

Subject Name: **Source Code Management**

Subject Code: **CS181**

**Cluster: Zeta**

Department: DCSE

**Submitted By:**

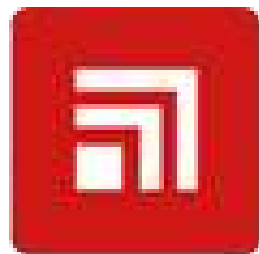
Vasu Khattar

2110992054

G27-A

**Submitted To:**

Dr. Anuj Jain



## List of Tasks

S. No	Task Title	Page No.
1	Setting up of Git Client.	
2	Setting up GitHub Account.	
3	Generate logs.	
4	Create and Visualize branches.	
5	Git lifecycle description.	

# Task 1

## Installing and Configuring the Git client

The following sections list the steps required to properly install and configure the Git clients - Git Bash and Git GUI - on a Windows 7 computer.

Git is also available for Linux and Mac. The remaining instructions here, however, are specific to the Windows installation.

**Be sure to carefully follow all of the steps in the first five sections.** The last section, 6, is optional.

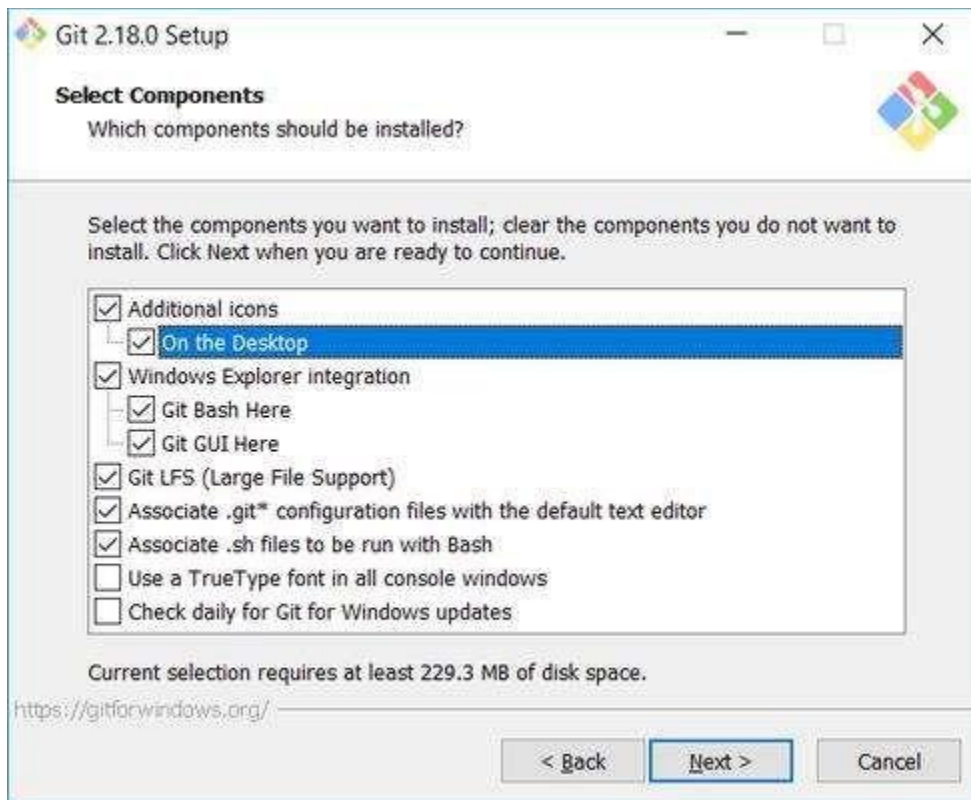
There is also a section on common problems and possible fixes at the bottom of the document.

### 1. Git installation

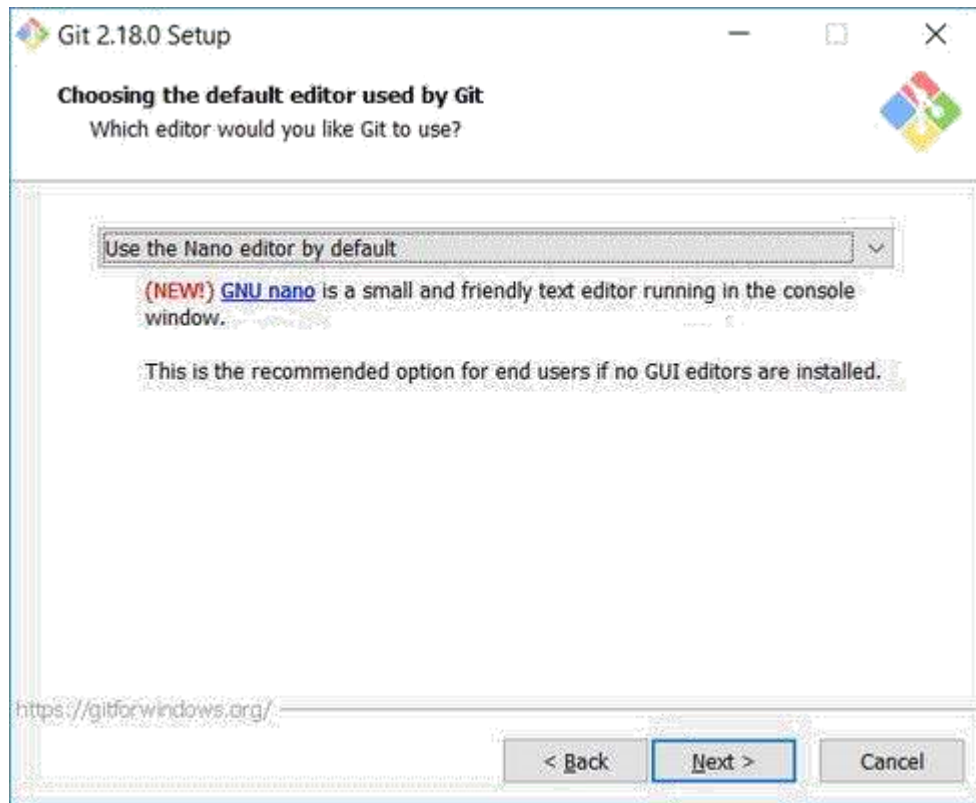
Download the Git installation program (Windows, Mac, or Linux) from <http://git-scm.com/downloads>.

When running the installer, various screens appear (Windows screens shown). Generally, you can accept the default selections, **except in the screens below where you do NOT want the default selections:**

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

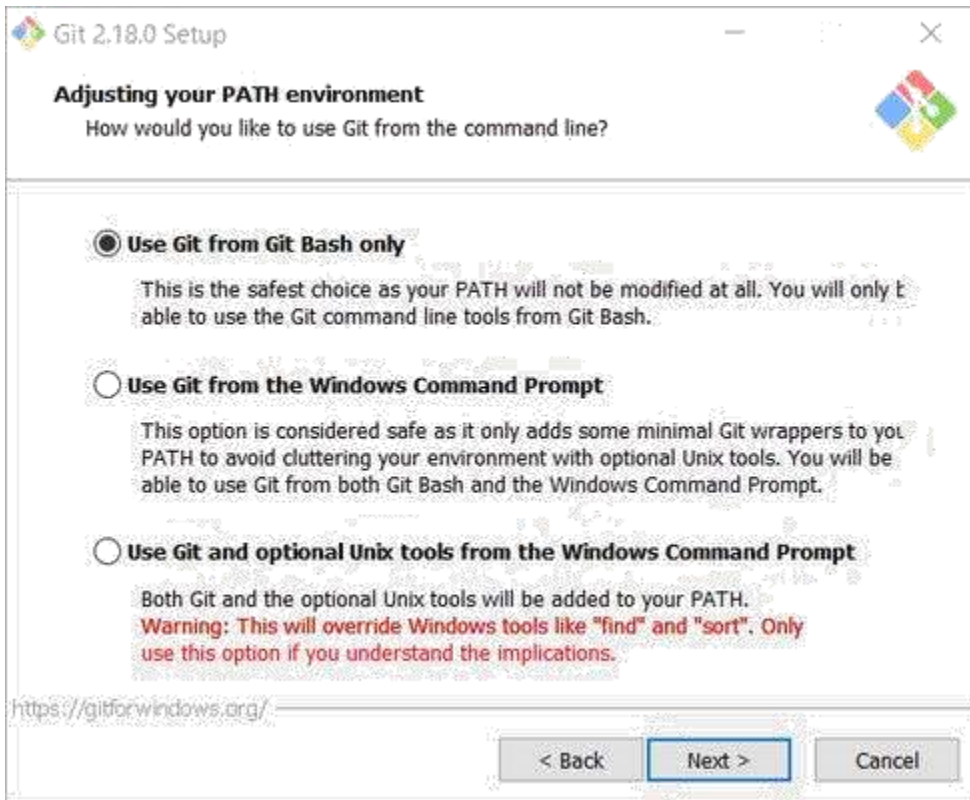


In the **Choosing the default editor used by Git** dialog, it is strongly recommended that you **DO NOT select the default VIM editor** - it is challenging to learn how to use it, and there are better modern editors available. Instead, choose **Notepad++ or Nano** - either of those is much easier to use. **It is strongly recommended that you select Notepad++, BUT YOU MUST INSTALL NOTEPAD++ first! Find the installation with Google.**

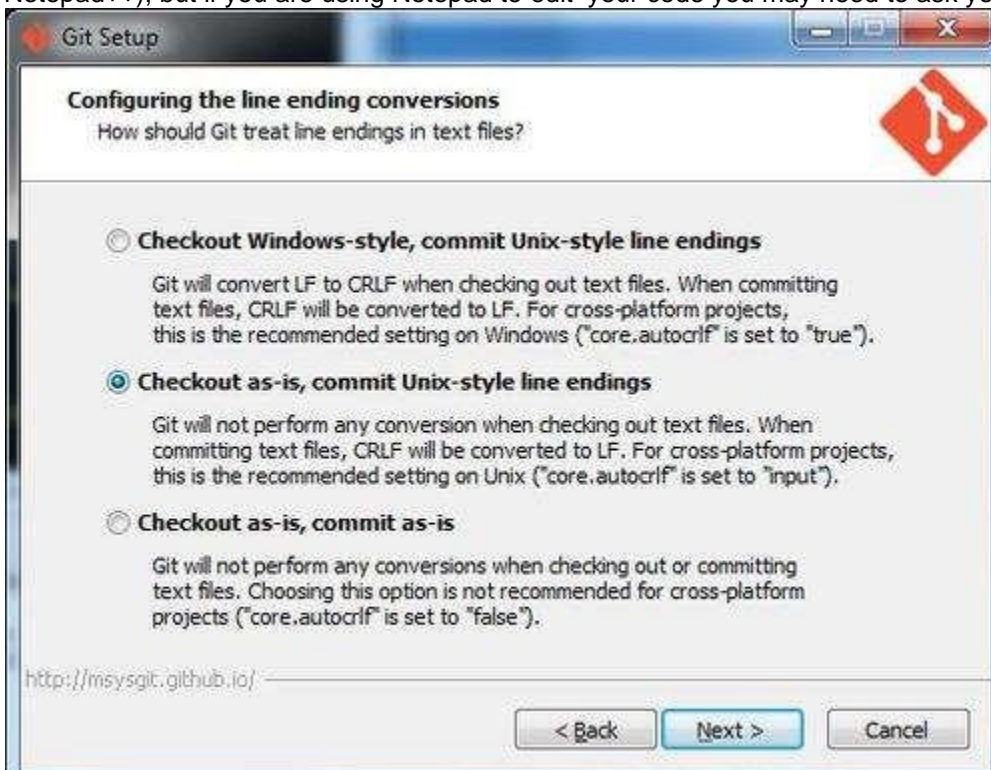


In the **Adjusting your PATH** screen, all three options are acceptable:

