Subject Name: **Source Code Management**

Subject Code: **CS181**

Cluster: **Beta**

Department: **DCSE**

EXPERIMENT	TOPIC	PAGE No.
EXPERIMENT -1	Setting up of Git Client	03-13
EXPERIMENT-2	Setting up Git Hub Account	14-19
EXPERIMENT-3	How to Use Git Log	20-23
EXPERIMENT-4	Create and visualize Branch	24-28

Submitted By:

ASTITAV MITTAL

2110990321

G08

Submitted To:

Dr.MONIT KAPOOR

EXPERIMENT-5	**Life Cycle of the Git**	**29-34**
EXPERIMENT-6	Add collaborators on Github repo	**35-38**
EXPERIMENT-7	Fork and Commit	**39-44**
EXPERIMENT-8	Merge and Resolve conflicts created due to own activity and collaborators activity	**45-54**
EXPERIMENT-9	Reset and Revert	**55-63**
EXPERIMENT-10	PROJECT	**64-93**

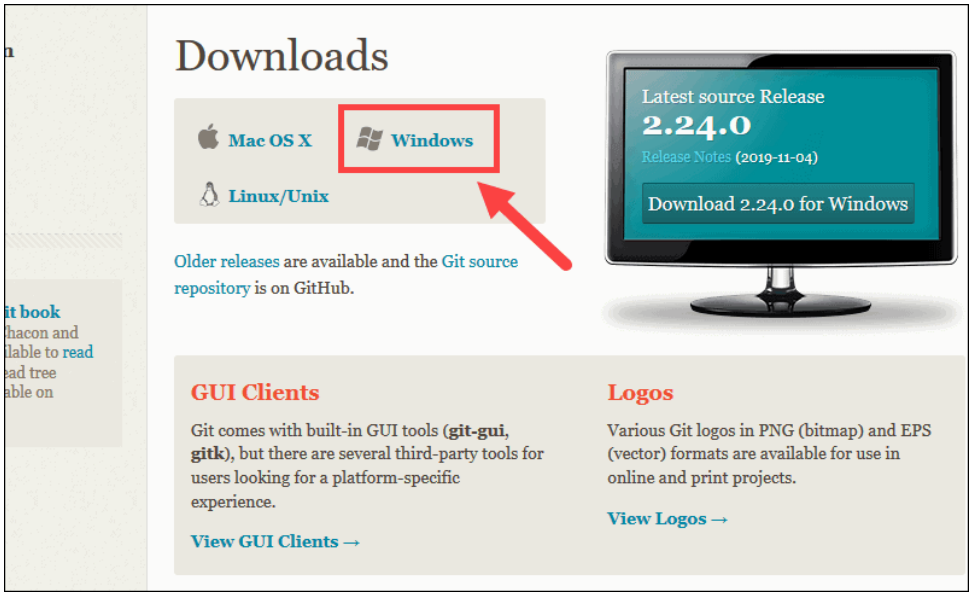
CHITKARA UNIVERSITY

AIM :-Setting up of Git Client

Steps For Installing Git for Windows

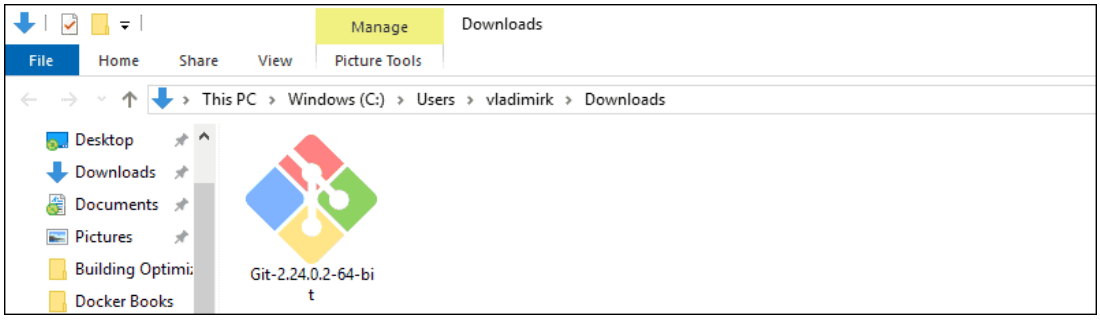
Download Git for Windows

1. Browse to the official Git Website: https://git-scm.com/downloads
2. click the download link for Windows and allow the download to complete.

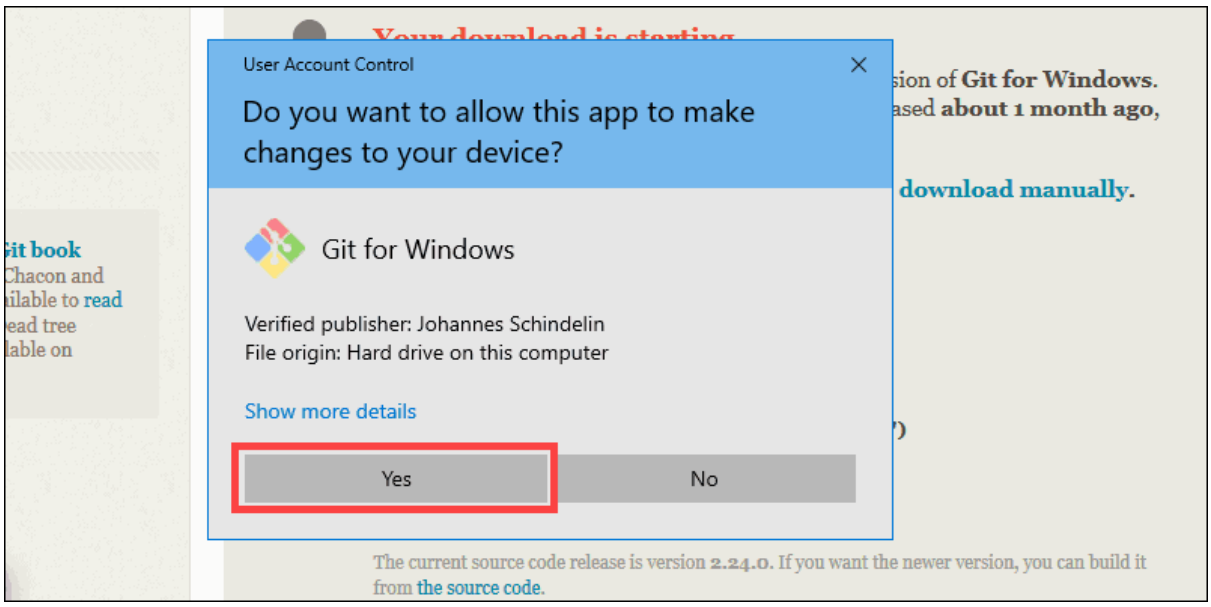


Extract and Launch Git Installer

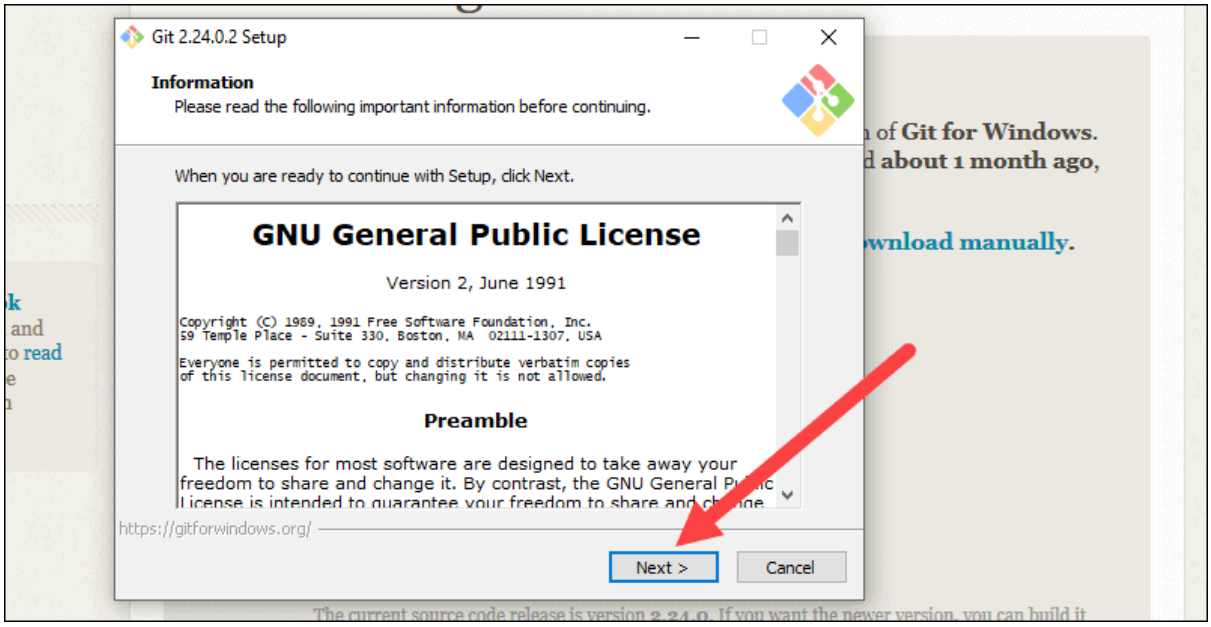
3. Browse to the download location (or use the download shortcut in your browser). Double-click the file to extract and launch the installer.



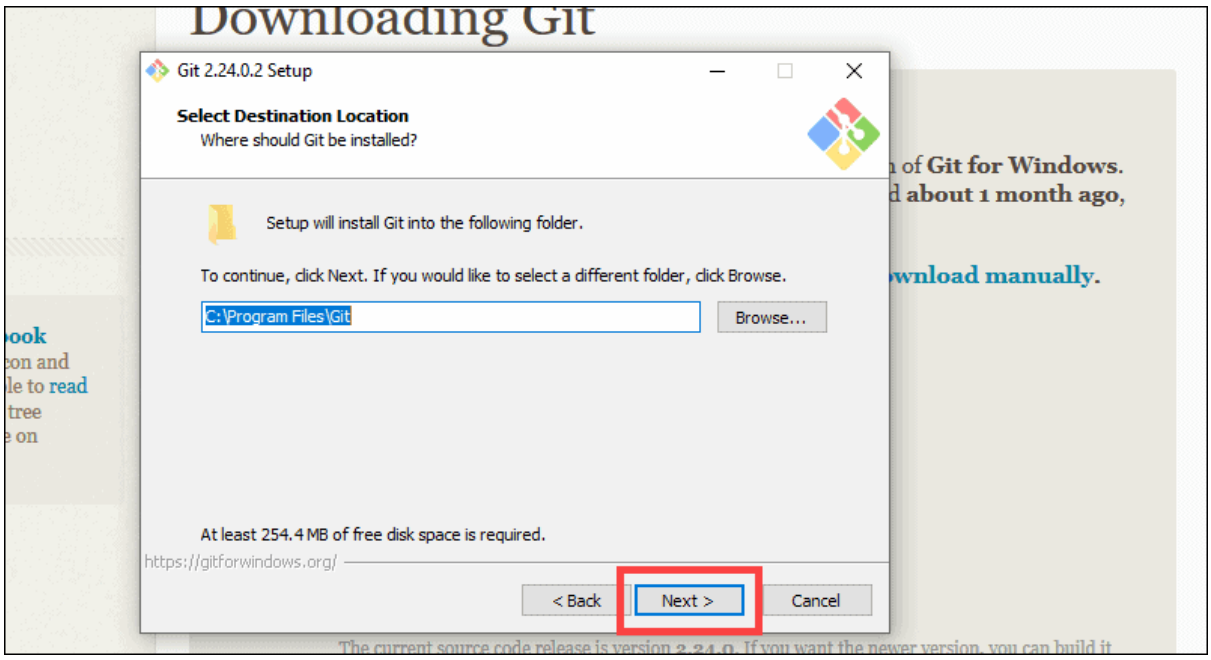
4.Allow the app to make changes to your device by clicking **Yes** on the User Account Control dialog that opens.



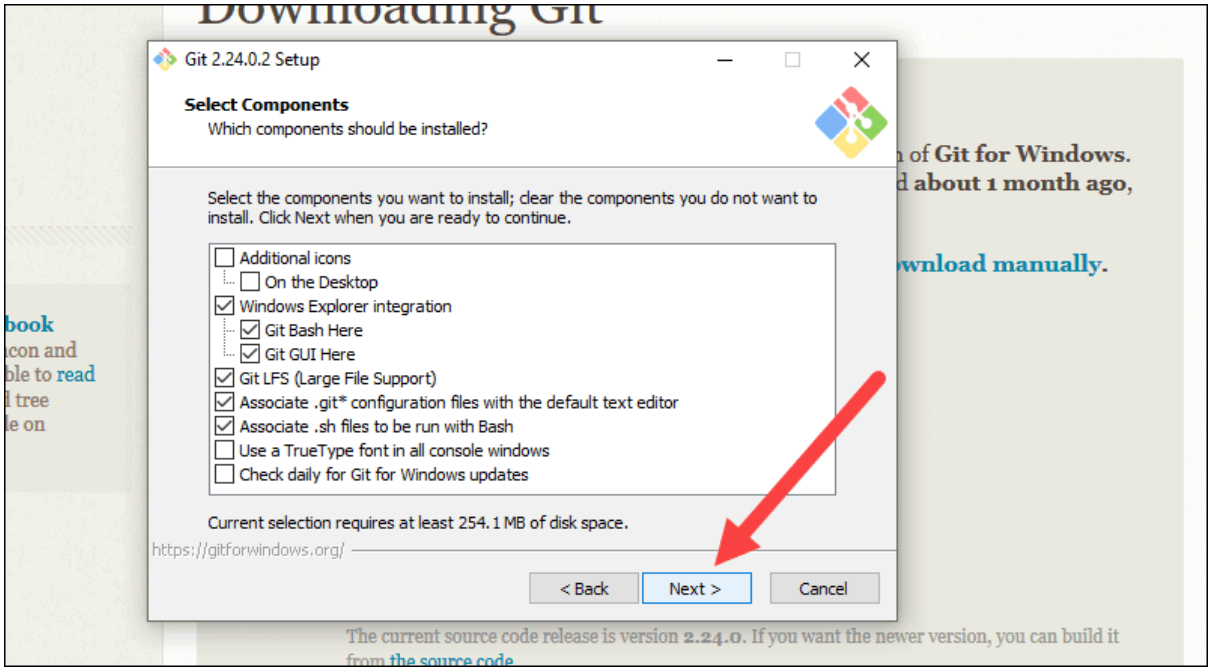
5. Review the GNU General Public License, and when you're ready to install, click **Next**.



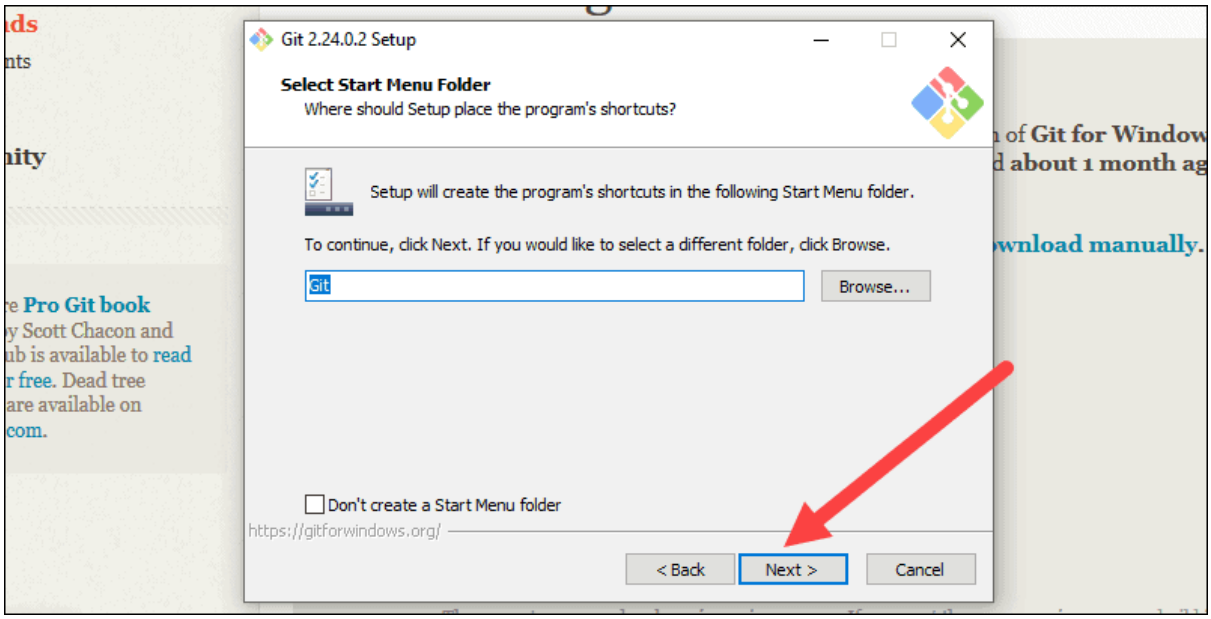
6. The installer will ask you for an installation location. Leave the default, unless you have reason to change it, and click **Next**.



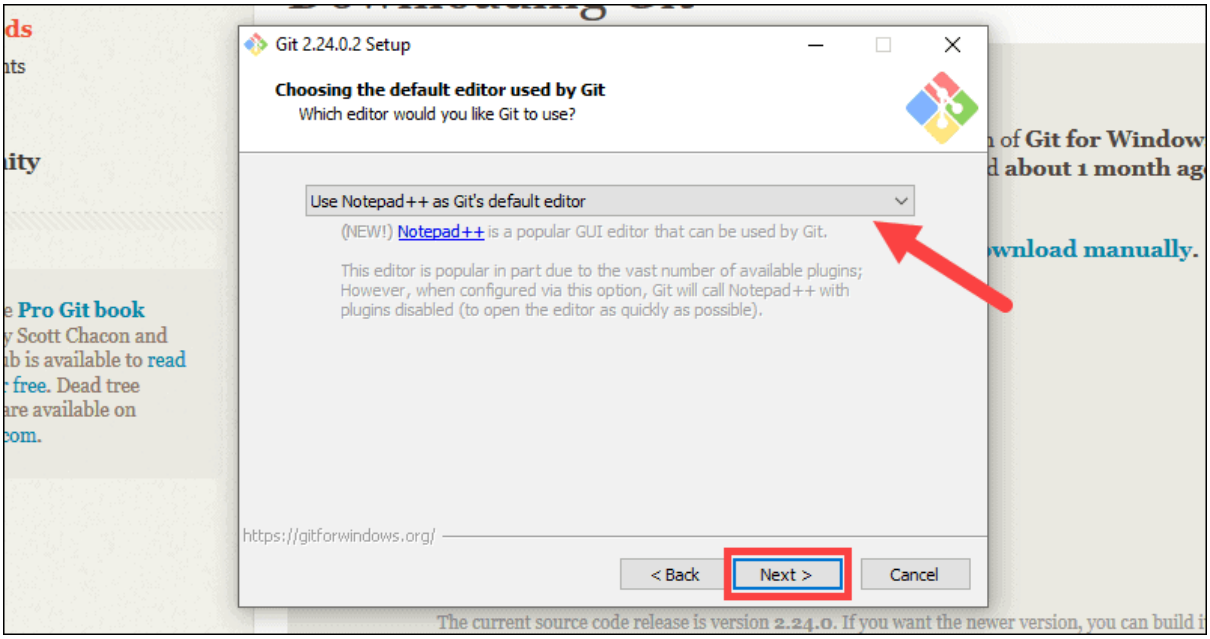
7. A component selection screen will appear. Leave the defaults unless you have a specific need to change them and click **Next**.



