Subject Name: **Source Code Management**

Subject Code: **CS181**

Cluster: **Zeta**

Department: **DCSE**

CHITKARA UNIVERSITY

Submitted By:
JASJOT SINGH
2110992022
G27

Submitted To:
Dr. ANUJ JAIN

LIST OF TASKS

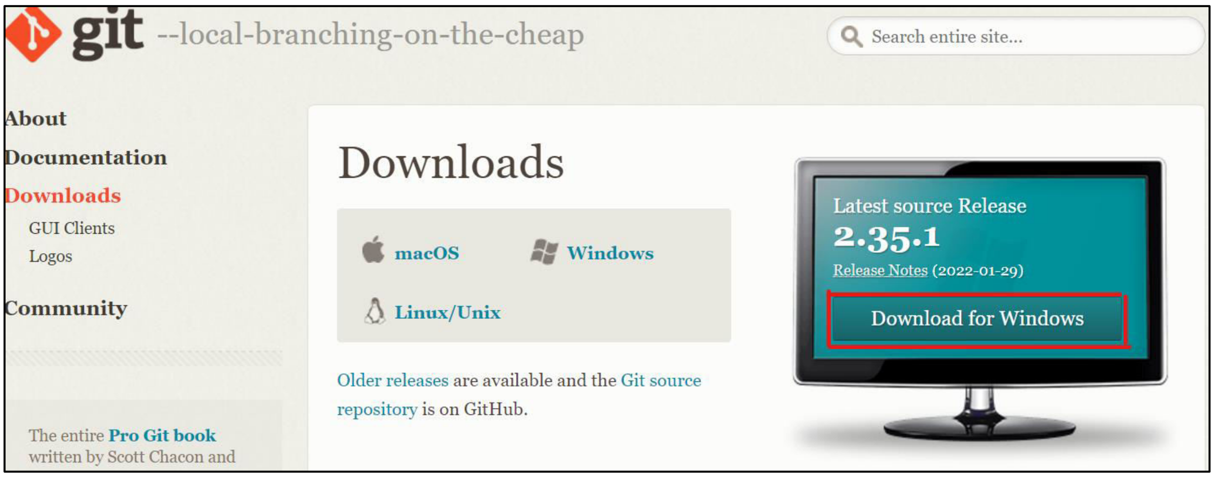
AIM : SETTING UP OF GIT CLIENT.

THEORY:

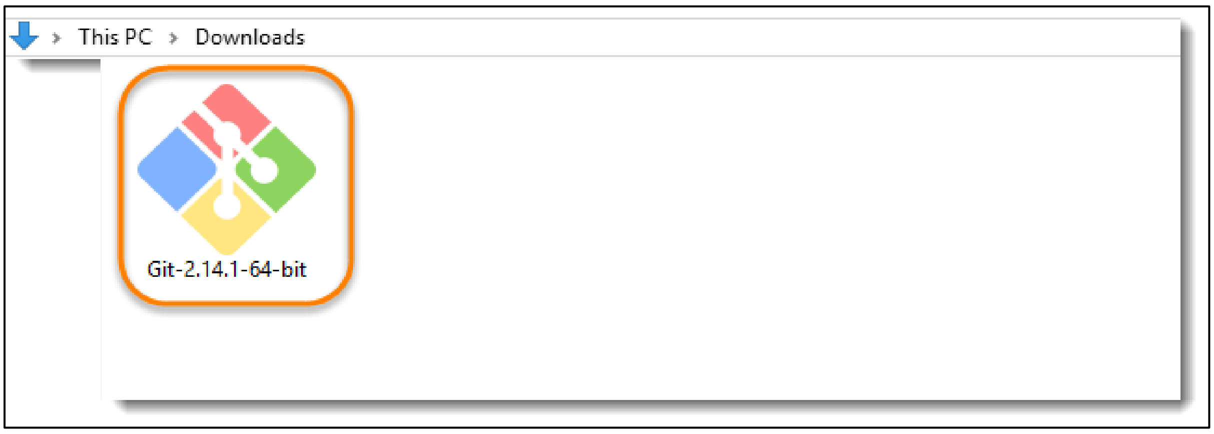
Git is basically used for pushing and pulling of code. We can use git and git-hub parallelly to work with multiple members or individually. We can make , edit , recreate ,copy or download any code on git hub using git. It's a Version Control System which means we are able to track all the previous changes in the code using this software.

PROCEDURE:

STEP1: We can install Git on Windows, using the most official build which is available for download on the GIT's official website or by just typing (scmgit) on any search engine.

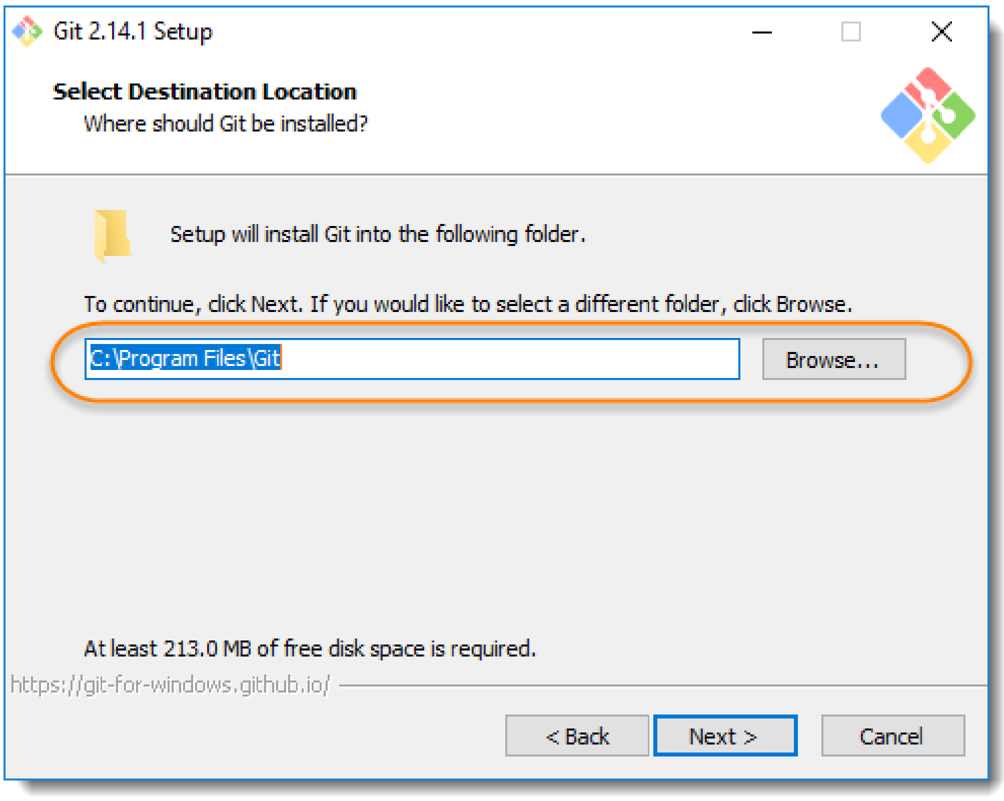

git --local-branching-on-the-cheap
Search entire site...
About
Documentation
Downloads
GUI Clients
Logos
Community
The entire Pro Git book written by Scott Chacon and
Downloads
macOS
Windows
Linux/Unix
Older releases are available and the Git source repository is on GitHub.
Latest source Release
2.35.1
Release Notes (2022-01-29)
Download for Windows

STEP2: Click on download for windows. Go to the folder where new downloads get store. Double click on the installer. The installer gets save on the machine as per the Windows OS configuration.



STEP 3: When you've successfully started the installer, you should see the Git Setup wizard screen. Follow the Next and Finish prompts to complete the installation.

STEP 4: You may like to keep the installation to another folder, so you can set the path of destination location using browse option or either let it be as shown and continue.


Git 2.14.1 Setup

Select Destination Location

Where should Git be installed?

Setup will install Git into the following folder.

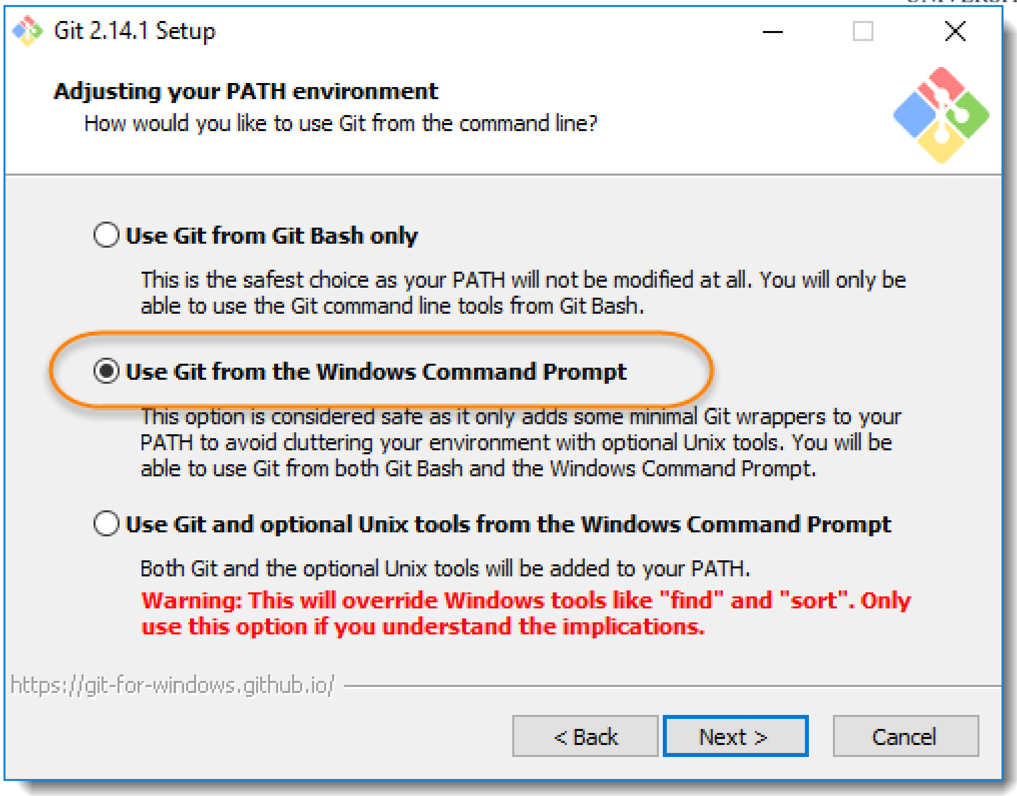
To continue, click Next. If you would like to select a different folder, click Browse.

C:\Program Files\Git

At least 213.0 MB of free disk space is required.

https://git-for-windows.github.io/

STEP 5: This is asking your choice that whether you like to Git from the **Windows Command Prompt** or you like to use some other program like **Git Bash**.


Git 2.14.1 Setup

Adjusting your PATH environment
How would you like to use Git from the command line?

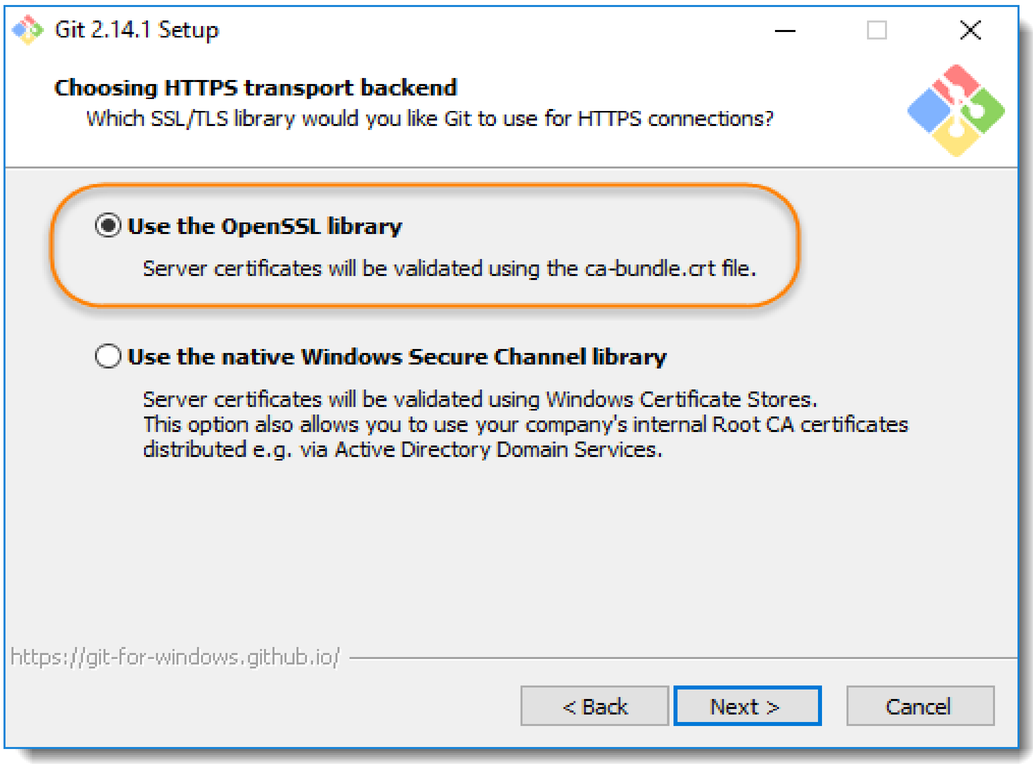
( ) **Use Git from Git Bash only**
This is the safest choice as your PATH will not be modified at all. You will only be able to use the Git command line tools from Git Bash.

[✓ **Use Git from the Windows Command Prompt**]
This option is considered safe as it only adds some minimal Git wrappers to your 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.

https://git-for-windows.github.io/

STEP 6: Select use the openSSL library


Git 2.14.1 Setup

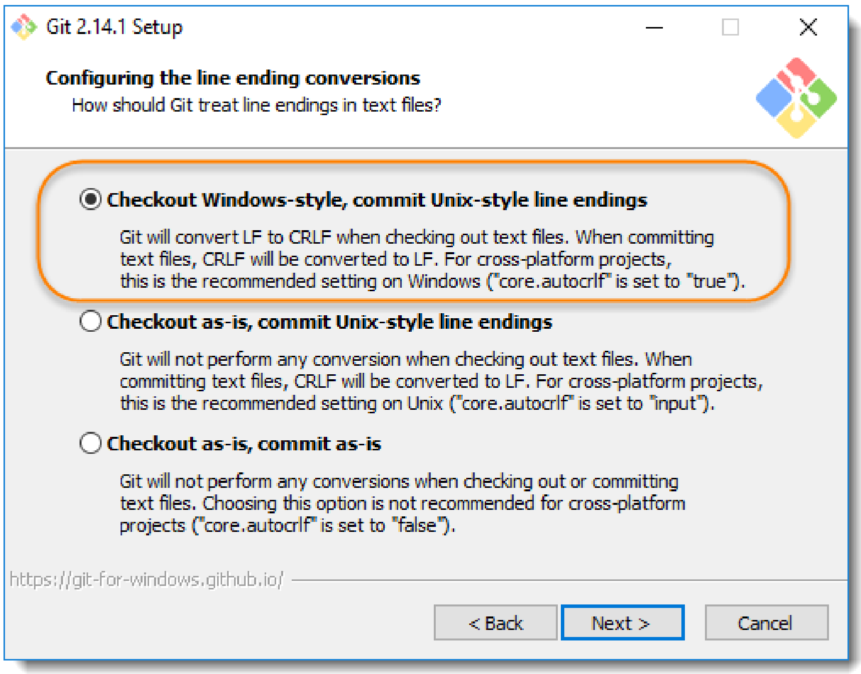
Choosing HTTPS transport backend
Which SSL/TLS library would you like Git to use for HTTPS connections?

[✓ **Use the OpenSSL library**]
Server certificates will be validated using the ca-bundle.crt file.

( ) **Use the native Windows Secure Channel library**
Server certificates will be validated using Windows Certificate Stores.
This option also allows you to use your company's internal Root CA certificates distributed e.g. via Active Directory Domain Services.

https://git-for-windows.github.io/

STEP 7: select the option and click on next.



Git 2.14.1 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").

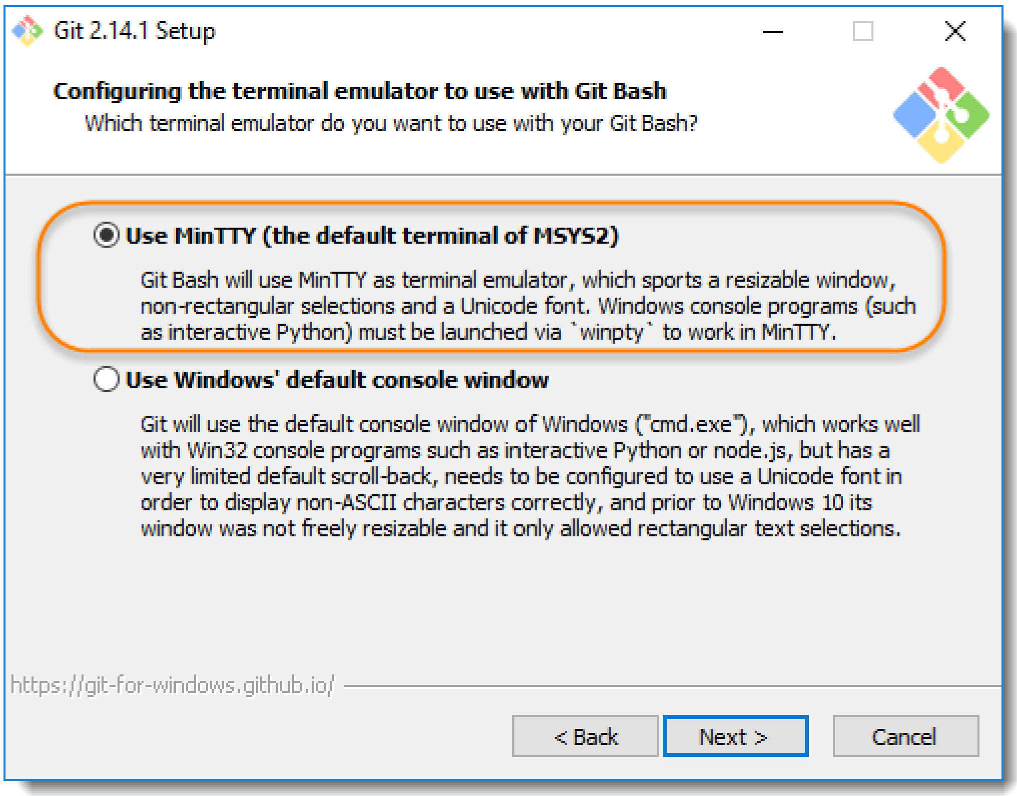
Checkout as-is, commit Unix-style line endings

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").

STEP 8: select the option and click on next.



Git 2.14.1 Setup

Configuring the terminal emulator to use with Git Bash

Which terminal emulator do you want to use with your Git Bash?

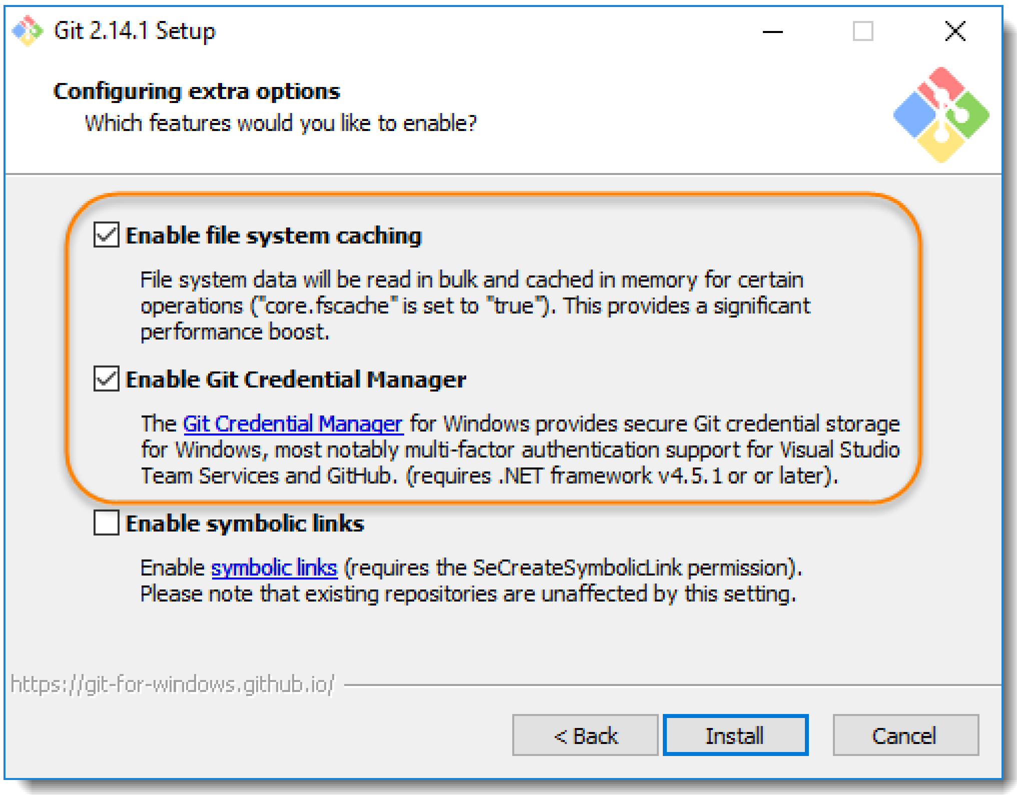
[✓ **Use MinTTY (the default terminal of MSYS2)**]

Git Bash will use MinTTY as terminal emulator, which sports a resizable window, non-rectangular selections and a Unicode font. Windows console programs (such as interactive Python) must be launched via `winpty` to work in MinTTY.

Use Windows' default console window

Git will use the default console window of Windows ("cmd.exe"), which works well with Win32 console programs such as interactive Python or node.js, but has a very limited default scroll-back, needs to be configured to use a Unicode font in order to display non-ASCII characters correctly, and prior to Windows 10 its window was not freely resizable and it only allowed rectangular text selections.

STEP 9: Just go with the default selections and click on install



Git 2.14.1 Setup

Configuring extra options

Which features would you like to enable?

☑ **Enable file system caching**

File system data will be read in bulk and cached in memory for certain operations ("core.fscache" is set to "true"). This provides a significant performance boost.

☑ **Enable Git Credential Manager**

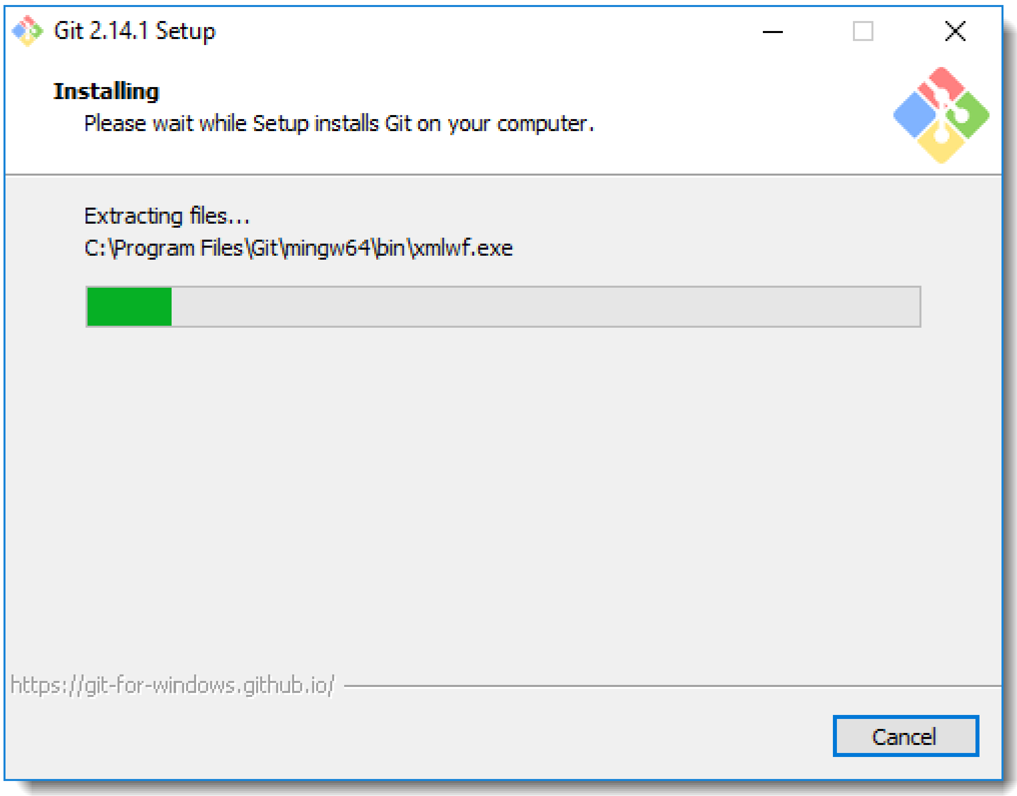
The Git Credential Manager for Windows provides secure Git credential storage for Windows, most notably multi-factor authentication support for Visual Studio Team Services and GitHub. (requires .NET framework v4.5.1 or or later).