- **Use Git from Git Bash only:** no integration, and no extra commands in your command path
- **Use Git from the Windows Command Prompt:** adds flexibility - you can simply run git from a Windows command prompt, and is often the setting for people in industry - but this does add some extra commands.
- **Use Git and optional Unix tools from the Windows Command Prompt:** this is also a robust choice and useful if you like to use Unix commands like grep.



In the **Configuring the line ending** screen, select the middle option (**Checkout as-is, commit Unix-style line endings**) as shown. This helps migrate files towards the Unix-style (LF) terminators that most modern IDE's and editors support. The Windows convention (CR-LF line termination) is only important for Notepad (as opposed to Notepad++), but if you are using Notepad to edit your code you may need to ask your instructor for help.



## 2. Configuring Git to ignore certain files

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

By default, Git tracks **all** files in a project. Typically, this is **NOT** what you want; rather, you want Git to ignore certain files such as **.bak** files created by an editor or **.class** files created by the Java compiler. To have Git automatically ignore particular files, create a file named **.gitignore** ( note that the filename begins with a dot) in the **C:\users\name** folder (where **name** is your MSOE login name).

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

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

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

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

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

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

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

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

### 3. Configuring Git default parameters

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

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

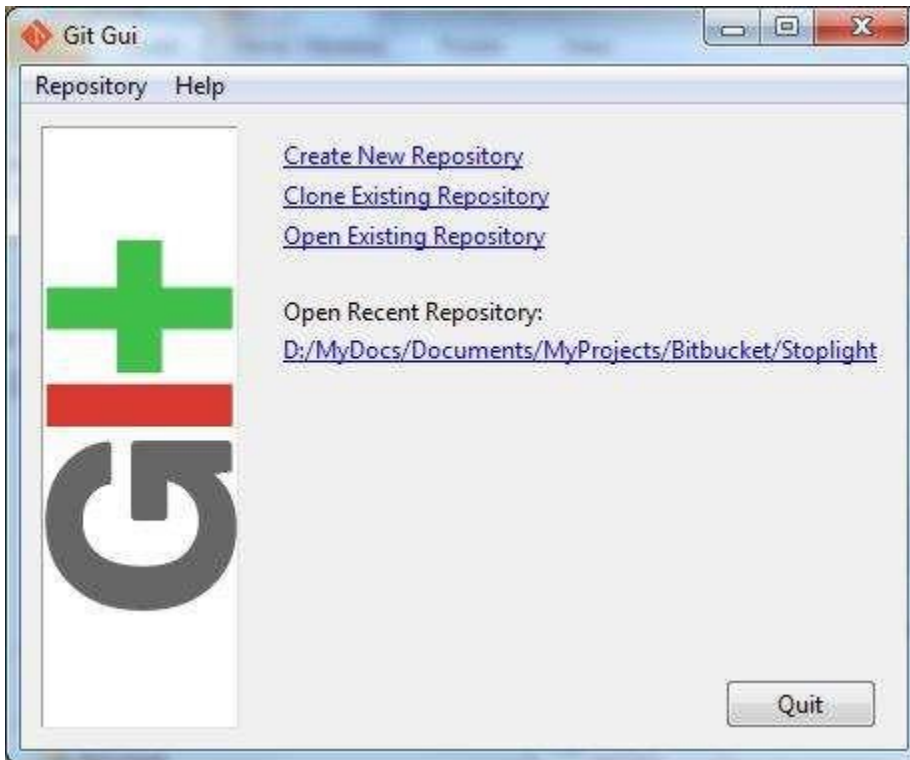
- b. Enter the command (replacing **name** as appropriate) `git config --global core.excludesfile c:/users/name/. gitignore`
- This tells Git to use the **.gitignore** file you created in step 2
  - **NOTE: TO avoid typing errors, copy and paste the commands shown here into the Git Bash window, using the arrow keys to edit the red text to match your information.**
- c. Enter the command `git config --global user.email "name@msoe.edu"`
- This links your Git activity to your email address. Without this, your commits will often show up
- as "unknown login".
- Replace **name** with your own MSOE email name.
- d. Enter the command `git config --global user.name "Your Name"`
- Git uses this to log your activity. Replace "**Your Name**" by your actual first and last name.
- e. Enter the command `git config --global push.default simple`
- This ensures that all pushes go back to the branch from which they were pulled. Otherwise pushes will go to the master branch, forcing a merge.

## 4. Generating public/private key pairs for authentication

**This part is critical and used to authenticate your access to the repository.**

You will eventually be storing your project files on a remote Bitbucket or other server using a secure network connection. The remote server requires you to authenticate yourself whenever you communicate with it so that it can be sure it is you, and not someone else trying to steal or corrupt your files. Bitbucket and Git together use public key authentication; thus you have to generate a pair of keys: a public key that you (or your instructor) put on Bitbucket, and a private key you keep to yourself (and guard with your life).

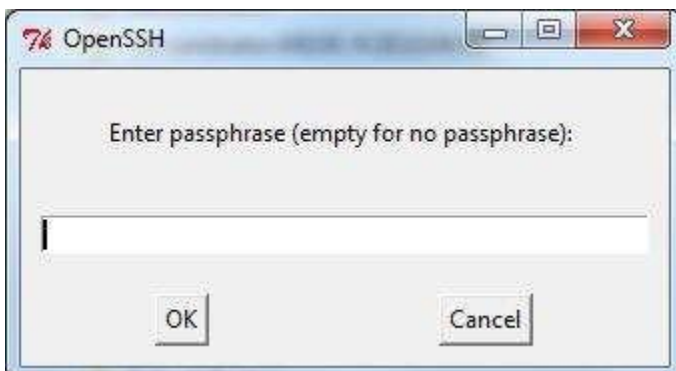
Generating the key pair is easy: From within File Explorer, right-click on any folder. From the context menu, select **Git GUI Here**. The following appears:



From the **Help** menu, select the **Show SSH Key** command. The following pup-up dialog appears:



Initially, you have no public/private key pair; thus the message "**No keys found**" appears within the dialog. Press the **Generate Key** button. The following dialog appears:






Do **NOT** enter a passphrase - just press **OK** twice. When you do, the dialog disappears and you should see something like the following - but your generated key will be different:



The keys have been written into two files named **id\_rsa** and **id\_rsa.pub** in your **c:/Users/*username*/.ssh** folder (where ***username*** is your MSOE user name). Don't ever delete these files! To configure Bitbucket to use this key:

1. Click on the **Copy to Clipboard** button in the Git GUI Public Key dialog.
2. Log in to Bitbucket
3. Click on your picture or the  icon in the left pane and select **Settings**.
4. Select **SSH keys** under **Security**.
5. Click on the **Add key** button.
6. Enter a name for your key in the **Label** box in the Bitbucket window. If your key is ever compromised (such as someone gets a copy off of your laptop), having a clear name will help you know which key to delete. A good pattern to follow is to name the computer used to generate the key followed by the date you generated it; for instance: "MSOE laptop key 2012-02-28".
7. Paste the key from the Clipboard into the **Key** text box in the Bitbucket window, and add it.

You should now be able to access your repository from your laptop using the ssh protocol without having to enter a password. Protect the key files - other people can use them to access your repository as well! If you have another computer you use, you can copy the **id\_rsa.pub** file to the **.ssh** folder on that computer or (better yet) you can generate another public/private key pair specific to that computer.

Configuring other repositories (such as GitLab) is very similar.

## 5. Authenticating with private keys

### Linux, Mac users:

1. Open a terminal prompt.
2. Type the commands

```
eval `ssh-agent`  
ssh-add
```

Note that backticks ( ``` ) are used, not forward ticks ( `'` ). The second command assumes your key is in the default location, `~/.ssh/id_dsa`. If it is somewhere else, type `ssh-add path-to-private-key-file`. Note that the directory containing the ssh key cannot be readable or writeable by other users; that is, it needs mode 700. You can add these commands to your `.bashrc` file so they are executed every time you log in. Otherwise the keys only remain active until you close the shell you are using or log out.

## Windows users:

1. Install Pageant if it is not installed. It is usually installed with PuTTY and PuTTYgen.
2. Start Pageant and select **Add Key**.
3. Browse to your `.ppk` file, open it, and enter the passphrase if prompted.

If git pull or get push cannot connect, you might need to add a system variable `GIT_SSH` set to the path to the `plink.exe` executable. Go to Windows Settings, enter "system environment" in the search box, open the "Edit the system environment variables" item, click on Environment Variables..., then New... in the System Variables section (the bottom half), enter `GIT_SSH` for the name, and browse to `plink.exe` for the value. Save the setting, then reboot your computer.

You will need to re-add the key to Pageant every time you log in to Windows (say, after a reboot). The command `start/b pageant c:\path\to\file.ppk` will open the file in pageant if you have pageant in your `%PATH%` variable.

## 6. Optional: Configure Git to use a custom application (WinMerge) for comparing file differences

It is recommended that you skip this step unless you really are attached to using WinMerge for file comparison tasks.

- a. Enter the command `git config --global merge.tool winmerge`

This configures Git to use the application WinMerge to resolve merging conflicts. **You must have WinMerge installed on your computer first.** Get WinMerge at <http://winmerge.org/downloads/>.