8. The installer will offer to create a start menu folder. Simply click **Next**.

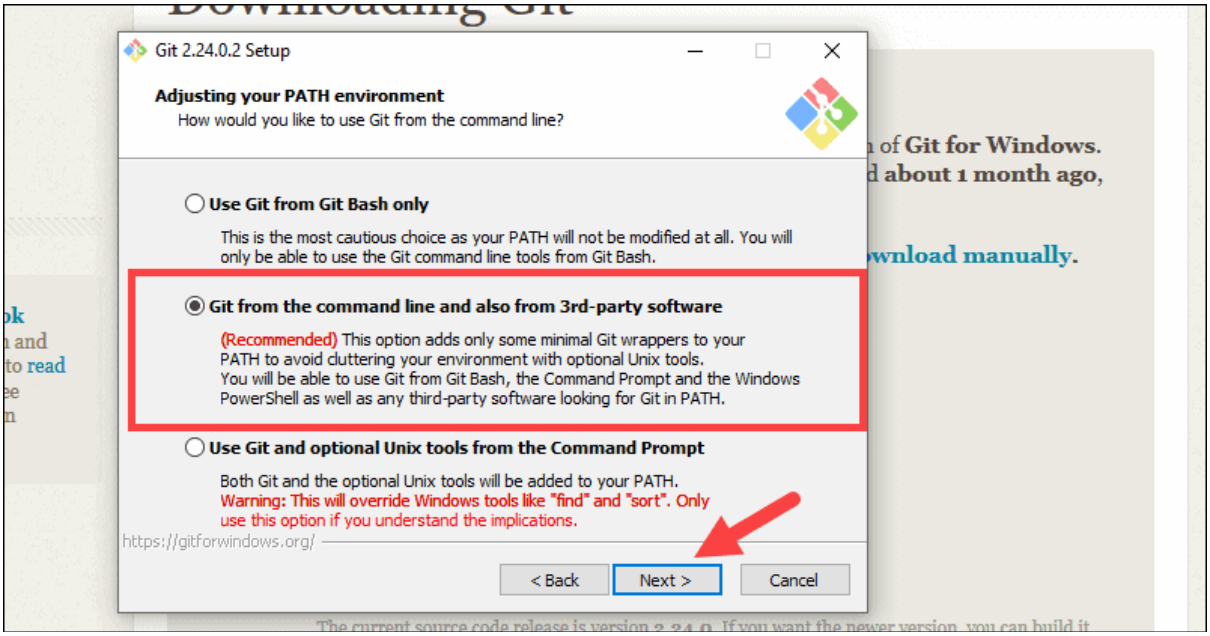


9. Select a text editor you'd like to use with Git. Use the drop-down menu to select Notepad++ (or whichever text editor you prefer) and click **Next**.



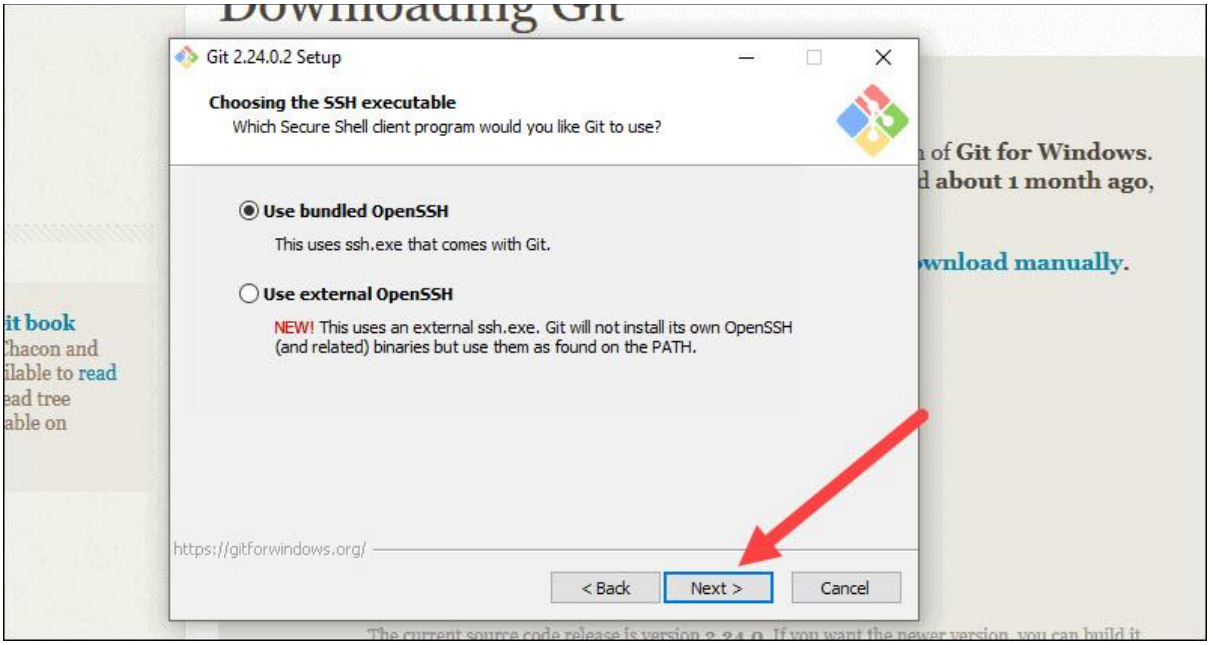
10. The next step allows you to choose a different name for your initial branch. The default is 'master.' Unless you're working in a team that requires a different name, leave the default option and click **Next.**

11. This installation step allows you to change the **PATH environment**. The **PATH** is the default set of directories included when you run a command from the command line. Leave this on the middle (recommended) selection and click **Next**.



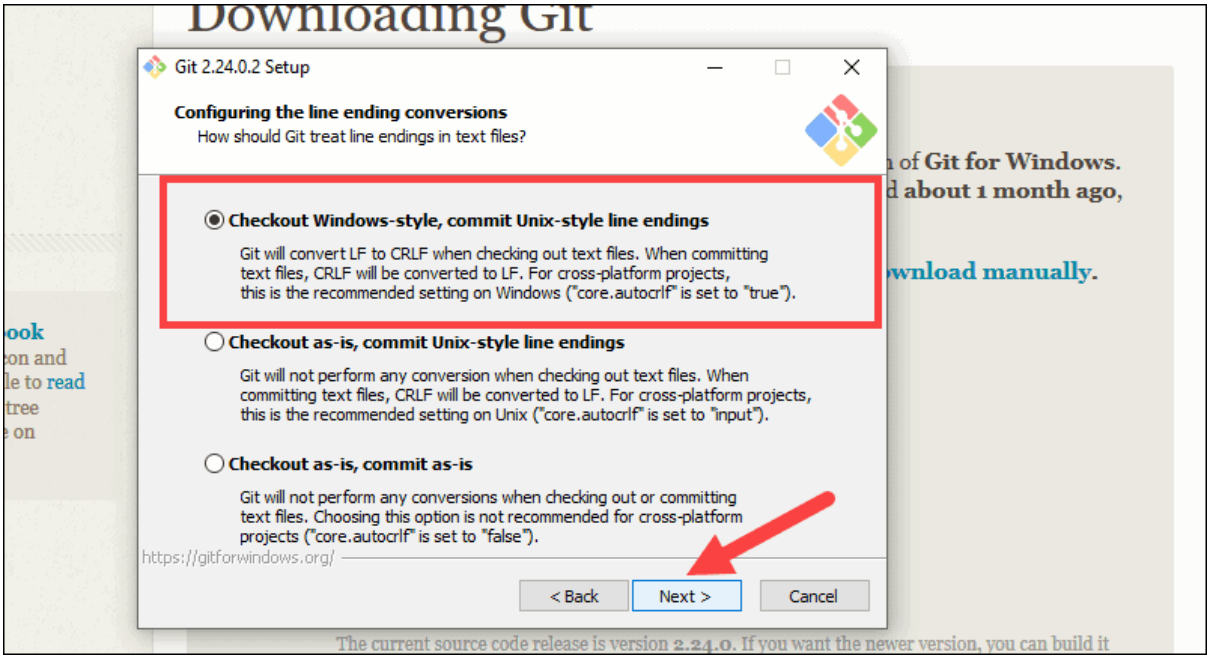
Server Certificates, Line Endings and Terminal Emulators

12. The installer now asks which SSH client you want Git to use. Git already comes with its own SSH client, so if you don't need a specific one, leave the default option and click **Next.**



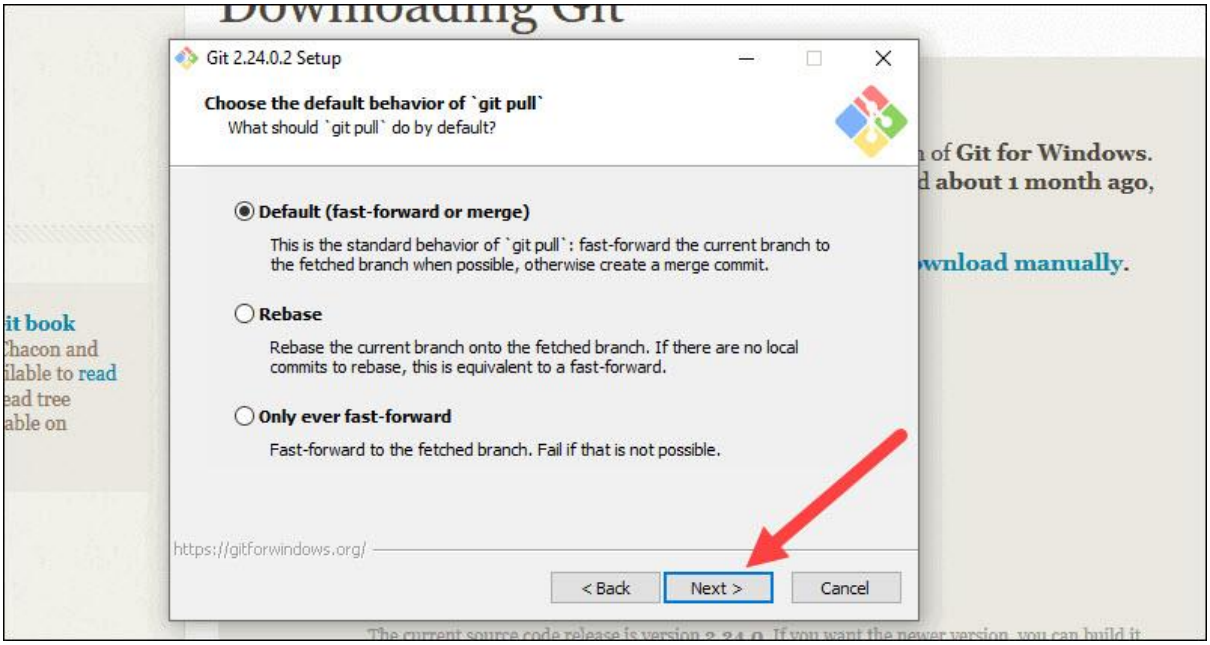
13. The next option relates to server certificates. Most users should use the default. If you're working in an Active Directory environment, you may need to switch to Windows Store certificates. Click **Next**.

14. The next selection converts line endings. It is recommended that you leave the default selection. This relates to the way data is formatted and changing this option may cause problems. Click **Next**.

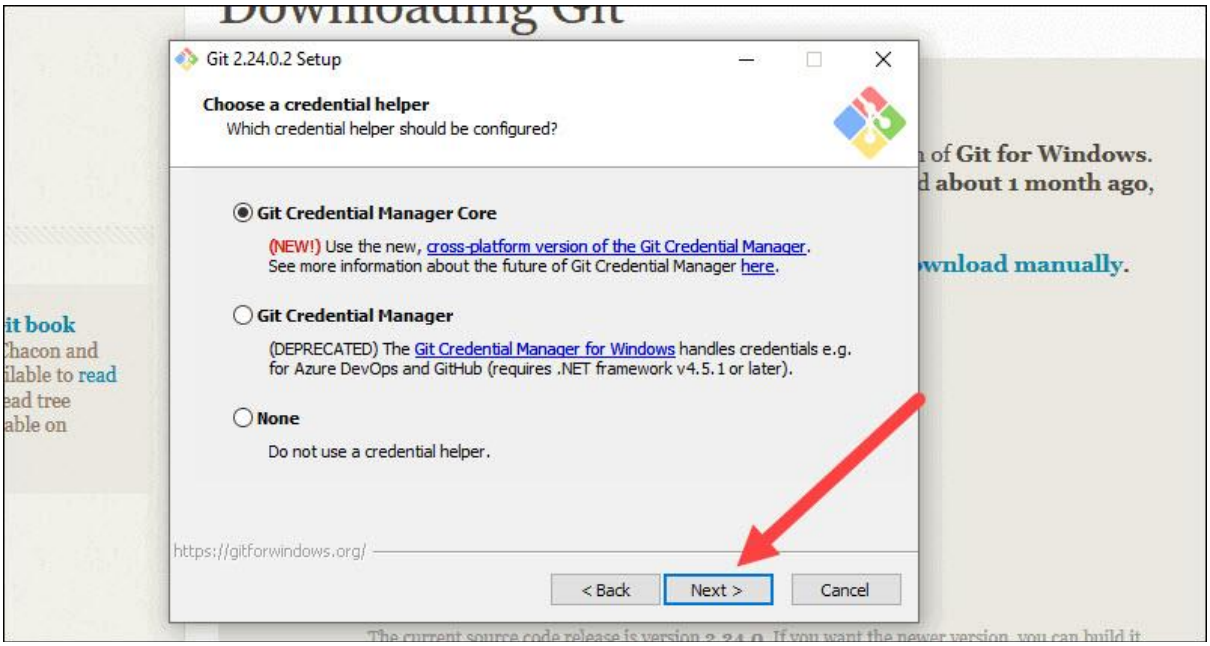


15. Choose the terminal emulator you want to use. The default MinTTY is recommended, for its features. Click **Next**.

16. The installer now asks what the **git pull** command should do. The default option is recommended unless you specifically need to change its behavior. Click **Next** to continue with the installation.

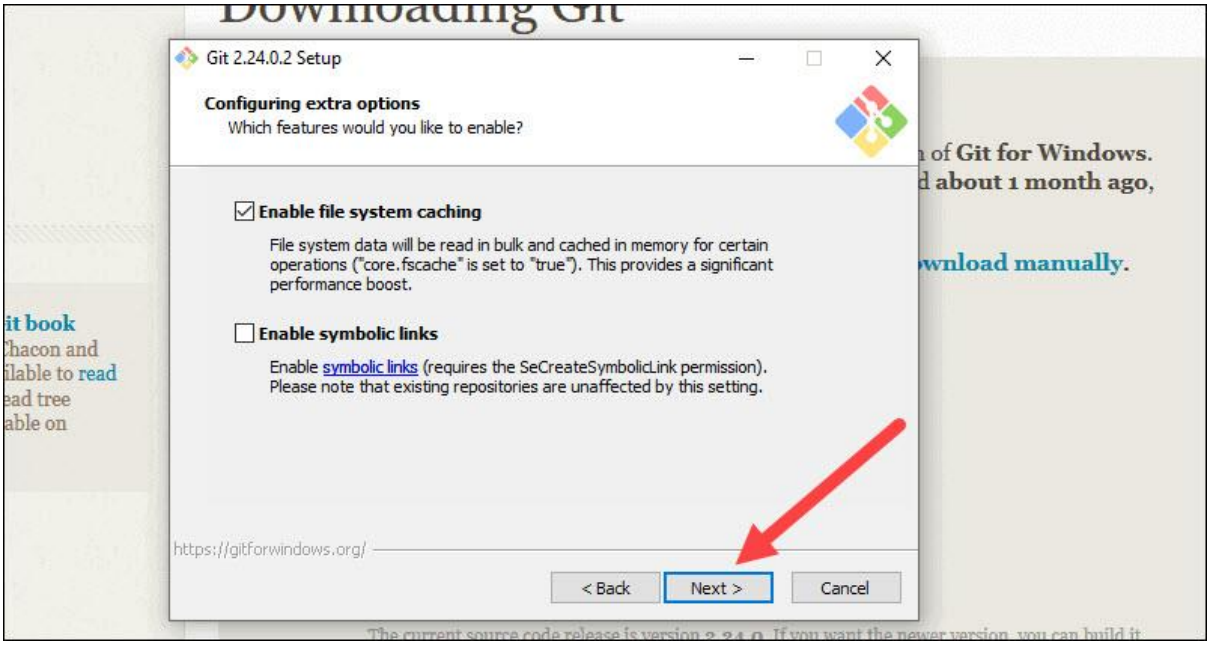


17. Next you should choose which credential helper to use. Git uses credential helpers to fetch or save credentials. Leave the default option as it is the most stable one, and click **Next**.

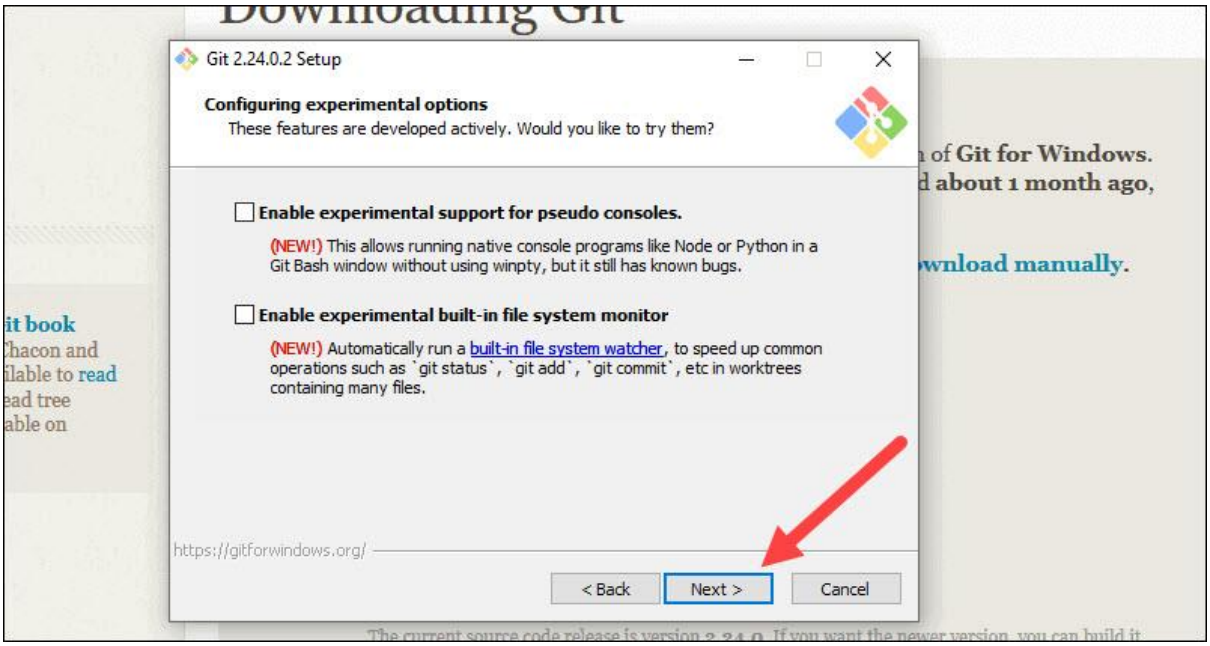


Additional Customization Options

18. The default options are recommended, however this step allows you to decide which extra option you would like to enable. If you use symbolic links, which are like shortcuts for the command line, tick the box. Click **Next**.

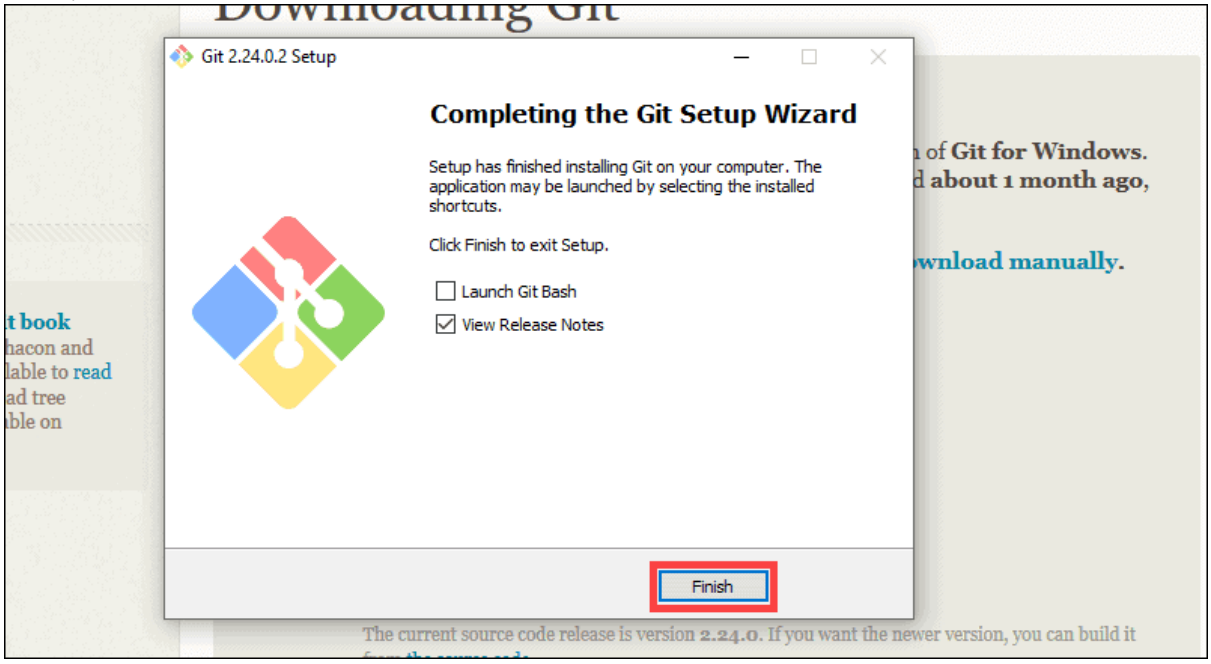


19. Depending on the version of Git you're installing, it may offer to install experimental features. At the time this article was written, the options to include support for pseudo controls and a built-in file system monitor were offered. Unless you are feeling adventurous, leave them unchecked and click **Install**.