☐ **Enable symbolic links**

Enable symbolic links (requires the SeCreateSymbolicLink permission). Please note that existing repositories are unaffected by this setting.

https://git-for-windows.github.io/

STEP 10: Wait for minutes for the installation to complete.



Git 2.14.1 Setup

Installing

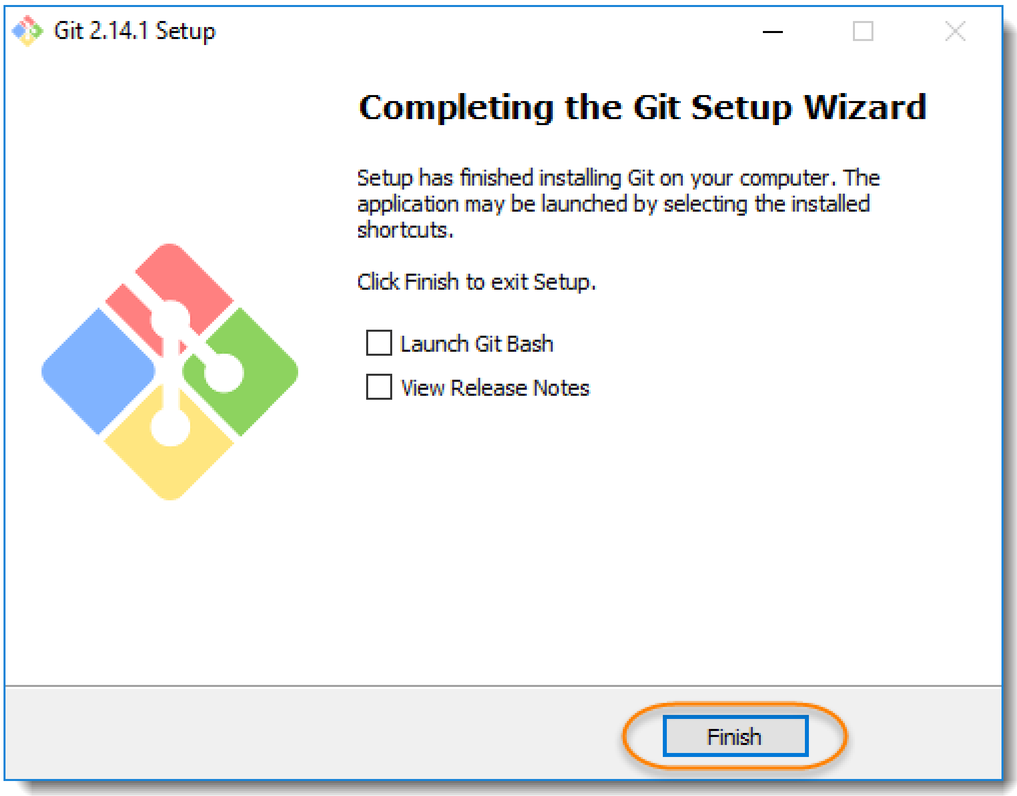
Please wait while Setup installs Git on your computer.

Extracting files...

C:\Program Files\Git\mingw64\bin\xmlwf.exe

https://git-for-windows.github.io/

STEP 11: Once done just click on finish button and the GitHub is installed on your systems.



Git 2.14.1 Setup

Completing the Git Setup Wizard

Setup has finished installing Git on your computer. The application may be launched by selecting the installed shortcuts.

Click Finish to exit Setup.

- [ ] Launch Git Bash
- [ ] View Release Notes

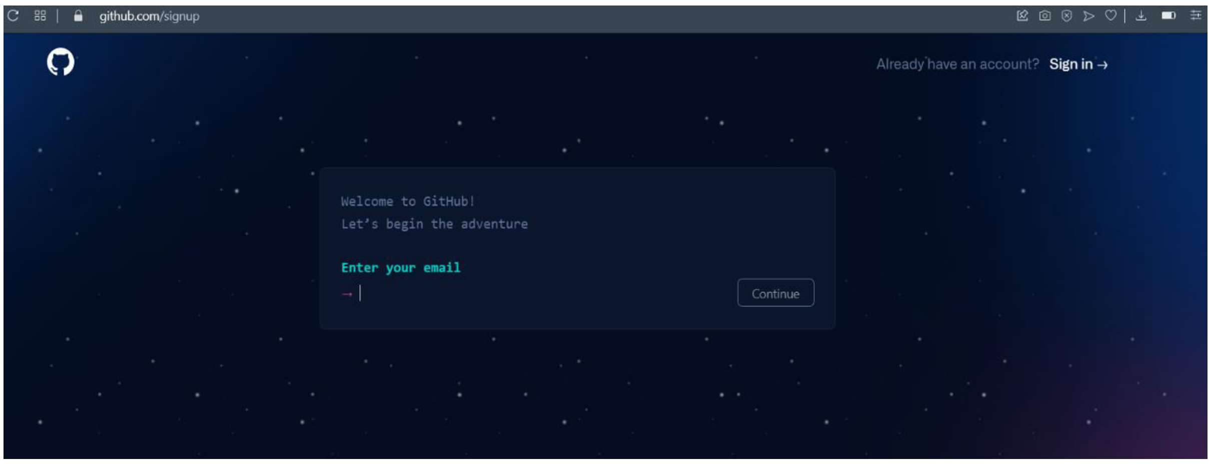
AIM : TO SETUP GITHUB ACCOUNT

THEORY:
GitHub is a website and cloud-based service (client) that helps an individual or a developer to store and manage their code. We can also track as well as control changes to our or public code. GitHub's has a user-friendly interface and is easy to use .

We can connect the git-hub and git but using some commands shown below in figure 001. Without GitHub we cannot use Git because it generally requires a host and if we are working for a project, we need to share it will our team members, which can only be done by making a repository . Additionally , anyone can sign up and host a public code repository for free, which makes GitHub especially popular with open-source projects

PROCEDURE:

STEP1: On any search engine like google, Microsoft edge, opera, etc. , search for git-hub.
A dialogue box would appear welcoming you to the git-hub.


github.com/signup

Already have an account? Sign in →

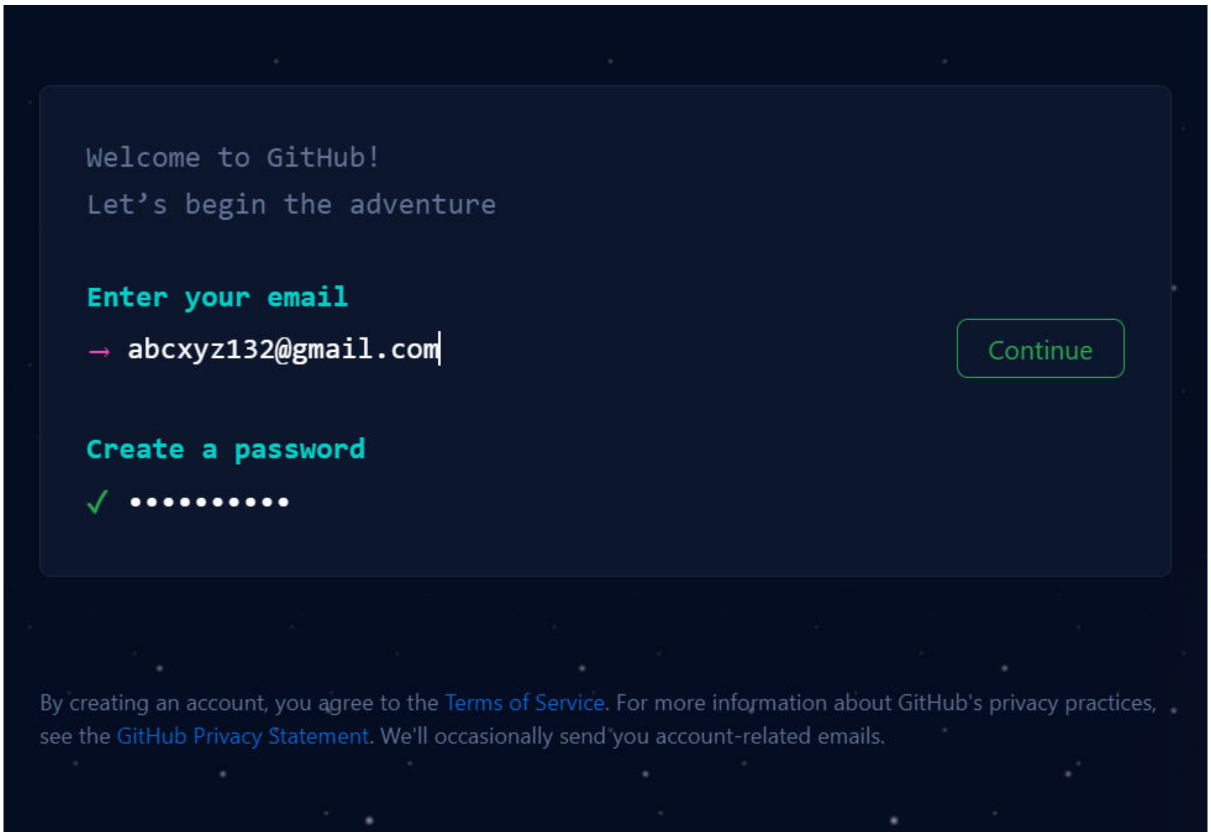
Welcome to GitHub!
Let's begin the adventure

Enter your email
→ |

STEP2: If you already have an account, then on the top right corner there's an option of signing up. Click on "SIGN IN ".

STEP 3: But if you don't have any account, then enter your email and continue

STEP 4: Then create a strong password for your GitHub account

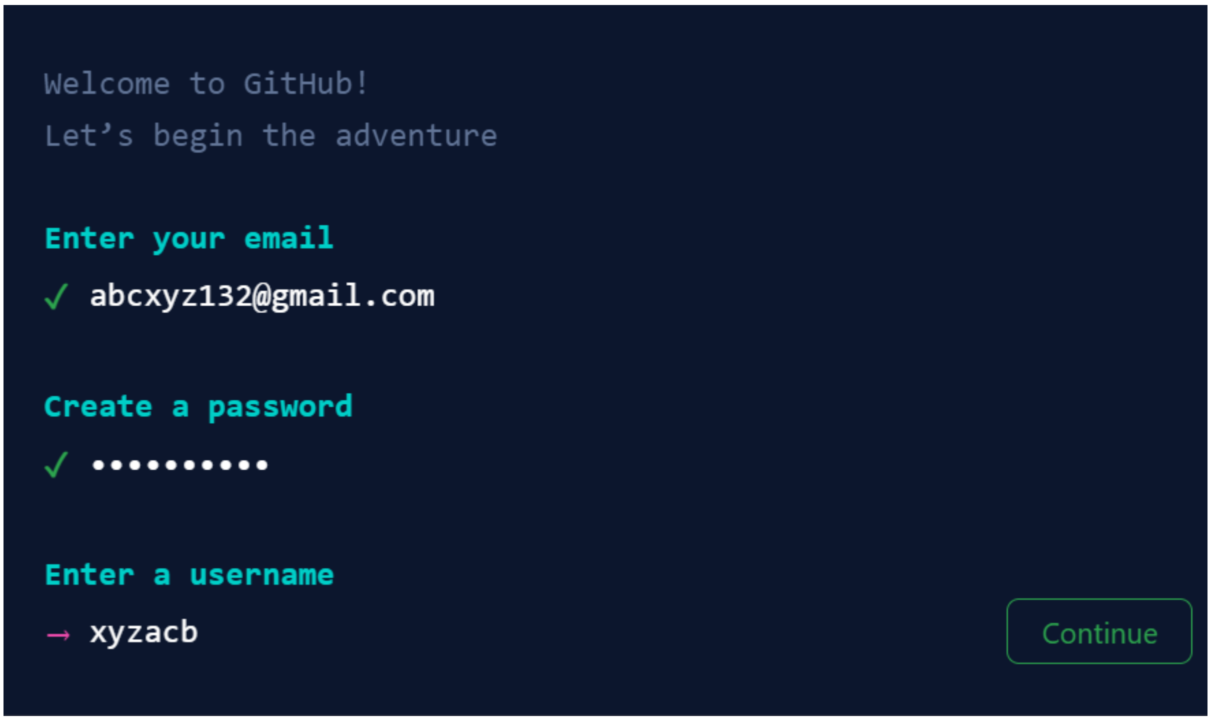

Welcome to GitHub!
Let's begin the adventure

Enter your email
→ abcxyz132@gmail.com

Create a password
✓ ••••••••••

By creating an account, you agree to the Terms of Service. For more information about GitHub's privacy practices, see the GitHub Privacy Statement. We'll occasionally send you account-related emails.

STEP 4: Create a username


Welcome to GitHub!
Let's begin the adventure

Enter your email
✓ abcxyz132@gmail.com

Create a password
✓ ••••••••••

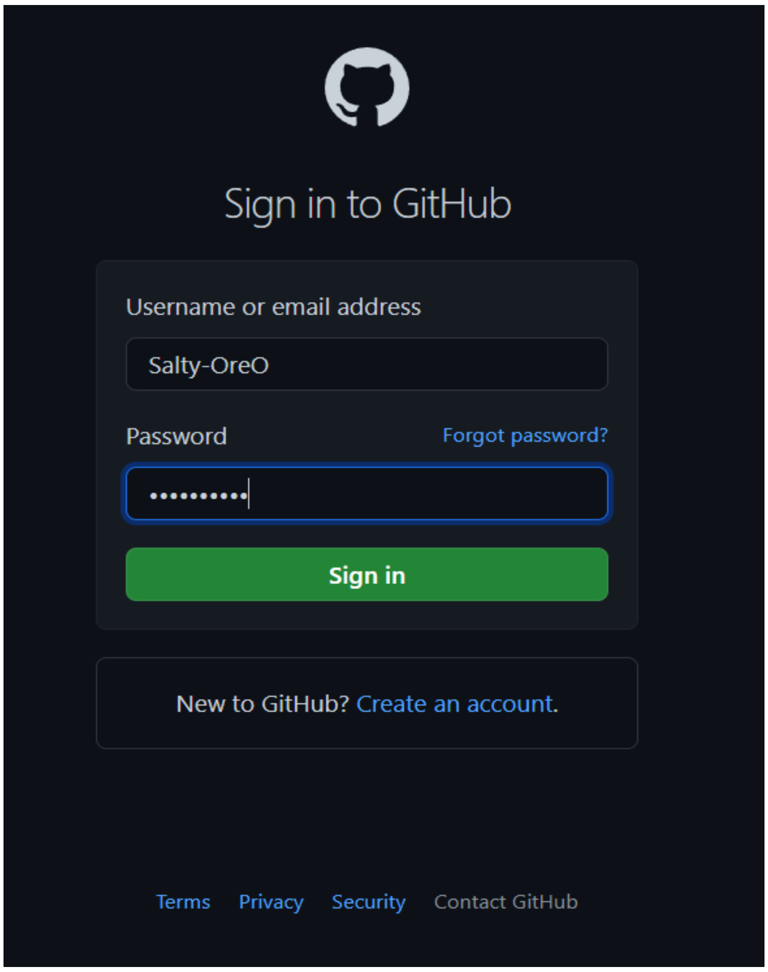
Enter a username
→ xyzacb

STEP 5: fill in all the details that are required and click on create account and your GitHub account would be created

BUT IF YOU ALREADY HAVE AN ACCOUNT, THEN FOLLOW THESE STEPS:

STEP 1: Sign In

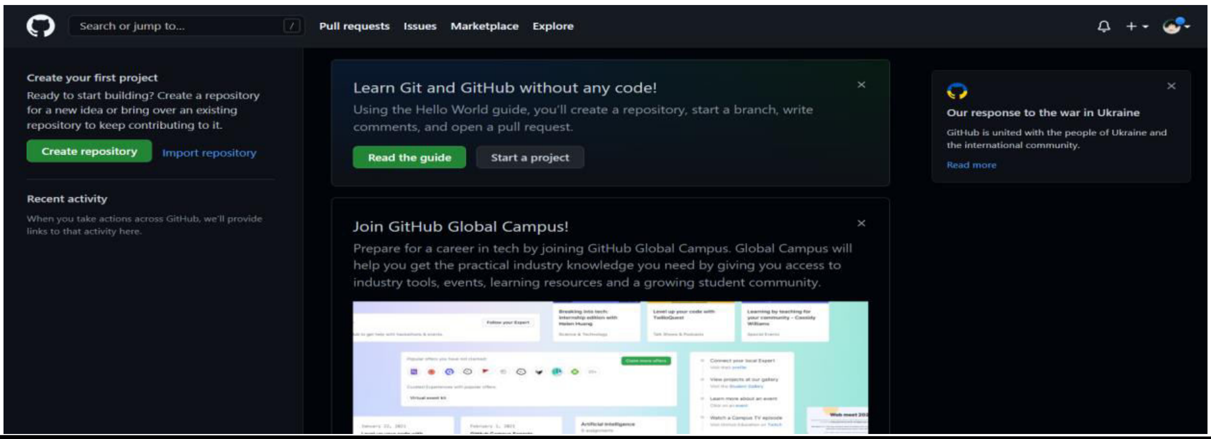
Enter your email-address or username , then the password and continue


Sign in to GitHub

Username or email address: Salty-OreO

Password: ••••••••••

STEP 2: you have logged in in your account. Now you can create and edit any project



Create your first project

Ready to start building? Create a repository for a new idea or bring over an existing repository to keep contributing to it.

Recent activity

When you take actions across GitHub, we'll provide links to that activity here.

Learn Git and GitHub without any code!

Using the Hello World guide, you'll create a repository, start a branch, write comments, and open a pull request.

Our response to the war in Ukraine

GitHub is united with the people of Ukraine and the international community.

Join GitHub Global Campus!

Prepare for a career in tech by joining GitHub Global Campus. Global Campus will help you get the practical industry knowledge you need by giving you access to industry tools, events, learning resources and a growing student community.

AIM : PROGRAM TO GENERATE LOGS

THEORY:

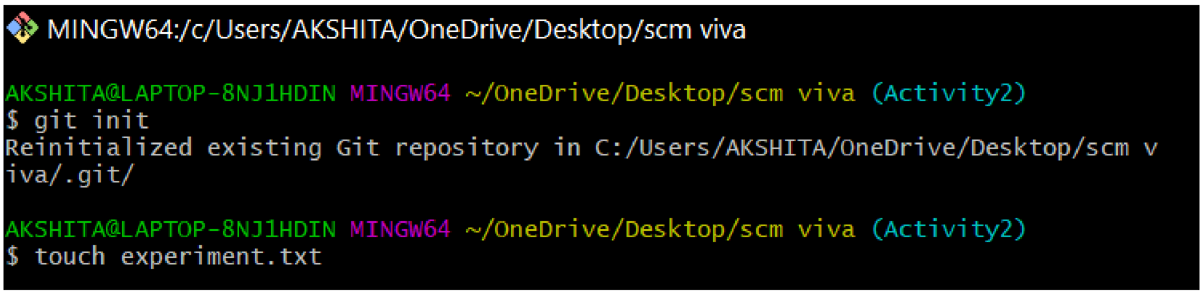
Logs are nothing but the history which we can see in git . It contains all the past commits, insertions and deletions in it which we can see any time.

Logs helps to check that what were the changes in the code or any other file and by whom. It also contains the number of insertions and deletions including at which time it was changed.

PROCEDURE:

The command used to generate logs in git is :

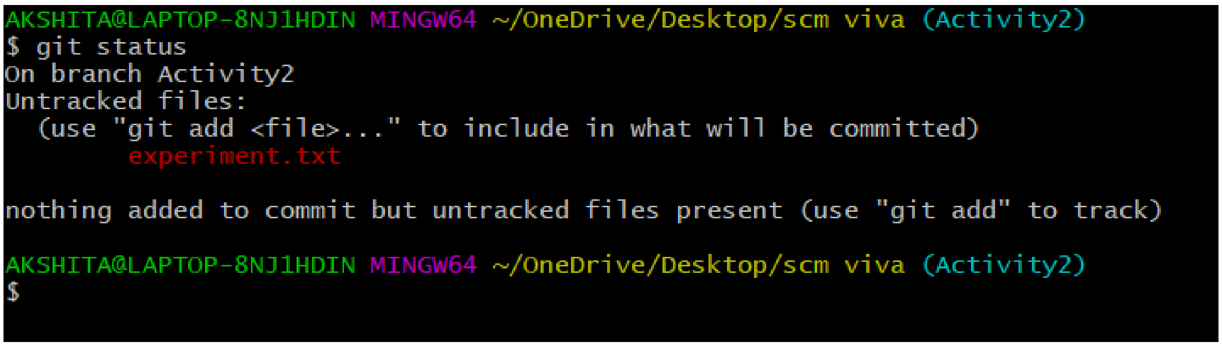
STEP1: Create a file in the folder.



```
MINGW64:/c/Users/AKSHITA/OneDrive/Desktop/scm viva

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ git init
Reinitialized existing Git repository in C:/Users/AKSHITA/OneDrive/Desktop/scm v
iva/.git/

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ touch experiment.txt
```

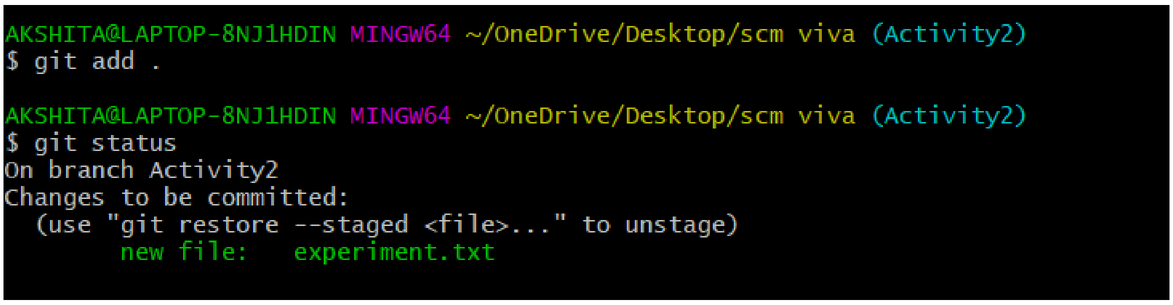
STEP 2: Check status

It will show the file name in red colour. This means that the file is untracked.