- B. Enter the following commands to complete the WinMerge configuration:

- i. `git config --global mergetool.winmerge.name WinMerge`
- ii. `git config --global mergetool.winmerge.trustExitCode true`
- iii. If you install WinMerge to the default location (that is, `C:\Program Files (x86)\WinMerge`), enter `git config --global mergetool.winmerge.cmd "\"C:\Program Files (x86)\WinMerge\WinMergeU.exe\" -u -e -dl \"%LOCAL%\" -dr \"%REMOTE%\" \"%LOCAL% $REMOTE $MERGED\"`
- iv. If you install WinMerge to an alternate location (for example, `D:\WinMerge`), enter

```
git config --global mergetool.winmerge.cmd "/d/WinMerge/WinMergeU.exe  
-u -e -dl \"Local\" -dr \"Remote\" \"$LOCAL $REMOTE $MERGED"
```

c. Enter the command `git config --global diff.tool winmerge`

This configures Git to use the application WinMerge to differences between versions of files.

d. Enter the commands to complete the WinMerge diff configuration:

i. `git config --global difftool.winmerge.name WinMerge`

ii. `git config --global difftool.winmerge.trustExitCode  
true`

iii. If you install WinMerge to the default location (that is, C:\Program Files

```
(x86)\WinMerge), enter git config --global  
difftool.winmerge.cmd "\"C:\Program Files  
(x86)\WinMerge\WinMergeU.exe\" -u -e \"$LOCAL $REMOTE"
```

iv. If you install WinMerge to an alternate location (for example, D:\WinMerge),  
enter

```
git config --global difftool.winmerge.cmd  
"/d/WinMerge/WinMergeU.exe -u -e \"$LOCAL $REMOTE"
```

## Common problems and possible fixes

Problem	Fix
In Linux, pushes and pulls do not work with the error message " <b>permission denied</b> ".	Check the ownership and permissions on the .git folder Use <code>ls -ld .git</code> to check this. If the owner is root, use the <code>chown</code> command to fix that. If the permissions are not rwx for the owner, use the <code>chmod</code> command to set the permissions to 770.
Even though your keys are set up and available through <code>ssh-agent</code> or <code>pageant</code> , <code>git push</code> and <code>pull</code> commands prompt you for username and password.	Confirm the repository address by entering <code>git remote -v show</code> . If it shows the HTTPS protocol (starts with HTTPS), then you need to clone the repository using the SSH protocol specifier. <code>your</code>

## Task 2

This guide will walk you through setting up your GitHub account and getting started with GitHub's features for collaboration and community.

### Part 1: Configuring your GitHub account

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

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

## **1. Creating an account**

To sign up for an account on GitHub.com, navigate to <https://github.com/> and follow the prompts.

To keep your GitHub account secure you should use a strong and unique password. For more information, see "[Creating a strong password](#)."

## **2. Choosing your GitHub product**

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

For more information on all of GitHub's plans, see "[GitHub's products](#)."

## **3. Verifying your email address**

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

## **4. Configuring two-factor authentication**

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

## **5. Viewing your GitHub profile and contribution graph**

Your GitHub profile tells people the story of your work through the repositories and gists you've pinned, the organization memberships you've chosen to publicize, the contributions you've made, and the projects you've created. For more information, see "[About your profile](#)" and "[Viewing contributions on your profile](#)."

## Part 2: Using GitHub's tools and processes

To best use GitHub, you'll need to set up Git. Git is responsible for everything GitHub-related that happens locally on your computer. To effectively collaborate on GitHub, you'll write in issues and pull requests using GitHub Flavored Markdown.

### 1. Learning Git

GitHub's collaborative approach to development depends on publishing commits from your local repository to GitHub for other people to view, fetch, and update using Git. For more information about Git, see the "[Git Handbook](#)" guide. For more information about how Git is used on GitHub, see "[GitHub flow](#)."

### 2. Setting up Git

If you plan to use Git locally on your computer, whether through the command line, an IDE or text editor, you will need to install and set up Git. For more information, see "[Set up Git](#)."

If you prefer to use a visual interface, you can download and use GitHub Desktop. GitHub Desktop comes packaged with Git, so there is no need to install Git separately. For more information, see "[Getting started with GitHub Desktop](#)."

Once you install Git, you can connect to GitHub repositories from your local computer, whether your own repository or another user's fork. When you connect to a repository on GitHub.com from Git, you'll need to authenticate with GitHub using either HTTPS or SSH. For more information, see "[About remote repositories](#)."

### 3. Choosing how to interact with GitHub

Everyone has their own unique workflow for interacting with GitHub; the interfaces and methods you use depend on your preference and what works best for your needs.

For more information about how to authenticate to GitHub with each of these methods, see "[About authentication to GitHub](#)."

Method	Description	Use cases
Browse to GitHub.com	If you don't need to work with files locally, This method is useful if you GitHub lets you complete most Git-related want a visual interface and need actions directly in the browser, from to do quick, simple changes creating and forking repositories to editing that don't require working files and opening pull requests. locally.	

GitHub Desktop	GitHub Desktop extends and simplifies your GitHub.com workflow, using a visual interface instead of text commands on the locally, but prefer using a visual command line. For more information on getting started with GitHub Desktop, see <a href="#">"Getting started with GitHub Desktop."</a>	This method is best if you need to work with files and want to interact with GitHub.
IDE or text editor	You can set a default text editor, or <a href="#">Visual Studio Code</a> to open your files with Git, use extensions, project structure. For more information, see <a href="#">"Associating text editors with Git."</a>	This is convenient if you are like <a href="#">Atom</a> or <a href="#">Visual Studio Code</a> working with more complex and edit files and projects and want and view the project structure. For more information, see <a href="#">"Associating text editors with Git."</a> text editors or IDEs often allow you to directly access the command line in the editor.
Command line, or GitHub CLI	For the most granular control and customization of how you use Git and interact with GitHub, you can use the command line. For more information on using Git commands, see <a href="#">"Git cheatsheet."</a> GitHub CLI is a separate command-line tool you can install that brings pull requests, issues, GitHub Actions, and other GitHub features to your terminal, so you can do all your work in one place. For more information, see <a href="#">"GitHub CLI."</a>	This is most convenient if you are already working from the without command line, allowing you to avoid switching context, or if you are more comfortable using the command line.
<b>Method</b>	<b>Description</b>	<b>Use cases</b>
GitHub API	GitHub has a REST API and GraphQL API that you can use to interact with GitHub. For more information, see <a href="#">"Getting started with the API."</a>	The GitHub API would be most helpful if you wanted to automate common tasks, back up your data, or create integrations that extend GitHub.

## 4. Writing on GitHub

To make your communication clear and organized in issues and pull requests, you can use GitHub Flavored Markdown for formatting, which combines an easy-to-read, easy-to-write syntax with some custom functionality. For more information, see ["About writing and formatting on GitHub."](#)

You can learn GitHub Flavored Markdown with the "[Communicating using Markdown](#)" course on GitHub Learning Lab.

## 5. Searching on GitHub

Our integrated search allows you to find what you are looking for among the many repositories, users and lines of code on GitHub. You can search globally across all of GitHub or limit your search to a particular repository or organization. For more information about the types of searches you can do on GitHub, see "[About searching on GitHub](#)."

Our search syntax allows you to construct queries using qualifiers to specify what you want to search for. For more information on the search syntax to use in search, see "[Searching on GitHub](#)."

## 6. Managing files on GitHub

With GitHub, you can create, edit, move and delete files in your repository or any repository you have write access to. You can also track the history of changes in a file line by line. For more information, see "[Managing files on GitHub](#)."

# Part 3: Collaborating on GitHub

Any number of people can work together in repositories across GitHub. You can configure settings, create project boards, and manage your notifications to encourage effective collaboration.

## 1. Working with repositories

### Creating a repository

A repository is like a folder for your project. You can have any number of public and private repositories in your user account. Repositories can contain folders and files, images, videos, spreadsheets, and data sets, as well as the revision history for all files in the repository. For more information, see "[About repositories](#)."

When you create a new repository, you should initialize the repository with a README file to let people know about your project. For more information, see "[Creating a new repository](#)."

## Cloning a repository

You can clone an existing repository from GitHub to your local computer, making it easier to add or remove files, fix merge conflicts, or make complex commits. Cloning a repository pulls down a full copy of all the repository data that GitHub has at that point in time, including all versions of every file and folder for the project. For more information, see "[Cloning a repository](#)."

## Forking a repository

A fork is a copy of a repository that you manage, where any changes you make will not affect the original repository unless you submit a pull request to the project owner. Most commonly, forks are used to either propose changes to someone else's project or to use someone else's project as a starting point for your own idea. For more information, see "[Working with forks](#)."

## 2. Importing your projects

If you have existing projects you'd like to move over to GitHub you can import projects using the GitHub Importer, the command line, or external migration tools. For more information, see "[Importing source code to GitHub](#)."

## 3. Managing collaborators and permissions

You can collaborate on your project with others using your repository's issues, pull requests, and project boards. You can invite other people to your repository as collaborators from the **Collaborators** tab in the repository settings. For more information, see "[Inviting collaborators to a personal repository](#)."

You are the owner of any repository you create in your user account and have full control of the repository. Collaborators have write access to your repository, limiting what they have permission to do. For more information, see "[Permission levels for a user account repository](#)."

## 4. Managing repository settings

As the owner of a repository you can configure several settings, including the repository's visibility, topics, and social media preview. For more information, see "[Managing repository settings](#)."



## 5. Setting up your project for healthy contributions

To encourage collaborators in your repository, you need a community that encourages people to use, contribute to, and evangelize your project. For more information, see "[Building Welcoming Communities](#)" in the Open Source Guides.

By adding files like contributing guidelines, a code of conduct, and a license to your repository you can create an environment where it's easier for collaborators to make meaningful, useful contributions. For more information, see "[Setting up your project for healthy contributions](#)."

## 6. Using GitHub Issues and project boards

You can use GitHub Issues to organize your work with issues and pull requests and manage your workflow with project boards. For more information, see "[About issues](#)" and "[About project boards](#)."

## 7. Managing notifications

Notifications provide updates about the activity on GitHub you've subscribed to or participated in. If you're no longer interested in a conversation, you can unsubscribe, unwatch, or customize the types of notifications you'll receive in the future. For more information, see "[About notifications](#)."

## 8. Working with GitHub Pages

You can use GitHub Pages to create and host a website directly from a repository on GitHub.com. For more information, see "[About GitHub Pages](#)."

## 9. Using GitHub Discussions

You can enable GitHub Discussions for your repository to help build a community around your project. Maintainers, contributors and visitors can use discussions to share announcements, ask and answer questions, and participate in conversations around goals. For more information, see "[About discussions](#)."