Complete Git Installation Process

20. Once the installation is complete, tick the boxes to view the Release Notes or Launch Git Bash, then click Finish**.**



Experiment No. 02

CHITKARA UNIVERSITY

AIM:-Setting Up GitHub Account

Steps For Making an Account in Git Hub

1

God to https://github.com/join in a web browser. You can use any web browser on your computer, phone, or tablet to join.

- Some ad blockers, including u Block Origin, prevent GitHub's verification CAPTCHA puzzle from appearing. For best results, disable your web browser's ad blocker when signing up for GitHub.

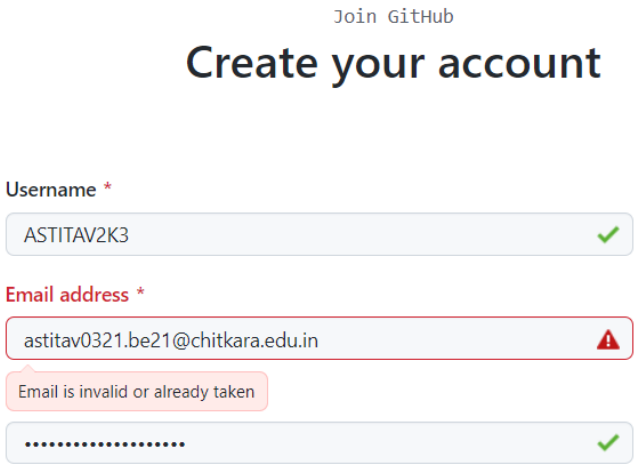
2

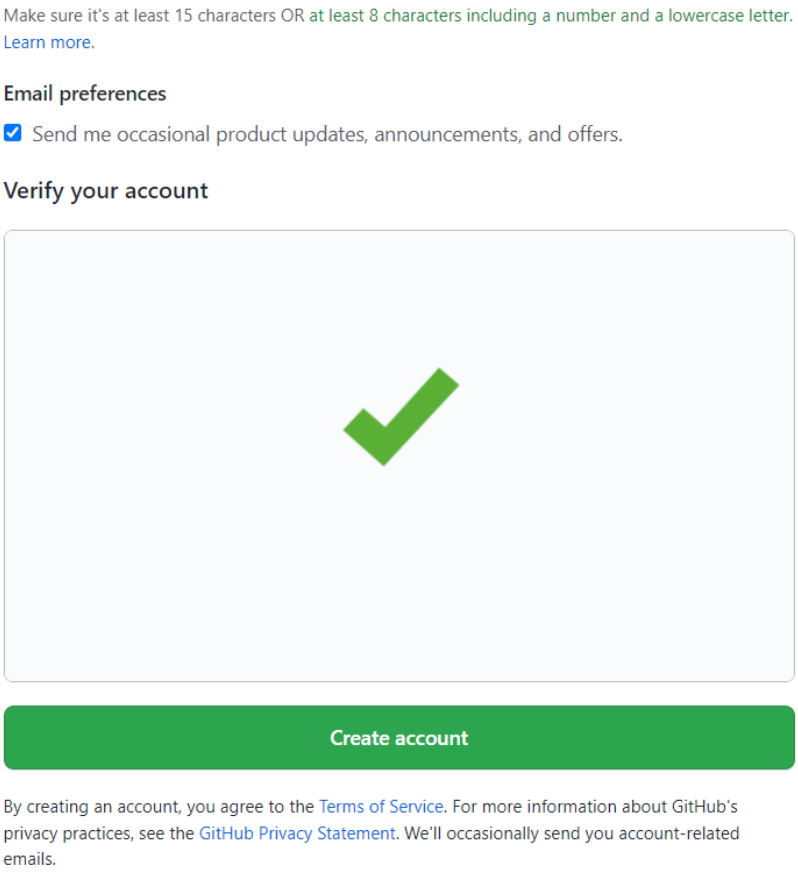
Enters your personal details. In addition to creating a username and entering an email address, you'll also have to create a password. Your password must be at least 15 characters in length *or* at least 8 characters with at least one number and lowercase letter.

- Carefully review the Terms of Service at https://help.github.com/en/articles/github-terms-of-service and the Privacy Statement at https://help.github.com/en/articles/github-privacy-statement before you continue. Continuing past the next step confirms that you agree to both documents.

3

Click the green Create an account **button.** It's below the form.

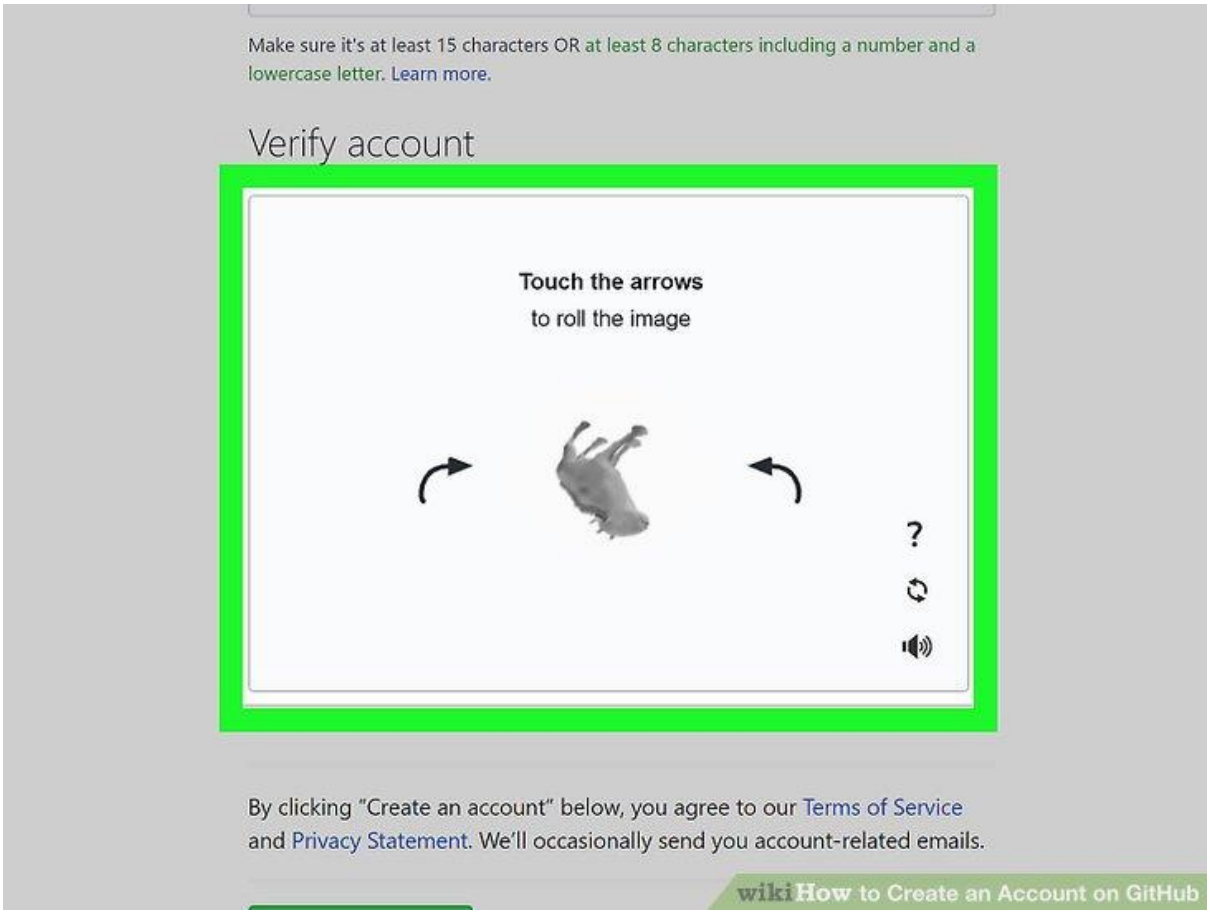




4

Complete the CAPTCHA puzzle. The instructions vary by puzzle, so just follow the on-screen instructions to confirm that you are a human.

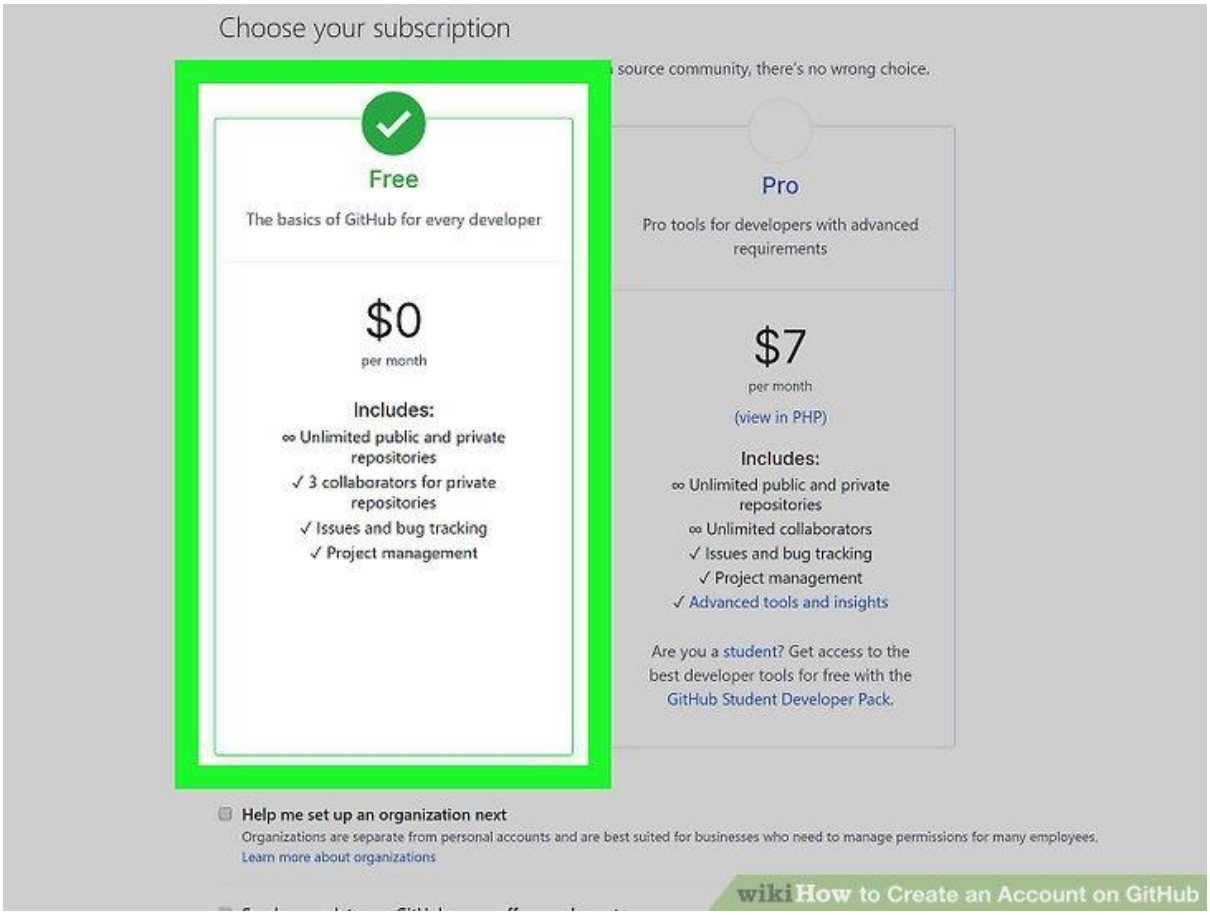
- If you see an error that says "Unable to verify your captcha response," it's because your web browser's ad blocking extension prevented the CAPTCHA puzzle from appearing. Disable all ad-blocking extensions, refresh the page, and then click **VERIFY** to start the CAPTCHA.



5

Click the Choose **button for your desired plan.** Once you select a plan, GitHub will send an email confirmation message to the address you entered. The plan options are:[2]

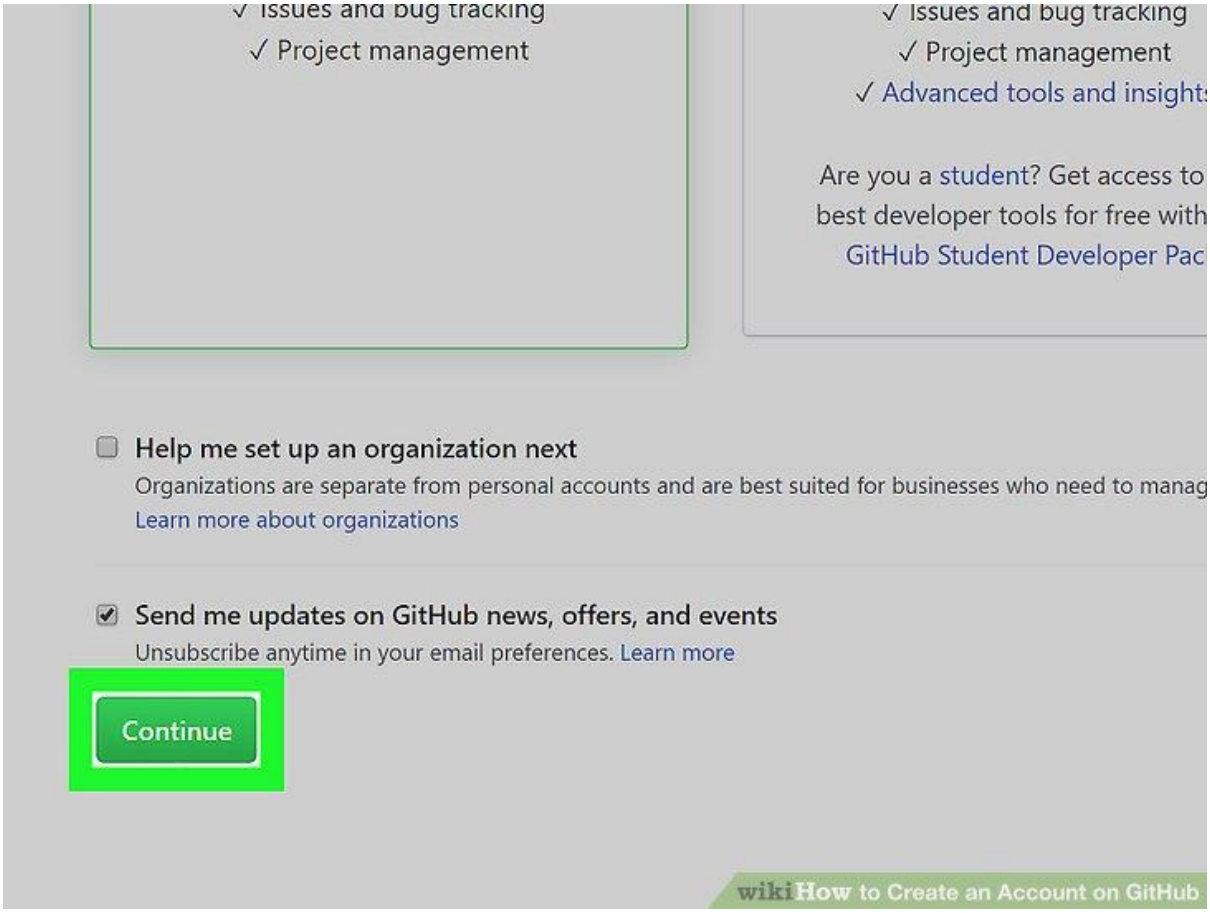
- **Free:** Unlimited public and private repositories, up to 3 collaborators, issues and bug tracking, and project management tools.
- **Pro:** Unlimited access to all repositories, unlimited collaborators, issue & bug tracking, and advanced insight tools.
- **Team:** All of the aforementioned features, plus team access controls and user management.
- **Enterprise:** All of the features of the Team plan, plus self-hosting or cloud hosting, priority support, single sign-on support, and more.



6

Review your plan selection and click Continue**.** You can also choose whether you want to receive updates from GitHub via email by checking or unchecking the "Send me updates" box.

- If you chose a paid plan, you'll have to enter your payment information as requested before you can continue.



7

Select your preferences and click Submit**.** GitHub displays a quick survey that can help you tailor your experience to match what you're looking for. Once you make

CHITKARA UNIVERSITY

AIM:-how to use Git Log

git status

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

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ echo "Add some text "> newfile3,txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        newfile3,txt

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

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

In conjunction with these commands, you'll also need git status to view the state of the working directory and the staging area.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   newfile.txt
        new file:   newfile1.txt
        new file:   newfile2.txt
        new file:   newfile3.txt

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory
)
        deleted:    newfile3.txt
```

Git Commit

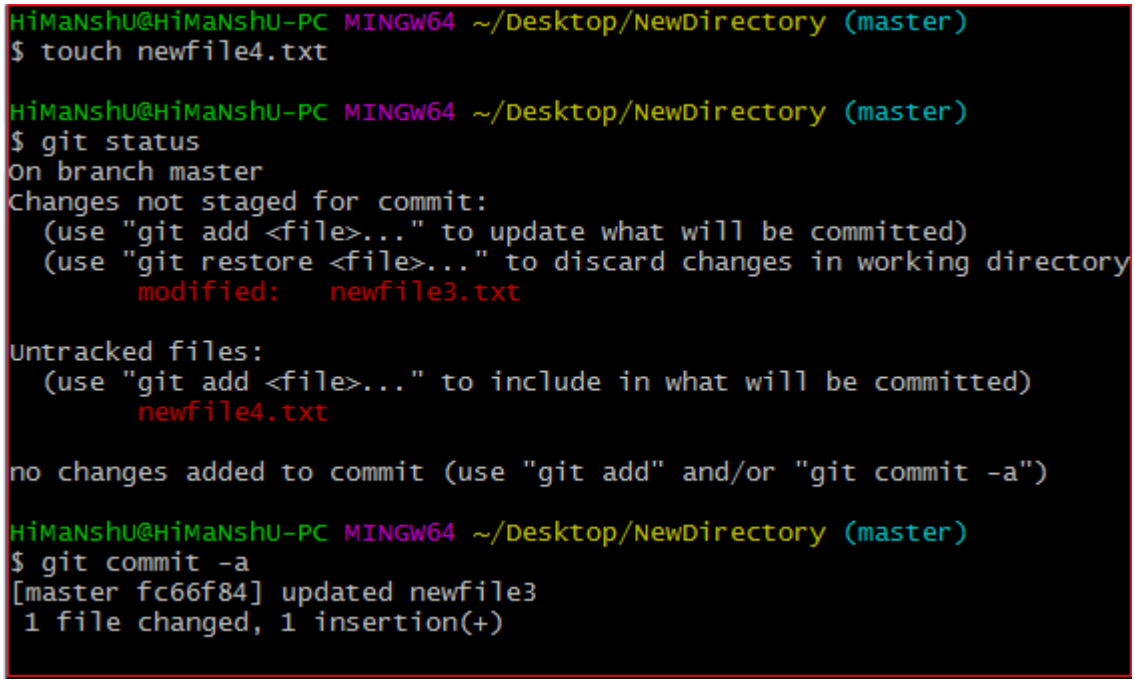
Since we have finished our work, we are ready move from stage to commit for our repo.

Adding commits keep track of our progress and changes as we work. Git considers each commit change point or "save point". It is a point in the project you can go back to if you find a bug, or want to make a change.

When we commit, we should **always** include a **message**.

By adding clear messages to each commit, it is easy for yourself (and others) to see what has changed and when.