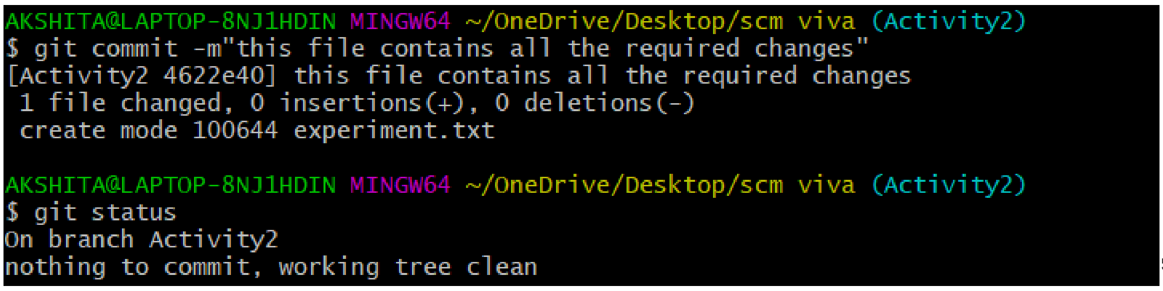


```
AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ git status
On branch Activity2
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        experiment.txt

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

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$
```

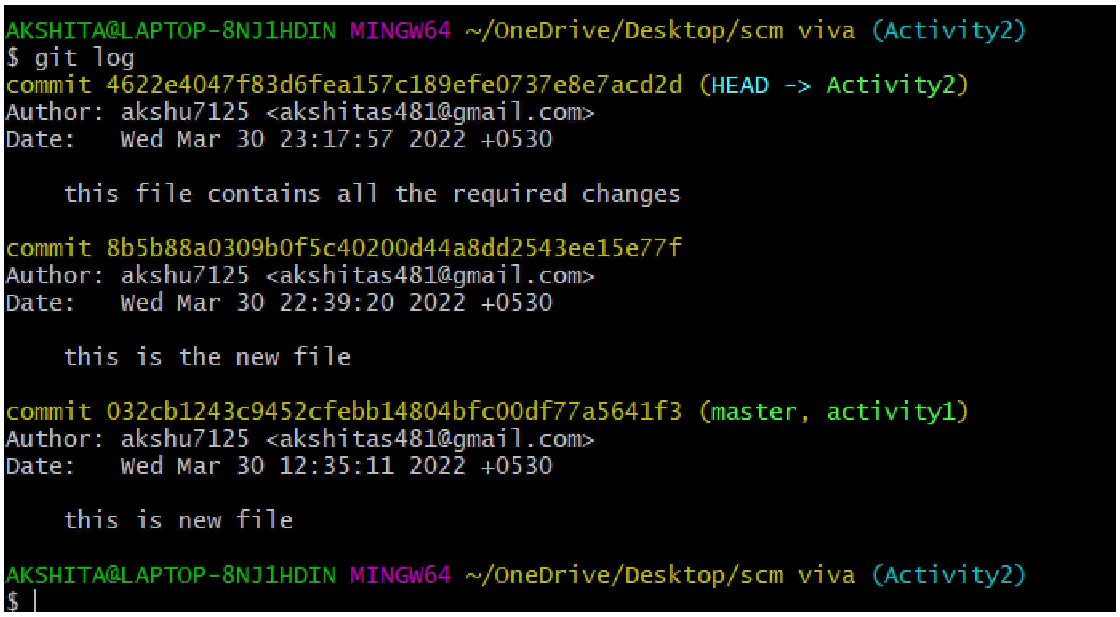
STEP 3: Staging of file


```
AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ git add .

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ git status
On branch Activity2
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   experiment.txt
```

STEP 4: Commit the file and check the status


```
AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ git commit -m"this file contains all the required changes"
[Activity2 4622e40] this file contains all the required changes
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 experiment.txt

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ git status
On branch Activity2
nothing to commit, working tree clean
```
,

STEP 5: Now run the command $ git log to generate all the commits in the repository


```
AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ git log
commit 4622e4047f83d6fea157c189efe0737e8e7acd2d (HEAD -> Activity2)
Author: akshu7125 <akshitas481@gmail.com>
Date:   Wed Mar 30 23:17:57 2022 +0530

    this file contains all the required changes

commit 8b5b88a0309b0f5c40200d44a8dd2543ee15e77f
Author: akshu7125 <akshitas481@gmail.com>
Date:   Wed Mar 30 22:39:20 2022 +0530

    this is the new file

commit 032cb1243c9452cfebb14804bfc00df77a5641f3 (master, activity1)
Author: akshu7125 <akshitas481@gmail.com>
Date:   Wed Mar 30 12:35:11 2022 +0530

    this is new file

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ |
```

- ✓ As it can be observed, on using this command, the system displays all the changes made in the file or list of all the commits in the history along with the information of the user.
- ✓ **This commands clearly defines the git as the 'version-controlled system' as it allows us to rollback to any of the previous working states and keeps track of all the versions.**

AIM : TO CREATE AND VISUALIZE BRANCHES.

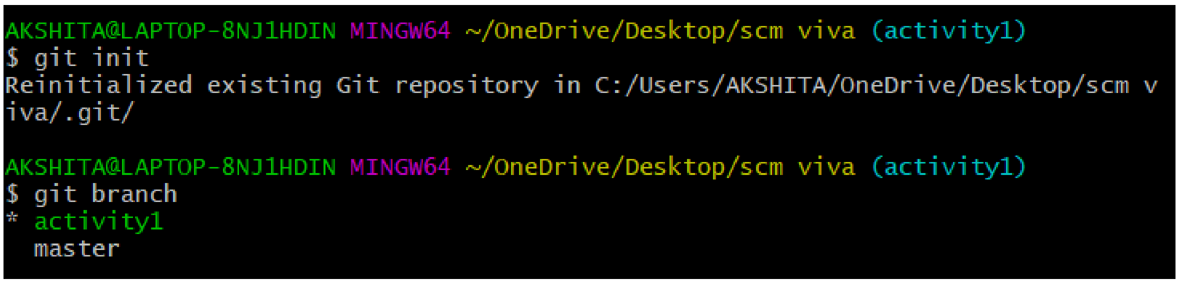
THEORY:

The main branch in git is called as 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 file which are created in branch are not shown in master branch. We can also merge both the parent (master) and child (other branches).

FOLLOW THESE STEPS TO CREATE A SEPARATE BRANCH IN GIT.

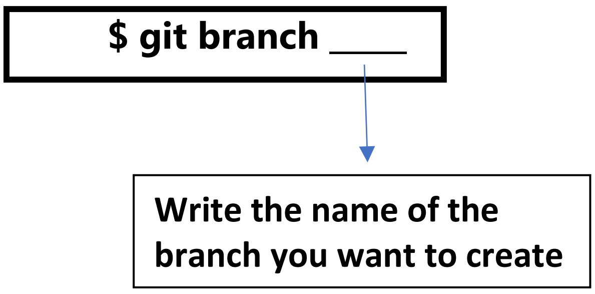
1. **Firstly, you can check the present branches in the master branch with the help of this command.**

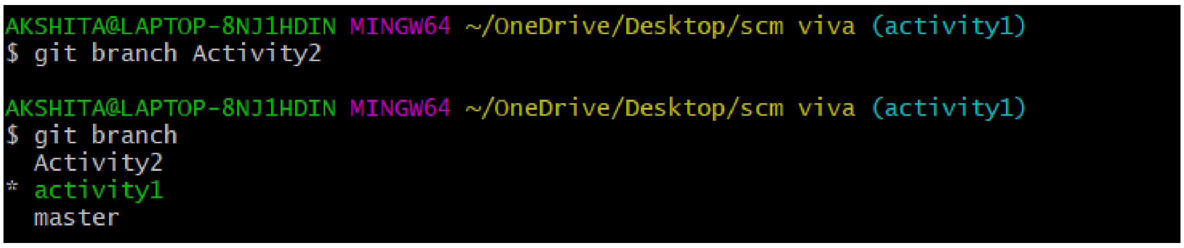
   $ git branch



```
AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (activity1)
$ git init
Reinitialized existing Git repository in C:/Users/AKSHITA/OneDrive/Desktop/scm v
iva/.git/

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (activity1)
$ git branch
* activity1
  master
```

2. **To create a branch, enter:**

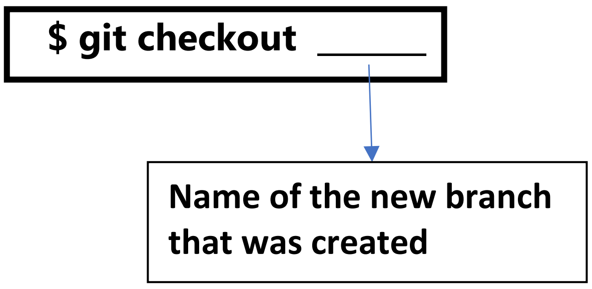


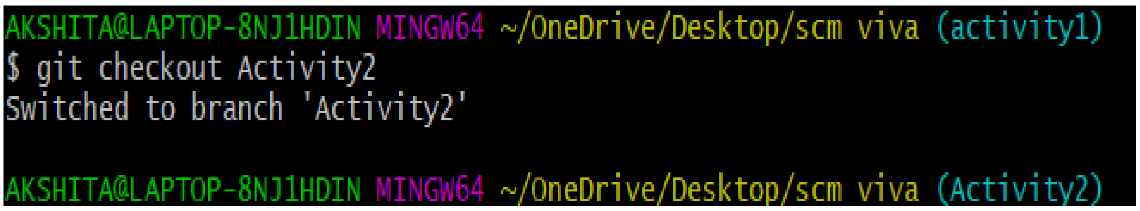


```
AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (activity1)
$ git branch Activity2

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (activity1)
$ git branch
  Activity2
* activity1
  master
```

The branch has been created.

3. The next step is to transfer the data from the master branch to the new branch. For this we use:



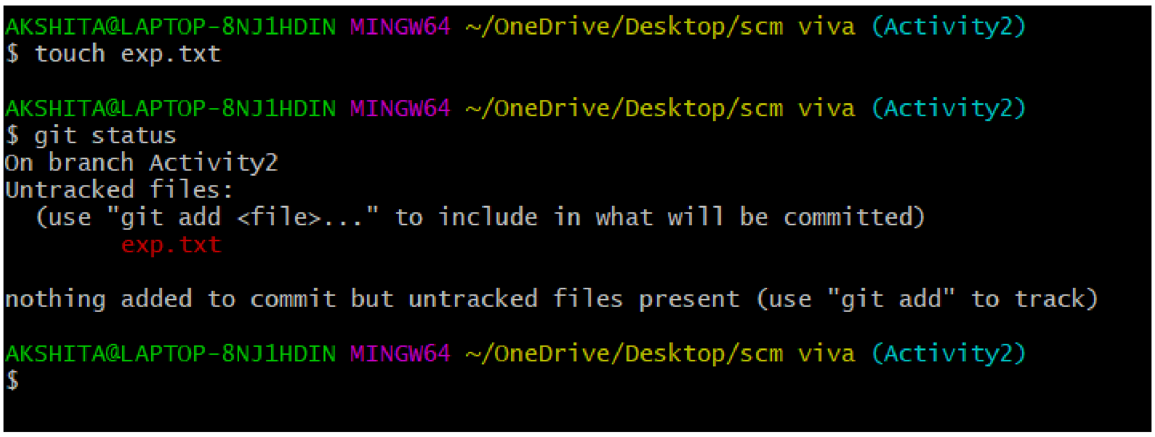


```
AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (activity1)
$ git checkout Activity2
Switched to branch 'Activity2'

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
```

All the data from the master branch has been transferred to this branch.

4. Staging the file.

Create a file and check the status.
You would observe the name of the file is in red colour with a notation "untracked ".

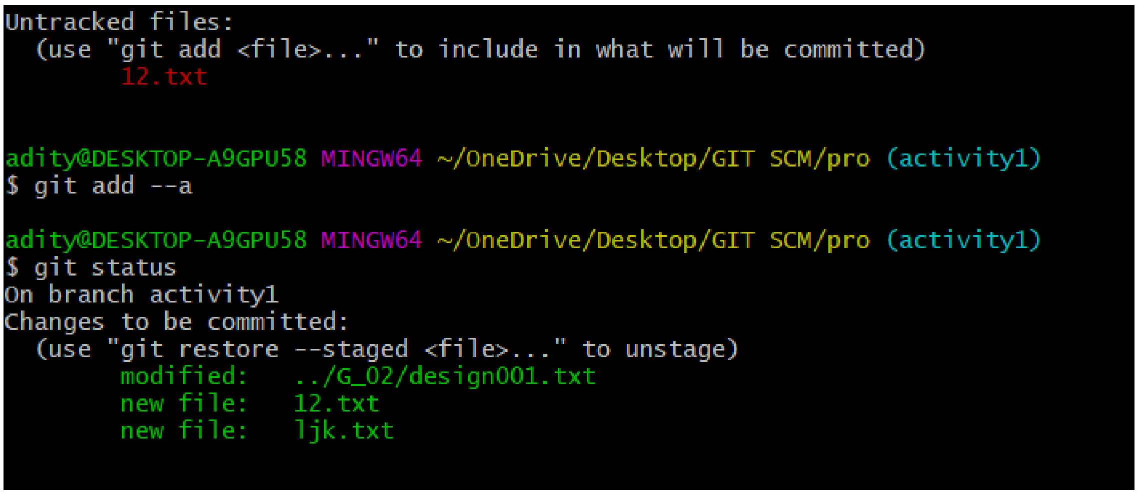


```
AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ touch exp.txt

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ git status
On branch Activity2
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        exp.txt

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

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$
```

This means that only the data has been transferred to the file but we cannot make changes in the same as the current working directory is the master branch.

To overcome this, we stage the file by using the command :

$ git add –a


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


adity@DESKTOP-A9GPU58 MINGW64 ~/OneDrive/Desktop/GIT SCM/pro (activity1)
$ git add --a

adity@DESKTOP-A9GPU58 MINGW64 ~/OneDrive/Desktop/GIT SCM/pro (activity1)
$ git status
On branch activity1
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   ../G_02/design001.txt
        new file:   12.txt
        new file:   ljk.txt
```

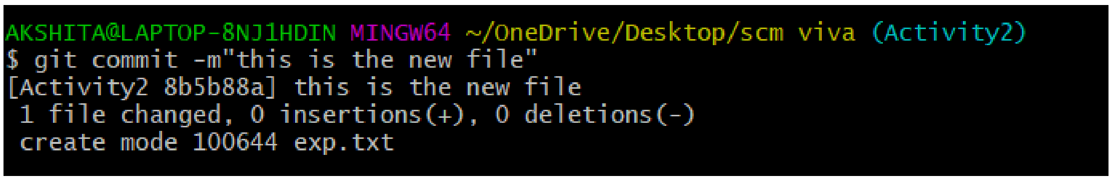
Check if the file is staged or not by using $ git status command.

✓ As you can observe the file name is green in colour now which denotes the file is staged.

5. Commit

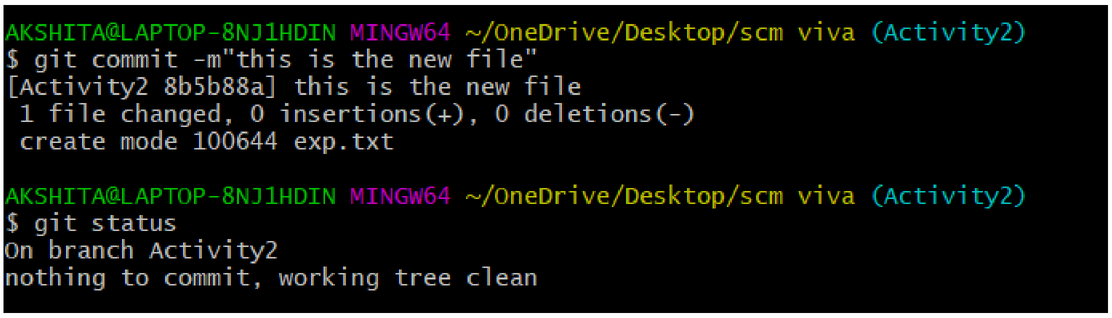
After staging we need to commit. This assures the system that the directory has been shifted to the new file. For this, the command used is:

$ git commit-m ______


```
AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ git commit -m"this is the new file"
[Activity2 8b5b88a] this is the new file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 exp.txt
```

6. Check the status

On checking the status, a message will be displayed as ‘ working tree clean’. Which means all the files inside that directory are tracked.


```
AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ git commit -m"this is the new file"
[Activity2 8b5b88a] this is the new file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 exp.txt

AKSHITA@LAPTOP-8NJ1HDIN MINGW64 ~/OneDrive/Desktop/scm viva (Activity2)
$ git status
On branch Activity2
nothing to commit, working tree clean
```

Now you can make any of the changes in the file but the modifications won’t be reflected in the master branch.

AIM: TO EXPLAIN GIT LIFECYCLE

THEORY: When a project is under Git version control system, they are present in three major Git states in addition to these basic ones. Here are the three Git states:

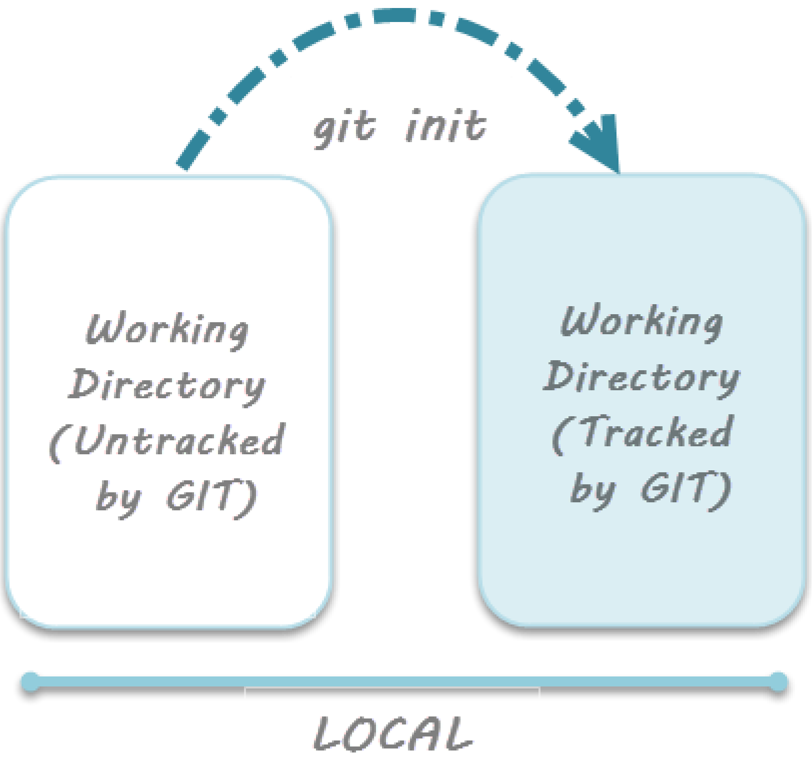
1. **Working directory**
2. **Staging area**
3. **Git directory**

1.Working Directory

Consider a project residing in your local system. This project may or may not be tracked by Git. In either case, this project directory is called your Working directory. Working directory is the directory containing hidden .git folder.

Working directory is the directory containing hidden .git folder.

git init - *Command to initialize a Git repository*


git init
Working Directory (Untracked by GIT)
Working Directory (Tracked by GIT)
LOCAL

reference for picture: www.toolsqa.com/git/git-life-cycle/

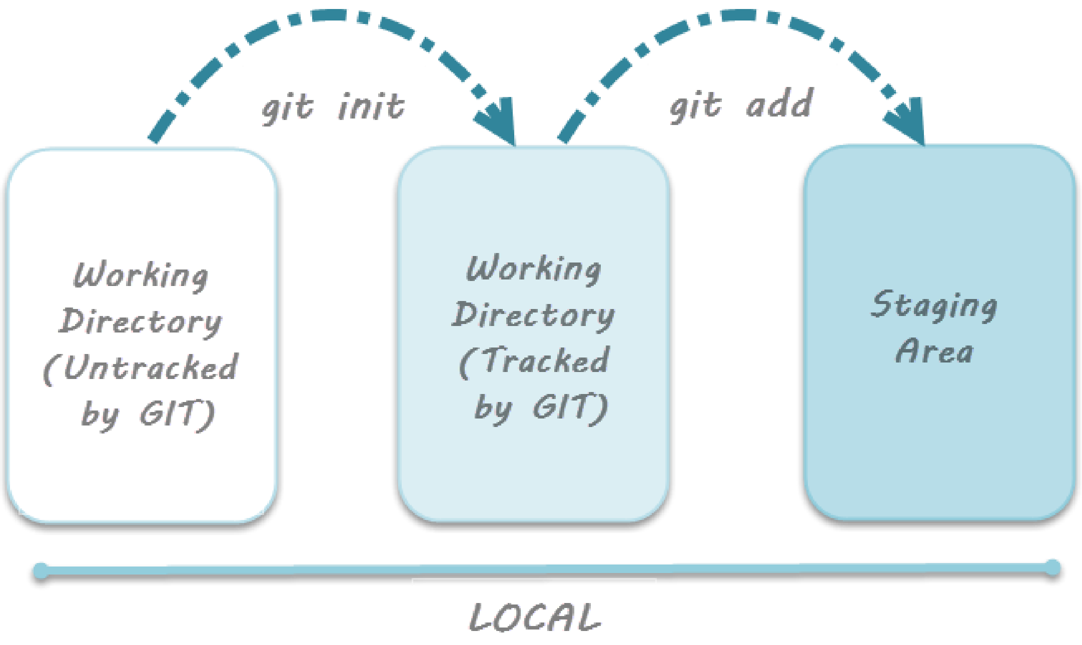
2.Staging area

some files in the project like class files, log files, result files and temporary data files are dynamically generated. It doesn't make sense to track the versions of these files.

Whereas the source code files, data files, configuration files and other project artifacts contain the business logic of the application. These files are to be tracked by Git are thus needs to be added to the staging area.

In other words, staging area is the playground where you group, add and organize the files to be committed to Git for tracking their versions.

git add *- Command to add files to staging area.*


git init
git add
Working Directory (Untracked by GIT)
Working Directory (Tracked by GIT)
Staging Area
LOCAL

{reference for picture: www.toolsqa.com/git/git-life-cycle/ }

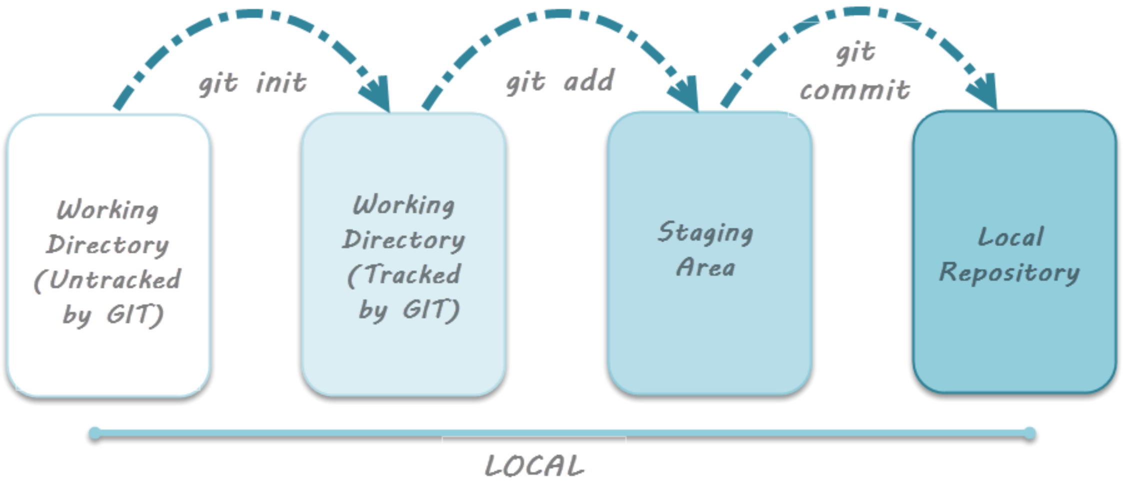
3.Git Directory

Now that the files to be committed are grouped and ready in the staging area, we can commit these files. So, we commit this group of files along with a commit message explaining what is the commit about. Apart from commit message, this step also records the author and time of the

commit. Now, a snapshot of the files in the commit is recorded by Git. The information related to this commit is stored in the Git directory.

Thus, Git directory is the database where metadata about project files' history will be tracked.

git commit -m"your message" - *Command to commit files to Git repository with message*.


git init
git add
git commit
Working Directory (Untracked by GIT)
Working Directory (Tracked by GIT)
Staging Area
Local Repository
LOCAL

{reference for picture: www.toolsqa.com/git/git-life-cycle/ }

Subject Name: **Source Code Management**

Subject Code: **CS181**

Cluster: **Zeta**

Department: **DCSE**

CHITKARA UNIVERSITY

Submitted By:
JASJOT SINGH
2110992022
G27

Submitted To:
Dr. ANUJ JAIN

Experiment No. 06

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:

```
$ git clone git@github.com:vlad/planets.git ~/Desktop/vlad-plane
ts
```

Replace 'vlad' with the Owner's username.

If you choose to clone without the clone path (`~/Desktop/vlad-planets`) specified at the end, you will clone inside your own planets folder! Make sure to navigate to the `Desktop` folder first.

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:

```
$ cd ~/Desktop/vlad-planets
$ nano pluto.txt
$ cat pluto.txt

It is so a planet!