# Part 4: Customizing and automating your work on GitHub

You can use tools from the GitHub Marketplace, the GitHub API, and existing GitHub features to customize and automate your work.

## 1. Using GitHub Marketplace

GitHub Marketplace contains integrations that add functionality and improve your workflow. You can discover, browse, and install free and paid tools, including GitHub Apps, OAuth Apps, and GitHub Actions, in [GitHub Marketplace](#). For more information, see "[About GitHub Marketplace](#)."

## 2. Using the GitHub API

There are two versions of the GitHub API: the REST API and the GraphQL API. You can use the GitHub APIs to automate common tasks, [back up your data](#), or [create integrations](#) that extend GitHub. For more information, see "[About GitHub's APIs](#)."

## 3. Building GitHub Actions

With GitHub Actions, you can automate and customize GitHub.com's development workflow on GitHub. You can create your own actions, and use and customize actions shared by the GitHub community. For more information, see "[Learn GitHub Actions](#)."

## 4. Publishing and managing GitHub Packages

GitHub Packages is a software package hosting service that allows you to host your software packages privately or publicly and use packages as dependencies in your projects. For more information, see "[Introduction to GitHub Packages](#)."

# Part 5: Building securely on GitHub

GitHub has a variety of security features that help keep code and secrets secure in repositories. Some features are available for all repositories, while others are only available for public repositories and repositories with a GitHub Advanced Security license. For an overview of GitHub security features, see "[GitHub security features](#)."

## 1. Securing your repository

As a repository administrator, you can secure your repositories by configuring repository security settings. These include managing access to your repository, setting a security policy, and managing dependencies. For public repositories, and for private repositories owned by organizations where GitHub Advanced Security is enabled, you can also configure code and secret scanning to automatically identify vulnerabilities and ensure tokens and keys are not exposed.

For more information on steps you can take to secure your repositories, see "[Securing your repository](#)."

## 2. Managing your dependencies

A large part of building securely is maintaining your project's dependencies to ensure that all packages and applications you depend on are updated and secure. You can manage your repository's dependencies on GitHub by exploring the dependency graph for your repository, using Dependabot to automatically raise pull requests to keep your dependencies up-to-date, and receiving Dependabot alerts and security updates for vulnerable dependencies.

For more information, see "[Securing your software supply chain](#)."

## Part 6: Participating in GitHub's community

There are many ways to participate in the GitHub community. You can contribute to open source projects, interact with people in the GitHub Community Support, or learn with GitHub Learning Lab.

### 1. Contributing to open source projects

Contributing to open source projects on GitHub can be a rewarding way to learn, teach, and build experience in just about any skill you can imagine. For more information, see "[How to Contribute to Open Source](#)" in the Open Source Guides.

You can find personalized recommendations for projects and good first issues based on your past contributions, stars, and other activities in [Explore](#). For more information, see "[Finding ways to contribute to open source on GitHub](#)."

### 2. Interacting with GitHub Community Support

You can connect with developers around the world in [GitHub Community Support](#) to ask and answer questions, learn, and interact directly with GitHub staff.

### 3. Reading about GitHub on GitHub Docs

You can read documentation that reflects the features available to you on GitHub. For more information, see "[About versions of GitHub Docs](#)."

### 4. Learning with GitHub Learning Lab

You can learn new skills by completing fun, realistic projects in your very own GitHub repository with [GitHub Learning Lab](#). Each course is a hands-on lesson created by the GitHub community and taught by the friendly Learning Lab bot.

For more information, see "[Git and GitHub learning resources](#)."

## 5. Supporting the open source community

GitHub Sponsors allows you to make a monthly recurring payment to a developer or organization who designs, creates, or maintains open source projects you depend on. For more information, see "[About GitHub Sponsors](#)."

## 6. Contacting GitHub Support

GitHub Support can help you troubleshoot issues you run into while using GitHub. For more information, see "[About GitHub Support](#)."

# Task 3

## Package 'logr'

Title Creates Log Files  
Version 1.2.9  
Description Contains functions to help create log files. The package aims to overcome the difficulty of the base R sink() command. The log\_print() function will print to both the console and the file log, without interfering in other write operations.  
License CC0  
Encoding UTF-8  
  
URL <https://logr.r-sassy.org>  
  
BugReports <https://github.com/dbosak01/logr/issues>  
Depends R (>= 3.4.0)  
Suggests knitr, rmarkdown, testthat, tidylog, dplyr, covr  
Imports withr, utils, this.path  
VignetteBuilder knitr  
RoxygenNote 7.1.2  
NeedsCompilation no  
Author David Bosak [aut, cre]  
Maintainer David Bosak <[dbosak01@gmail.com](mailto:dbosak01@gmail.com)>  
Repository CRAN  
Date/Publication 2022-03-08 16:10:02 UTC

# R topics documented:

logr .....	log_close .....
.....	log_code .....
.....	log_path .....
.....	log_status ..
.....	

## Index

logr	Creates log files
------	-------------------

## Description

The logr package contains functions to easily create log files.

## Details

The logr package helps create log files for R scripts. The package provides easy logging, without the complexity of other logging systems. It is designed for analysts who simply want a written log of the their program execution. The package is designed as a wrapper to the base R sink() function.

## How to use

There are only three logr functions:

- 8. [log\\_open](#)
- 9. [log\\_print](#)
- 10. [log\\_close](#)

The log\_open() function initiates the log. The log\_print() function prints an object to the log. The log\_close() function closes the log. In normal situations, a user would place the call to log\_open at the top of the program, call log\_print() as needed in the program body, and call log\_close() once at the end of the program.

Logging may be controlled globally using the options "logr.on" and "logr.notes". Both options accept TRUE or FALSE values, and control log printing or log notes, respectively.

See function documentation for additional details.

---

log_close	Close the log
-----------	---------------

---

## Description

The log\_close function closes the log file.

## Usage

```
log_close()
```

## Details

The log\_close function terminates logging. As part of the termination process, the function prints any outstanding warnings to the log. Errors are printed at the point at which they occur. But warnings can be captured only at the end of the logging session. Therefore, any warning messages will only be printed at the bottom of the log.

The function also prints the log footer. The log footer contains a date-time stamp of when the log was closed.

## Value

None

## See Also

[log\\_open](#) to open the log, and [log\\_print](#) for printing to the log.

## Examples

```
# Create temp file location
tmp <- file.path(tempdir(), "test.log")

# Open log
lf <- log_open(tmp)

# Send message to log
log_print("High Mileage Cars Subset")

# Perform operations
hmc <- subset(mtcars, mtcars$mpg > 20)
```

3. Print data to log log\_print(hmc)

4. Close log  
log\_close()

4. View results

```
writeLines(readLines(lf))
```

---

log_code	Log the current program code
----------	------------------------------

---

## Description

A function to send the program/script code to the currently opened log. The log must be opened first with [log\\_open](#). Code will be prefixed with a right arrow (">") to differentiate it from standard logging lines. The log\_code function may be called from anywhere within the program. Code will be inserted into the log at the point where it is called. The log\_code function will log the code as it is saved on disk. It will not capture any unsaved changes in the editor. If the current program file cannot be found, the function will return FALSE and no code will be written

## Usage

```
log_code()
```

## Value

A TRUE or FALSE value to indicate success or failure of the function.

## See Also

[log\\_open](#) to open the log, and [log\\_close](#) to close the log.

## Examples

```
# Create temp file location
tmp <- file.path(tempdir(), "test.log")
```

```
# Open log
lf <- log_open(tmp)
```

7. Write code to the log log\_code()

8. Send message to log  
log\_print("High Mileage Cars Subset")

```
# Perform operations
hmc <- subset(mtcars, mtcars$mpg > 20)
```

c. Print data to log  
log\_print(hmc)

d. Close log  
log\_close()

```
v. View results
writeLines(readLines(lf))
```

log_open	Open a log
----------	------------

## Description

A function to initialize the log file.

## Usage

```
log_open(file_name = "", logdir = TRUE, show_notes = TRUE, autolog = NULL)
```

## Arguments

file_name	The name of the log file. If no path is specified, the working directory will be used. As of v1.2.7, the name and path of the program or script will be used as a default if the file_name parameter is not supplied.
logdir	Send the log to a log directory named "log". If the log directory does not exist, the function will create it. Valid values are TRUE and FALSE. The default is TRUE.
show_notes	If true, will write notes to the log. Valid values are TRUE and FALSE. Default is TRUE.
autolog	Whether to turn on autolog functionality. Autolog automatically logs functions from the dplyr, tidyr, and sassy family of packages. To enable autolog, either set this parameter to TRUE or set the "logr.autolog" option to TRUE. A FALSE value on this parameter will override the global option. The global option will

## Details

The log\_open function initializes and opens the log file. This function must be called first, before any logging can occur. The function determines the log path, attaches event handlers, clears existing log files, and initiates a new log.

The file\_name parameter may be a full path, a relative path, or a file name. An relative path or file name will be assumed to be relative to the current working directory. If the file\_name does not have a '.log' extension, the log\_open function will add it.

As of v1.2.7, if the file\_name parameter is not supplied, the function will use the program/script name as the default log file name, and the program/script path as the default path.

If requested in the logdir parameter, the log\_open function will write to a 'log' subdirectory of the path specified in the file\_name. If the 'log' subdirectory does not exist, the function will create it.

The log file will be initialized with a header that shows the log file name, the current working directory, the current user, and a timestamp of when the log\_open function was called.



All errors, the last warning, and any `log_print` output will be written to the log. The log file will exist in the location specified in the `file_name` parameter, and will normally have a `'log'` extension.

If errors or warnings are generated, a second file will be written that contains only error and warning messages. This second file will have a `'msg'` extension and will exist in the specified log directory. If the log is clean, the msg file will not be created. The purpose of the msg file is to give the user a visual indicator from the file system that an error or warning occurred. This indicator msg file is useful when running programs in batch.

To use `logr`, call `log_open`, and then make calls to `log_print` as needed to print variables or data frames to the log. The `log_print` function can be used in place of a standard print function. Anything printed with `log_print` will be printed to the log, and to the console if working interactively.

This package provides the functionality of `sink`, but in much more user-friendly way. Recommended usage is to call `log_open` at the top of the script, call `log_print` as needed to log interim state, and call `log_close` at the bottom of the script.