```
git commit -m "commit message"
```



The git log Command

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

Whereas the git status command is focused on the current working directory, git log allows you to see the history of your repository.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: --
d, tag: projectv1.1, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530

    added a new image to prject

commit 828b9628a873091ee26ba53c0fcfc0f2a943c544 (tag: olderversion)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Thu Oct 3 11:17:25 2019 +0530

    Update design2.css

commit 0a1a475d0b15ecec744567c910eb0d8731ae1af3 (test)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Tue Oct 1 12:30:40 2019 +0530

    CSS file

    See the proposed CSS file.

commit f1ddc7c9e765bd688e2c5503b2c88cb1dc835891
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sat Sep 28 12:31:30 2019 +0530
```

AIM:-Create and Visualize branch

How it works

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

The git branch command lets you create, list, rename, and delete branches. It doesn't let you switch between branches or put a forked history back together again. For this reason, git branch is tightly integrated with the git checkout and git merge commands.

Common Options

```
git branch
```

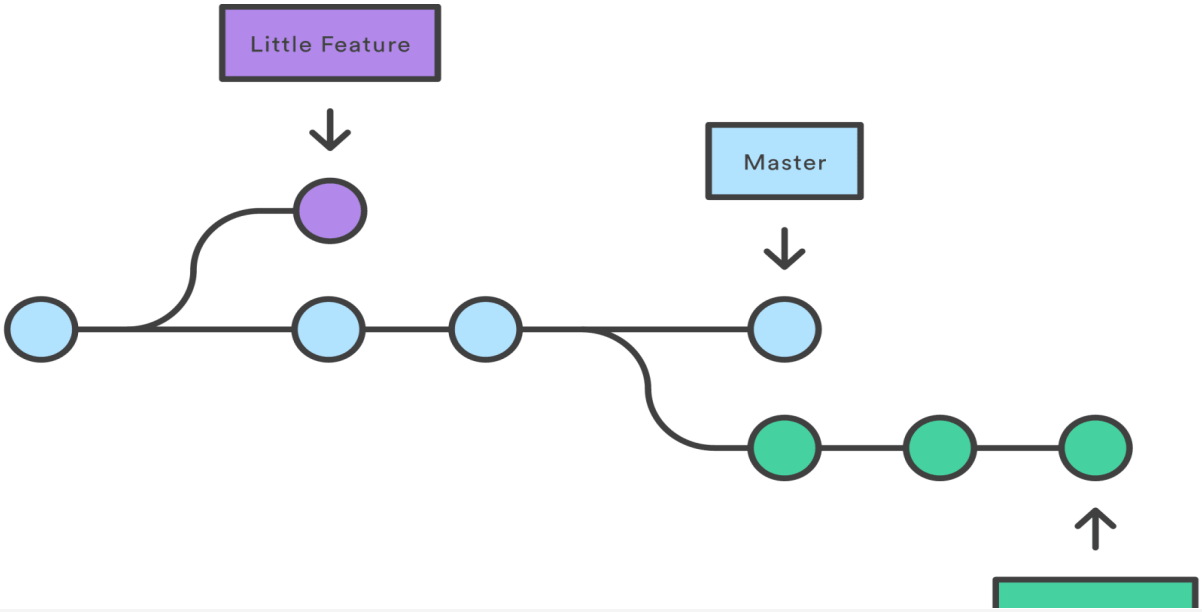
List all of the branches in your repository. This is synonymous with git branch --list.

```
git branch <branch>
```

Create a new branch called <branch>. Thisdoes *not* check out the new branch.

Creating Branches

It's important to understand that branches are just pointers to commits. When you create a branch, all Git needs to do is create a new pointer, it doesn't change the repository in any other way. If you start with a repository that looks like this:

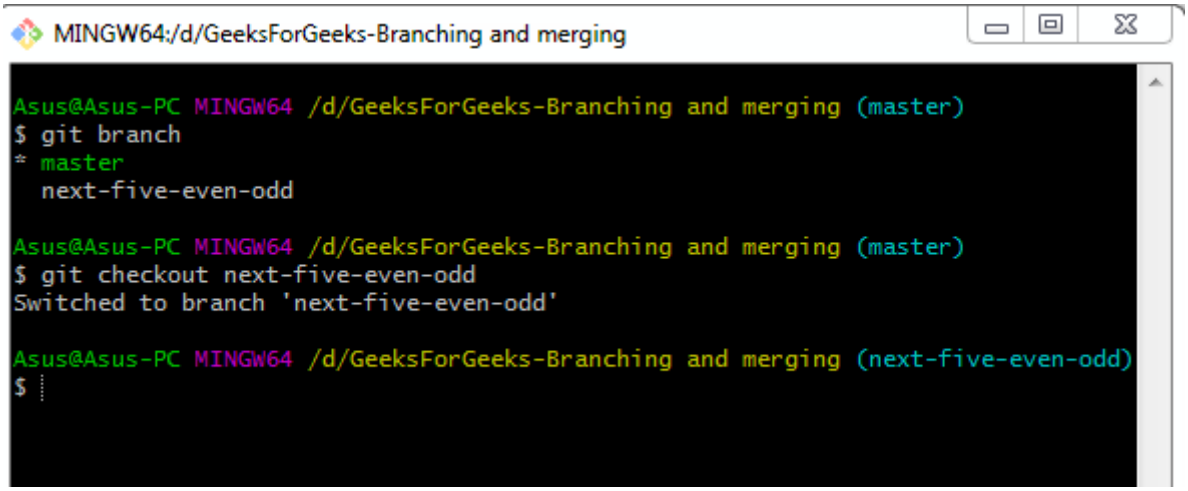


Then, you create a branch using the following command:

```
gitbranch crazy-experiment
```

The repository history remains unchanged. All you get is a new pointer to the current commit:

Note that this only *creates* the new branch. To start adding commits to it, you need to select it with git checkout, and then use the standard git add and git commit commands.



Merging into Master — Fun Part

This is the only step that is different from the steps followed in the rebase article.

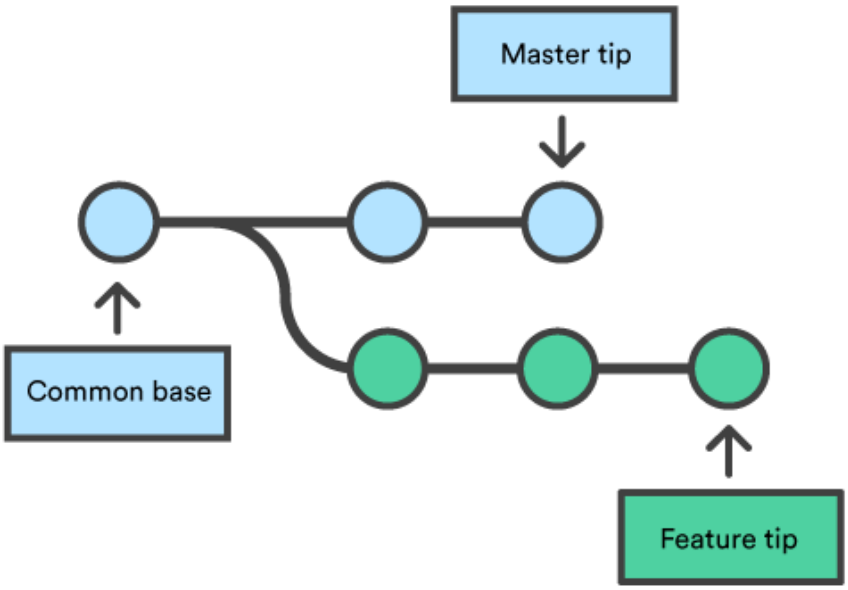
Note that the changes are added to the current branch from the selected branch when doing a merge. Therefore we should first move to the **master** branch to start the merge.

```
git checkout master
```

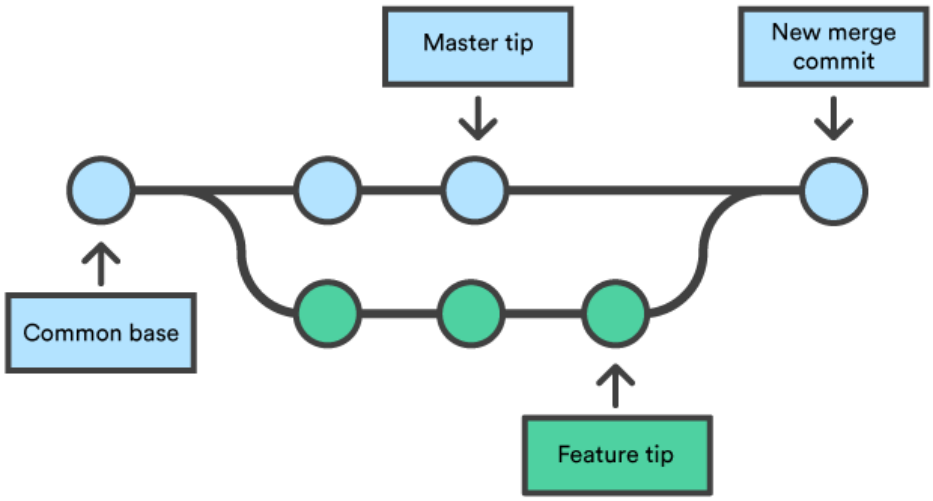
Now you can use the following command to move the changes from **dev** to **master**.

```
git merge <Branch name>
```

When we use this command, Git will find a common commit between the 2 branches and create a new *merge commit* on the **master** combining the queued changes in the **dev** branch.



Before Merge(Image from [Git Documentation])




```
MINGW64:/f/Git_Branching_Demo

CodeTej@LAPTOP-07TVC84J MINGW64 /f/Git_Branching_Demo (navbar-new-feature)
$ git checkout master
Switched to branch 'master'

CodeTej@LAPTOP-07TVC84J MINGW64 /f/Git_Branching_Demo (master)
$ git merge navbar-new-feature
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.

CodeTej@LAPTOP-07TVC84J MINGW64 /f/Git_Branching_Demo (master|MERGING)
$
```

Experiment No. 05

CHITKARA UNIVERSITY

AIM:- Git Life Cycle Description

Introduction to Git Life Cycle

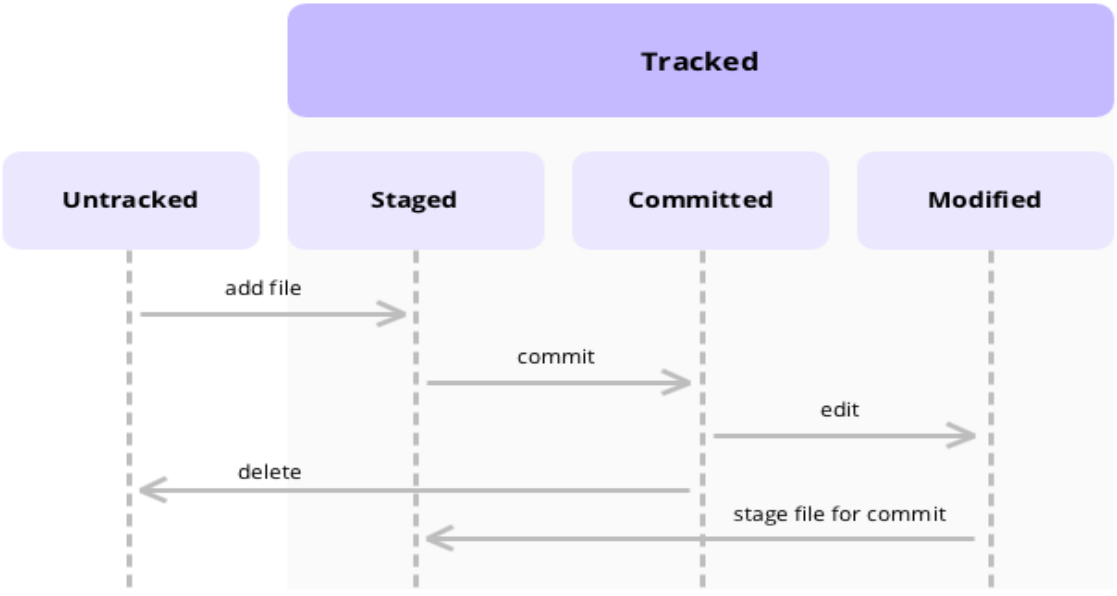
Git is one of the premier distributed version control systems available for programmers and corporates. In this article, we will see details about how a project that is being tracked by git proceeds with workflow i.e Git Life Cycle. As the name suggests is regarding different stages involved after cloning the file from the repository. It covers the git central commands or main commands that are required for this particular version control system

Workflow of Git Life Cycle

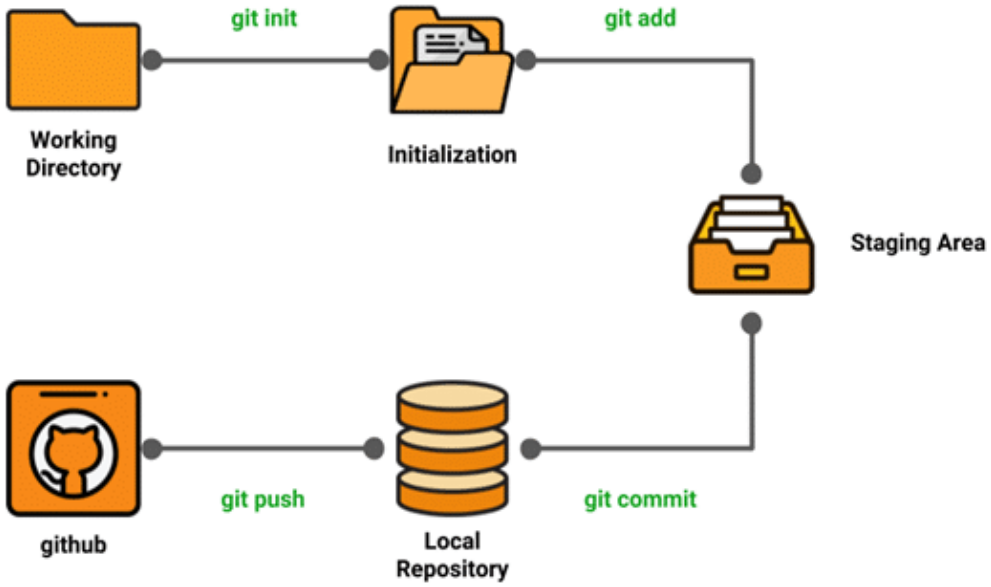
The workflow of the Git as follows:

- We will create a branch on which we can work on and later we will merge it with master

- Clone: First, when we have code present in the remote repository, we clone to local to form something called a local repository.
- Modifications/Adding Files: we perform several developments on the existing files or may as well add new files. Git will monitor all these activities and will log them.



- We need to move the content that we require to transform to the master to the staging area by using git commands and the snapshot of staged files will be saved in the git staging area.
- We need to perform commits on the files that are staged and the recorded snapshot from the above steps will be permanently saved on the local repo and this particular is recorded by commit message for future referrals.

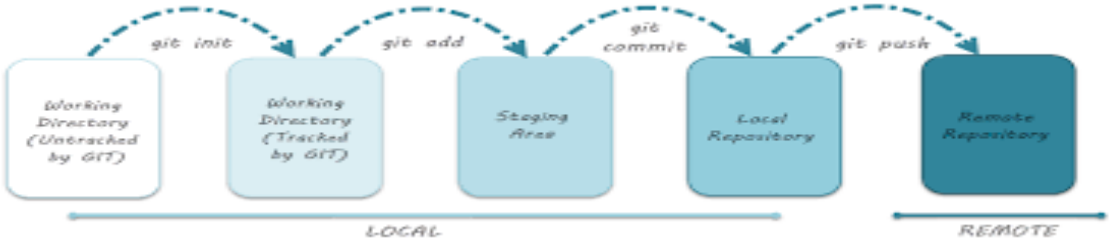


- Once we commit the code is available on the local repo but to send it to the master repo we need to perform PUSH operation
- If someone else is working on the same branch then there will be a possibility that he might have added his changes to the master by push. So we need to perform

PULL operation before the PUSH operation if multiple people are working on the same branch and this workflow as shown below.

- Once the target branch is updated we need to get all the required approvals so that merge operation with the master is allowed.

This is the basic workflow of git was lots of intermediate commands like git add, git status, git commit, git push origin, git rebase, git merge, git diff, etc will be used depending upon the requirement of the user.



Stages of Git Life Cycle

So we have seen the workflow of the git life cycle above. But we need to know that we have a project linked with git then that project can reside in there of the following areas. Below mentioned areas are ingredients to the recipe of Git and having an idea of them will help you a lot to track the files that you are working on.

The stages are as discussed:

- Working Directory

- Staging Area
- Git Directory

These Three Stages are explained below:

1. Working Directory

- If you have your project residing on to your local machines then basically it is called even though it is linked to git or not. In either case, it will be called as the working directory. But when the available project is linked with git then basically there will be .git folder hidden in the so-called working directory. So the presence of the .git folder is enough to say that the folder is working copy on the machine and it is tracked by the git.
- At this stage, git knows what are the files and folders that it's tracking that's it. No other info will be available regarding this. To make sure that the newly added files get tracked in the working copy we need to make sure that those files are staged and this is our second residence for the files.

2. Staging Area

- When we make changes to the existing files in the working repo or if we add any folder of files and if we want these changes to need to be tracked and also need to be moved to the local repo for tracking then we need to move these changed files

or newly added folder of file to the staging area. Git add is the basic command which will be used to move the modified files to the staged area.

- It's ticked that been give to modified files or newly added folder of file to travel to the local repo for further traction. Those files that don't have that ticket will be tracked by the git but they won't be able to move to the target easily. Here index plays a critical role. GIT Index is something that comes in between local repo and working directory and it is the one that decides what needs to be sent to the local repo and in fact, it decides what needs to be sent to the central repo.

3. GIT Directory

- When we have done the modifications or addition of files or folder and want them to be part of the repository they first we do is to move them to the staging area and they will commit ready. When we commit then provide the appropriate commit message and files will be committed and get updated in the working directory.
- Now git tracks the commits and commit messages and preserves the snapshot of commit files and this is done in the Git specific directory called Git Directory. Information related to all the files that were committed and their commit messages will be stored in this directory. We can say that this git directory stores the metadata of the files that were committed.

Experiment No. 06

UNIVERSITY

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 accessto 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 theOwner's repository to a different location than her own repository with the same name.

To clone the Owner's repo into her Desktopfolder, the Collaborator enters:

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

Replace 'vlad' with the Owner's username.

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

The Collaborator can now make a change in her clone of the Owner's repository, exactly the sameway 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 weclone 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 upstreamto 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 pullbefore making our changes. The basic collaborativeworkflow would be:

- update your local repo with git pull origin main,
- make your changes and stage them with git add,
- commit your changes with git commit -m, and
- upload the changes to GitHub with git push origin main

It is better to make many commits with smaller changes rather than of one commit with massive changes: small commits are easier to read and review.

Switch Roles and Repeat-

Switch roles and repeat the whole process.

Review Changes-

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

Comment Changes in GitHub-

The Collaborator has some questions about one line change made by the Owner and has some suggestions to propose. With GitHub, it is possible to comment the diff of a commit. Over the line of code to comment, ablue 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 someof the benefits of using version control, Git and GitHub?

Experiment No. 07

Aim: Fork and Commit

About forks

Most commonly, forks are used to either propose changes to someone else's project orto 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 thanlogging 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 toinclude 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.

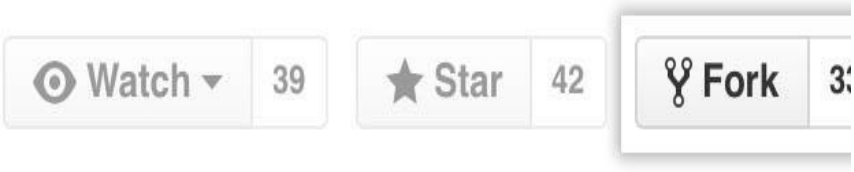
Prerequisites

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

Forking a repository

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

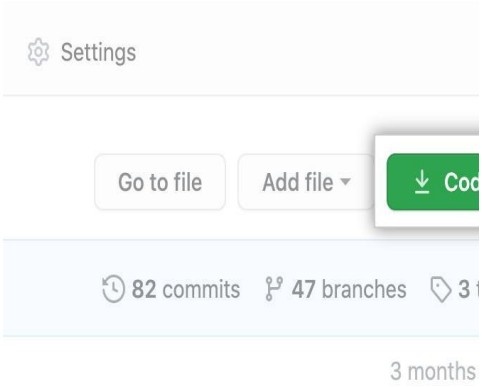
- On GitHub.com, navigate to the octocat/Spoon-Knife repository.
- In the top-right corner of the page, click Fork.



