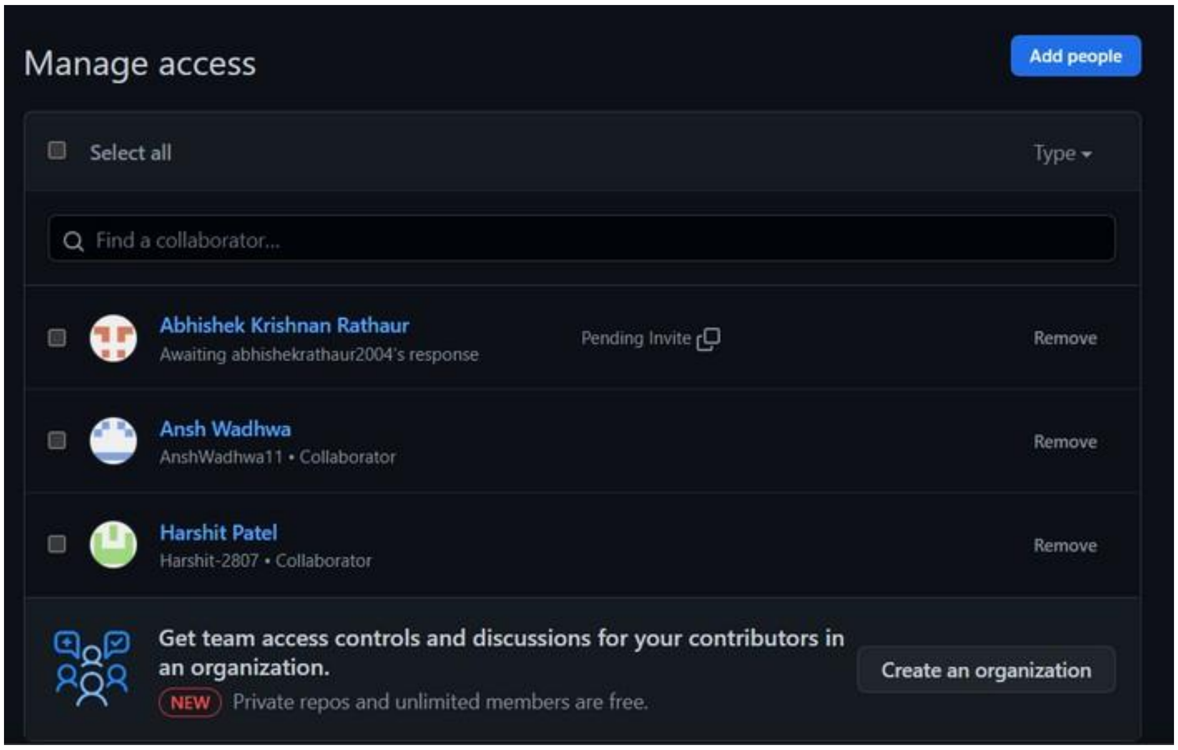
You can accept or decline this invitation. You can also head over to https://github.com/GautamSharma20/Project to check out the repository or visit @GautamSharma20 to learn a bit more about them.

This invitation will expire in 7 days.

Note: This invitation was intended for gunand0527.be21@chitkara.edu.in. If you were not expecting this invitation, you can ignore this email. If @GautamSharma20 is sending you too many emails, you can block them or report abuse.

Getting a 404 error? Make sure you're signed in as **Gunand186**.

Added GautamCollaborater to projectRepo


Manage access

Add people

Select all

Type

Find a collaborator...

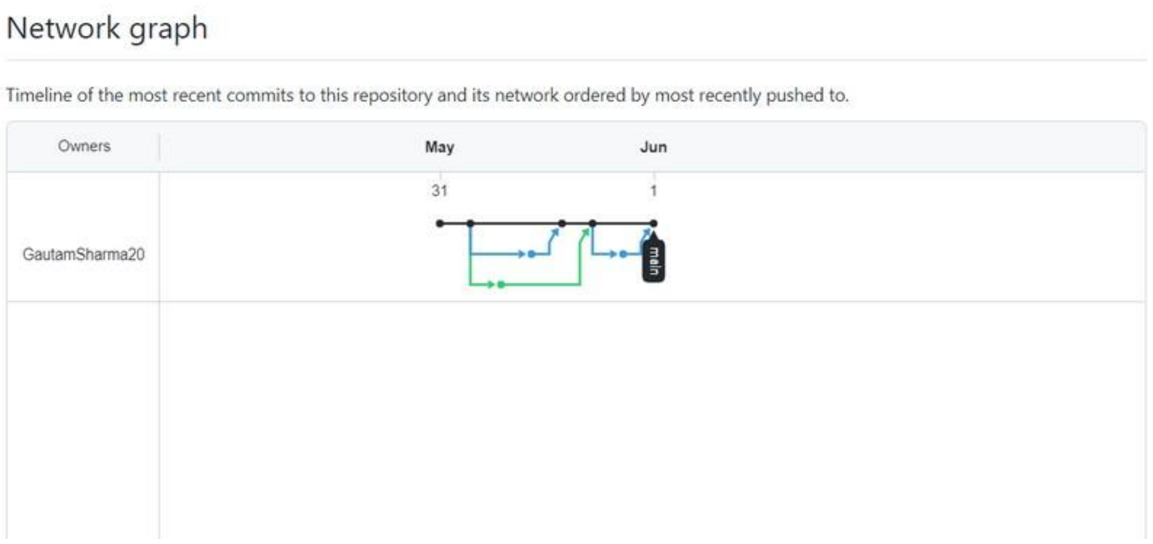
Abhishek Krishnan Rathaur
Awaiting abhishekrathaur2004's response
Pending Invite
Remove

Ansh Wadhwa
AnshWadhwa11 • Collaborator
Remove

Harshit Patel
Harshit-2807 • Collaborator
Remove

Get team access controls and discussions for your contributors in an organization.
NEW Private repos and unlimited members are free.
Create an organization

6. Network Graph


Network graph

Timeline of the most recent commits to this repository and its network ordered by most recently pushed to.

6. Network Graph

Reference for few code snips of CSS and HTML were taken from Coding Ninjas video lectures and from few crash courses available on YouTube.
Reference for few code snippets of JavaScript were taken from the Udemy Courses available online.

Links used for reference

https://www.coursera.org/learn/introduction-git-github

https://www.freecodecamp.org/news/git-and-github-crash-course/

https://www.udemy.com/course/github-ultimate/

https://www.udemy.com/course/git-started-with-github/?

LSNPUBID=JVFxdTr9V80&ranEAID=JVFxdTr9V80&ranMID=39197&ranSiteID=JVFxdTr

9V80-GmotHk.p_rwC77qokoBs_w&utm_medium=udemyads&utm_source=aff-campaign