Logging may be controlled globally using the `"logr.on"` option. This option accepts a `TRUE` or `FALSE` value. If the option is set to `FALSE`, `logr` will print to the console, but not to the log. Example: `options("logr.on" = TRUE)`

Notes may be controlled globally using the `"logr.notes"` option. This option also accepts a `TRUE` or `FALSE` value, and determines whether or not to print notes in the log. The global option will override the `show_notes` parameter on the `log_open` function. Example: `options("logr.notes" = FALSE)`

Version v1.2.0 of the `logr` package introduced `autolog`. The `autolog` feature provides automatic log-ging for `dplyr`, `tidyr`, and the `sassy` family of packages. To use `autolog`, set the `autolog` parameter to `TRUE`, or set the global option `logr.autolog` to `TRUE`. To maintain backward compatibility with prior versions, `autolog` is disabled by default.

## Value

The path of the log.

## See Also

[log\\_print](#) for printing to the log (and console), and [log\\_close](#) to close the log.

## Examples

```
# Create temp file location
tmp <- file.path(tempdir(), "test.log")

# Open log
lf <- log_open(tmp)

# Send message to log
log_print("High Mileage Cars Subset")

# Perform operations
hmc <- subset(mtcars, mtcars$mpg > 20)

# Print data to log log_print(hmc)

# Close log
log_close()
```

```
# View results
writeLines(readLines(lf))
```

---

log_path	Get the path of the current log
----------	---------------------------------

---

#### Description

The log\_path function gets the path to the currently opened log. This function may be useful when you want to manipulate the log in some way, and need the path. The function takes no parameters.

#### Usage

log\_path()

#### Value

The full path to the currently opened log, or NULL if no log is open.

#### Examples

```
# Create temp file location
tmp <- file.path(tempdir(), "test.log")

# Open log log_open(tmp)

# Get path
lf <- log_path()

# Close log log_close()

lf
```

---

log_print	Print an object to the log
-----------	----------------------------

---

#### Description

The log\_print function prints an object to the currently opened log.

#### Usage

```
log_print(x, ..., console = TRUE, blank_after = TRUE, msg = FALSE, hide_notes = FALSE)
put(x, ..., console = TRUE, blank_after = TRUE, msg = FALSE, hide_notes = FALSE)
sep(x, console = TRUE)
```

log\_hook(x)

## Arguments

x	The object to print.
...	Any parameters to pass to the print function.
console	Whether or not to print to the console. Valid values are TRUE and FALSE. Default is TRUE.
blank_after	Whether or not to print a blank line following the printed object. The blank line helps readability of the log. Valid values are TRUE and FALSE. Default is TRUE.
msg	Whether to print the object to the msg log. This parameter is intended to be used internally. Value values are TRUE and FALSE. The default value is FALSE.
hide_notes	If notes are on, this parameter gives you the option of not printing notes for a particular log entry. Default is FALSE, meaning notes will be displayed. Used internally.

## Details

The log is initialized with log\_open. Once the log is open, objects like variables and data frames can be printed to the log to monitor execution of your script. If working interactively, the function will print both to the log and to the console. The log\_print function is useful when writing and debugging batch scripts, and in situations where some record of a scripts' execution is required.

If requested in the log\_open function, log\_print will print a note after each call. The note will contain a date-time stamp and elapsed time since the last call to log\_print. When printing a data frame, the log\_print function will also print the number and rows and column in the data frame. These counts may also be useful in debugging.

Notes may be turned off either by setting the show\_notes parameter on log\_open to FALSE, or by setting the global option "logr.notes" to FALSE.

The put function is a shorthand alias for log\_print. You can use put anywhere you would use log\_print. The functionality is identical.

The sep function is also a shorthand alias for log\_print, except it will print a separator before and after the printed text. This function is intended for documentation purposes, and you can use it to help organize your log into sections.

The log\_hook function is for other packages that wish to integrate with logr. The function prints to the log only if autolog is enabled. It will not print to the console.

## Value

The object, invisibly

## See Also

[log\\_open](#) to open the log, and [log\\_close](#) to close the log.

### Examples

```
# Create temp file location
tmp <- file.path(tempdir(), "test.log")

# Open log
lf <- log_open(tmp)

# Send message to log
log_print("High Mileage Cars Subset")

# Perform operations
hmc <- subset(mtcars, mtcars$mpg > 20)

# Print data to log log_print(hmc)

# Close log
log_close()

# View results
writeLines(readLines(lf))
```

---

log_status	Get the status of the log
------------	---------------------------

---

## Description

The `log_status` function gets the status of the log. Possible status values are 'on', 'off', 'open', or 'closed'. The function takes no parameters.

## Usage

```
log_status()
```

## Value

The status of the log as a character string.

## Examples

```
# Check status before the log is opened
log_status()
# [1] "closed"

# Create temp file location
tmp <- file.path(tempdir(), "test.log")

# Open log
```

```
lf <- log_open(t

# Check status after log is opened
log_status()
# [1] "open"

# Close log log_close()
```

## Task 4

This document is an in-depth review of the `git branch` command and a discussion of the overall Git branching model. Branching is a feature available in most modern version control systems. Branching in other VCS's can be an expensive operation in both time and disk space. In Git, branches are a part of your everyday development process. Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes. This makes it harder for unstable code to get merged into the main code base, and it gives you the chance to clean up your future's history before merging it into the main branch.

The diagram above visualizes a repository with two isolated lines of development, one for a little feature, and one for a longer-running feature. By developing them in branches, it's not only possible to work on both of them in parallel, but it also keeps the `main` branch free from questionable code.

The implementation behind Git branches is much more lightweight than other version control system models. Instead of copying files from directory to directory, Git stores a branch as a reference to a commit. In this sense, a branch represents the tip of a series of commits—it's not a container for commits. The history for a branch is extrapolated through the commit relationships.

As you read, remember that Git branches aren't like SVN branches. Whereas SVN branches are only used to capture the occasional largescale development effort, Git branches are an integral part of your everyday workflow. The following content will expand on the internal Git branching architecture.

# 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>`. This does *not* check out the new branch.

```
git branch -d <branch>
```

Delete the specified branch. This is a "safe" operation in that Git prevents you from deleting the branch if it has unmerged changes.

```
git branch -D <branch>
```

Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

```
git branch -m <branch>
```

Rename the current branch to `<branch>`.

```
git branch -a
```

List all remote branches.

## 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:

```
git branch 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.

## Creating remote branches

So far these examples have all demonstrated local branch operations. The `git branch` command also works on remote branches. In order to operate on remote branches, a remote repo must first be configured and added to the local repo config.

```
$ git remote add new-remote-  
repo https://bitbucket.com/user/repo.git  
# Add remote repo to local repo config  
$ git push <new-remote-repo> crazy-experiment~  
# pushes the crazy-experiment branch to new-remote-repo
```

This command will push a copy of the local branch `crazyexperiment` to the remote repo `<remote>`.



# Deleting Branches

Once you've finished working on a branch and have merged it into the main code base, you're free to delete the branch without losing any history:

```
git branch -d crazy-experiment
```

However, if the branch hasn't been merged, the above command will output an error message:

```
error: The branch 'crazy-experiment' is not fully merged. If
you are sure you want to delete it, run 'git branch -D
crazy-experiment'.
```

This protects you from losing access to that entire line of development. If you really want to delete the branch (e.g., it's a failed experiment), you can use the capital `-D` flag:

```
git branch -D crazy-experiment
```

This deletes the branch regardless of its status and without warnings, so use it judiciously.

The previous commands will delete a local copy of a branch. The branch may still exist in remote repos. To delete a remote branch execute the following.