$ git add pluto.txt
$ git commit -m "Add notes about Pluto"

 1 file changed, 1 insertion(+)
 create mode 100644 pluto.txt
```

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

```
$ git push origin main

Enumerating objects: 4, done.
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 306 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/vlad/planets.git
9272da5..29aba7c  main -> main
```

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` to `fred`.

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

```
$ git pull origin main remote: Enumerating
objects: 4, done. remote: Counting objects:
100% (4/4), done. remote: Compressing
objects: 100% (2/2), done.
```

```
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/vlad/planets
 * branch            main     -> FETCH_HEAD
   9272da5..29aba7c  main     -> origin/main
Updating 9272da5..29aba7c
Fast-forward  pluto.txt |
1 +
 1 file changed, 1 insertion(+)
create mode 100644 pluto.txt
```

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?

Solution

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.

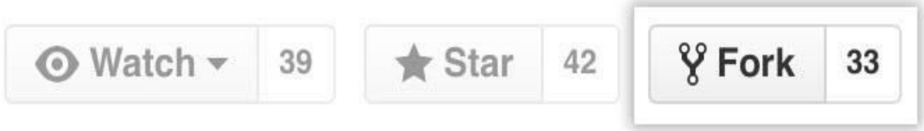
Prerequisites

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.

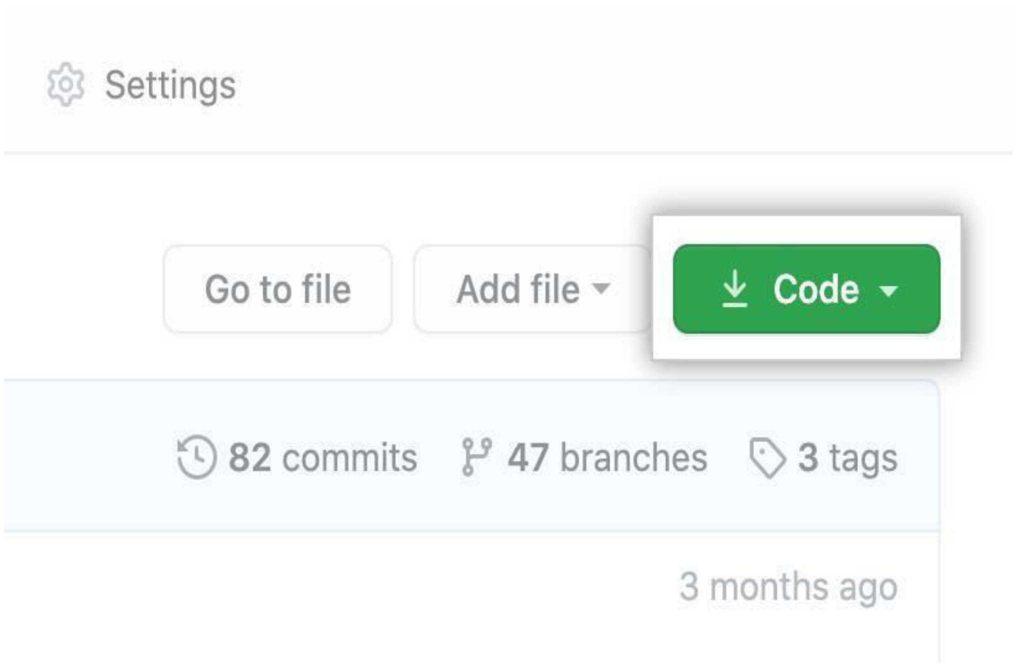
1. On GitHub.com, navigate to the octocat/Spoon-Knife repository.
2. In the top-right corner of the page, click **Fork**.



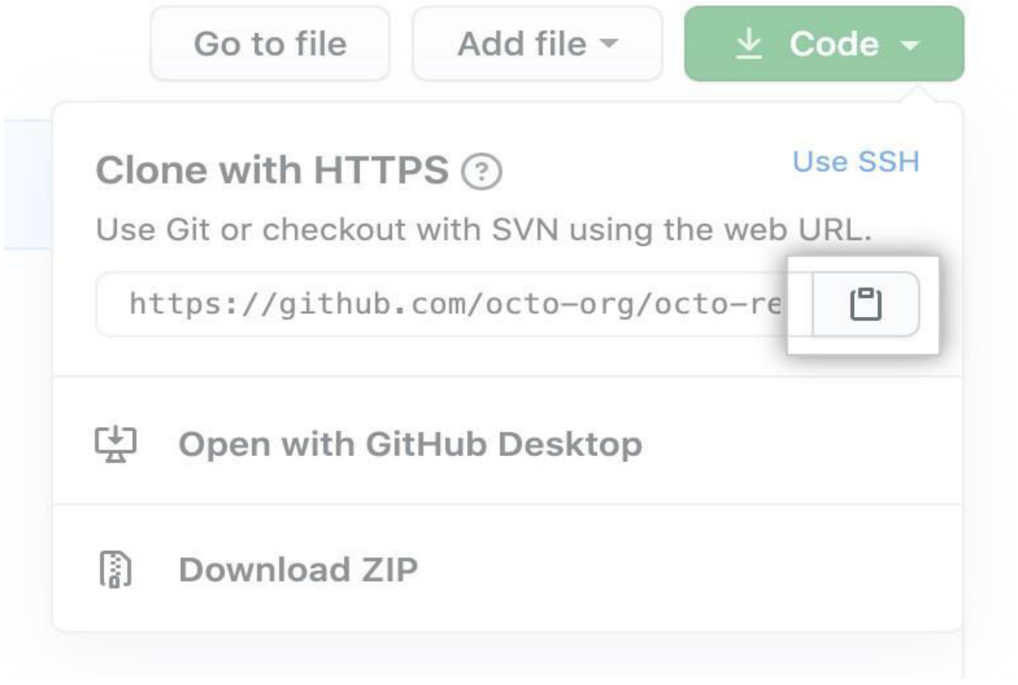
Cloning your forked repository

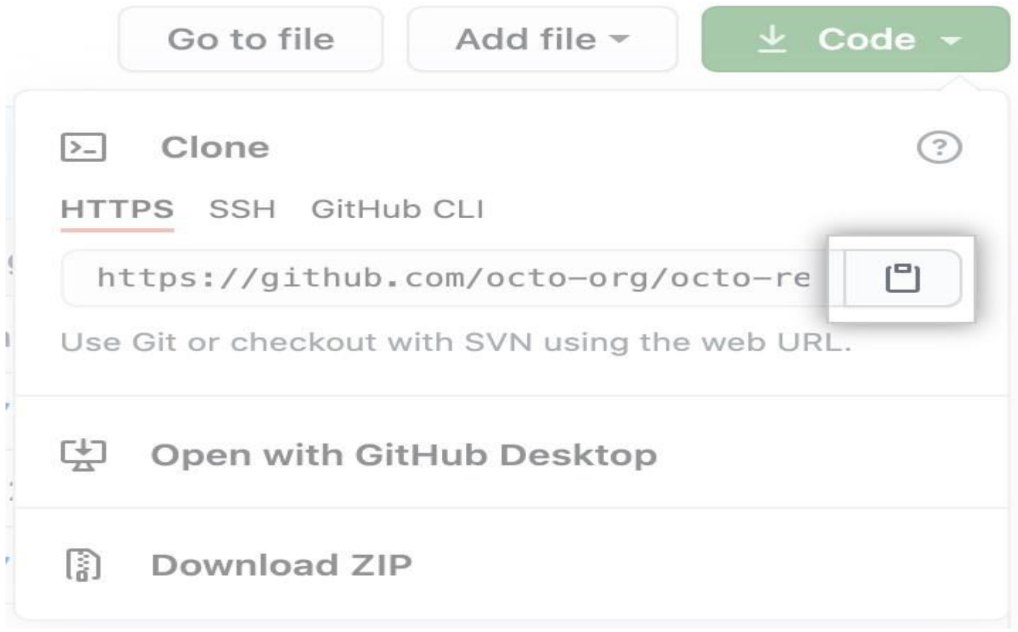
Right now, you have a fork of the Spoon-Knife repository, but you don't have the files in that repository locally on your computer.

1. On GitHub.com, navigate to **your fork** of the 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 .


Go to file
Add file
Code
Clone with HTTPS
Use SSH
Use Git or checkout with SVN using the web URL.
https://github.com/octo-org/octo-re
Open with GitHub Desktop
Download ZIP


Go to file
Add file
Code
Clone
HTTPS
SSH
GitHub CLI
https://github.com/octo-org/octo-re
Use Git or checkout with SVN using the web URL.
Open with GitHub Desktop
Download ZIP

4. Open Git Bash.
5. Change the current working directory to the location where you want the cloned directory.
6. 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`

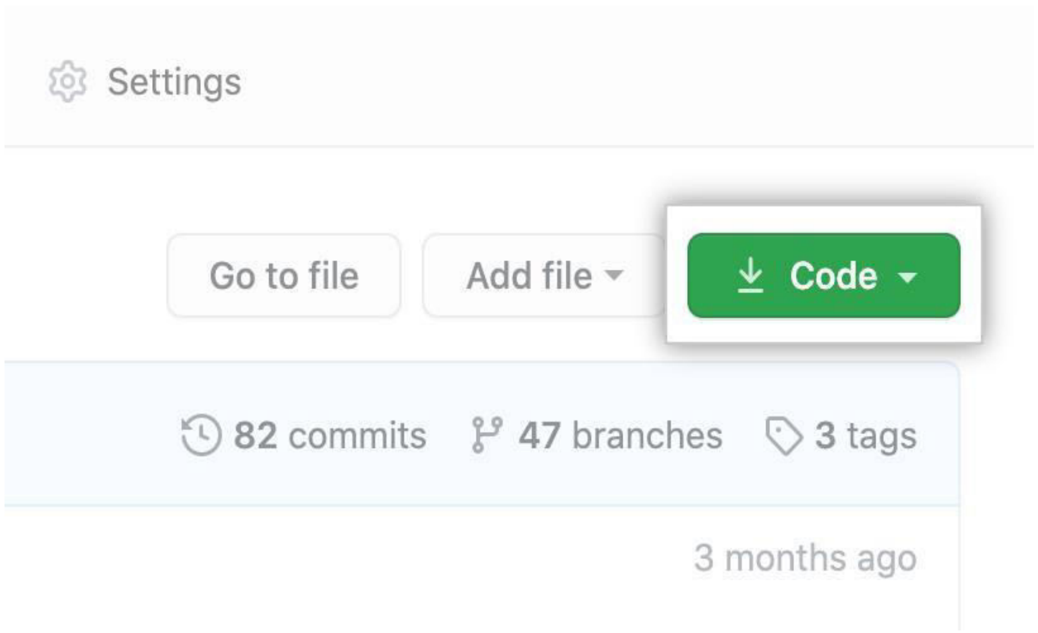
7. Press **Enter**. Your local clone will be created.
8. `$ git clone https://github.com/YOUR-USERNAME/Spoon-Knife`

```
   9. > Cloning into `Spoon-Knife`...
10.  > remote: Counting objects: 10, done.
11.  > remote: Compressing objects: 100% (8/8), done.
12.  > remove: Total 10 (delta 1), reused 10 (delta 1) >
     Unpacking objects: 100% (10/10), done.
```

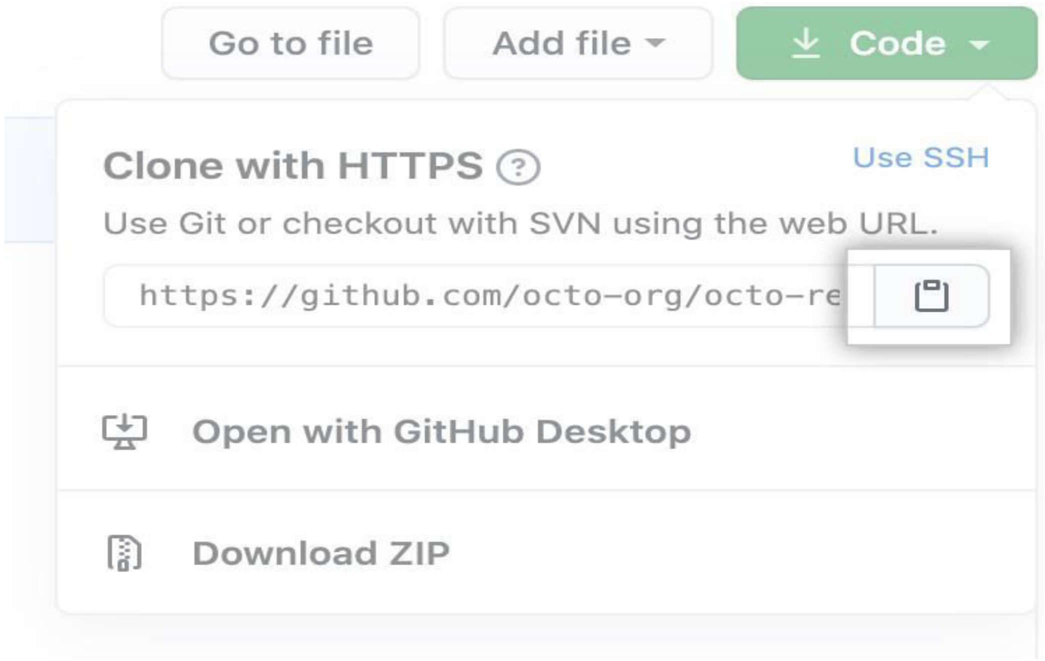
Configuring Git to sync your fork with the original repository

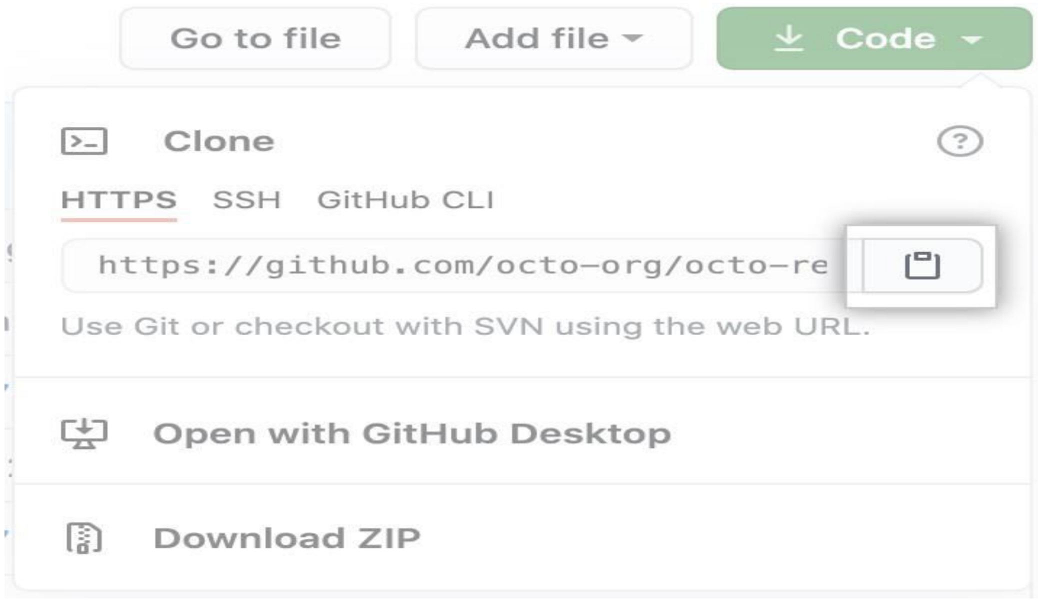
When you fork a project in order to propose changes to the original repository, you can configure Git to pull changes from the original, or upstream, repository into the local clone of your fork.

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 .


Go to file
Add file
Code
Clone with HTTPS
Use SSH
Use Git or checkout with SVN using the web URL.
https://github.com/octo-org/octo-re
Open with GitHub Desktop
Download ZIP


Go to file
Add file
Code
Clone
HTTPS
SSH
GitHub CLI
https://github.com/octo-org/octo-re
Use Git or checkout with SVN using the web URL.
Open with GitHub Desktop
Download ZIP

4. Open Git Bash.
5. Change directories to the location of the fork you cloned.
   - To go to your home directory, type just `cd` with no other text. ○ To list the files and folders in your current directory, type `ls`. ○ To go into one of your listed directories, type `cd your_listed_directory`.
   - To 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.
    ```
    > origin
    https://github.com/YOUR_USERNAME/YOUR_FORK.git (fetch)
    ```

13.
    ```
    > origin
    https://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 Explore 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.

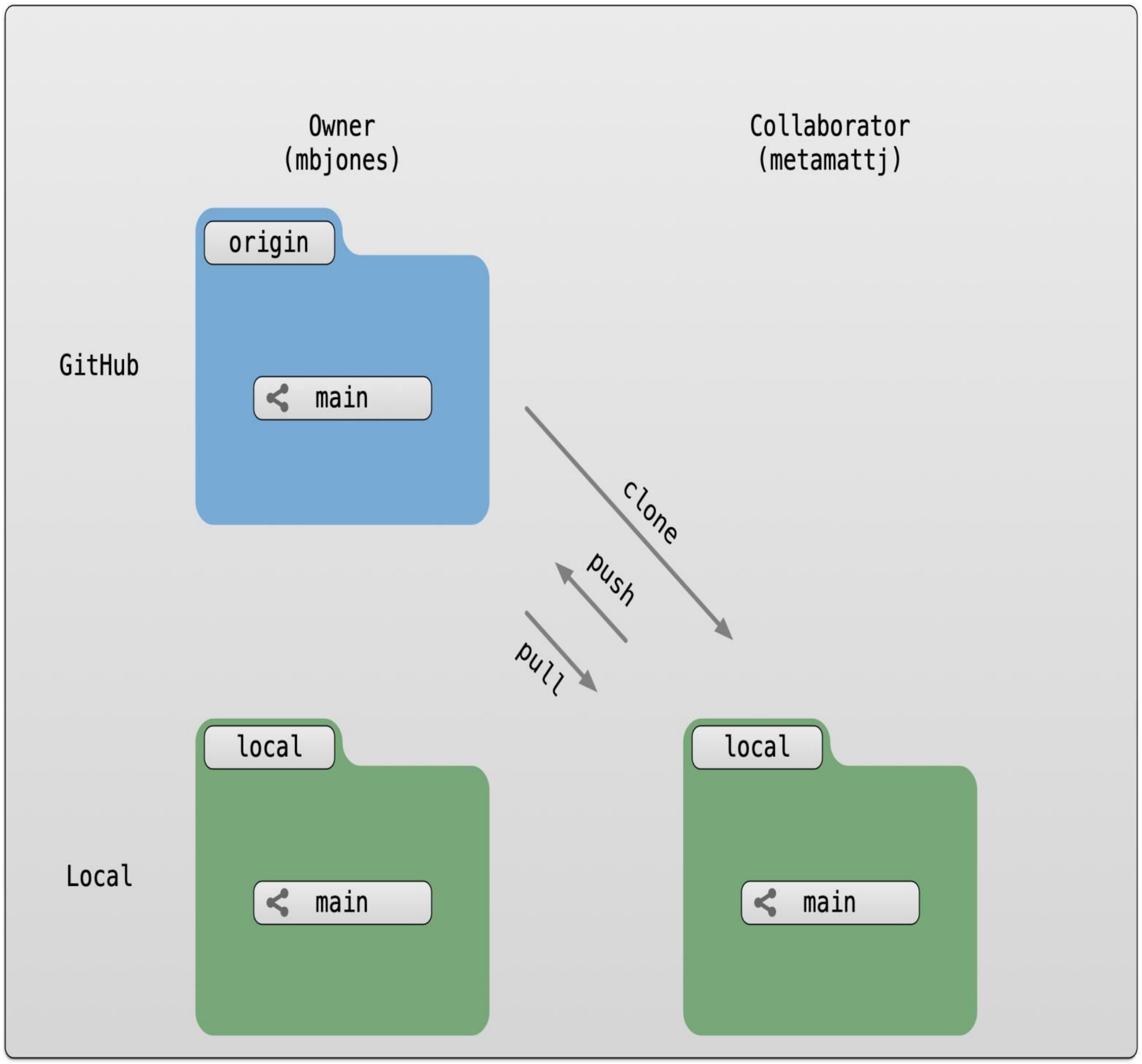
Experiment No. 08

Aim: Merge and Resolve conficts created due to own actvity and collaborators actvity.

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.



5.3 Collaborating with a trusted colleague without conflicts

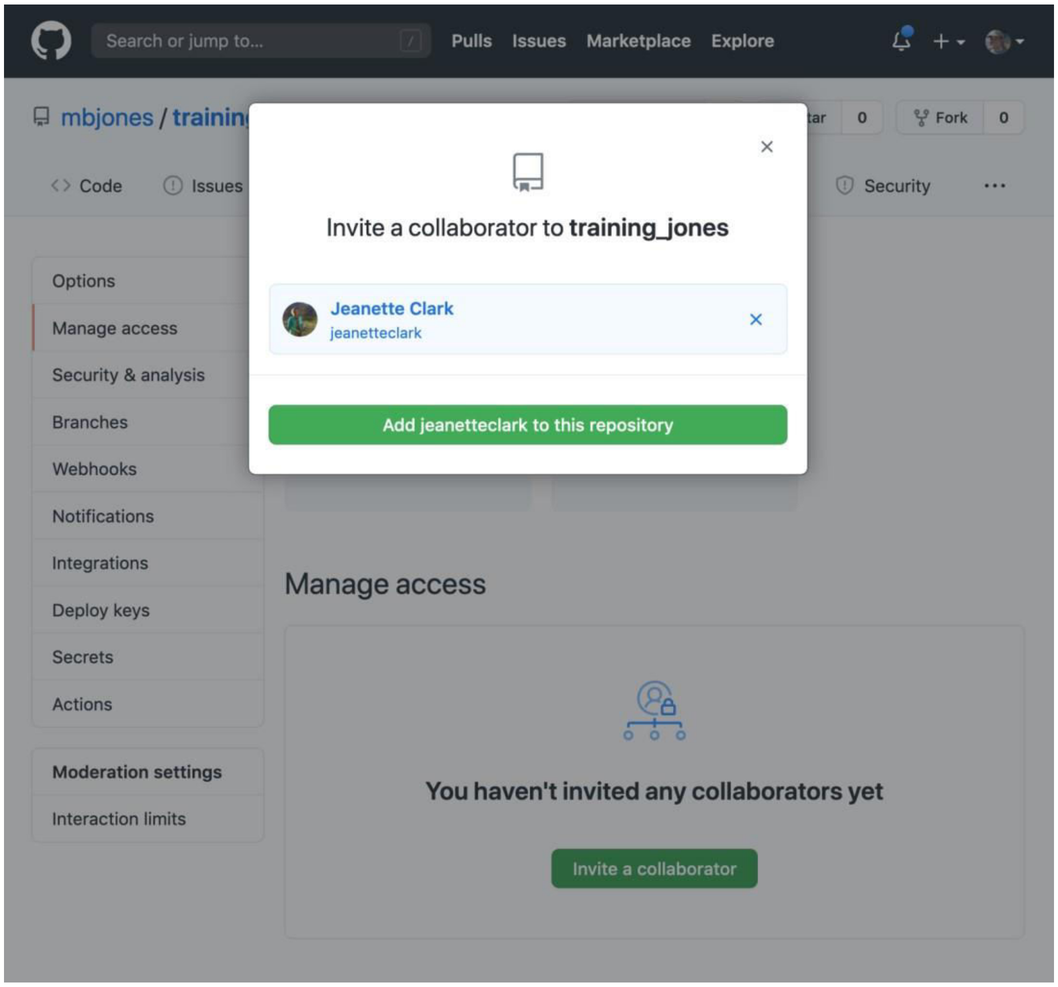
We start by enabling collaboration with a trusted colleague. We will designate the `Owner` as the person who owns the shared repository, and the `Collaborator` as the person that they wish to grant the ability to make changes to their reposity. We start by giving that person access to our GitHub repository.

Setup

- We will break you into pairs, so choose one person as the `Owner` and one as the `Collaborator`
- Log into GitHub as the `Owner

- Navigate to the `Owner`’s training repository (e.g., `training_jones`)

Then, have the Owner visit their training repository created earlier, and visit the *Settings* page, and select the *Manage access* screen, and add the username of your Collaborator in the box.



Invite a collaborator to **training_jones**

Jeanette Clark
jeanetteclark

Manage access

You haven't invited any collaborators yet

Once the collaborator has been added, they should check their email for an invitation from GitHub, and click on the acceptance link, which will enable them to collaborate on the repository.

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.

5.3.1 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).

5.3.2 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.

5.3.3 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.

5.3.4 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!**

5.3.5 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.

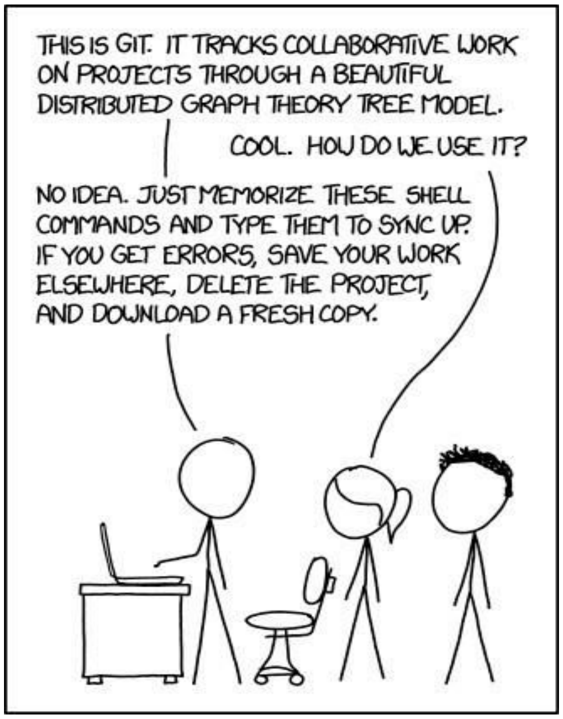
5.3.6 Step 6: Collaborator pull

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 `Collborator`, and then repeat the steps described above:

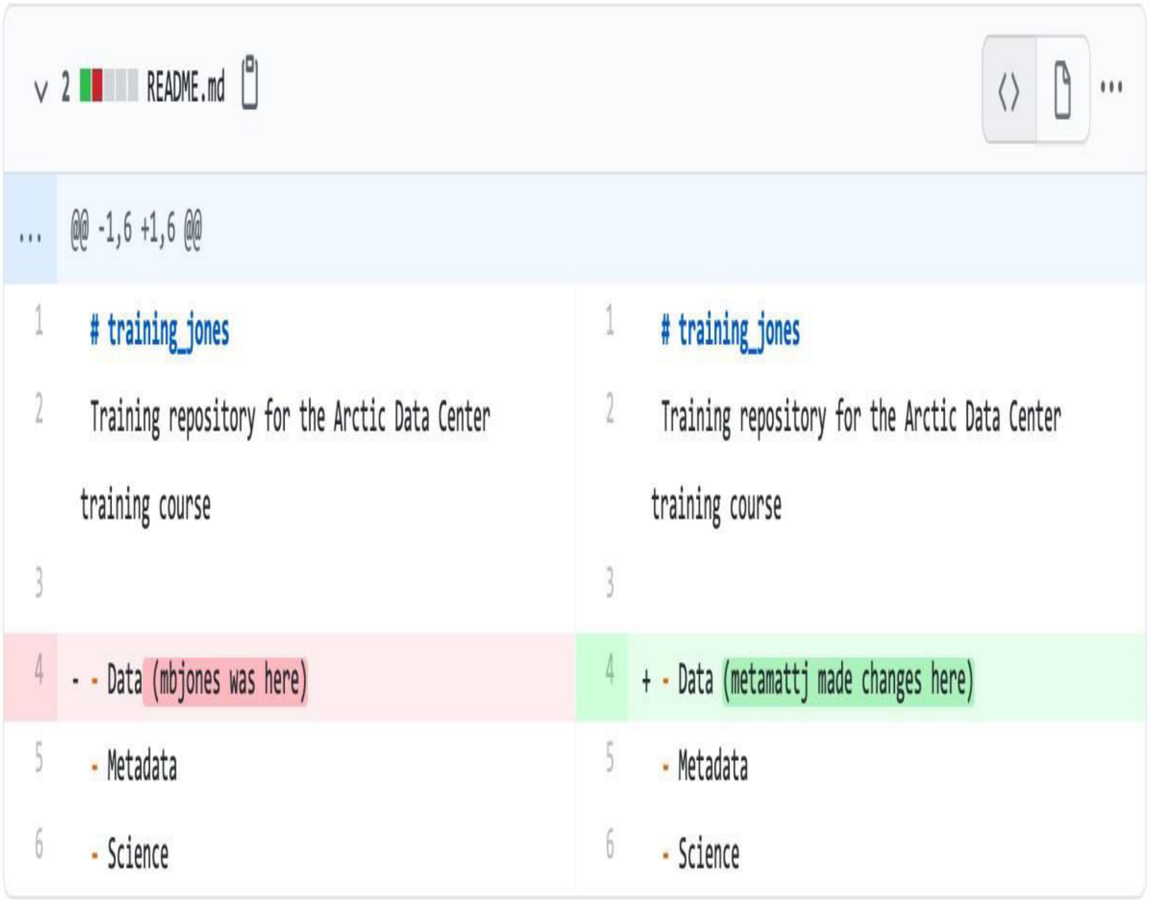
- Step 0: Setup permissions for your collaborator
- Step 1: Collaborator clones the Owner repository
- Step 2: Collaborator Edits the README file
- Step 3: Collaborator commits and pushes the file to GitHub
- Step 4: Owner pulls the changes that the Collaborator made
- Step 5: Owner edits, commits, and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


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

5.4 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.