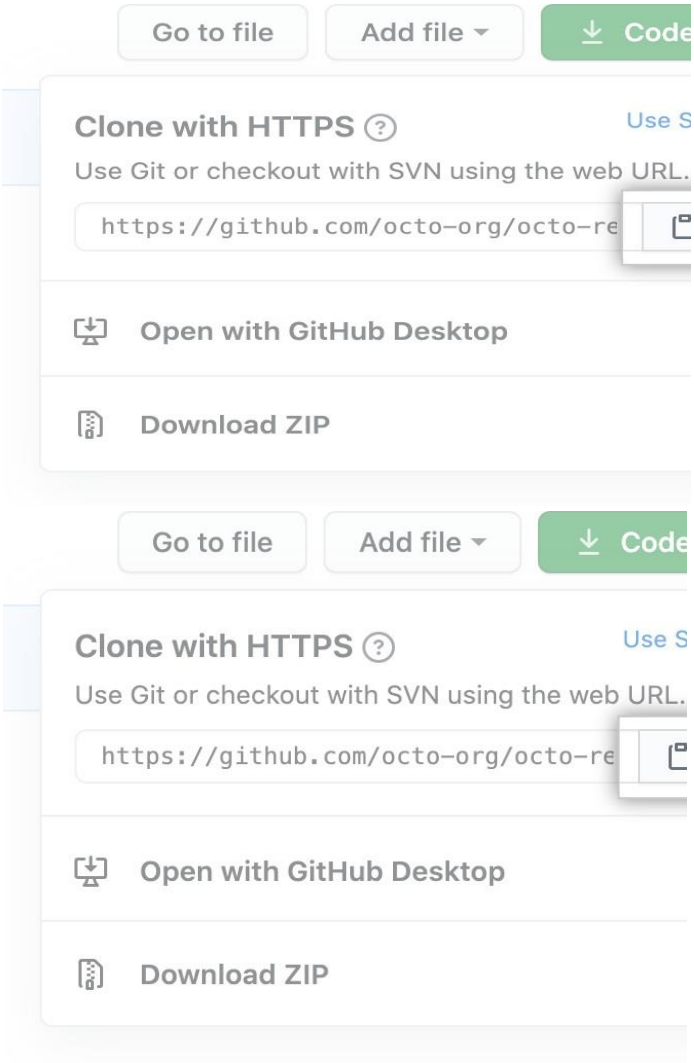
Cloning your forked repository

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

- On GitHub.com, navigate to your fork of the Spoon-Knife repository.
- Above the list of files, click Code



- 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 .



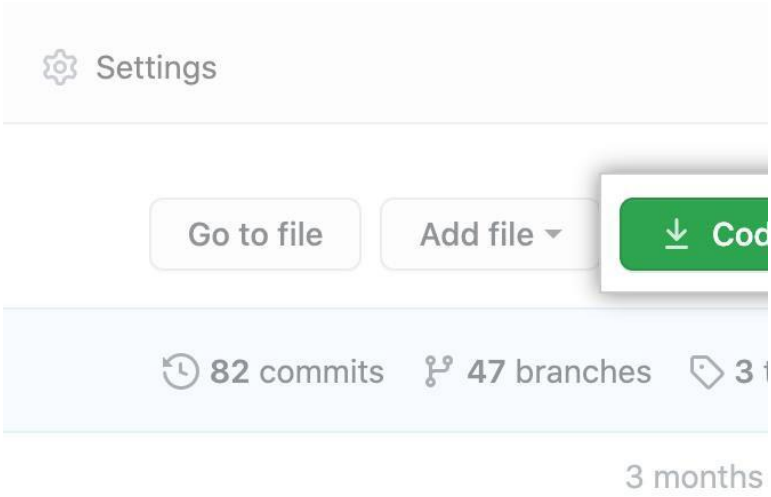
- Open Git Bash.

- Change the current working directory to the location where you want the cloned directory.
- Type git clone, and then paste the URL you copied earlier. It will look like this, with your GitHub username instead of YOUR-USERNAME:
- $ git clone https://github.com/*YOUR-USERNAME*/Spoon-Knife
- Press Enter. Your local clone will be created.
- $ git clone https://github.com/*YOUR-USERNAME*/Spoon-Knife
- > Cloning into `Spoon-Knife`...
- > remote: Counting objects: 10, done.
- > remote: Compressing objects: 100% (8/8), done.
- > remove: Total 10 (delta 1), reused 10 (delta 1)
- > Unpacking objects: 100% (10/10), done.

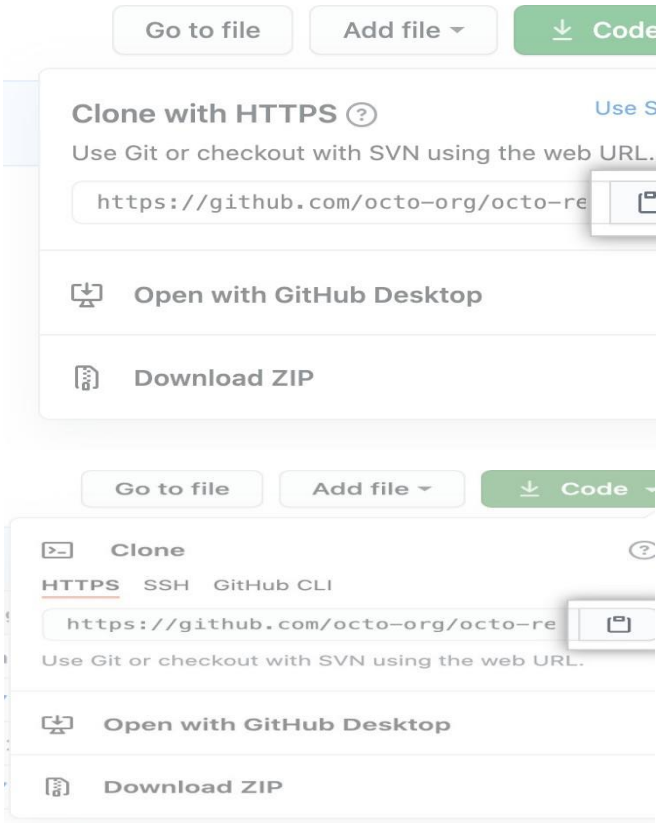
Configuring Git to sync your fork with the originalrepository

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

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



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



4. Open Git Bash.
5. Change directories to the location of the fork you cloned.
    - To go to your home directory, type just cdwith 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 andpress 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.

```
$ git remote -v
> origin      https://github.com/YOUR_USERNAME/YOUR_FORK.git (fetch)
> origin      https://github.com/YOUR_USERNAME/YOUR_FORK.git (push)
> 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 ideaswithout 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 intotheir 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. Formore information, see "Roles in an organization." If you have access to a private repository and the owner permits forking, you can forkthe 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 onGitHub."

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 thechanges 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 thecode on GitHub. For more information see, "Create a repository."

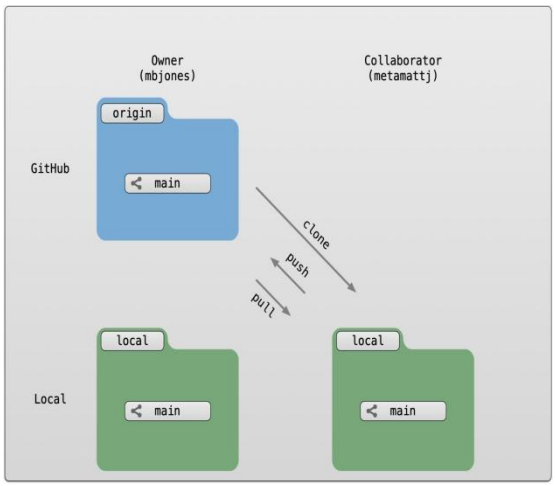
Each repository in GitHub is owned by a person or an organization. You can interact with the people, repositories, and organizations by connecting and following them onGitHub. For more information see "Be social." GitHub has a great support community where you can ask for help and talk to peoplefrom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

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

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

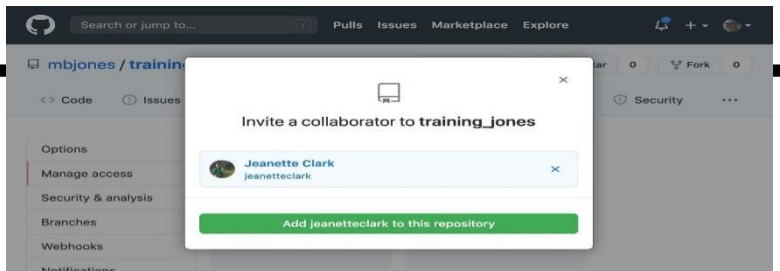
The simplest way to collaborate with Git is to use a shared repository on a hosting service such as GitHub, and use this shared repository as the mechanism to move changes from one collaborator to another. While there are other more advanced ways to sync git repositories, this "hub and spoke" model works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

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



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.



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

Step 1: Collaborator clone

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

Step 2: Collaborator Edits

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

Step 3: Collaborator commit and push

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

Step 4: Owner pull

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

Step 5: Owner edits, commit, and push

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

Step 6: Collaborator pull

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

Challenge

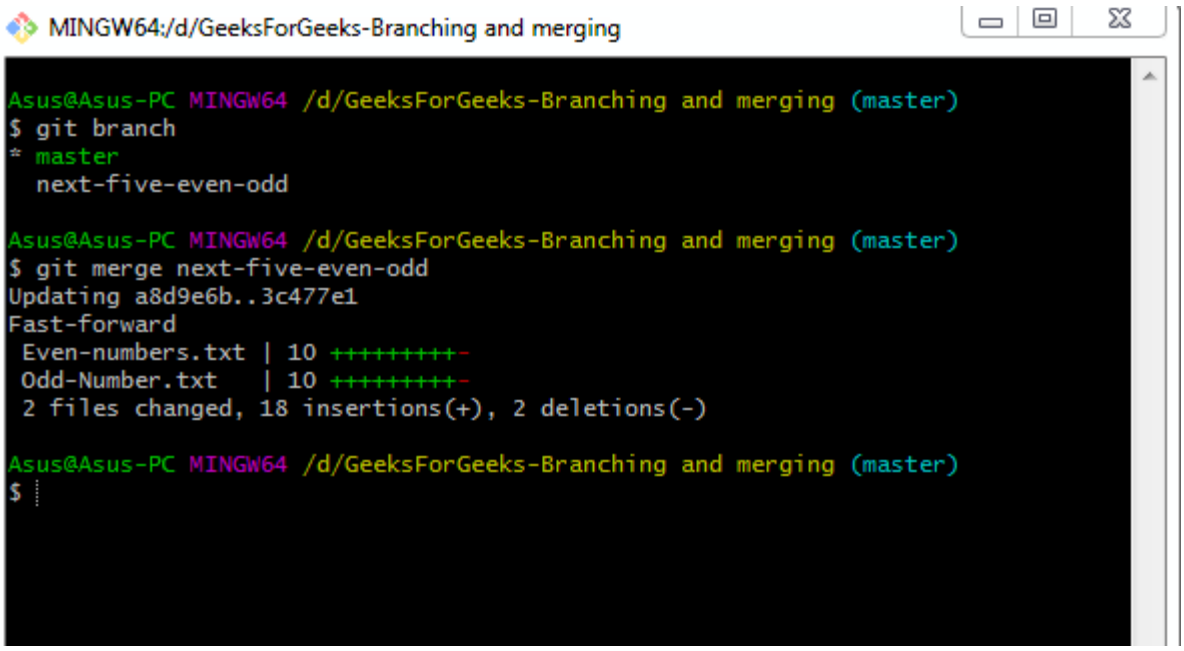
Now that the instructors have demonstrated this conflict-free process, break into pairs and try the same with your partner. Start by designating one person as the Owner and one as the Collaborator, and then repeat the steps described above:

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

Merge conflicts

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

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



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

How to resolve a conflict

Abort, abort, abort…

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

```
git merge --abort
```

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

Checkout

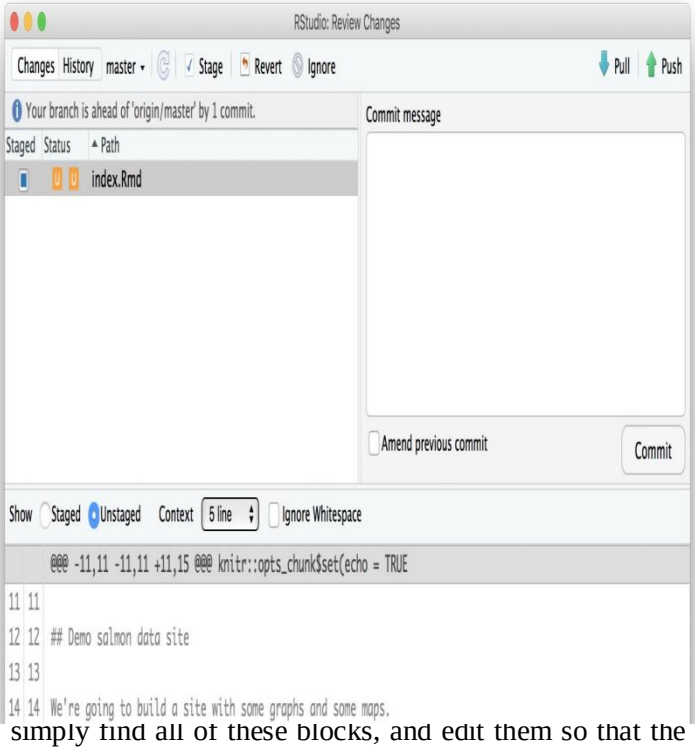
The simplest way to resolve a conflict, given that you know whose version of the file you want to keep, is to use the commandline `git` program to tell git to use either your changes (the person doing the merge), or their changes (the other collaborator).

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

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

Pull and edit the file

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



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

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

Producing and resolving merge conflicts

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

Owner and collaborator ensure all changes are updated

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

Owner makes a change and commits

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

Collaborator makes a change and commits on the same line

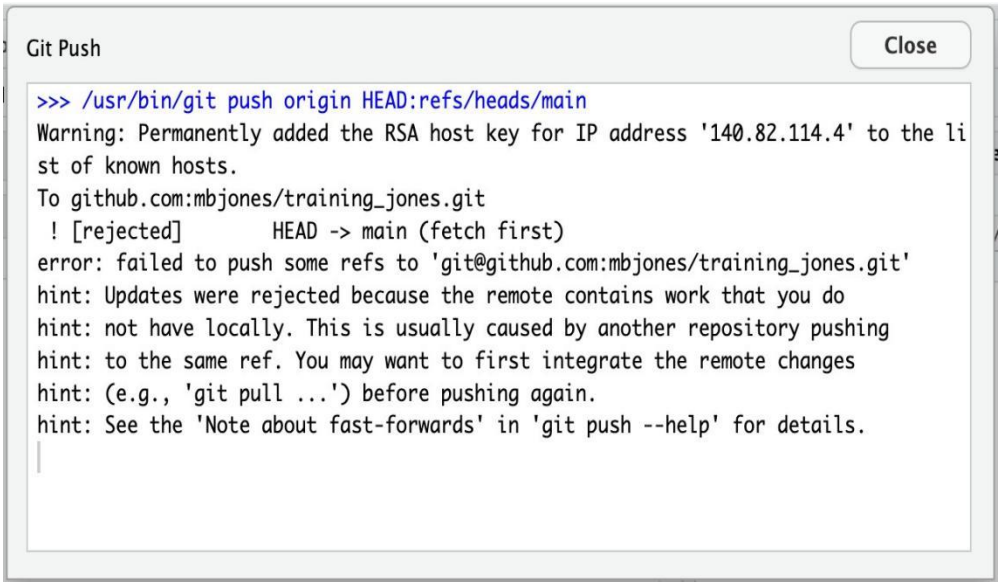
Now the collaborator also makes changes to the same (line 4) of the README.md file in their RStudio copy of the project, adding their name to the line. They then commit. At this point, both the owner and collaborator have committed changes based on their shared version of the README.md file, but neither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

Collaborator pushes the file to GitHub

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

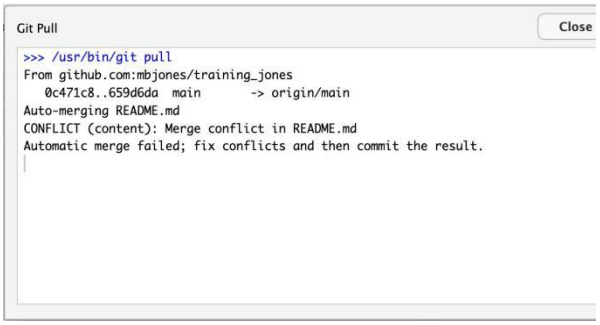
Owner pushes their changes and gets an error

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

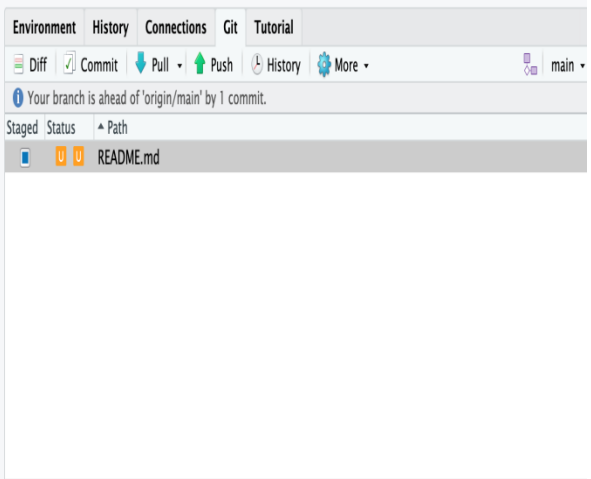


Owner pulls from GitHub to get Collaborator changes

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.

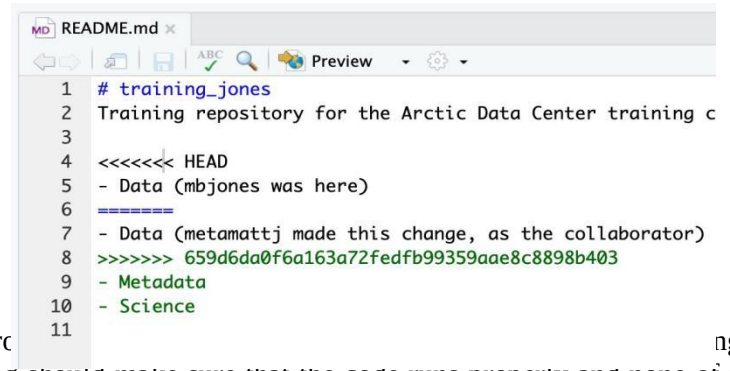


In the Git pane of RStudio, the file is also flagged with an orange 'U', which stands for an unresolved merge conflict.

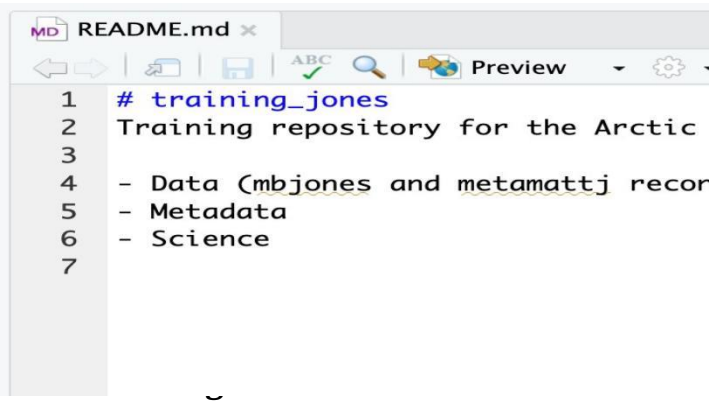


Owner edits the file to resolve the conflict

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



Of course, for scripts and pro ng 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.

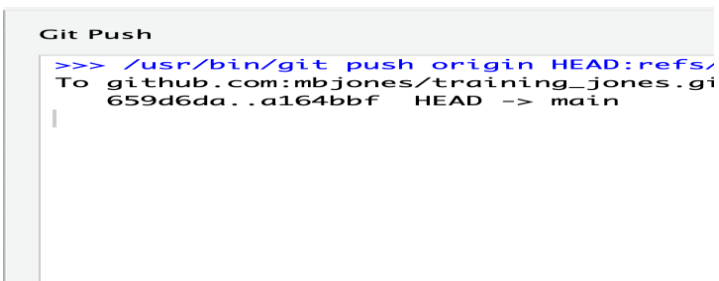


Owner commits the r

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

Owner pushes the resolved changes to GitHub

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



Collaborator pulls the resolved changes from GitHub

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

Both can view commit history

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

Merge Conflict Challenge

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

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

Workflows to avoid merge conflicts

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

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

A good workflow is encapsulated as follows:

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

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

Aim: Reset and Revert

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

How to reset a Git commit

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

Take a look at Figure 1. Here we have a representation of a series of commits in Git. A branch in Git is simply a named, movable pointer to a specific commit. In this case, our branch master is a pointer to the latest commit in the chain.

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

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

Programming and development

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

What happens if we want to roll back to a previous commit. Simple—we can just move the branch pointer. Git supplies the resetcommand 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 logcommand 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, suchas:

$ git revert HEAD

Because this adds a new commit, Git will prompt for the commit message:

```
Revert "File with three lines"


This reverts commit
b764644bad524b80457
7684bf74e7bca3117f5
54.
```

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

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

If we do a git lognow, 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->C0and *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 Cdesignators 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
```

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

So, we can rebase a feature onto master to pick up C4 (e.g., insert it into feature's chain). Using the basic Git commands, it might look like this:

```
$ git checkout feature
$ git rebase master


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

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

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

```
$ git log --oneline master
6a92e7a C4
259bf36 C2
f33ae68
```

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" C3and C5are 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 rebaseoperation—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 catthe file, we can see its contents.

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

We could use the resetcommand, 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: refs/heads/feature returning to

c4533a5 HEAD@{2}: rebase: C5 rebase:
64f2047 HEAD@{3}: 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 commit: from master to feature
6a92e7a HEAD@{8}: C4
259bf36 HEAD@{9}: checkout: moving from feature to master
```

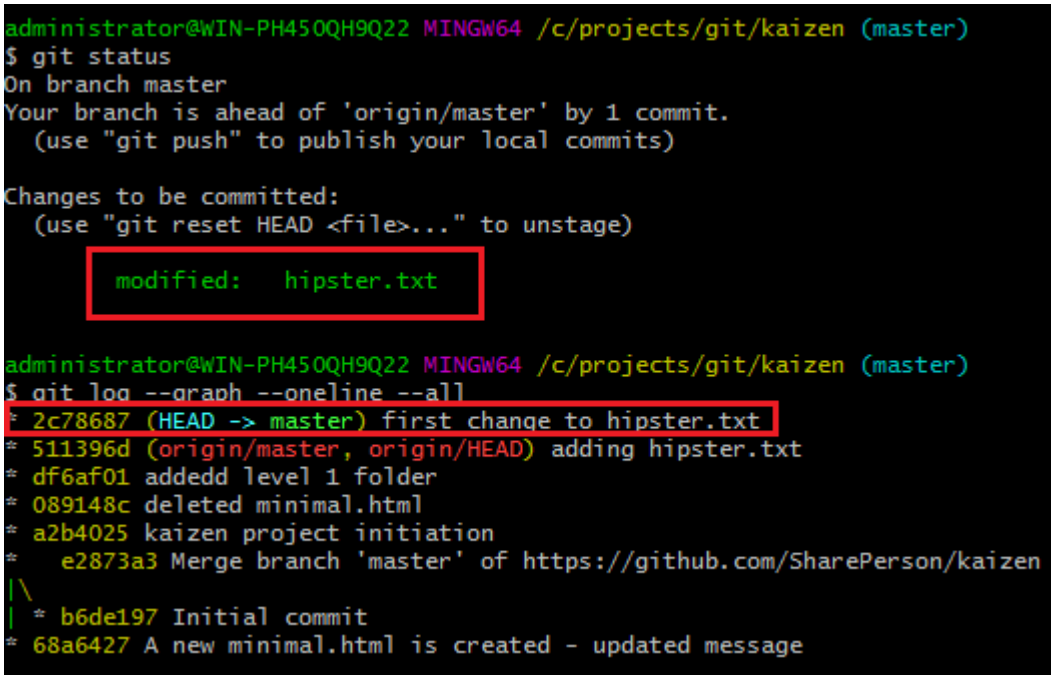
```
000f9ae HEAD@{10}: commit: C3
259bf36 HEAD@{11}: checkout: moving from master to feature
259bf36 HEAD@{12}: commit: C2 f33ae68
HEAD@{13}: commit: C1
5043e79 HEAD@{14}: commit (initial): C0
```

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.

Snapshots

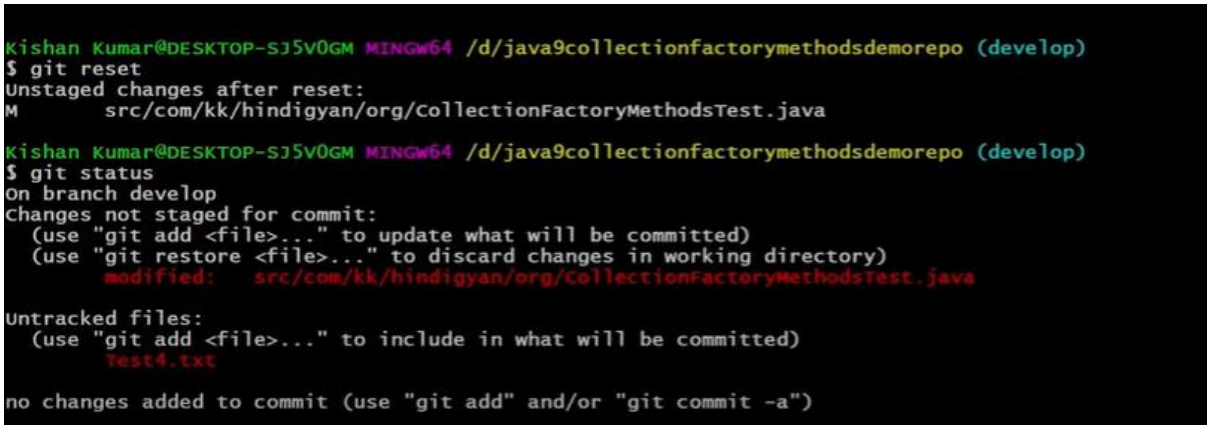
Git log for file before reset



git reset --soft

```
$ git reset --soft HEAD~1

$ git status

On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file1

$ git log --oneline --graph

* 90f8bb1 (HEAD -> master) Second commit
* 7083e29 Initial repository commit
```

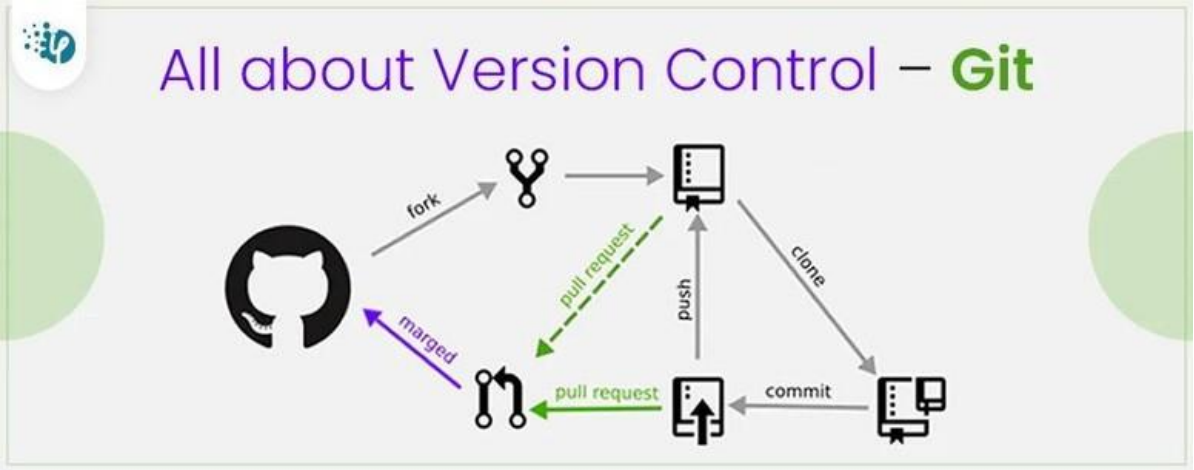
git reset mixed(default)



git reset --hard

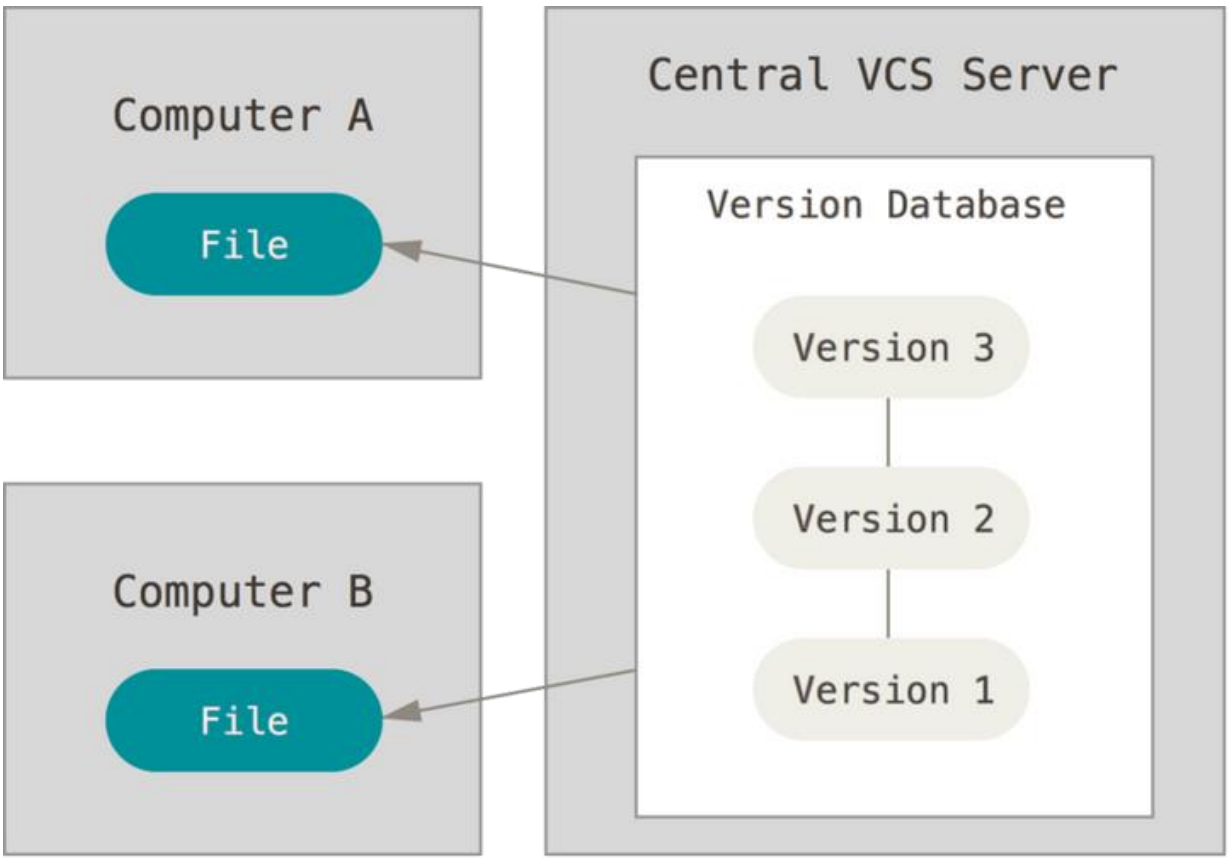
```
saraf@TheBlueDog  ~/Source/Repos/scrap/foo5 (master)
$ git reset --hard e99ff27
HEAD is now at e99ff27 adding file 4

saraf@TheBlueDog  ~/Source/Repos/scrap/foo5 (master)
$ git log --oneline
e99ff27 adding file 4
09ee738 adding file 3
ad63551 adding file 2
334bc8a adding file 1
```

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



Local Version Control Systems

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

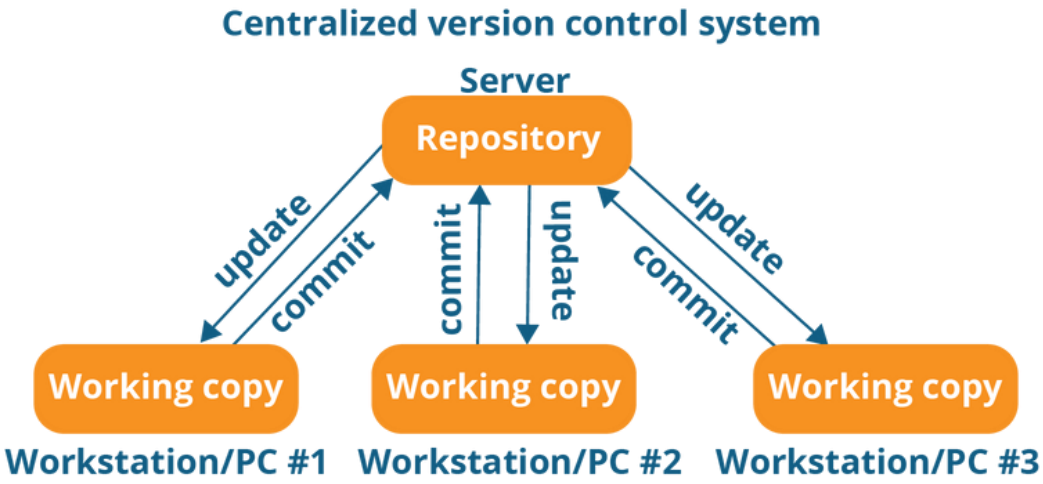


One of the most popular VCS tools was a system called RCS, which is still distributedwith many computers today. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the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

Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a

number of clients that check out files from that central place. For many years, this has been the standard for version control.



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

Distributed Version Control Systems

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

the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that

server, any of the client repositories can be copied back up to the server to restoreit. Every clone is really a full backup of all the data.

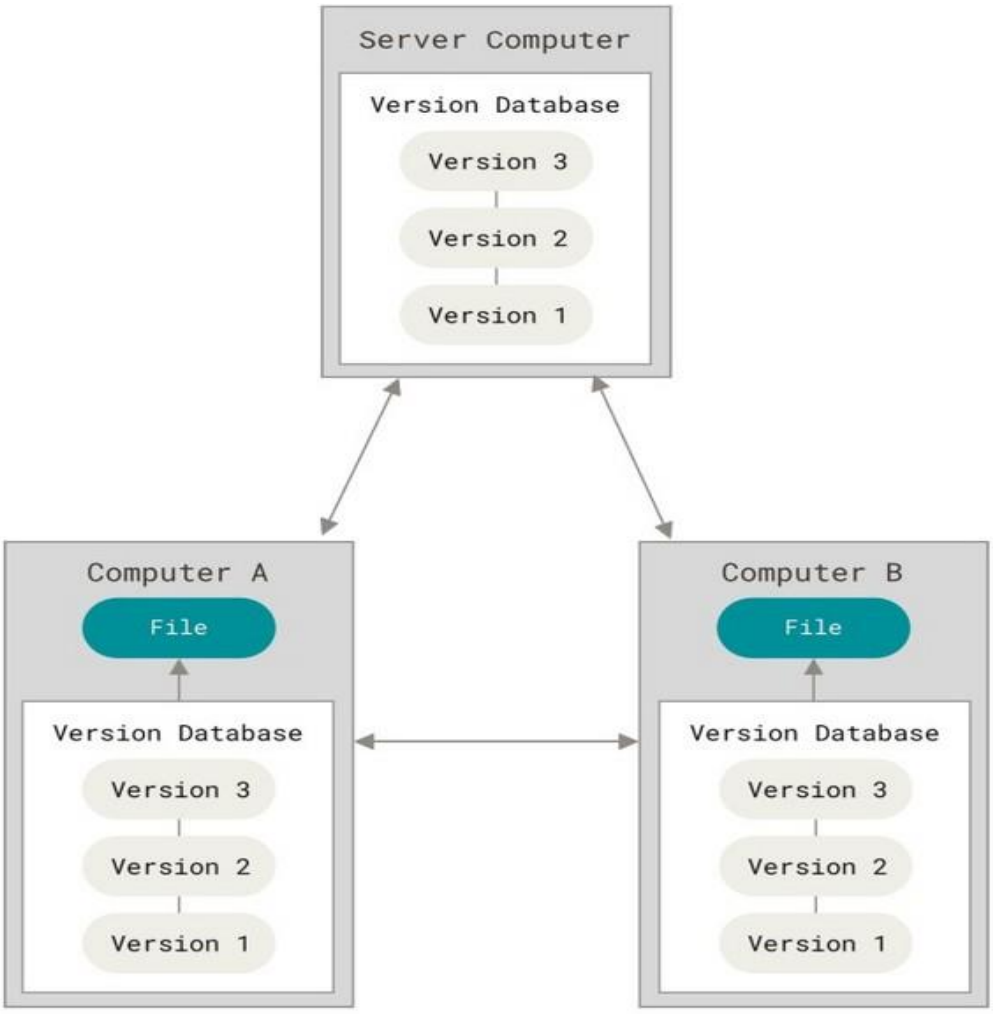


Figure 3. Distributed version control

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

2. Problem

Statement

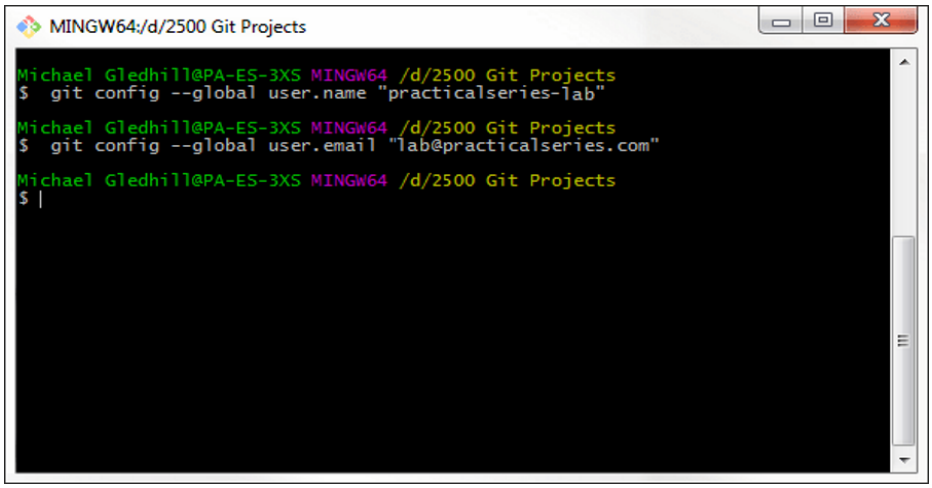
In the project The AWD clone, we created a clone of a non-profit website called help-u using HTML, CSS, and JavaScript. Many causes are listed under the ngos section of this website. The website's eye-catching design aids in attracting visitors. It is a day-to-day application that is useful in a person's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