```
git push origin --delete crazy-experiment
```

Or

```
git push origin :crazy-experiment
```

This will push a delete signal to the remote origin repository that triggers a delete of the remote `crazy-experiment` branch.

## Summary

In this document we discussed Git's branching behavior and the `git branch` command. The `git branch` commands primary functions are to create, list, rename and delete branches. To operate further on the resulting branches the command is commonly used with other commands like `git checkout`. Learn more about `git checkout`

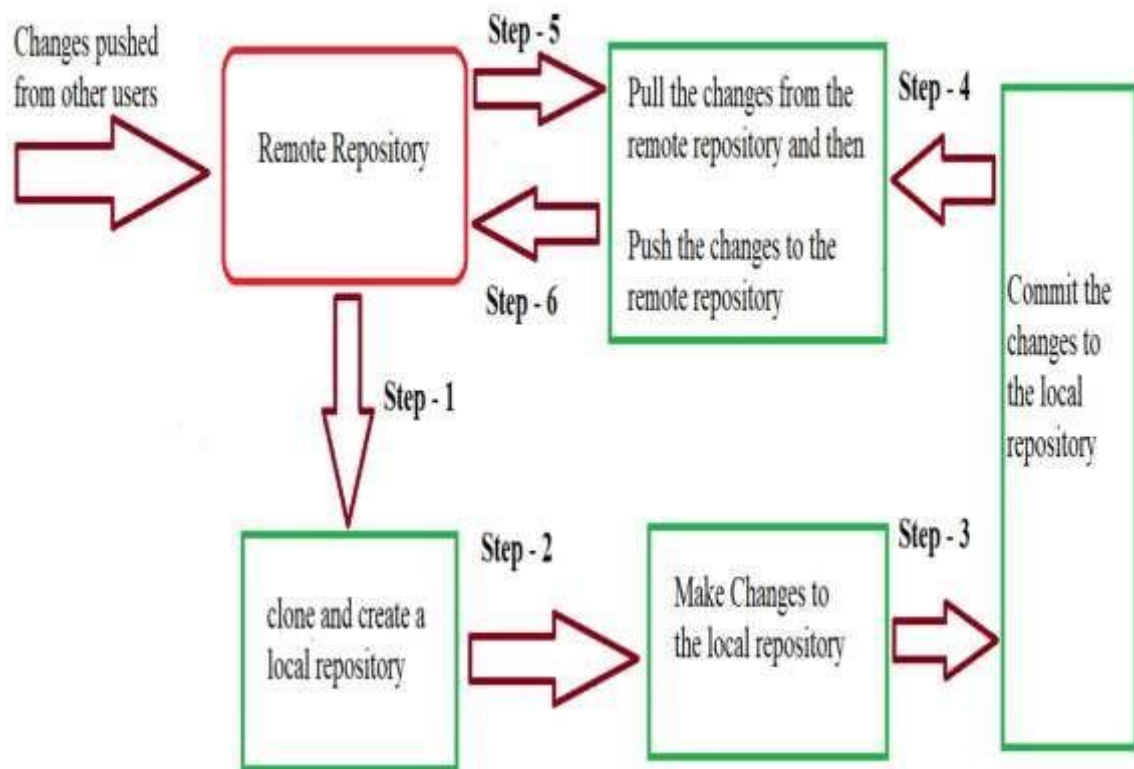
branch operations; such as switching branches and merging branches, on the [git checkout](#) page.

Compared to other VCSs, Git's branch operations are inexpensive and frequently used. This flexibility enables powerful [Git workflow](#) customization. For more info on Git workflows visit our extended workflow discussion pages: [The Feature Branch Workflow](#), [GitFlow Workflow](#), and [Forking Workflow](#).

## Task 5

### Git – Life Cycle

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



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

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

- Working Directory
- Staging Area
- Git Directory

Let us understand in detail about each state.

## 1. Working Directory

Whenever we want to initialize our local project directory to make it a git repository, we use the **git init** command. After this command, git becomes aware of the files in the project although it doesn't track the files yet. The files are further tracked in the staging area.

*git init*

## 2. Staging Area

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

*// to specify which file to add to the staging area git*

*add <filename>*

*// to add all files of the working directory to the staging area git*

*add .*

## 3. Git Directory

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



Subject Name: **Source Code Management**

Subject Code: **CS181**

Cluster: **Zeta**

Department: **DCSE**

**CHITKARA**  
UNIVERSITY



**Submitted By:**

Vasu Khattar

2110992054

G27

**Submitted To:**

Dr. Anuj Jain

## **List of Programs**

<b>S. No</b>	<b>Program Title</b>	<b>Page No.</b>
1	Add collaborators on GitHub Repository	3-4
2	Fork and Commit	5-8
3	Merge and Resolve conflicts created due to own activity and collaborators activity.	9-12
4	Reset and Revert	13-17

# Task 1 - Add collaborators on GitHub Repository

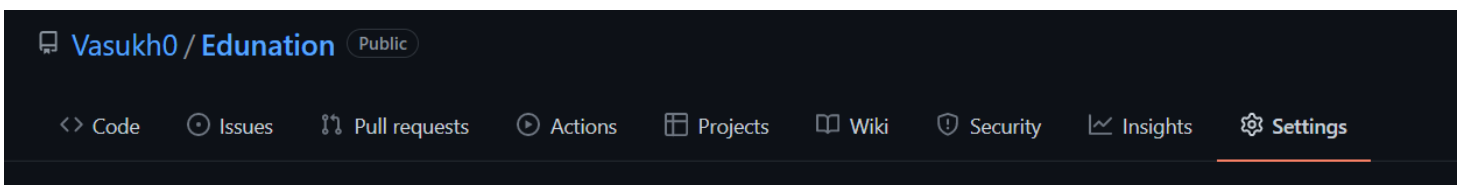
You can invite users to become collaborators to your personal repository. If you're using GitHub Free, you can add unlimited collaborators on public and private repositories. Repositories owned by an organization can grant more granular access.

For more information, see "Access permissions on GitHub."

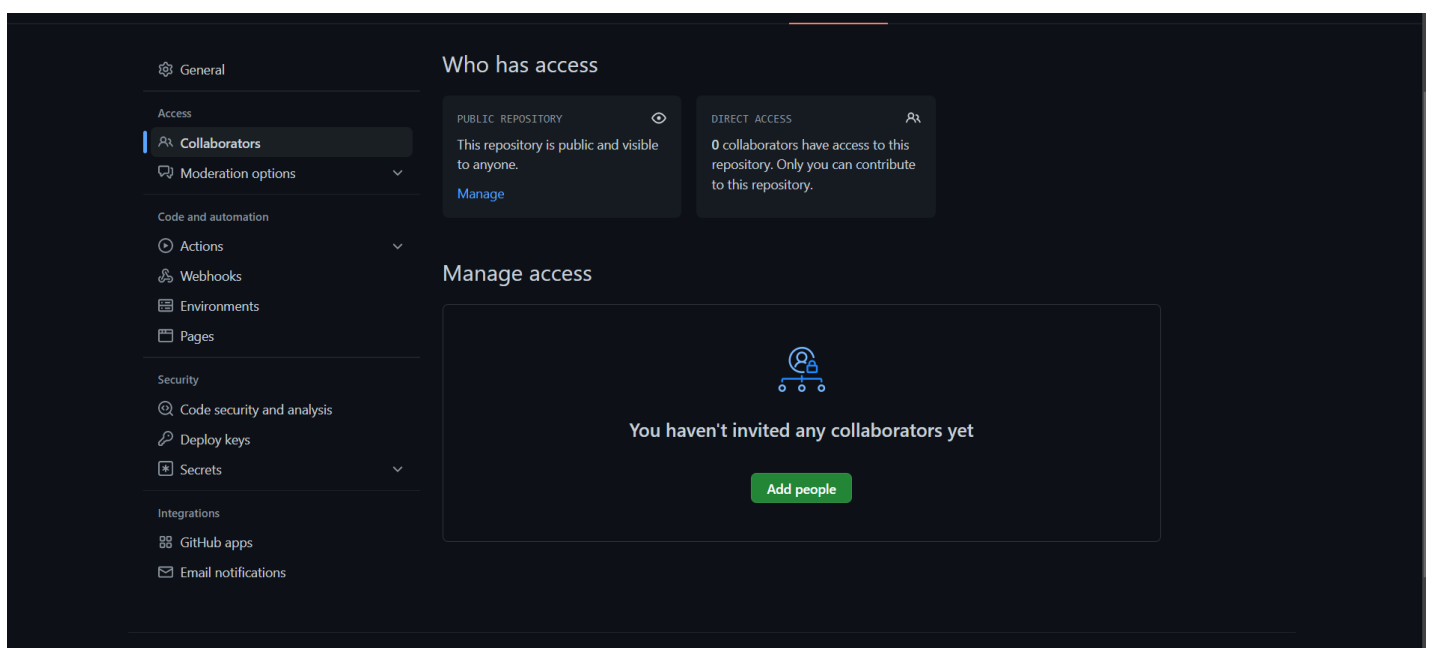
Pending invitations will expire after 7 days, restoring any unclaimed licenses. If you're a member of an enterprise with managed users, you can only invite other members of your enterprise to collaborate with you.

For more information, see "Types of GitHub accounts." Note: GitHub limits the number of people who can be invited to a repository within a 24-hour period. If you exceed this limit, either wait 24 hours or create an organization to collaborate with more people.

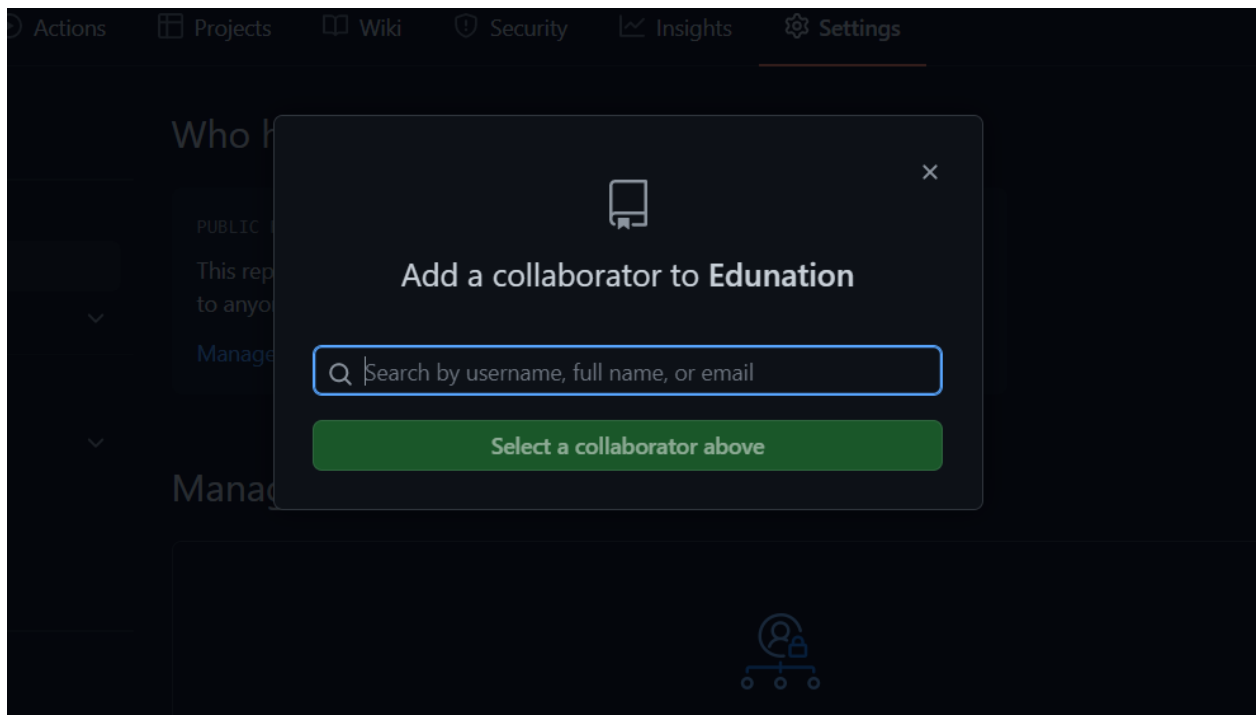
1. Ask for the username of the person you're inviting as a collaborator. If they don't have a username yet, they can sign up for GitHub For more information, see "[Signing up for a new GitHub account](#)".
2. On GitHub.com, navigate to the main page of the repository.
3. Under your repository name, click Settings.



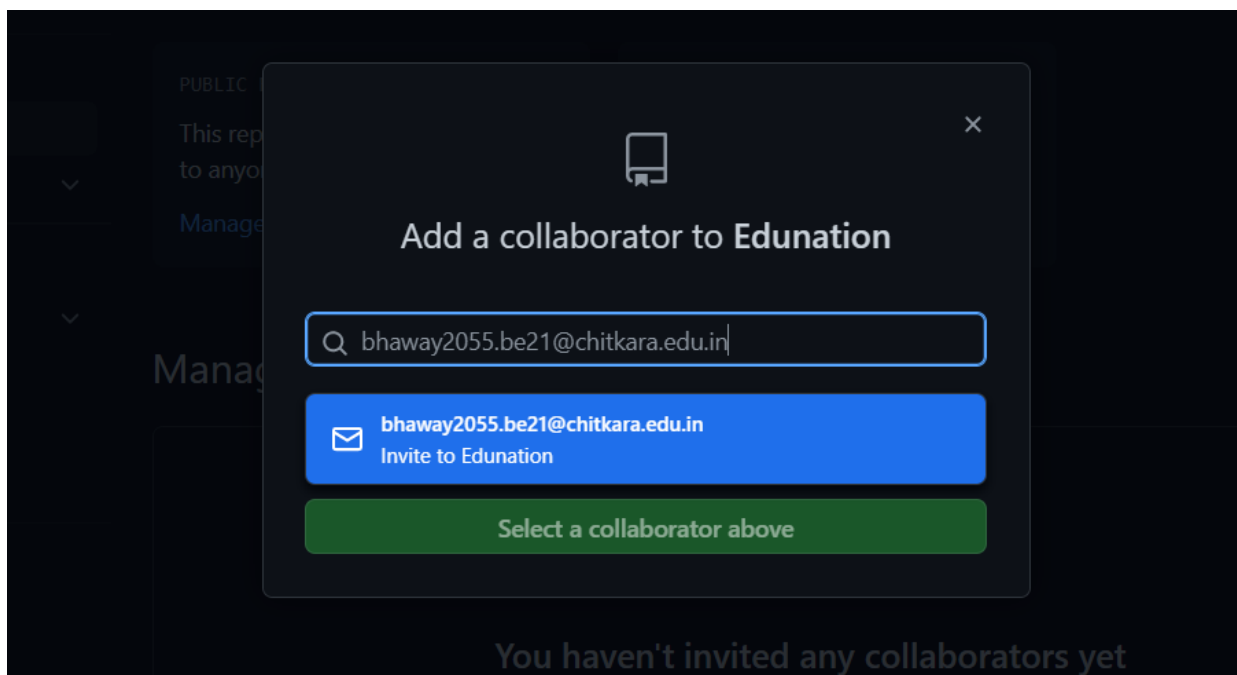
4. In the "Access" section of the sidebar, click Collaborators & teams.
5. Click Invite a collaborator.



6. In the search field, start typing the name of person you want to invite, then click a name in the list of matches.



7. Click Add NAME to REPOSITORY.



8. The user will receive an email inviting them to the repository. Once they accept your invitation, they will have collaborator access to your repository.



# Task 2 - Fork and Commit

## Git forks

Most commonly, forks are used to either propose changes to someone else's project to which you don't have write access, or to use someone else's project as a starting point for your own idea. You can fork a repository to create a copy of the repository and make changes without affecting the upstream repository.

### Propose changes to someone else's project

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

- Fork the repository.
- Make the fix.
- Submit a pull request to the project owner.
- Use someone else's project as a starting point for your own idea.

Open source software is based on the idea that by sharing code, we can make better, more reliable software. For more information, see the "About the Open Source Initiative" on the Open Source Initiative.

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](https://choosealicense.com).

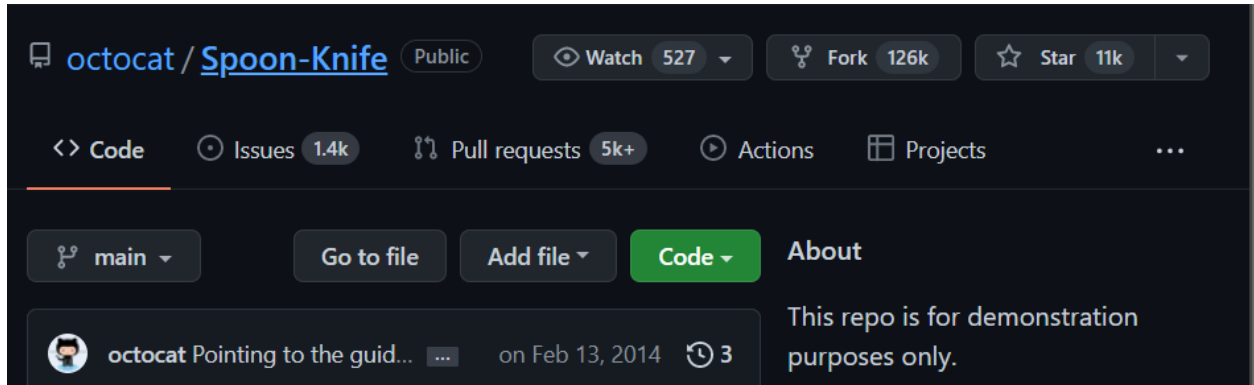
For more information on open source, specifically how to create and grow an open source project, we've created Open Source Guides that will help you foster a healthy open source community by recommending best practices for creating and maintaining repositories for your open source project. You can also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

## Forking a repository

You might fork a project to propose changes to the upstream, or original, repository. In this case, it's good practice to regularly sync your fork with the upstream repository. To do this,

you'll need to use Git on the command line. You can practice setting the upstream repository using the same octocat/Spoon-Knife repository you just forked.

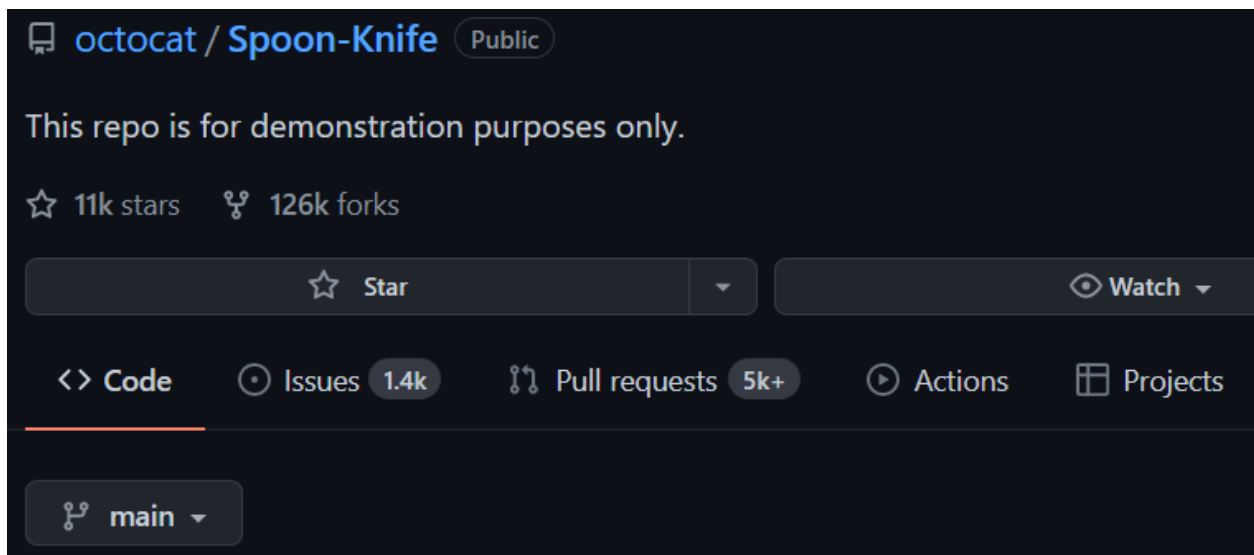
1. On GitHub.com, navigate to the octocat/Spoon-Knife repository.
2. In the top-right corner of the page, click Fork.



## Cloning your forked repository

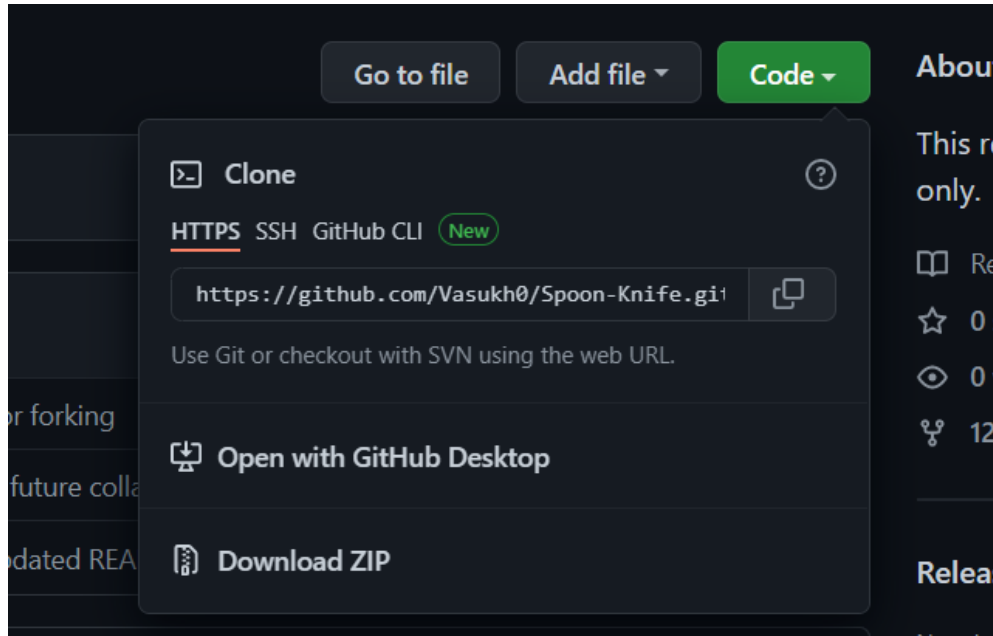
Right now, you have a fork of the Spoon-Knife repository, but you don't have the files in that repository locally on your computer.

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





- 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.
- \$ git clone https://github.com/Vasukh0/Spoon-Knife**
- Press Enter. Your local clone will be created.