```
v 2 README.md
@@ -1,6 +1,6 @@
1  # training_jones                                  1  # training_jones
2  Training repository for the Arctic Data Center    2  Training repository for the Arctic Data Center
   training course                                      training course
3                                                   3
4  - - Data (mbjones was here)                      4  + - Data (metamattj made changes here)
5  - Metadata                                       5  - Metadata
6  - Science                                        6  - Science
```

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.

5.5 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

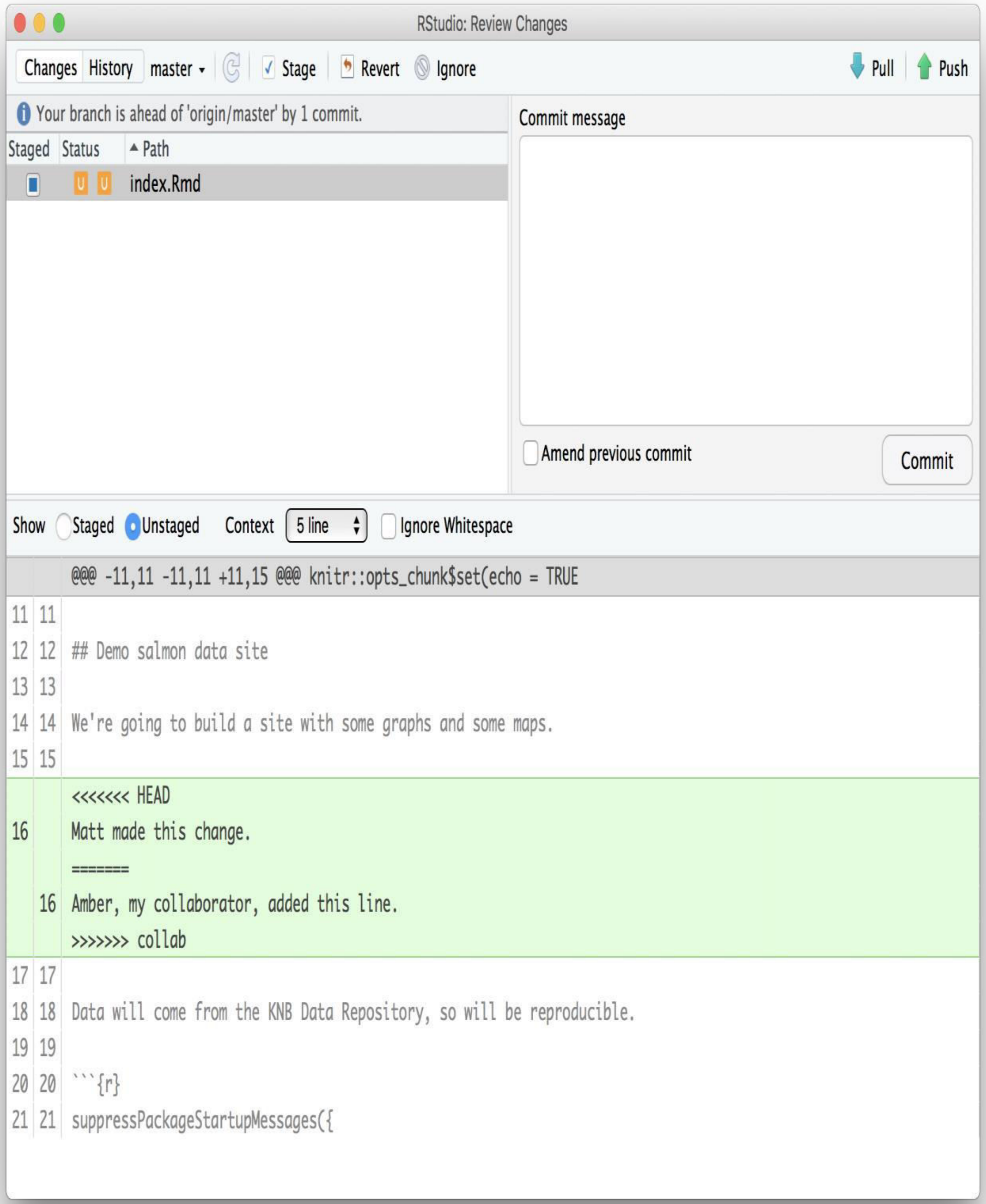
The simplest way to resolve a conflict, given that you know whose version of the file you want to keep, is to use the commandline `git` program to tell git to use either *your* changes (the person doing the merge), or *their* changes (the other collaborator).

- 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 command line. 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 `pull`ed 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.

Staged	Status	Path
■	U U	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.

<<<<<<< HEAD
Matt made this change.
=======
Amber, my collaborator, added this line.
>>>>>>> collab

Data will come from the KNB Data Repository, so will be reproducible.

```{r}
suppressPackageStartupMessages({
~~~

To resolve the conflicts, 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.

5.5.1 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.

5.5.1.1 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 repository 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.

5.5.1.2 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.

5.5.1.3 Collaborator makes a change and commits on the same line

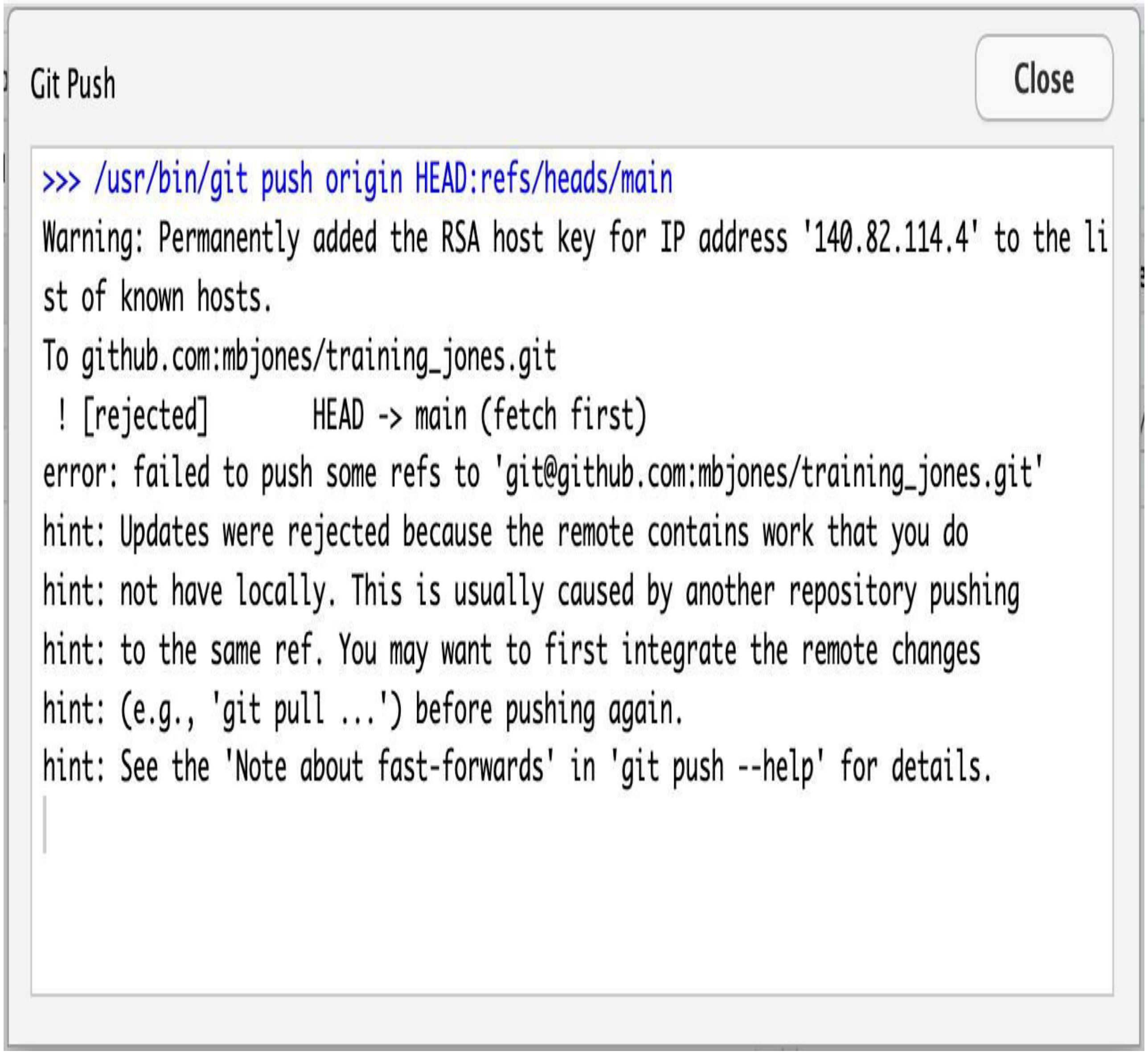
Now the collaborator also makes changes to the same (line 4) of the README.md file in their RStudio copy of the project, adding their name to the line. They then commit. At this point, both the owner and collaborator have committed changes based on their shared version of the README.md file, but neither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

5.5.1.4 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.

5.5.1.5 Owner pushes their changes and gets an error

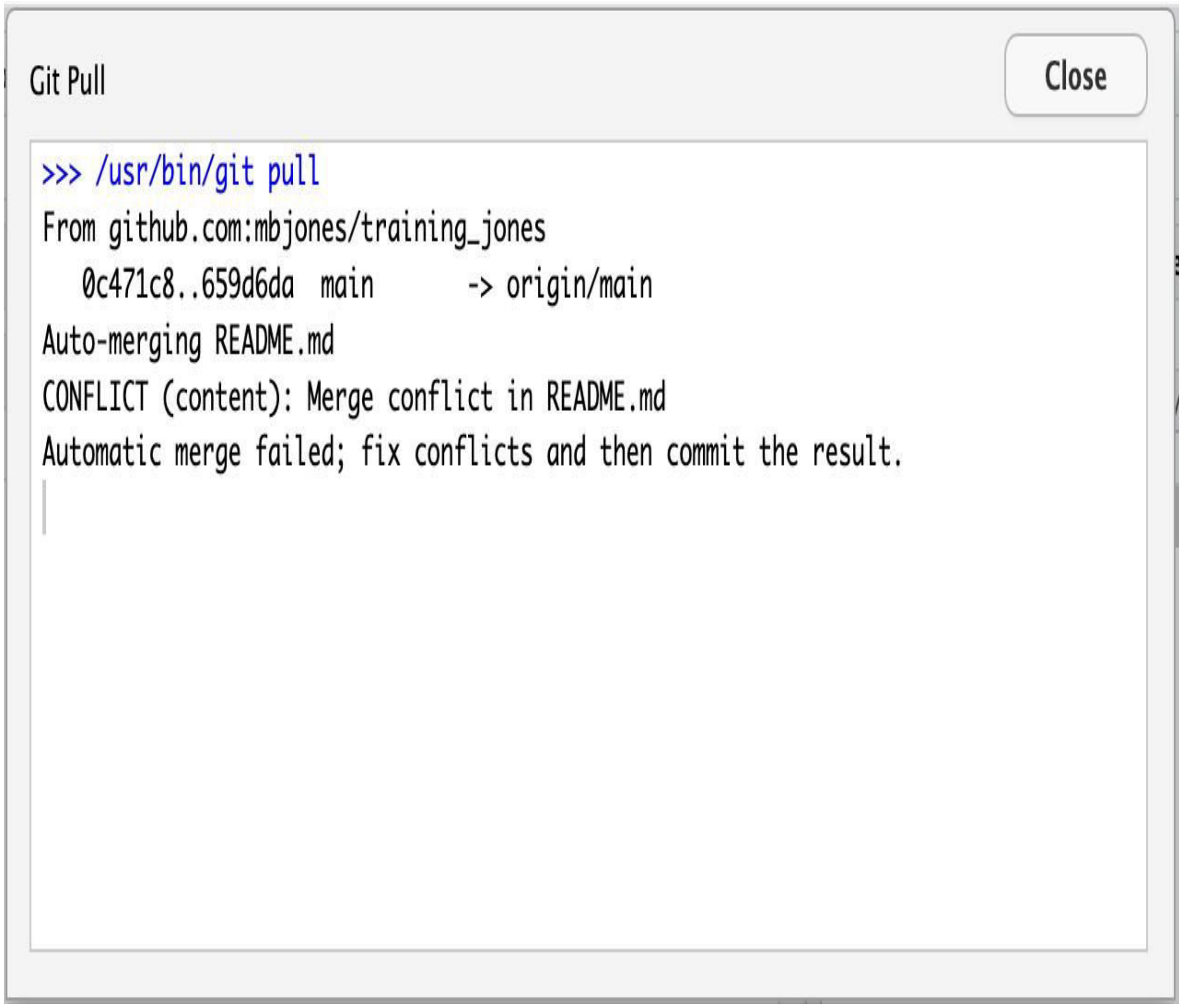
At this point, the owner tries to push their change to the repository, which triggers an error from GitHub. While the error message is long, it basically tells you everything needed (that the owner's repository doesn't reflect the changes on GitHub, and that they need to *pull* before they can push).



```
>>> /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.
```

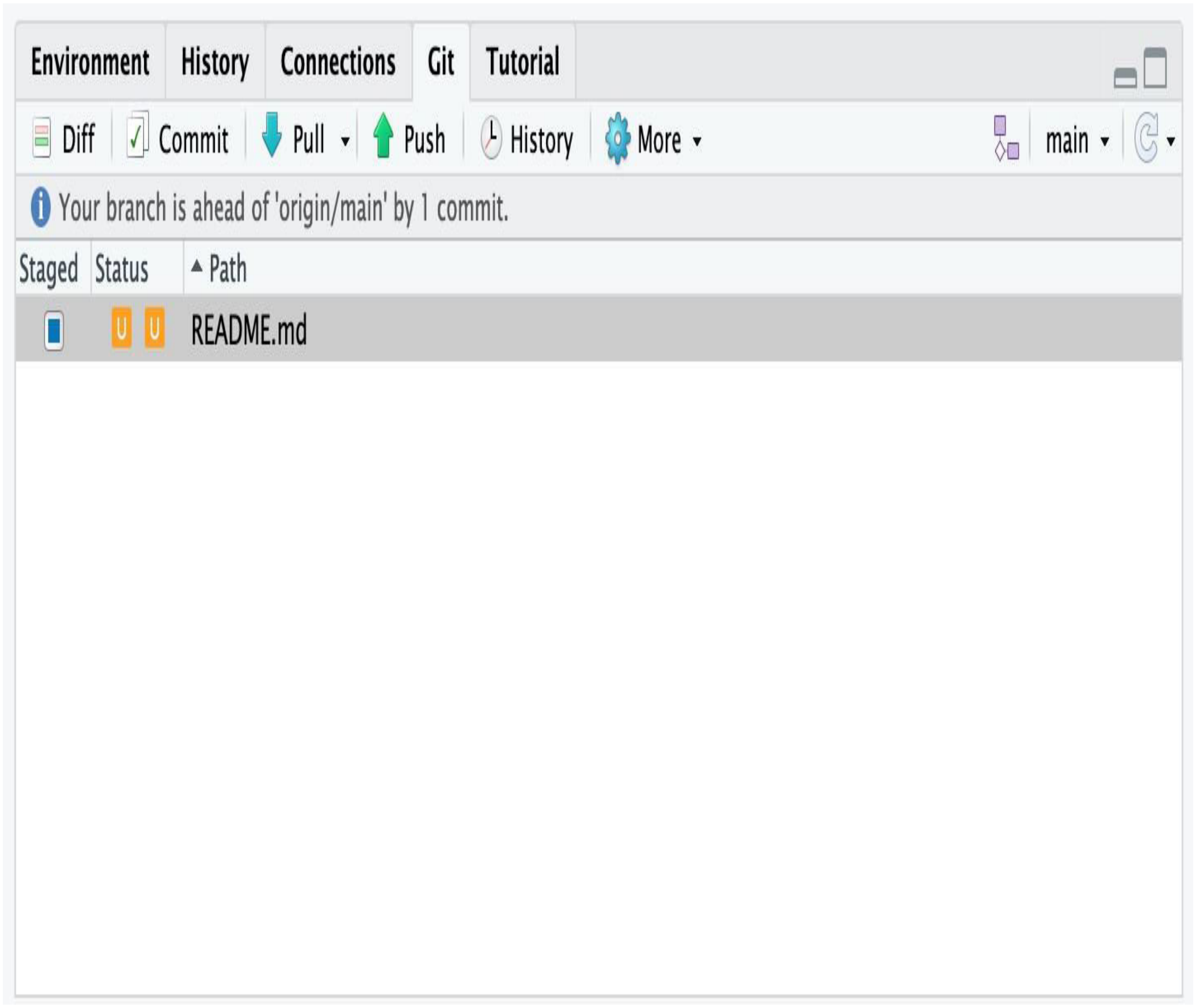
5.5.1.6 Owner pulls from GitHub to get Collaborator changes

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.



```
>>> /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.
```

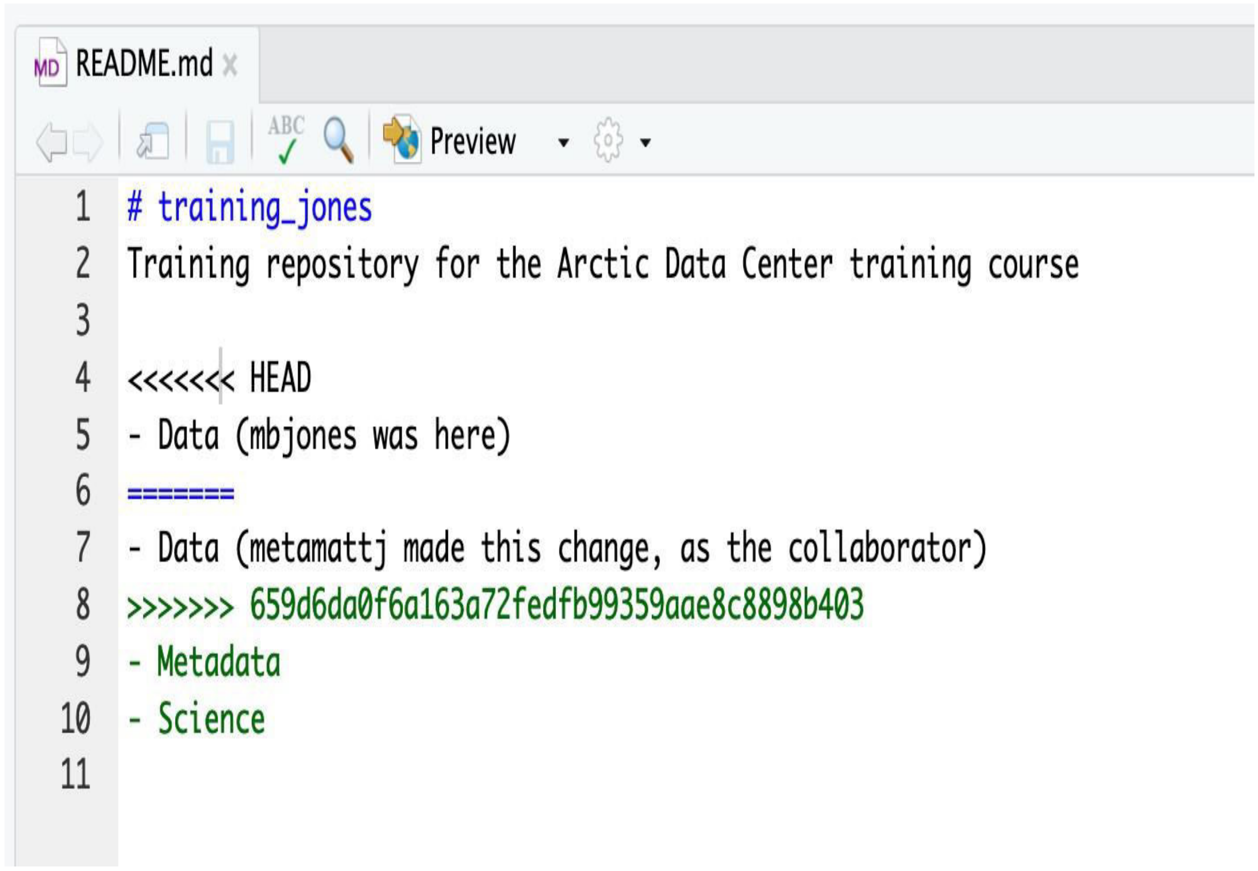
In the Git pane of RStudio, the file is also flagged with an orange ‘U’, which stands for an unresolved merge conflict.



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

5.5.1.7 Owner edits the file to resolve the conflict

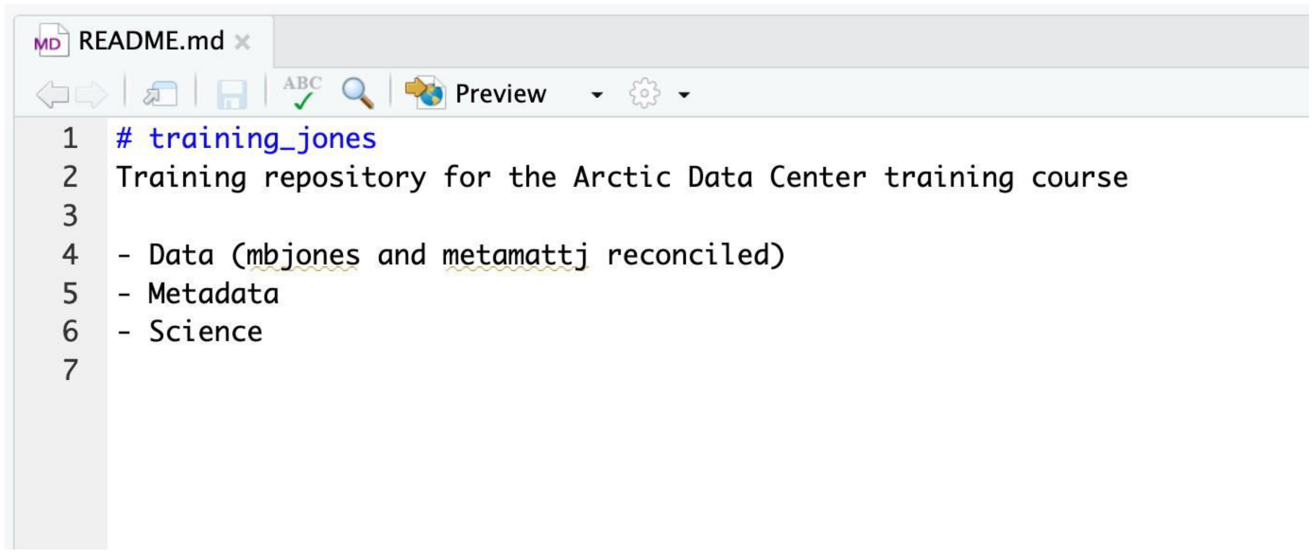
To resolve the conflict, the Owner now needs to edit the file. Again, as indicated above, git has flagged the locations in the file where a conflict occurred with <<<<<<<, =======, and >>>>>>>. The Owner should edit the file, merging whatever changes are appropriate until the conflicting lines read how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


README.md

Preview

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

<<<<<<< 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.

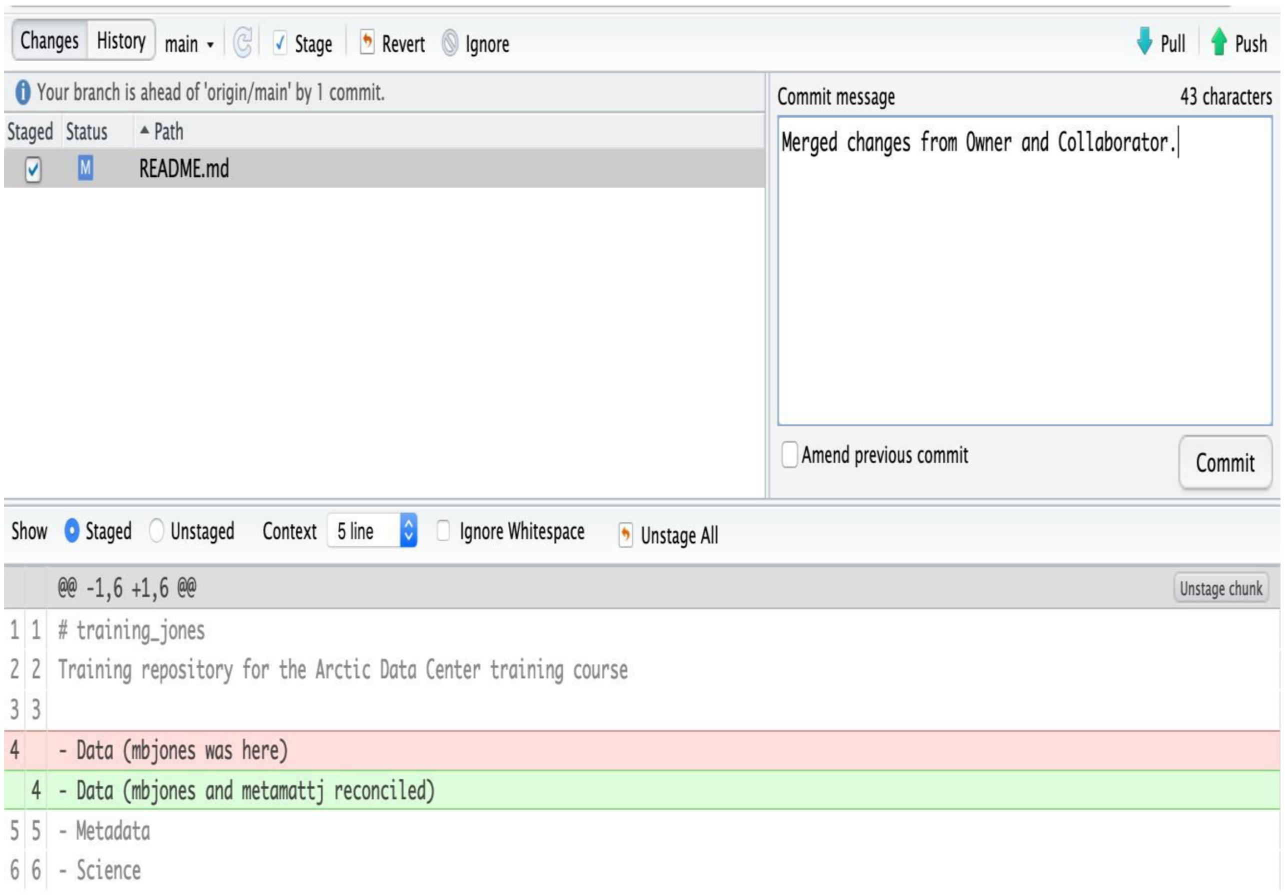

README.md

Preview

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

- Data (mbjones and metamattj reconciled)
- Metadata
- Science