3. Objective

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

4. Concepts and Commands

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



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

git config user.email

To verify linked username

git init

To make folder git ready

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

git add –a

To push all the files to repo

git add filename

To push a particular file to repo

```
gauta@gautam MINGW64 ~/OneDrive/Desktop/Project (main)
$ git add .
```

git status

git branch name

To make a new branch

git checkout name

To switch between branch

git branch -d branch name

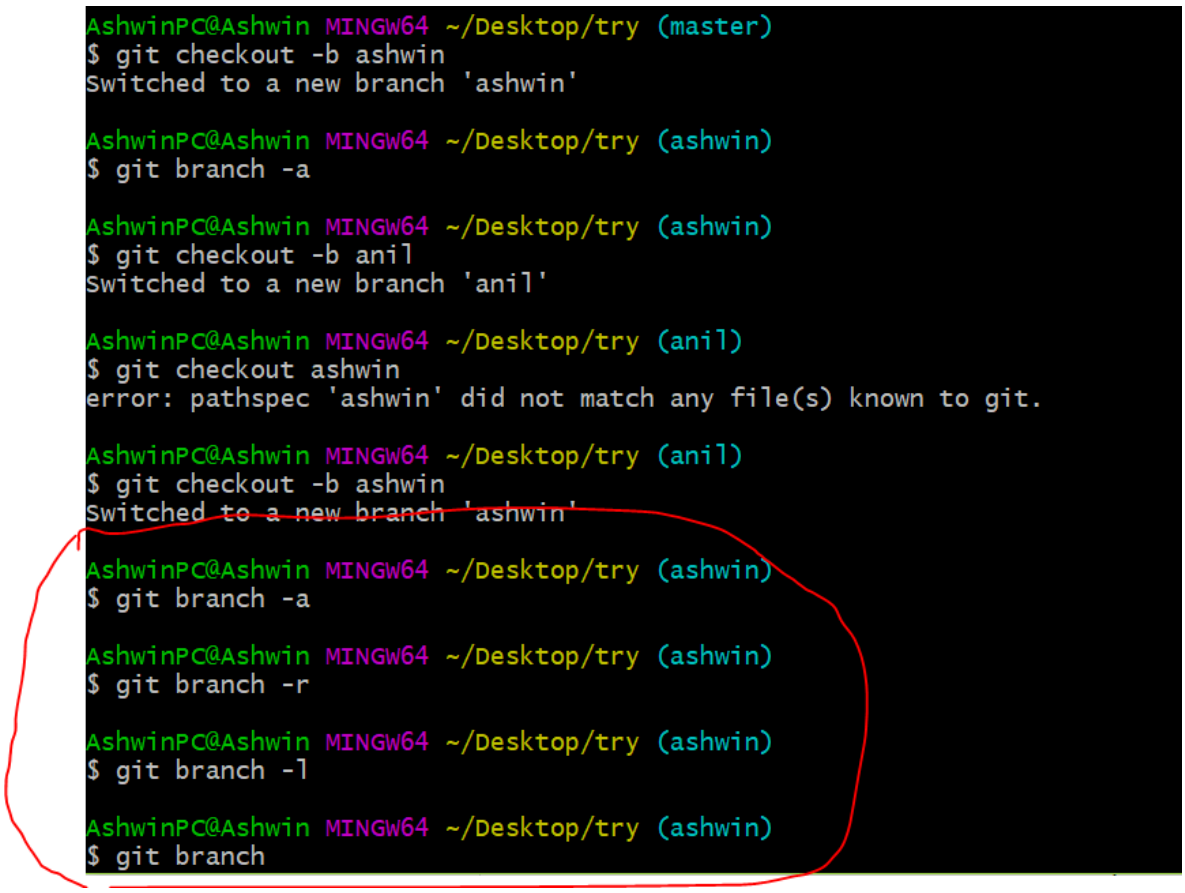
To delete branch (**Soft delete** because it ask to merge **)**

git branch -D branch name

To delete branch **(Hard delete** because it don’t ask to merge**)**

git branch

To see number of branches



git branch -m new branch name

To rename a branch (we need to be in that branch)

git branch -r
To see number of branches

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

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

git log --online –graph

To get log in graph format

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

pwd
Present working Directory

```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ pwd
/e/Coding-Ninjas-java-
```

```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ git commit -m "added files"
On branch activity
nothing to commit, working tree clean
```

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

```
HP@LAPTOP-PNM35MOK MINGW64 ~/OneDrive/Desktop/SCM PROJECT/combined project (ma
er)
$ git clone https://github.com/ASTITAV-2K3/Project.git
Cloning into 'Project'...
remote: Enumerating objects: 80, done.
remote: Counting objects: 100% (80/80), done.
remote: Compressing objects: 100% (72/72), done.
Receiving objects:  20% (16/80), 2.69 MiB | 1.07 MiB/s
```

git merge branch name
To merge sub branch with master

```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ git merge 'activity'
Already up to date.
```

mv old-file-name new-file-name

To rename a flolder (we have to do staging for this)

git mv old-file-name new-file-name

To rename a folder (no need for staging)

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git rm --cached newfile1.txt
rm 'newfile1.txt'

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    newfile1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        newfile1.txt
```

git restore --staged filed name

To reverse to to previous version

git restore filename

To go to previous command

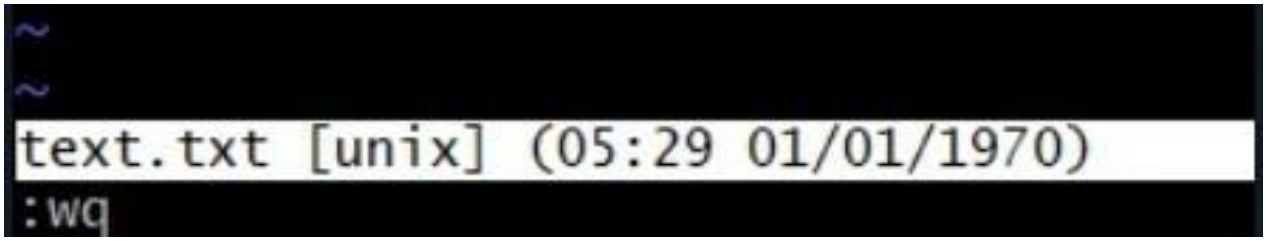
```
wardah@wardah:~/git_project$ git restore my_git.txt
wardah@wardah:~/git_project$ git status
On branch master

No commits yet

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

wardah@wardah:~/git_project$
```

git mergetool

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

:wq

To quit from special screen



rm -rf .git

To delete whole git folder

rm -rf filename

To delete a particular file

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

To make new remote

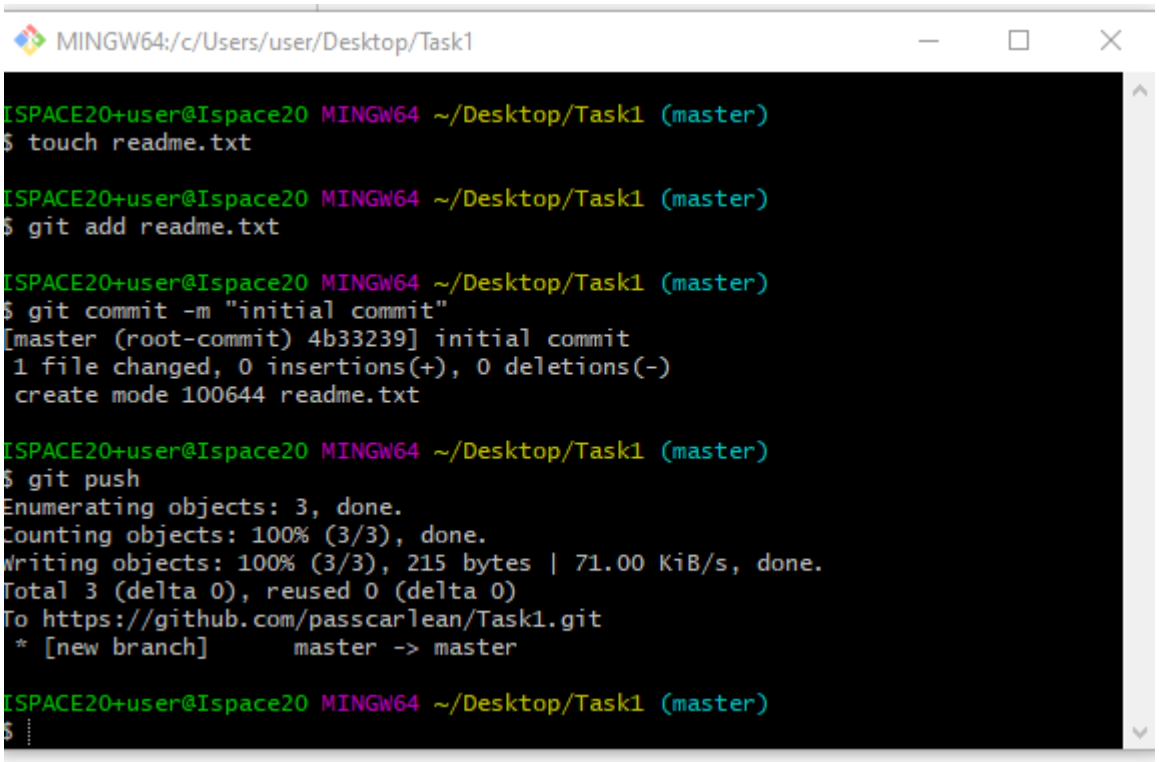
git push -u master remote-nam To make data visible on sscloud

git pull link-of-repo-on-github
To make changes done on cloud visible on the system.

git remote -v
to see the location where remote is being stored

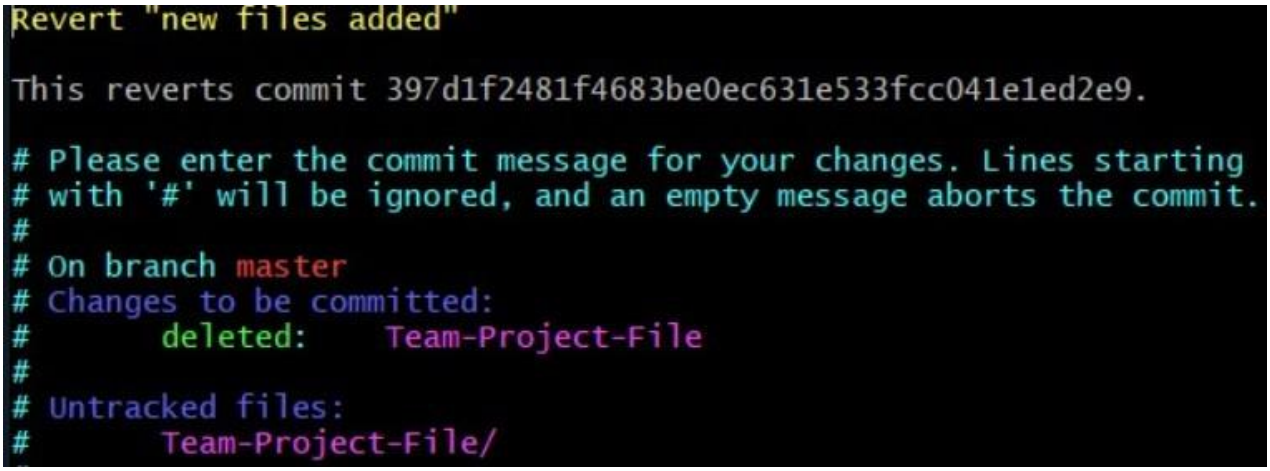
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote set-url origin  https://github.com/URLChanged

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote -v
origin  https://github.com/URLChanged (fetch)
origin  https://github.com/URLChanged (push)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

touch filename
To make css/html/c++ files



git revert checksum

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



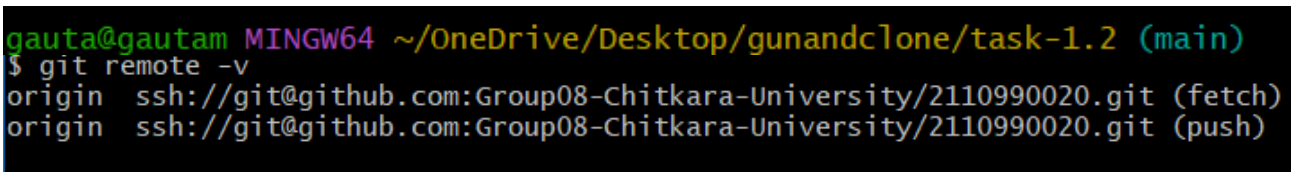
rm -rf .git
To delete whole git folder
rm -rf filename
To delete a particular file

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

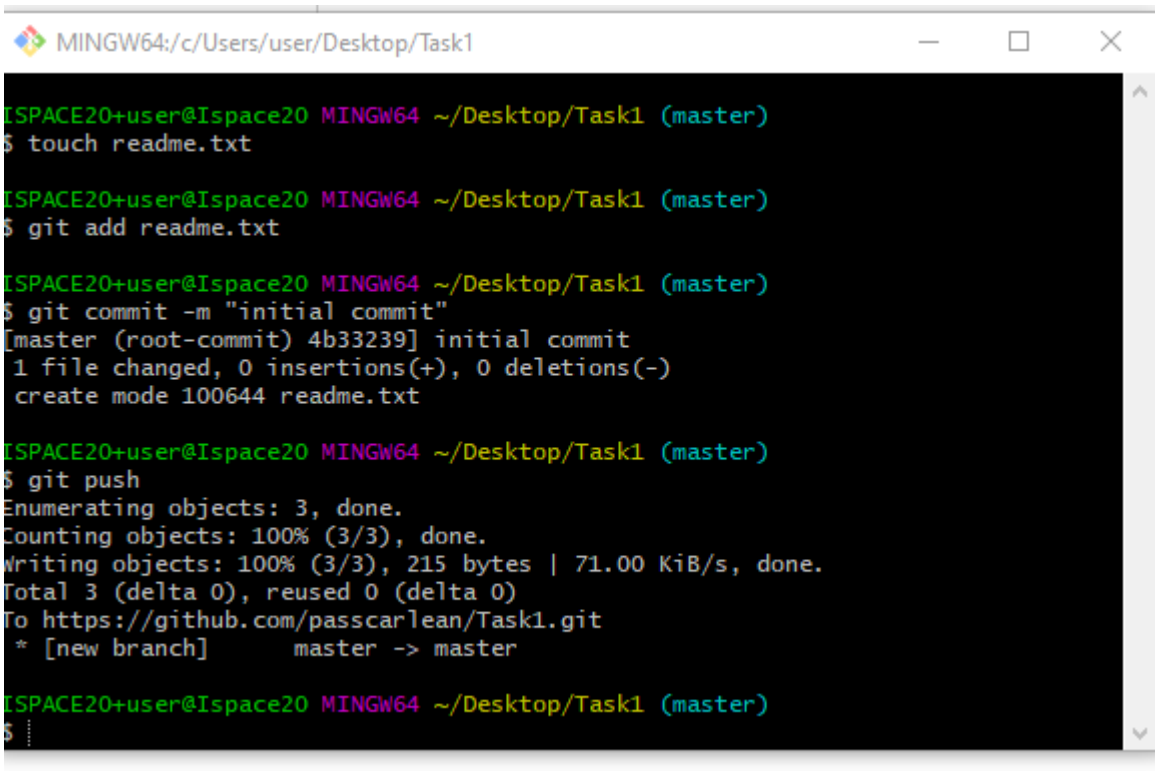
on-github
To make new remote

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

git pull link-of-repo-on-github
To make changes done on cloud visible on the system.
git remote -v
to see the location where remote is being stored



touch filename
To make css/html/c++ files



git revert checksum

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

```
Revert "new files added"

This reverts commit 397d1f2481f4683be0ec631e533fcc041e1ed2e9.

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

touch .gitignore

git reset --hard checksum

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

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

git reset --mixed checksum

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

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

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

git reset --soft checksum

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

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

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

git remote

To see the name of origin formed

```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ git remote
origin
```

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

```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ git remote rename origin origindev

HP@Aaryan-Sooc          /e/Coding-Ninjas-java-/Graphs (master)
$ git remote
origindev
```

git remote remove name

To delete remote

```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ git remote remove origin
```

cat file-name

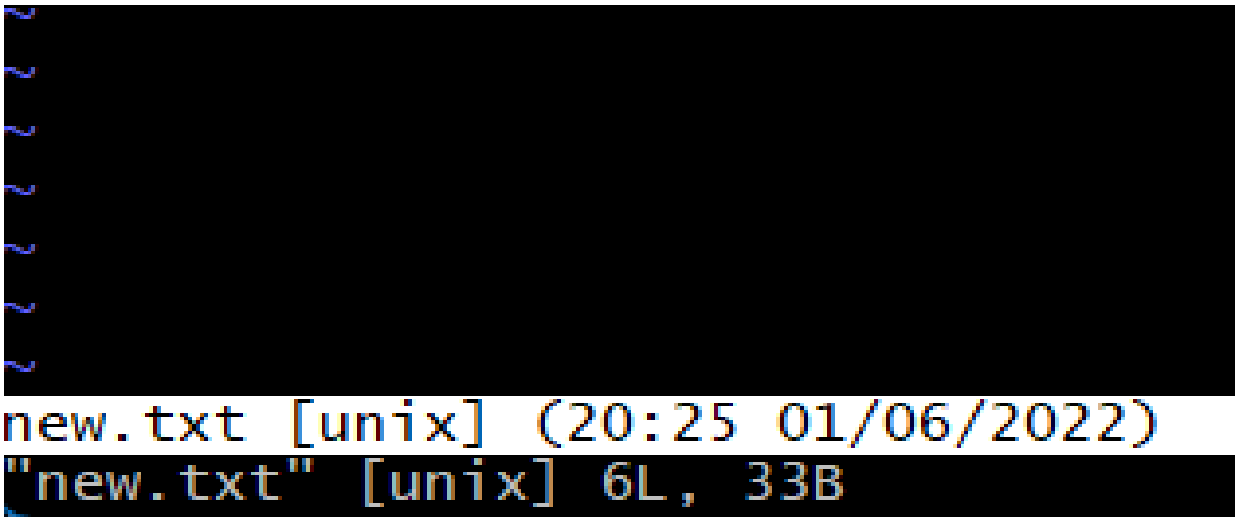
To see the data stored in file in git bash without opening the file

```
gauta@gautam MINGW64 ~/OneDrive/Desktop/gunandclone/task-1.2 (main)
$ git remote remove origindev
```

vi file-name

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

5. Workflow and discussion



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

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

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

I made a new repository

Create a new repository

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

Owner * Repository name *

ASTITAV-2K3 / demo

Great repository names are short and memorable. Need inspiration? How about studious-octo-funicular?

Description (optional)

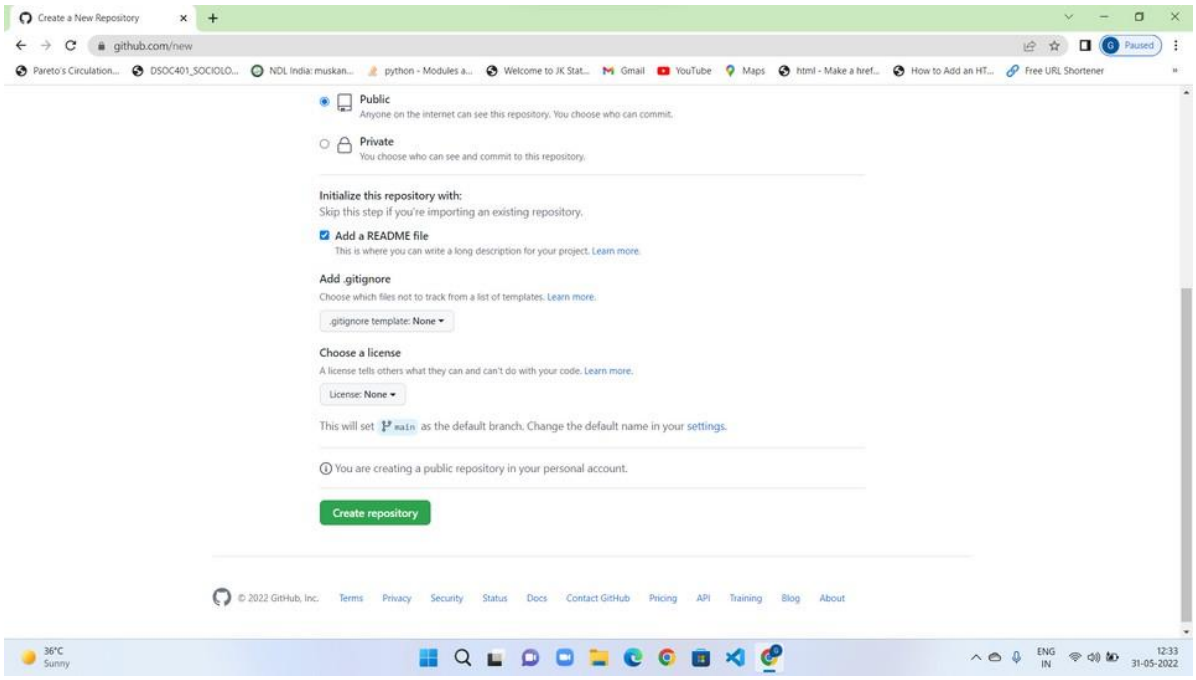
Public
Anyone on the internet can see this repository. You choose who can commit.