```
view Insert Format Tools Add-ons Help Accessibility Last edit w
MINGW64:/c/Users/vkhat

vkhat@MSI MINGW64 ~
$ git clone https://github.com/Vasukh0/Spoon-Knife
Cloning into 'Spoon-Knife'...
remote: Enumerating objects: 16, done.
remote: Total 16 (delta 0), reused 0 (delta 0), pack-reused 16
Receiving objects: 100% (16/16), done.
Resolving deltas: 100% (3/3), done.
```



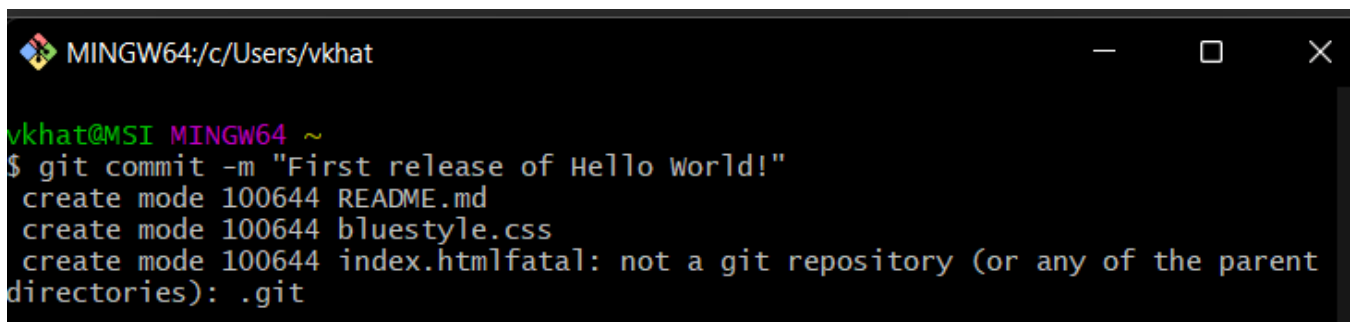
## Git Commit

Since we have finished our work, we are ready to 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.