```

5.5.1.8 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.

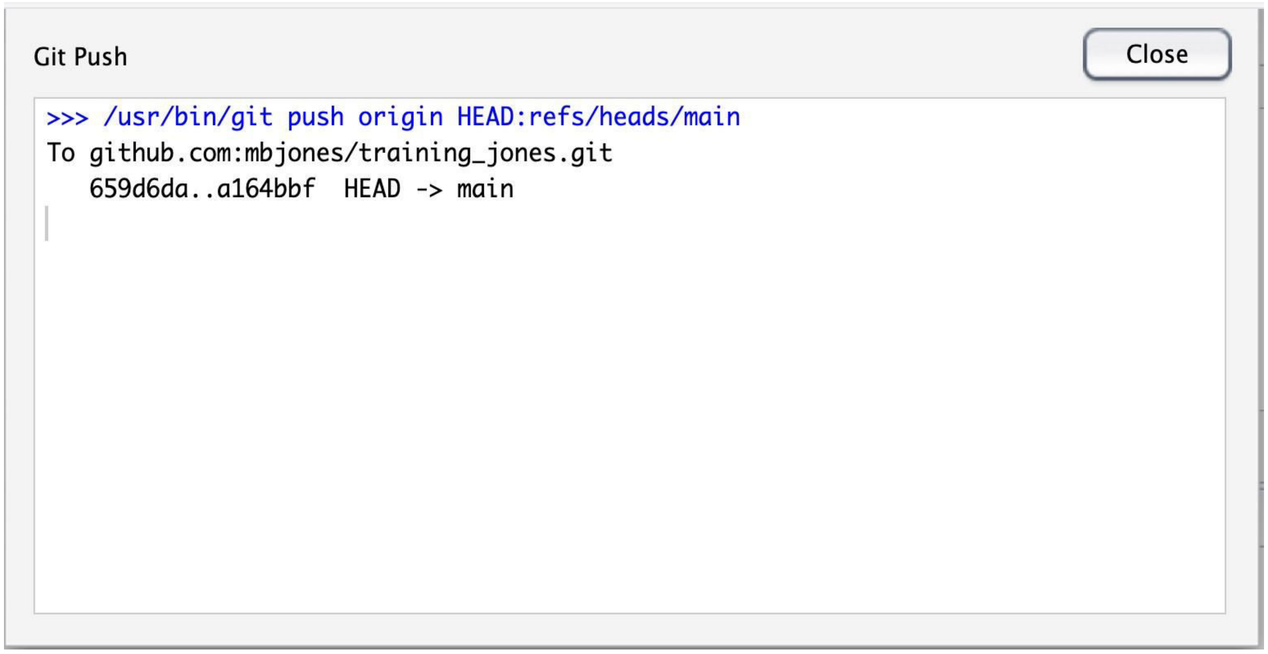


Merged changes from Owner and Collaborator.

```
@@ -1,6 +1,6 @@
# training_jones
Training repository for the Arctic Data Center training course

- Data (mbjones was here)
- Data (mbjones and metamattj reconciled)
- Metadata
- Science
```

5.5.1.9 Owner pushes the resolved changes to GitHub

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



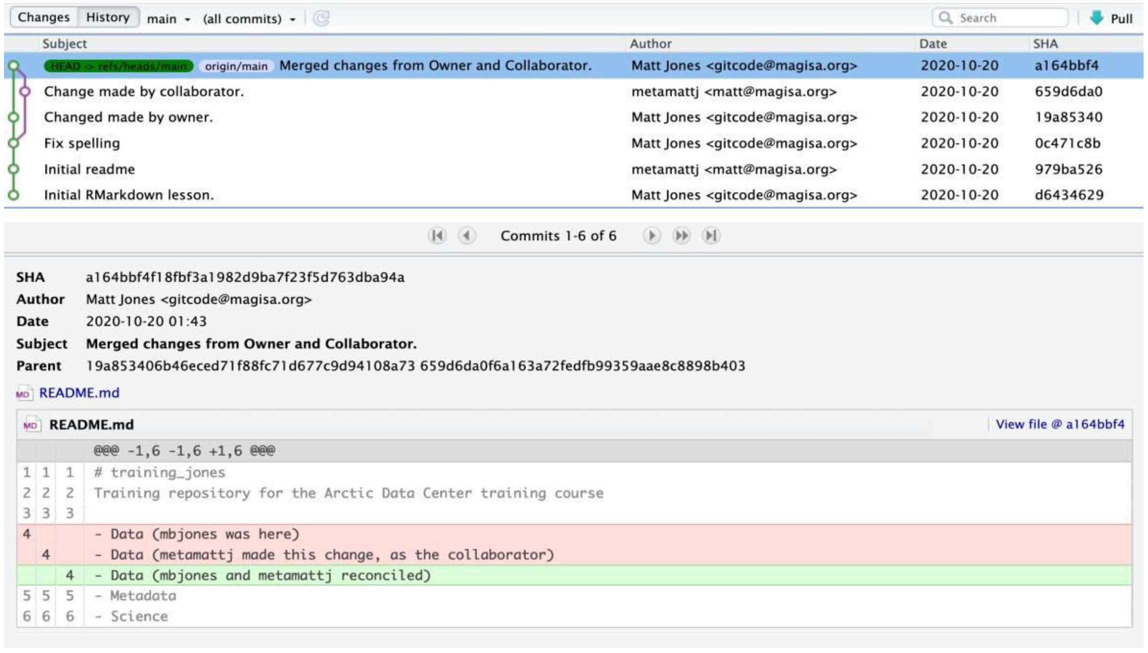
```
>>> /usr/bin/git push origin HEAD:refs/heads/main
To github.com:mbjones/training_jones.git
   659d6da..a164bbf  HEAD -> main
```

5.5.1.10 Collaborator pulls the resolved changes from GitHub

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

5.5.1.11 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.



Subject	Author	Date	SHA
HEAD -> refs/heads/main origin/main Merged changes from Owner and Collaborator.	Matt Jones <gitcode@magisa.org>	2020-10-20	a164bbf4
Change made by collaborator.	metamattj <matt@magisa.org>	2020-10-20	659d6da0
Changed made by owner.	Matt Jones <gitcode@magisa.org>	2020-10-20	19a85340
Fix spelling	Matt Jones <gitcode@magisa.org>	2020-10-20	0c471c8b
Initial readme	metamattj <matt@magisa.org>	2020-10-20	979ba526
Initial RMarkdown lesson.	Matt Jones <gitcode@magisa.org>	2020-10-20	d6434629

Commits 1-6 of 6

SHA	a164bbf4f18fbf3a1982d9ba7f23f5d763dba94a
Author	Matt Jones <gitcode@magisa.org>
Date	2020-10-20 01:43
Subject	Merged changes from Owner and Collaborator.
Parent	19a853406b46eced71f88fc71d677c9d94108a73 659d6da0f6a163a72fedfb99359aae8c8898b403

README.md

README.md — View file @ a164bbf4

```
        @@@ -1,6 -1,6 +1,6 @@@
1 1 1   # training_jones
2 2 2   Training repository for the Arctic Data Center training course
3 3 3
4       - Data (mbjones was here)
  4     - Data (metamattj made this change, as the collaborator)
    4   - Data (mbjones and metamattj reconciled)
5 5 5   - Metadata
6 6 6   - Science
```

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

5.6 Workflows to avoid merge conflicts

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

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

A good workflow is encapsulated as follows:

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

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

Experiment No. 09

Aim: Reset and Revert

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 b764644 File
with three lines 7c709f0 File
with two lines
9ef9173 File with one line
```

Programming and development

- **Red Hat Developers Blog**
- **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
b764644bad524b804577684bf74e7bca3117f554.

# 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:
#       modified:   file1.txt
#
```

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
f33ae68 C1
5043e79 C0
```

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
Applying: C5
```

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
f33ae68 C1
```

```
5043e79 C0

$ git log --oneline feature
c4533a5 C5
64f2047 C3
6a92e7a C4
259bf36 C2
f33ae68 C1
5043e79 C0
```

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 to 79768b
c4533a5 HEAD@{1}: rebase finished: returning to
refs/heads/feature
c4533a5 HEAD@{2}: rebase: C5
64f2047 HEAD@{3}: rebase: C3
6a92e7a HEAD@{4}: rebase: checkout master
79768b8 HEAD@{5}: checkout: moving from feature to feature
79768b8 HEAD@{6}: commit: C5
000f9ae HEAD@{7}: checkout: moving from master to feature
6a92e7a HEAD@{8}: commit: C4
259bf36 HEAD@{9}: checkout: moving from 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.

-A Project report

on

"SneakFreak - Sneakers"

with

Source Code Management

(CS181)

Submitted by

Team Member 1	**Priyanshu Vashishtha**	**2110992074**
Team Member 2	**Aashirwad**	**2110992099**
Team Member 3	**Kuber Bansal**	**2110992020**
Team Member 4	**Jasjot Singh**	**2110992022**

Department of Computer Science & Engineering

Chitkara University Institute of Engineering and Technology, Punjab

Jan- June

(2021-22)

Institute/School Name	**Chitkara University Institute of Engineering and Technology**		
Department Name	**Department of Computer Science & Engineering**		
Programme Name	**Bachelor of Engineering (B.E.), Computer Science & Engineering**		
Course Name	**Source Code Management**	Session	**2021-22**
Course Code	**CS181**	Semester/Batch	**2nd/2021**
Vertical Name	**Zeta**	Group No	G27

Course Coordinator **Dr. Neeraj Singla**

Faculty Name **Dr. Sachendra Singh Chauhan**

Submission

Name: JASJOT SINGH

Date: 23.05.2022

CHITKARA UNIVERSITY

Table of Content

1. Version control with Git

What is GIT and why is it used?

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

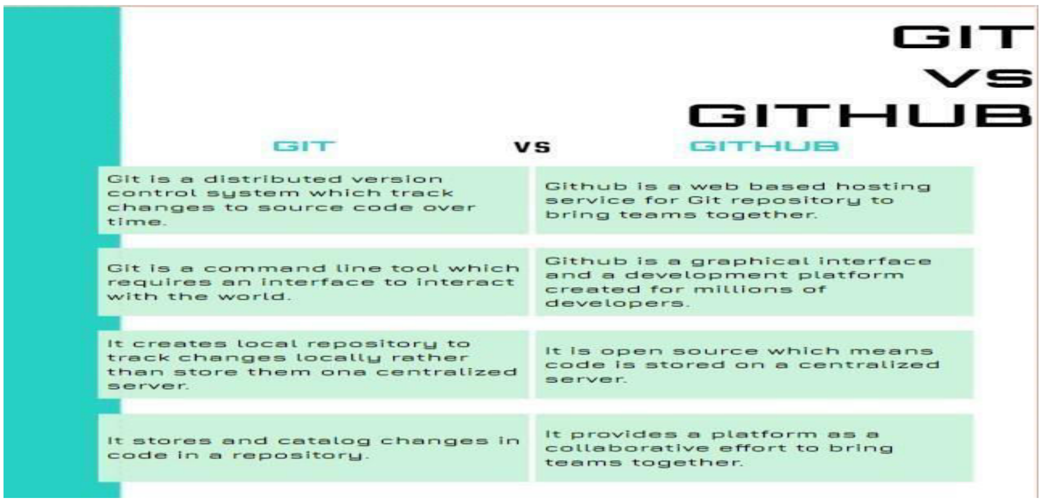
Git is a speedy and efficient distributed VCS tool that can handle projects of any size, from small to very large ones. Git provides cheap local branching, convenient staging areas, and multiple workflows. It is free, open-source software that lowers the cost because developers can use Git without paying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

Git is an example of a distributed version control system (DVCS) (hence Distributed Version Control System).

What is GITHUB?

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

What is the difference between GIT and GITHUB?



GIT VS GITHUB

GIT	GITHUB
Git is a distributed version control system which track changes to source code over time.	Github is a web based hosting service for Git repository to bring teams together.
Git is a command line tool which requires an interface to interact with the world.	Github is a graphical interface and a development platform created for millions of developers.
It creates local repository to track changes locally rather than store them ona centralized server.	It is open source which means code is stored on a centralized server.
It stores and catalog changes in code in a repository.	It provides a platform as a collaborative effort to bring teams together.

What is Repository?

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

What is Version Control System (VCS)?

A version control system is a tool that helps you manage “versions” of your code or changes to your code while working with a team over remote distances. Version control keeps track of every modification in a special kind of database that is accessible to the version control software. Version control software (VCS) helps you revert back to an older version just in case a bug or issue is introduced to the system or fixing a mistake without disrupting the work of other team members.

Types of VCS

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

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

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

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

2. Problem Statement

Looking and buying new shoes , sneakers can sometimes become difficult and inaccessible owing to repetitive and distracting advertisements, and occasionally due to the paid policy of these programs.
These websites' user interfaces and user experiences are both poor (UX). Apart from this there are many websites on which you can look for Sneakers but this only factor makes the decision more confusing.

3. Objective

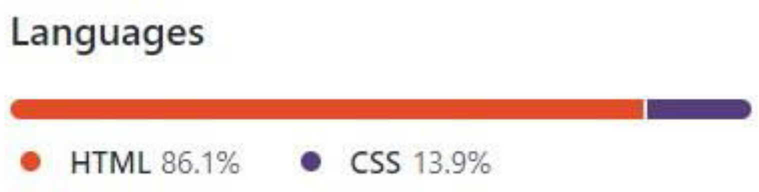
PROJECT NAME: SneakFreak

Our goal is to create a web app for our users that will provide them with a variety and trending sneakers for all categories whether it is male, female , kids. Our prime objective is to create a lightweight online app with a very basic yet classic UI/UX that allows our customers to search for their best and Fitting Shoes/Sneakers without any sort of disturbances like advertisements.

4. Resources Required.

Frontend – HTML5, CSS

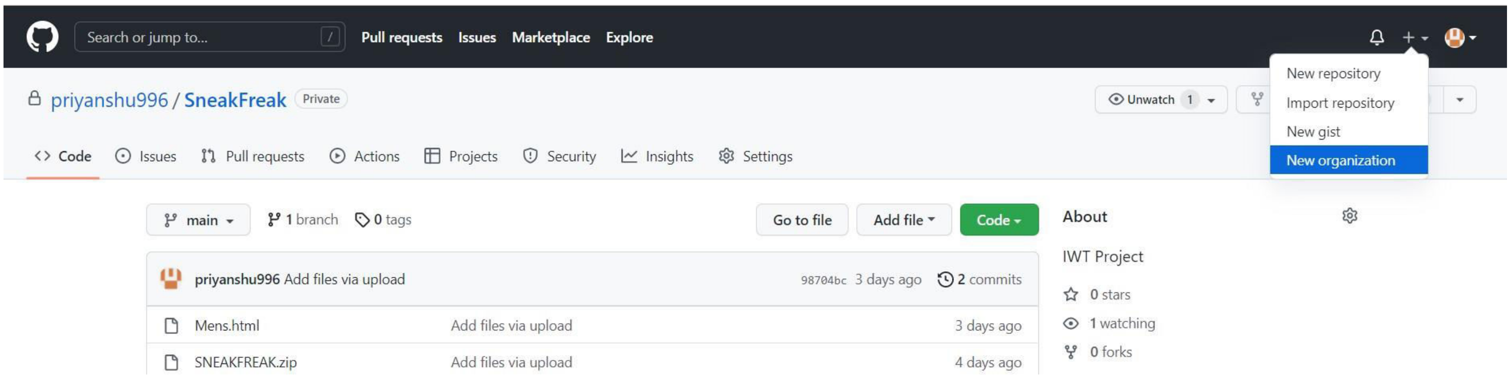
Backend – NodeJS



5. Concepts , Commands, Workflow and Discussions.

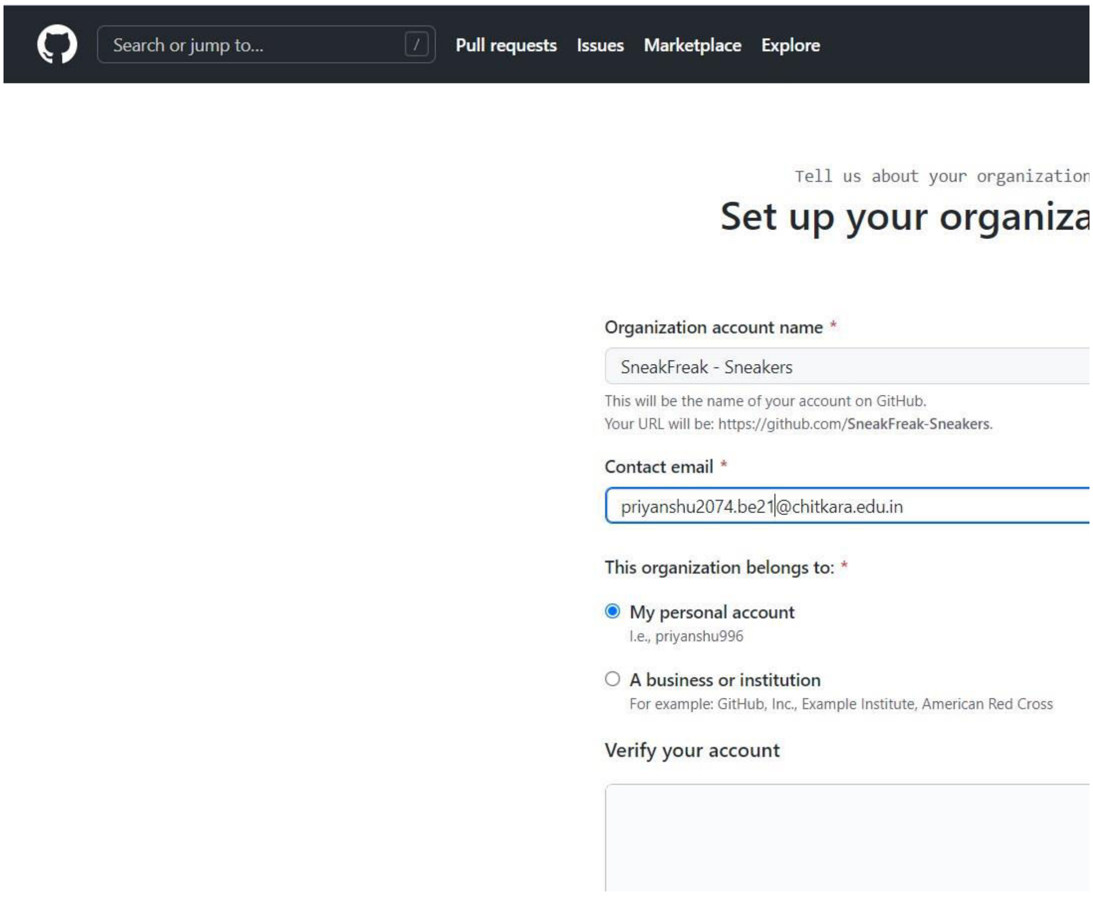
Aim: Create a distributed Repository and add members in project team

- Login to your GitHub account and you will land on the homepage as shown below. Click on the button shown in the menu bar and then click on New Organization.



priyanshu996 Add files via upload 98704bc 3 days ago 2 commits
Mens.html Add files via upload 3 days ago
SNEAKFREAK.zip Add files via upload 4 days ago

- Set Up Your Organization. Fill Your Organization's Name and Other Details. .



Tell us about your organization

Set up your organiza

Organization account name *

SneakFreak - Sneakers

This will be the name of your account on GitHub.
Your URL will be: https://github.com/SneakFreak-Sneakers.

Contact email *

priyanshu2074.be21@chitkara.edu.in

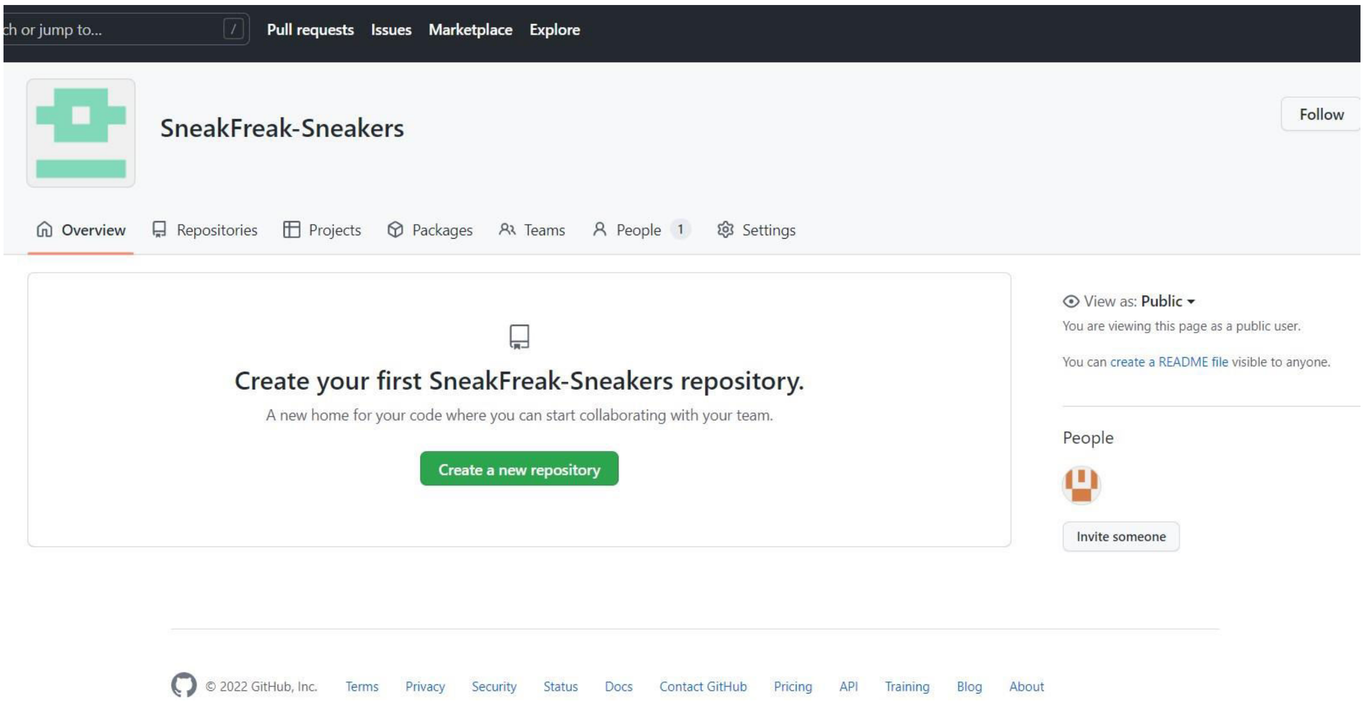
This organization belongs to: *

(●) My personal account
I.e., priyanshu996

( ) A business or institution
For example: GitHub, Inc., Example Institute, American Red Cross

Verify your account

- After creating the repository, we have to create a Repository.



SneakFreak-Sneakers

Create your first SneakFreak-Sneakers repository.

A new home for your code where you can start collaborating with your team.

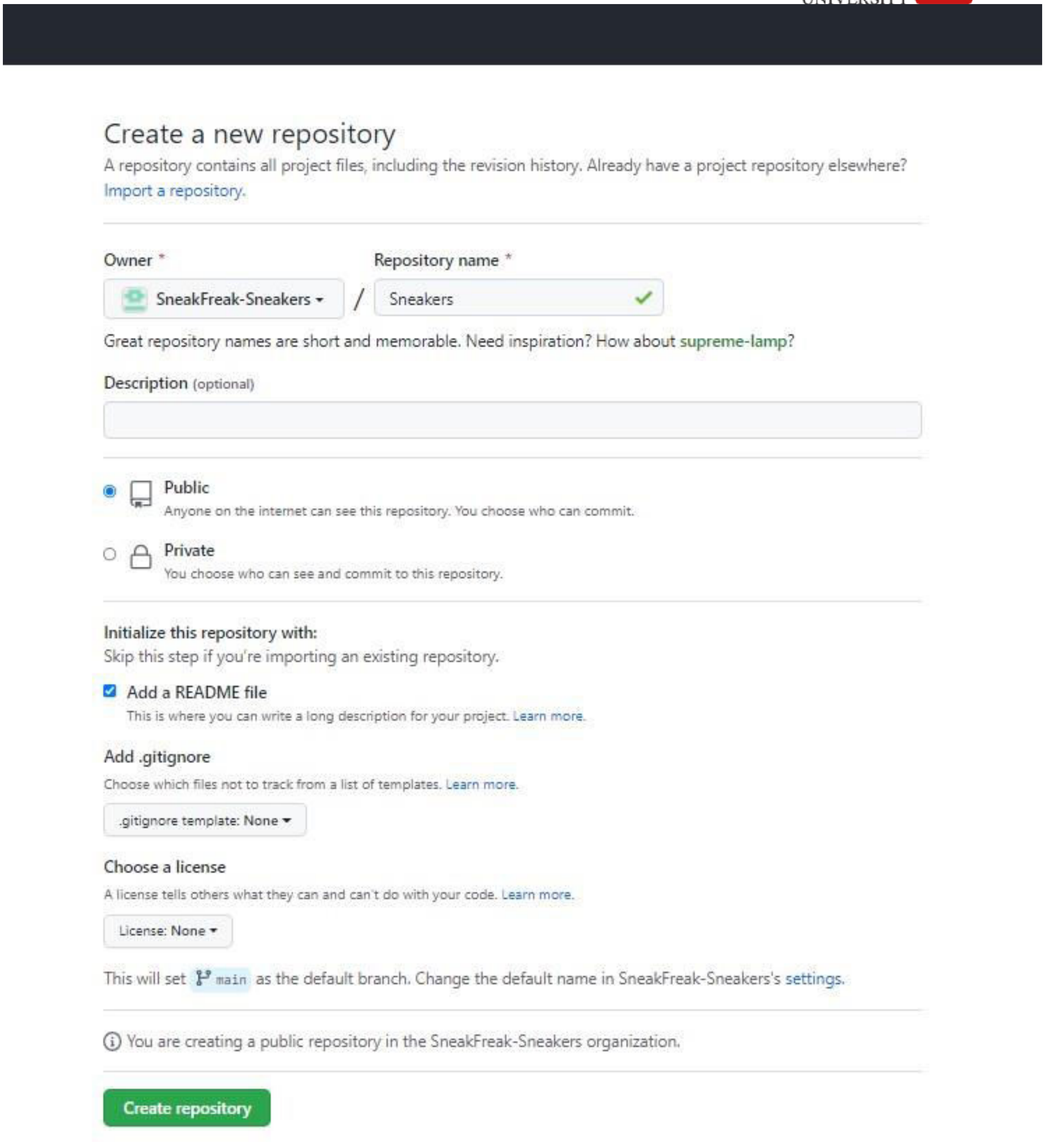
View as: Public

You are viewing this page as a public user.

You can create a README file visible to anyone.

People

- Create a New Repo by Pressing the New repository button. Fill in the required details.



Create a new repository

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

Owner * SneakFreak-Sneakers / Repository name * Sneakers ✓

Great repository names are short and memorable. Need inspiration? How about supreme-lamp?

Description (optional)

● Public
Anyone on the internet can see this repository. You choose who can commit.

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

Initialize this repository with:
Skip this step if you're importing an existing repository.

☑ Add a README file
This is where you can write a long description for your project. Learn more.

Add .gitignore
Choose which files not to track from a list of templates. Learn more.

.gitignore template: None

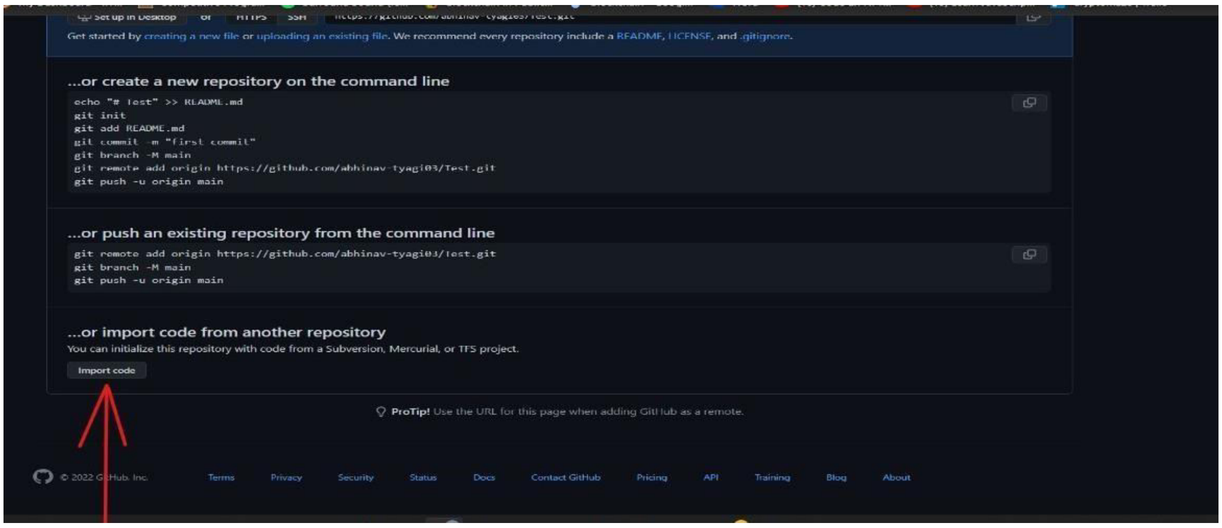
Choose a license
A license tells others what they can and can't do with your code. Learn more.

License: None

This will set `main` as the default branch. Change the default name in SneakFreak-Sneakers's settings.

ⓘ You are creating a public repository in the SneakFreak-Sneakers organization.

- If you want to import code from an existing repository select the import code option.



Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# Test" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/abhinav-tyagi03/Test.git
git push -u origin main
```