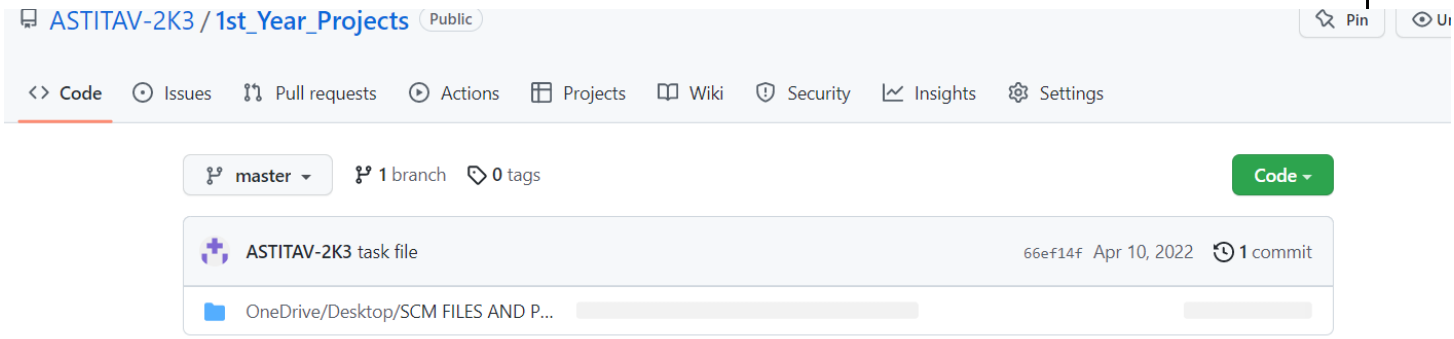
Private
You choose who can see and commit to this repository.

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

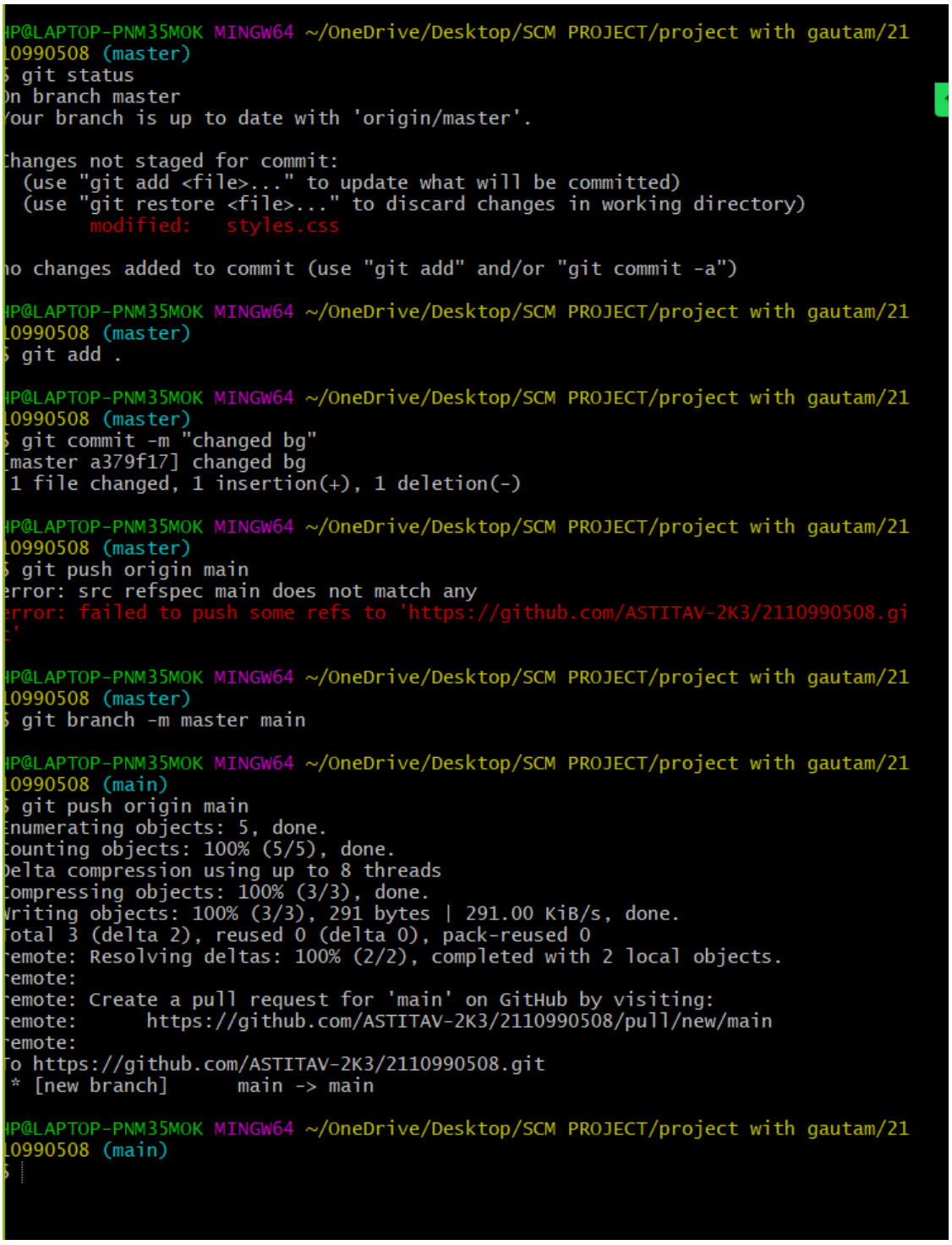
Add a README file
This is where you can write a long description for your project. Learn more.

Add .gitignore

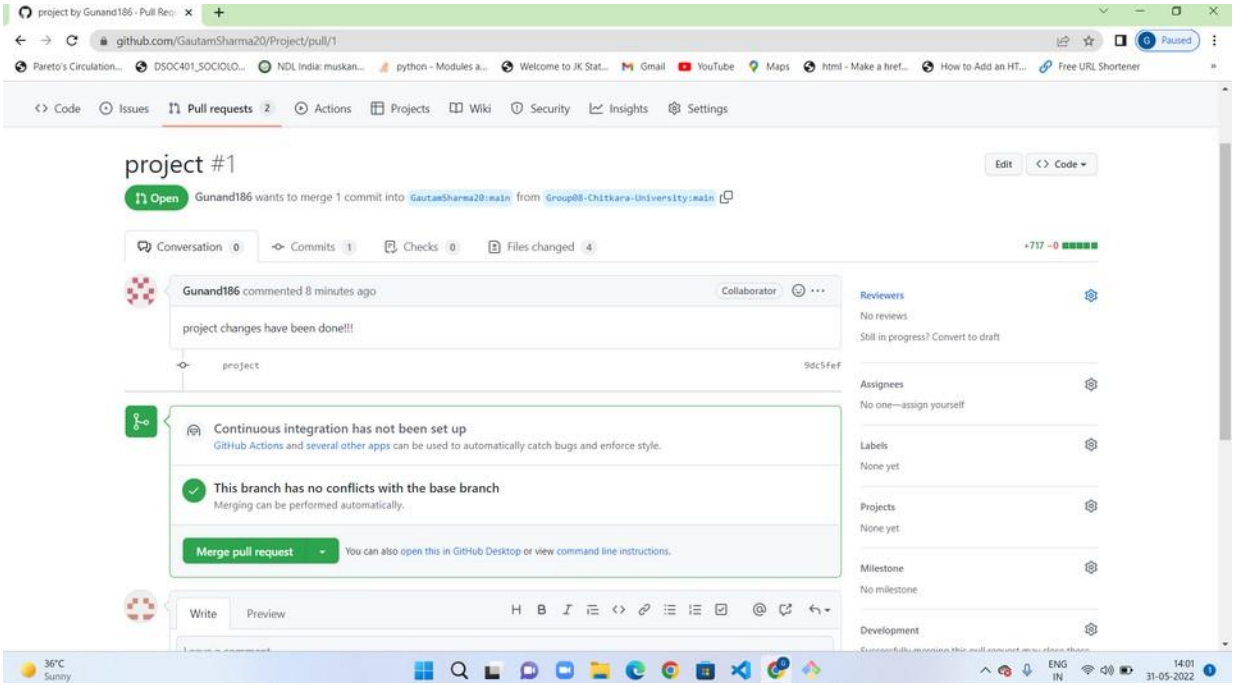


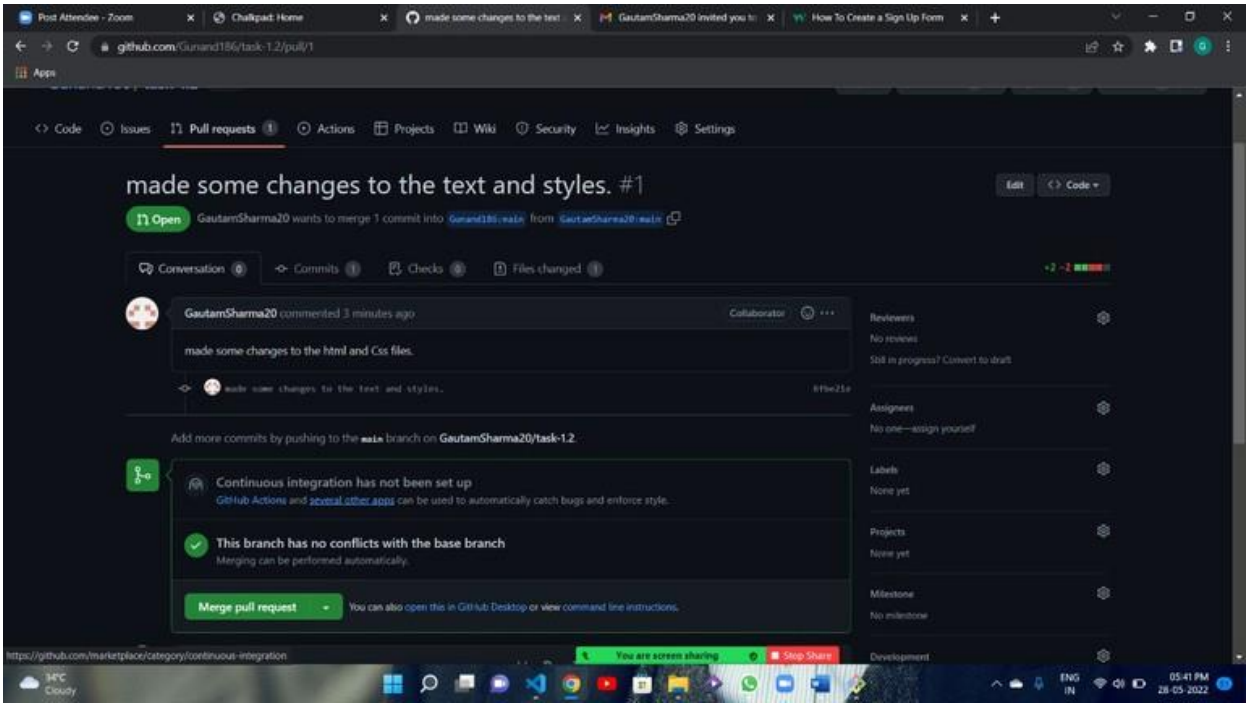


I pushed some files into the project



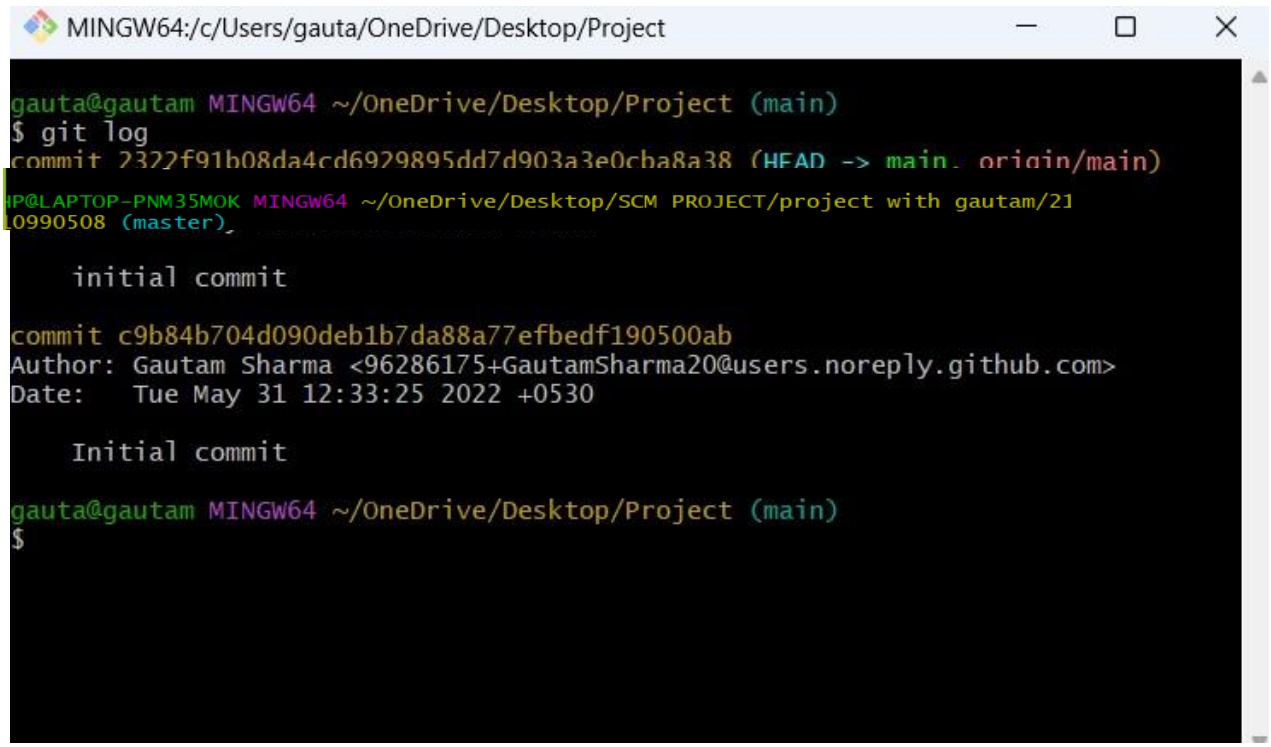
All the collaborators made some changes to thefiles or added new files:


project #1
Open
Gunand186 wants to merge 1 commit into GautamSharma20:main from Group08-Chitkara-University:main
Conversation 0
Commits 1
Checks 0
Files changed 4
Gunand186 commented 8 minutes ago
Collaborator
project changes have been done!!!
project
9dc5fef
Continuous integration has not been set up
GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop or view command line instructions.
Write
Preview
Reviewers
No reviews
Still in progress? Convert to draft
Assignees
No one—assign yourself
Labels
None yet
Projects
None yet
Milestone
No milestone
Development


made some changes to the text and styles. #1
Open
GautamSharma20 wants to merge 1 commit into Gunand186:main from GautamSharma20:main
Conversation 0
Commits 1
Checks 0
Files changed 1
GautamSharma20 commented 3 minutes ago
Collaborator
made some changes to the html and Css files.
Add more commits by pushing to the main branch on GautamSharma20/task-1.2.
Continuous integration has not been set up
GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop or view command line instructions.
Reviewers
No reviews
Still in progress? Convert to draft
Assignees
No one—assign yourself
Labels
None yet
Projects
None yet
Milestone
No milestone
Development

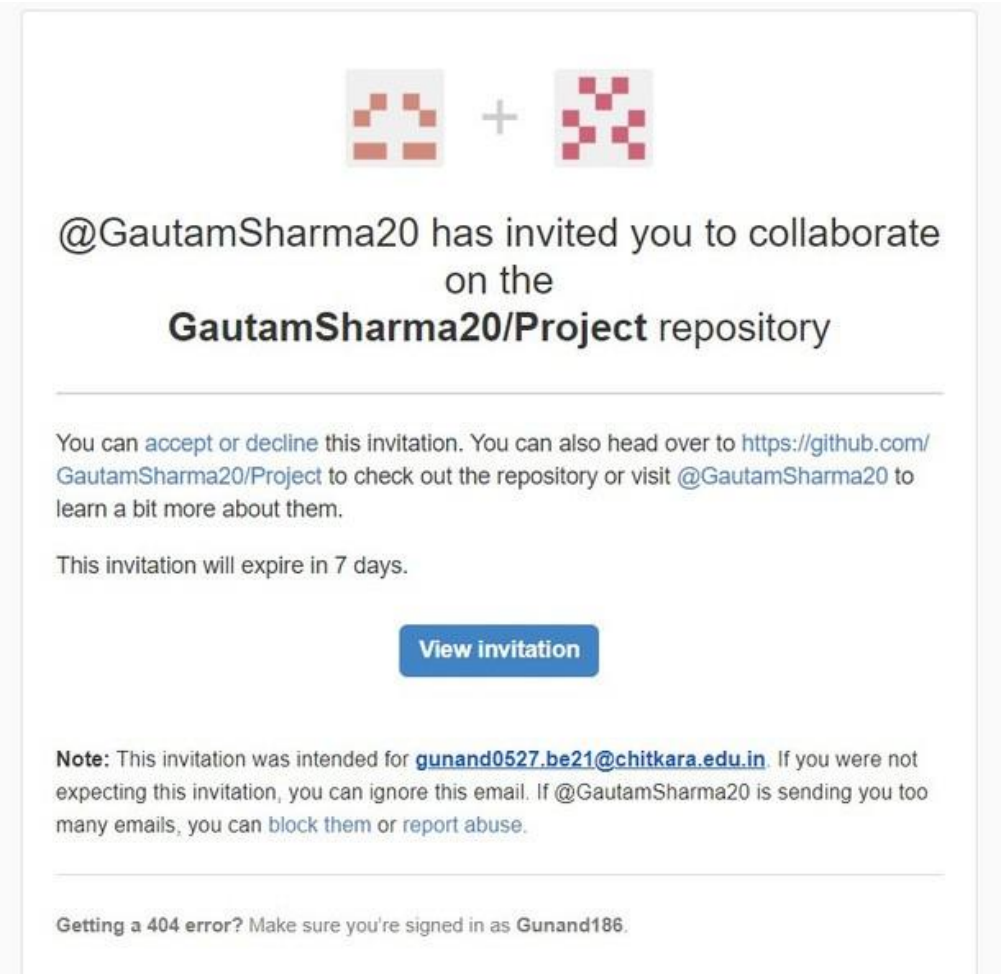
I also made some changes and send the pull request to the collaborators and they merged it.

git log after forking and collaborating



Collab with -

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



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

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

This invitation will expire in 7 days.

View invitation

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

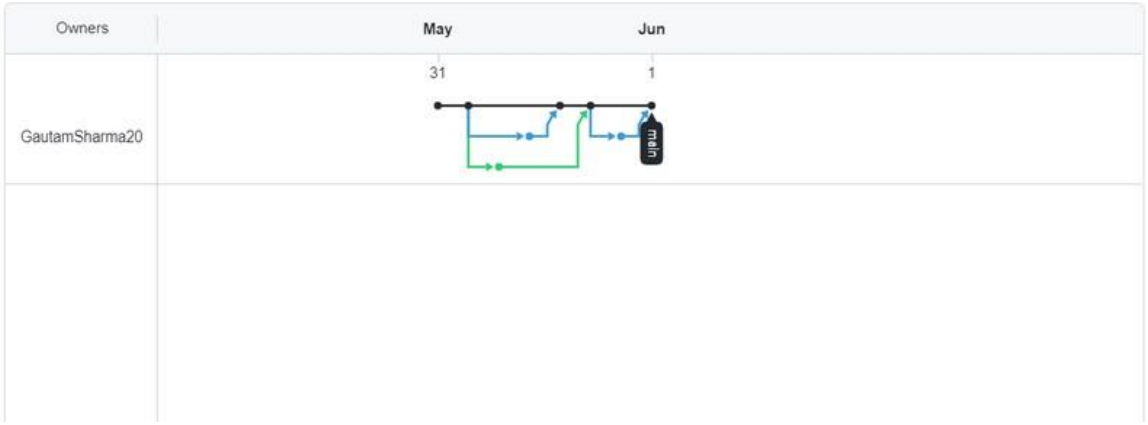
Getting a 404 error? Make sure you're signed in as **Gunand186**.

Added Guanand Collaborater to projectRepo

6. Network Graph

Network graph

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



6. Network Graph

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

Links used for reference

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

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

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

https://www.udemy.com/course/git-started-with-github/?LSNPUBID=JVFxdTr9V80&ranEAID=JVFxdTr9V80&ranMID=39197&ranSiteID=JVFxdTr9V80-GmotHk.p_rwC77qokoBs_w&utm_medium=udemyads&utm_source=aff-campaign