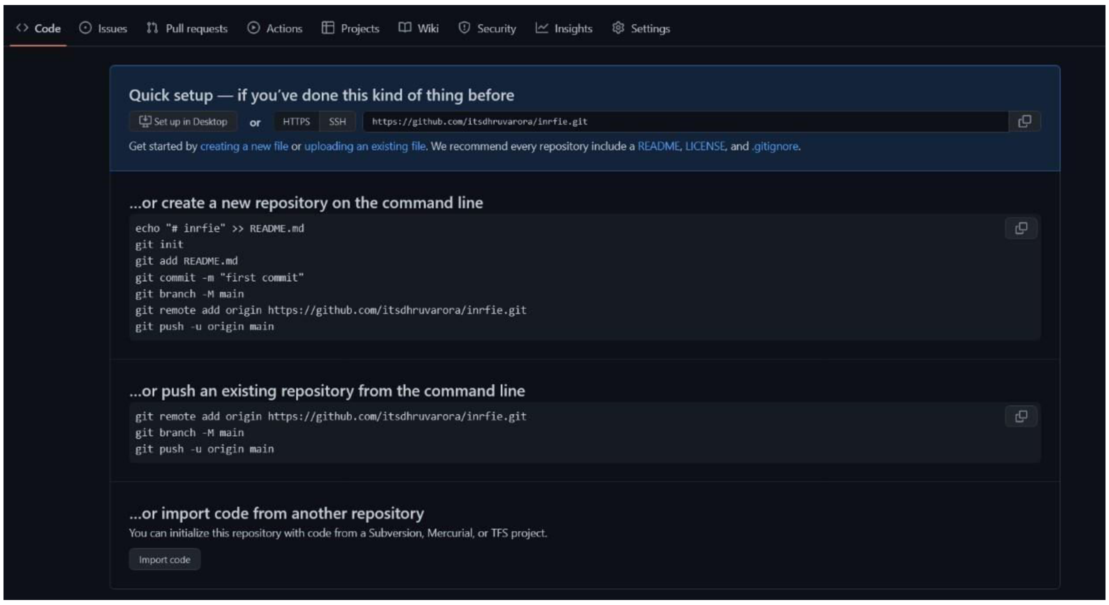
...or push an existing repository from the command line

```
git remote add origin https://github.com/abhinav-tyagi03/Test.git
git branch -M main
git push -u origin main
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

- To create a new file or upload an existing file into your repository select the option in the following box.



Quick setup — if you've done this kind of thing before

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# inrfie" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/itsdhruvarora/inrfie.git
git push -u origin main
```

...or push an existing repository from the command line

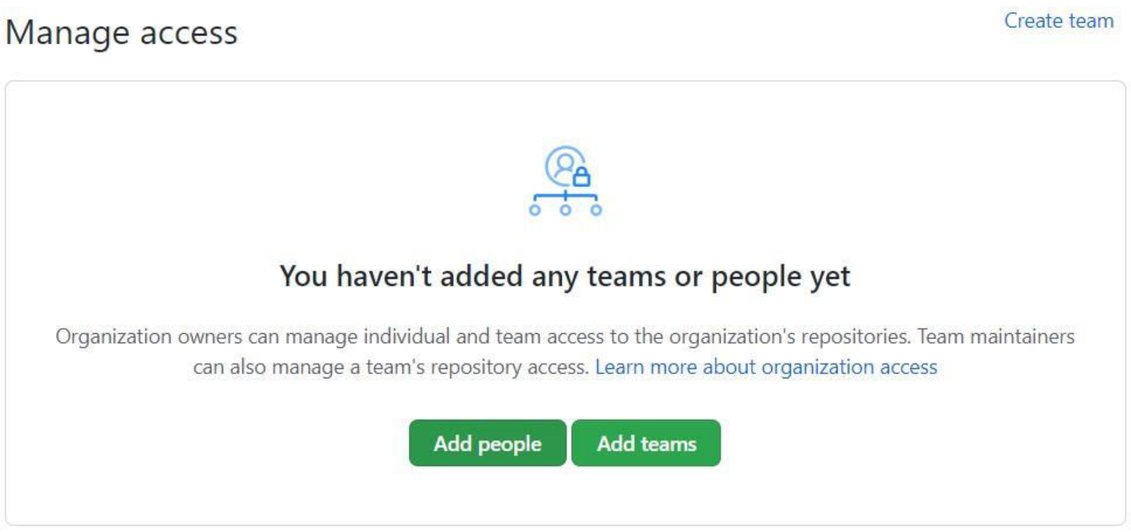
```
git remote add origin https://github.com/itsdhruvarora/inrfie.git
git branch -M main
git push -u origin main
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

- Now, you have created your repository successfully.
- To add members to your repository, open your Organization and select People option in the navigation bar.
- Click on Collaborators option under the access tab.

- To add members click on the add people option and search the id of your respective team member.

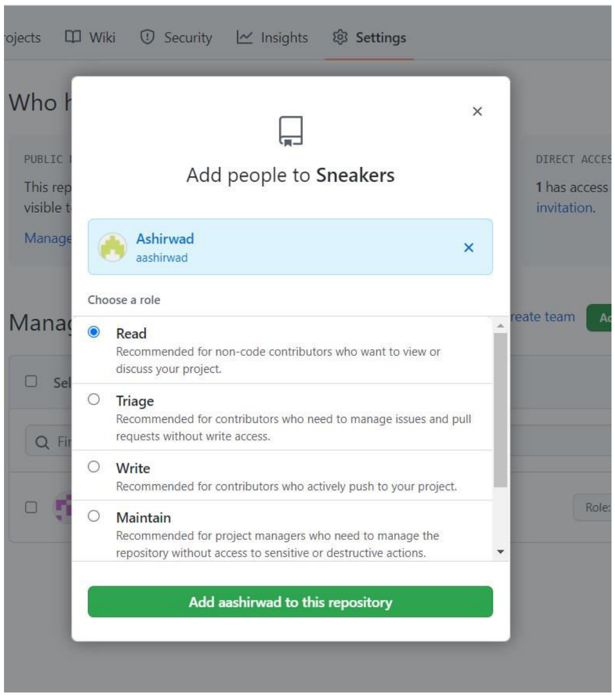


Manage access

Create team

You haven't added any teams or people yet

Organization owners can manage individual and team access to the organization's repositories. Team maintainers can also manage a team's repository access. Learn more about organization access



Add people to Sneakers

Ashirwad
aashirwad

Choose a role

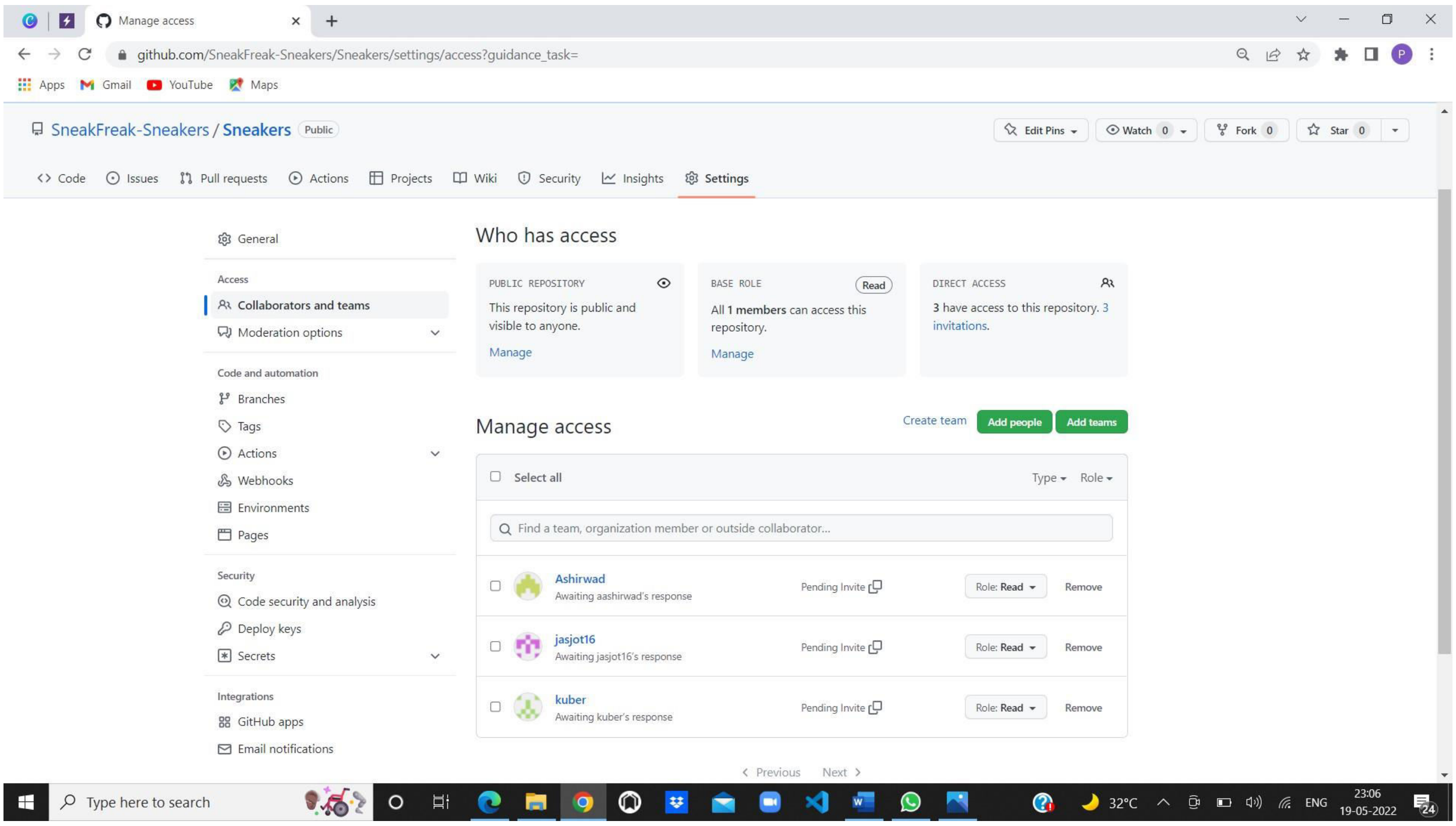
- [✓ Read] Recommended for non-code contributors who want to view or discuss your project.
- Triage Recommended for contributors who need to manage issues and pull requests without write access.
- Write Recommended for contributors who actively push to your project.
- Maintain Recommended for project managers who need to manage the repository without access to sensitive or destructive actions.

- To accept the invitation from your team member, open your email registered with GitHub.

- You will receive an invitation mail from the repository owner. Open the email and click on accept invitation.

- You will be redirected to GitHub where you can either select to accept or decline the invitation.

- Next, Open the desired Repository in the Organisation. Look and click on Settings -> Collaborators and Teams. Here you can Manage the roles of each collaborator.



Manage access

github.com/SneakFreak-Sneakers/Sneakers/settings/access?guidance_task=

SneakFreak-Sneakers / **Sneakers** Public

Code · Issues · Pull requests · Actions · Projects · Wiki · Security · Insights · **Settings**

General

Access
- Collaborators and teams
- Moderation options

Code and automation
- Branches
- Tags
- Actions
- Webhooks
- Environments
- Pages

Security
- Code security and analysis
- Deploy keys
- Secrets

Integrations
- GitHub apps
- Email notifications

Who has access

PUBLIC REPOSITORY
This repository is public and visible to anyone.
Manage

BASE ROLE (Read)
All 1 **members** can access this repository.
Manage

DIRECT ACCESS
3 have access to this repository. 3 invitations.

Manage access

Create team · Add people · Add teams

Select all · Type · Role

Find a team, organization member or outside collaborator...

Ashirwad Awaiting aashirwad's response	Pending Invite	Role: Read	Remove
jasjot16 Awaiting jasjot16's response	Pending Invite	Role: Read	Remove
kuber Awaiting kuber's response	Pending Invite	Role: Read	Remove

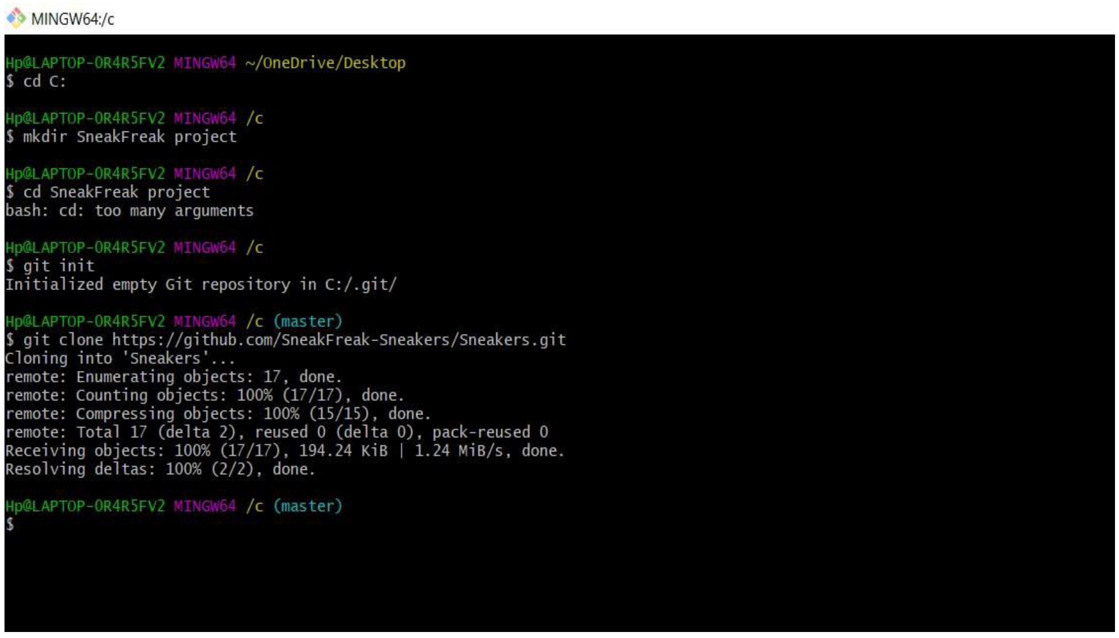
< Previous Next >

CHITKARA UNIVERSITY

Experiment No. 02

Aim: Open And Close a Pull Request

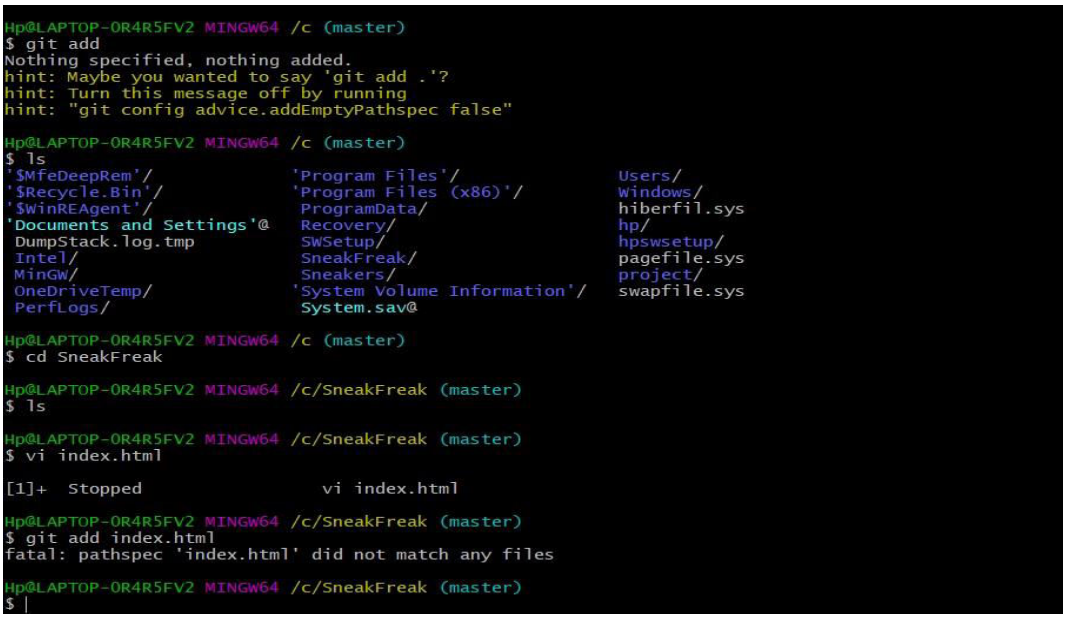
- To Open a Pull Request, First of All, it will be required to fork the repository and commit changes into your own.



MINGW64:/c

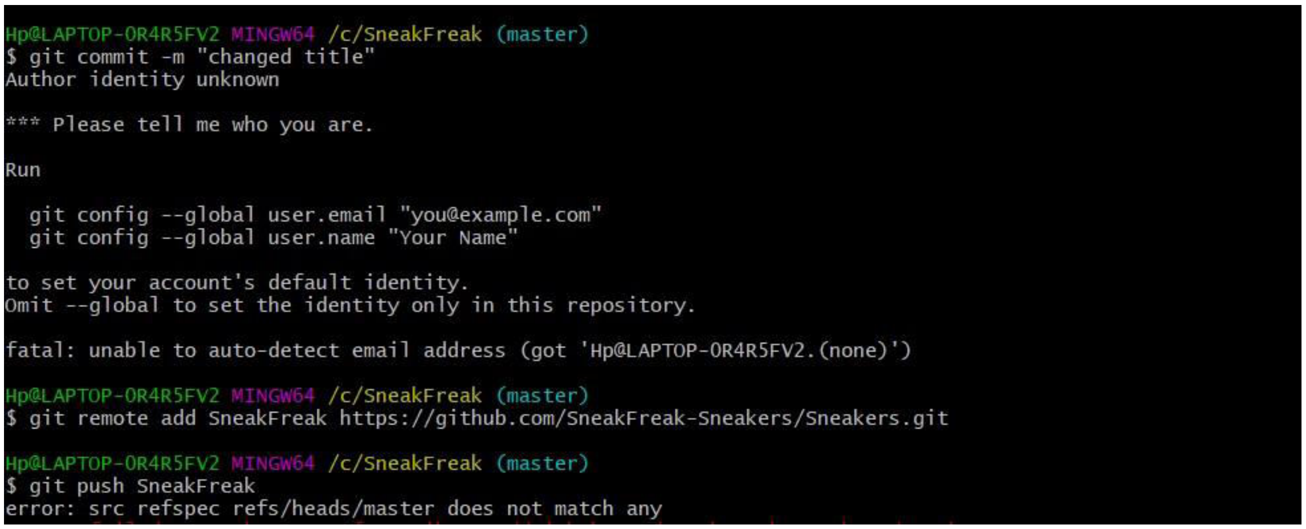
```
Hp@LAPTOP-0R4R5FV2 MINGW64 ~/OneDrive/Desktop
$ cd C:

Hp@LAPTOP-0R4R5FV2 MINGW64 /c
$ mkdir SneakFreak project

Hp@LAPTOP-0R4R5FV2 MINGW64 /c
$ cd SneakFreak project
bash: cd: too many arguments

Hp@LAPTOP-0R4R5FV2 MINGW64 /c
$ git init
Initialized empty Git repository in C:/.git/

Hp@LAPTOP-0R4R5FV2 MINGW64 /c (master)
$ git clone https://github.com/SneakFreak-Sneakers/Sneakers.git
Cloning into 'Sneakers'...
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 17 (delta 2), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (17/17), 194.24 KiB | 1.24 MiB/s, done.
Resolving deltas: 100% (2/2), done.

Hp@LAPTOP-0R4R5FV2 MINGW64 /c (master)
$
```

- Add and commit the changes to the local repository.



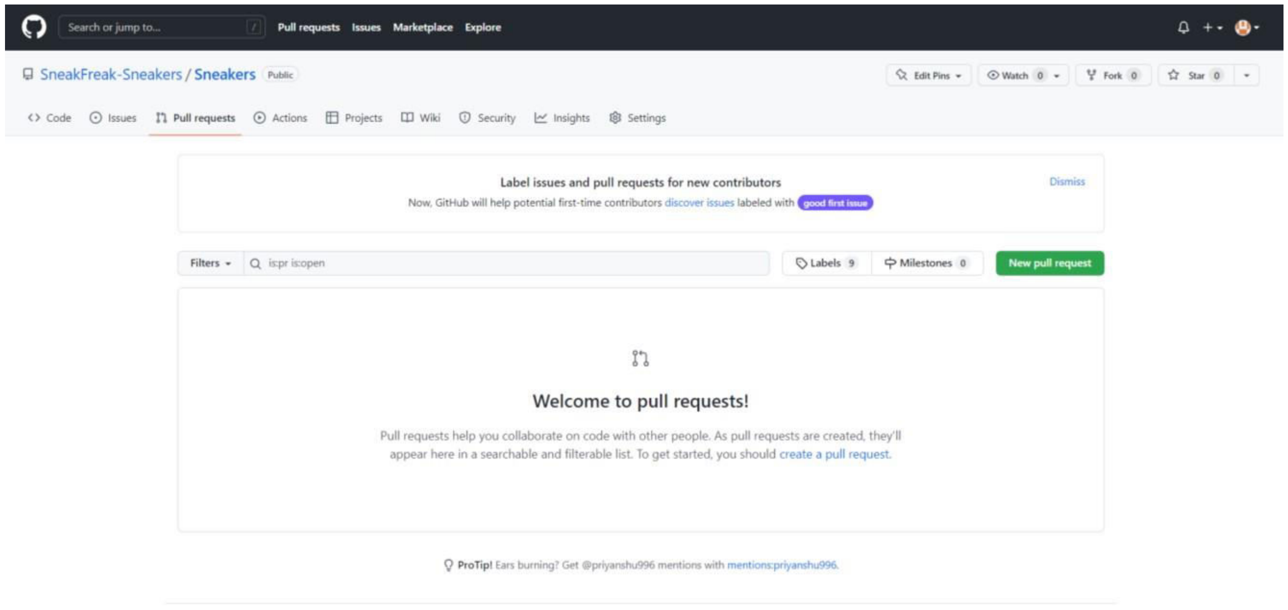
```
Hp@LAPTOP-OR4R5FV2 MINGW64 /c (master)
$ git add
Nothing specified, nothing added.
hint: Maybe you wanted to say 'git add .'?
hint: Turn this message off by running
hint: "git config advice.addEmptyPathspec false"

Hp@LAPTOP-OR4R5FV2 MINGW64 /c (master)
$ ls
'$MfeDeepRem'/              'Program Files'/              Users/
'$Recycle.Bin'/             'Program Files (x86)'/        Windows/
'$WinREAgent'/               ProgramData/                 hiberfil.sys
'Documents and Settings'@    Recovery/                    hp/
 DumpStack.log.tmp           SWSetup/                     hpswsetup/
 Intel/                      SneakFreak/                  pagefile.sys
 MinGW/                      Sneakers/                    project/
 OneDriveTemp/              'System Volume Information'/  swapfile.sys
 PerfLogs/                   System.sav@

Hp@LAPTOP-OR4R5FV2 MINGW64 /c (master)
$ cd SneakFreak

Hp@LAPTOP-OR4R5FV2 MINGW64 /c/SneakFreak (master)
$ ls

Hp@LAPTOP-OR4R5FV2 MINGW64 /c/SneakFreak (master)
$ vi index.html

[1]+  Stopped                 vi index.html

Hp@LAPTOP-OR4R5FV2 MINGW64 /c/SneakFreak (master)
$ git add index.html
fatal: pathspec 'index.html' did not match any files

Hp@LAPTOP-OR4R5FV2 MINGW64 /c/SneakFreak (master)
$ |
```

- Use git push origin branch name option to push the new branch to the main repository.



```
Hp@LAPTOP-0R4R5FV2 MINGW64 /c/SneakFreak (master)
$ git commit -m "changed title"
Author identity unknown

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'Hp@LAPTOP-0R4R5FV2.(none)')

Hp@LAPTOP-0R4R5FV2 MINGW64 /c/SneakFreak (master)
$ git remote add SneakFreak https://github.com/SneakFreak-Sneakers/Sneakers.git

Hp@LAPTOP-0R4R5FV2 MINGW64 /c/SneakFreak (master)
$ git push SneakFreak
error: src refspec refs/heads/master does not match any
```

- After pushing new branch GitHub will either automatically ask you to create a pull request or you can create your own pull request.
- To create your own pull request click on pull request option.



Label issues and pull requests for new contributors

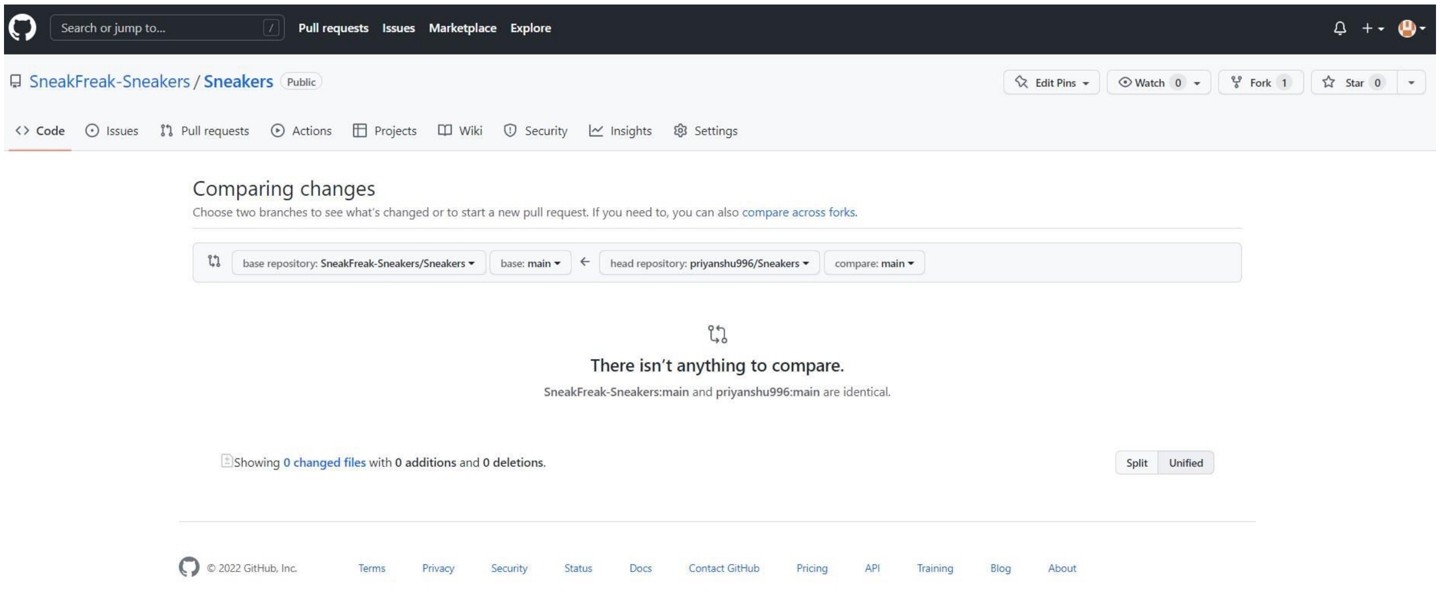
Now, GitHub will help potential first-time contributors discover issues labeled with good first issue

Welcome to pull requests!

Pull requests help you collaborate on code with other people. As pull requests are created, they'll appear here in a searchable and filterable list. To get started, you should create a pull request.

ProTip! Ears burning? Get @priyanshu996 mentions with mentions:priyanshu996.

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



SneakFreak-Sneakers / Sneakers Public

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base repository: SneakFreak-Sneakers/Sneakers ▾ | base: main ▾ | ← | head repository: priyanshu996/Sneakers ▾ | compare: main ▾

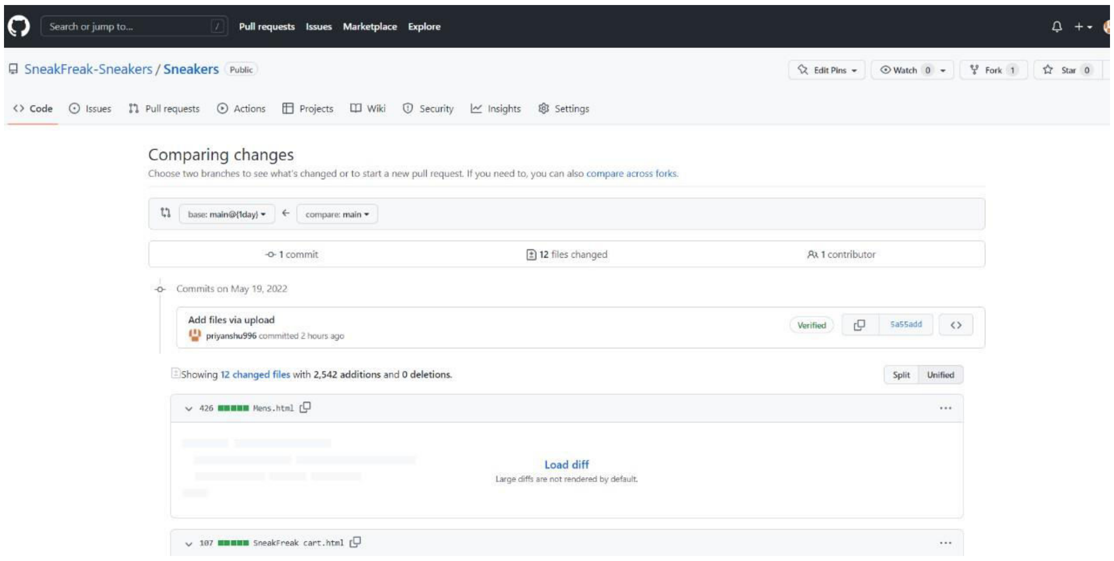
There isn't anything to compare.

SneakFreak-Sneakers:main and priyanshu996:main are identical.

Showing 0 changed files with 0 additions and 0 deletions.

© 2022 GitHub, Inc.

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



SneakFreak-Sneakers / Sneakers Public

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base: main@{1day} ▾ | ← | compare: main ▾

1 commit | 12 files changed | 1 contributor

Commits on May 19, 2022

Add files via upload
priyanshu996 committed 2 hours ago — Verified 5a55add

Showing 12 changed files with 2,542 additions and 0 deletions.

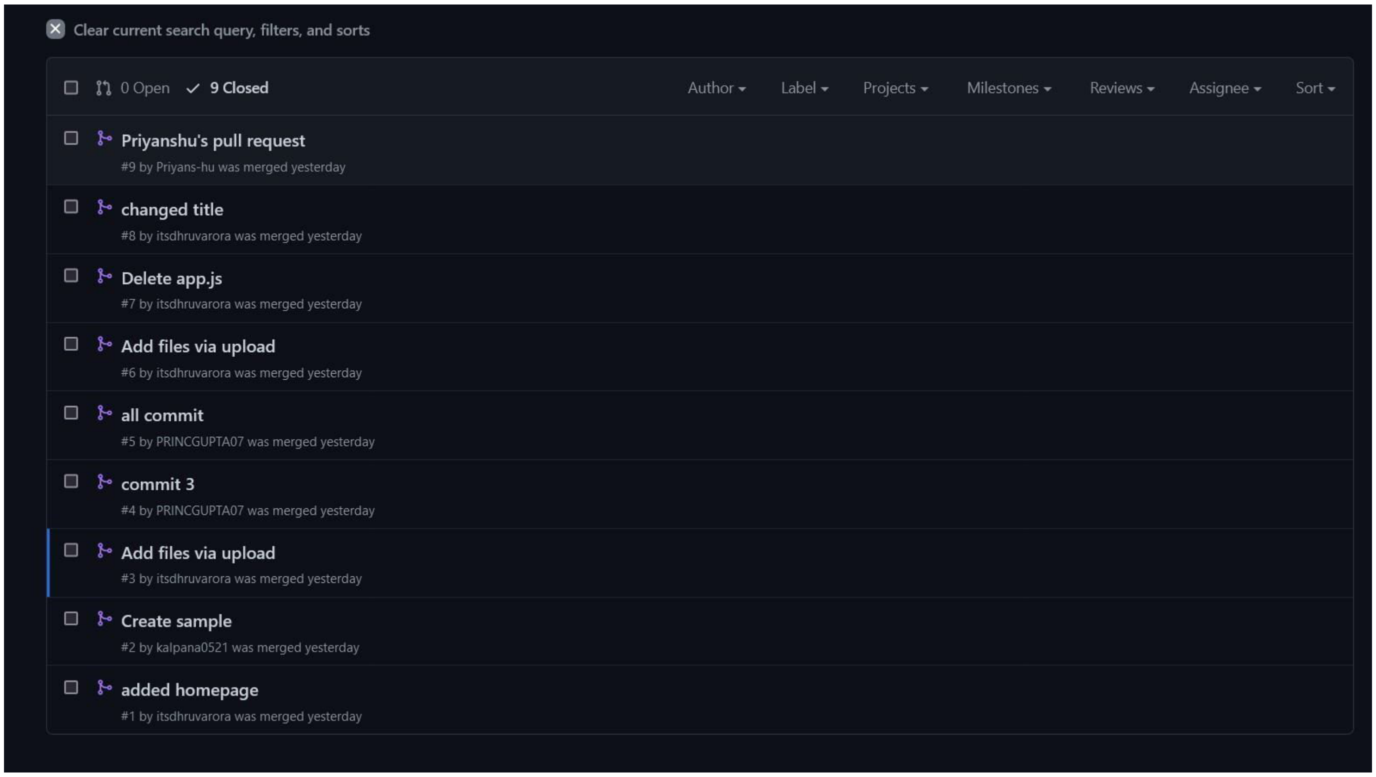
426 Mens.html

Load diff
Large diffs are not rendered by default.

107 SneakFreak cart.html

- If the team member chooses not to merge your pull request they will close you're the pull request.

- To close the pull request simply click on close pull request and add comment/ reason why you closed the pull request.
- You can see all the pull request generated and how they were dealt with by clicking on pull request option.



Clear current search query, filters, and sorts

0 Open ✓ 9 Closed

Priyanshu's pull request
#9 by Priyans-hu was merged yesterday

changed title
#8 by itsdhruvarora was merged yesterday

Delete app.js
#7 by itsdhruvarora was merged yesterday

Add files via upload
#6 by itsdhruvarora was merged yesterday

all commit
#5 by PRINCGUPTA07 was merged yesterday

commit 3
#4 by PRINCGUPTA07 was merged yesterday

Add files via upload
#3 by itsdhruvarora was merged yesterday

Create sample
#2 by kalpana0521 was merged yesterday

added homepage
#1 by itsdhruvarora was merged yesterday

Experiment No. 03

Aim: Publish and print network graphs

The network graph is one of the useful features for developers on GitHub. It is used to display the branch history of the entire repository network, including branches of the root repository and branches of forks that contain commits unique to the network.

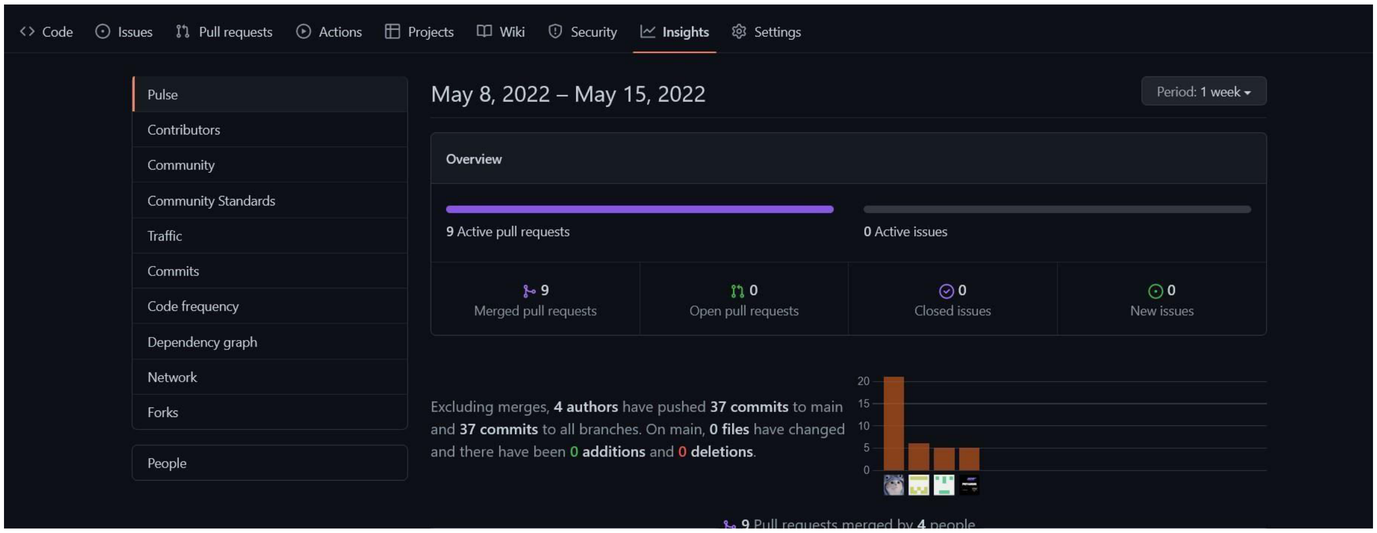
A repository's graphs give you information on traffic, projects that depend on the repository, contributors and commits to the repository, and a repository's forks and network. If you maintain a repository, you can use this data to get a better understanding of who's using your repository and why they're using it.

Some repository graphs are available only in public repositories with GitHub Free:

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

<u>Steps to access network graphs of respective repository</u>

1. On GitHub.com, navigate to the main page of the repository.
2. Under your repository name, click Insights.
3. At the left sidebar, click on Network.



May 8, 2022 – May 15, 2022

9 Active pull requests

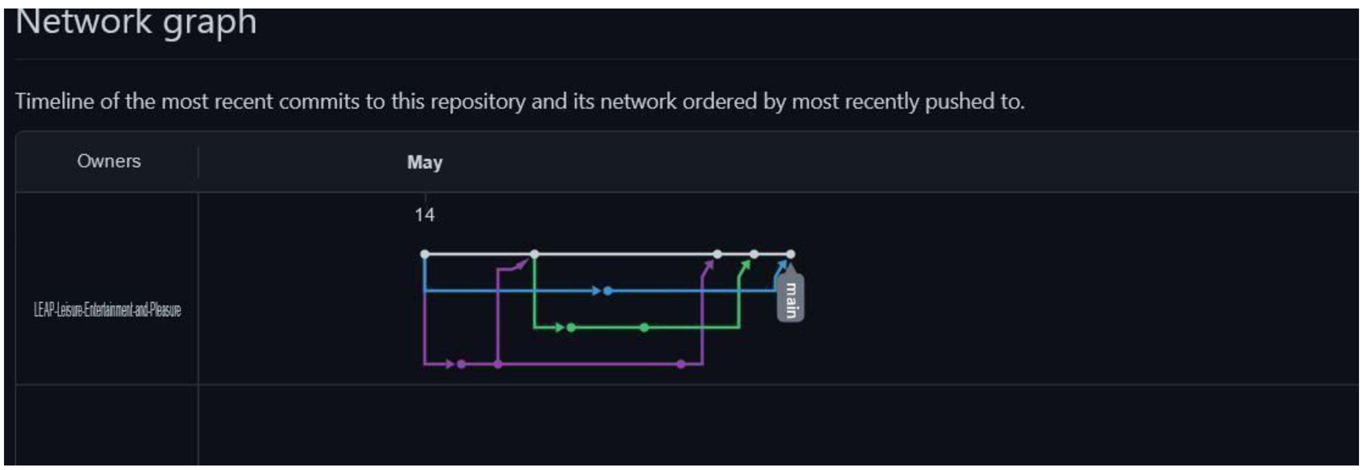
0 Active issues

9 Merged pull requests	0 Open pull requests	0 Closed issues	0 New issues

Excluding merges, **4 authors** have pushed **37 commits** to main and **37 commits** to all branches. On main, **0 files** have changed and there have been **0 additions** and **0 deletions**.

9 Pull requests merged by 4 people

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



Network graph

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