```
MINGW64:/c/Users/vkhat
vkhat@MSI MINGW64 ~
$ git commit -m "First release of Hello World!"
create mode 100644 README.md
create mode 100644 bluestyle.css
create mode 100644 index.htmlfatal: not a git repository (or any of the parent
directories): .git
```

The **commit** command performs a commit, and the **-m "message"** adds a message.

The Staging Environment has been committed to our repo, with the message:

"First release of Hello World!"

## Git Commit without Stage

Sometimes, when you make small changes, using the staging environment seems like a waste of time. It is possible to commit changes directly, skipping the staging environment. The **-a** option will automatically stage every changed, already tracked file.

And check the status of our repository. But this time, we will use the **--short** option to see the changes in a more compact way:

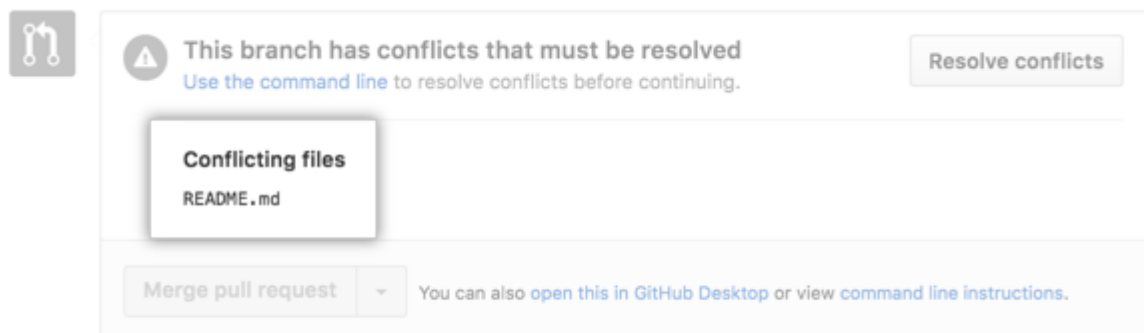
## Task 3 - Merge and Resolve conflicts created due to own activity and collaborators activity.

### About merge conflicts

Merge conflicts happen when you merge branches that have competing commits, and Git needs your help to decide which changes to incorporate in the final merge.

Git can often resolve differences between branches and merge them automatically. Usually, the changes are on different lines, or even in different files, which makes the merge simple for computers to understand. However, sometimes there are competing changes that Git can't resolve without your help. Often, merge conflicts happen when people make different changes to the same line of the same file, or when one person edits a file and another person deletes the same file.

You must resolve all merge conflicts before you can merge a pull request on GitHub. If you have a merge conflict between the compare branch and base branch in your pull request, you can view a list of the files with conflicting changes above the Merge pull request button. The Merge pull request button is deactivated until you've resolved all conflicts between the compare branch and base branch.



### Resolving merge conflicts

To resolve a merge conflict, you must manually edit the conflicted file to select the changes that you want to keep in the final merge. There are a couple of different ways to resolve a merge conflict:

If your merge conflict is caused by competing line changes, such as when people make different changes to the same line of the same file on different branches in your Git repository, you can resolve it on GitHub using the conflict editor. For more information, see "Resolving a merge conflict on GitHub."

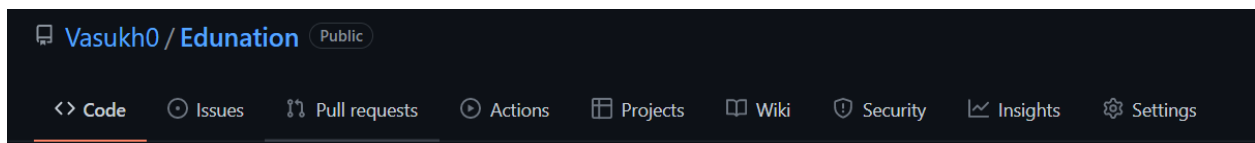
For all other types of merge conflicts, you must resolve the merge conflict in a local clone of the repository and push the change to your branch on GitHub. You can use the command line or a tool like GitHub Desktop to push the change. For more information, see "Resolving a merge conflict on the command line."

## Resolving a merge conflict on GitHub

You can resolve simple merge conflicts that involve competing line changes on GitHub, using the conflict editor.

You can only resolve merge conflicts on GitHub that are caused by competing line changes, such as when people make different changes to the same line of the same file on different branches in your Git repository. For all other types of merge conflicts, you must resolve the conflict locally on the command line.

1. Under your repository name, click **Pull requests**.



2. In the "Pull Requests" list, click the pull request with a merge conflict that you'd like to resolve.
3. Near the bottom of your pull request, click **Resolve conflicts**.

**Tip:** If the conflict is complex, use a graphical client, or resolve the conflict using the command line.

**This branch has conflicts that must be resolved**

Use the [command line](#) to resolve conflicts before continuing.

Resolve conflicts

**Conflicting files**

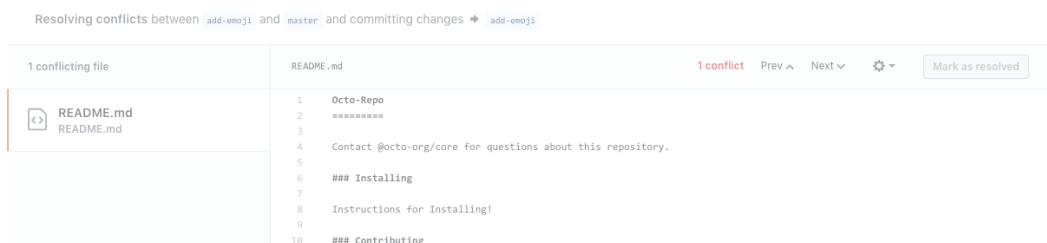
README.md

Merge pull request

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Go to  
GitHub  
to  
resolve

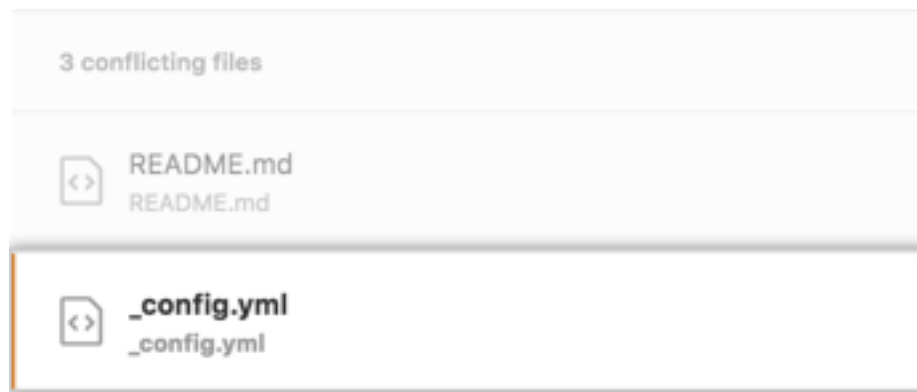
4. Decide if you want to keep only your branch's changes, keep only the other branch's changes, or make a brand new change, which may incorporate changes from both branches. Delete the conflict markers <<<<<<<, =====, >>>>>>> and make the changes you want in the final merge.



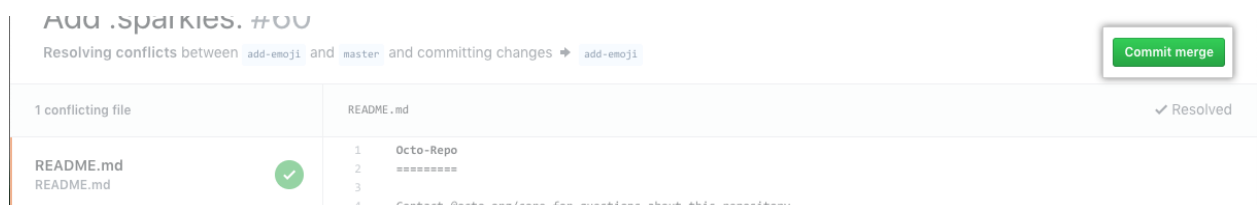
5. If you have more than one merge conflict in your file, scroll down to the next set of conflict markers and repeat steps four and five to resolve your merge conflict.
6. Once you've resolved all the conflicts in the file, click Mark as resolved.



7. If you have more than one file with a conflict, select the next file you want to edit on the left side of the page under "conflicting files" and repeat steps four through seven until you've resolved all of your pull request's merge conflicts.

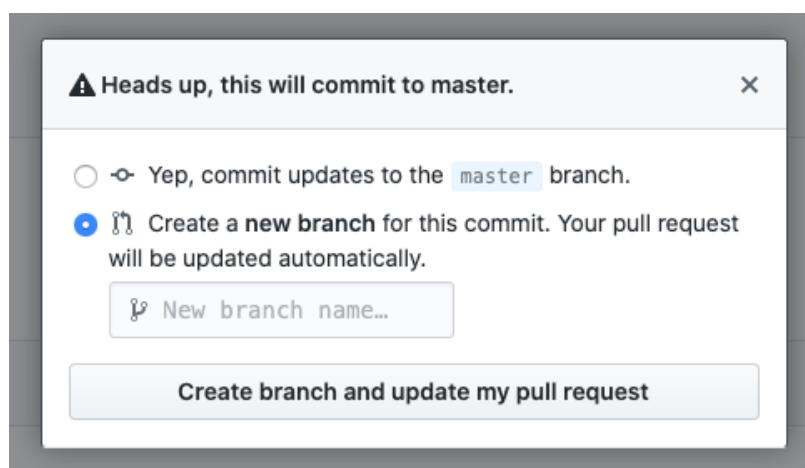


8. Once you've resolved all your merge conflicts, click Commit merge. This merges the entire base branch into your head branch.



9. If prompted, review the branch that you are committing to.

If the head branch is the default branch of the repository, you can choose either to update this branch with the changes you made to resolve the conflict, or to create a new branch and use this as the head branch of the pull request.



If you choose to create a new branch, enter a name for the branch.

If the head branch of your pull request is protected you must create a new branch. You won't get the option to update the protected branch.

Click Create branch and update my pull request or I understand, continue updating BRANCH. The button text corresponds to the action you are performing.

10. To merge your pull request, click Merge pull request. For more information about other pull request merge options, see "Merging a pull request."

## Task 4 - Reset and Revert

### Git Reset

The git reset command allows you to RESET your current head to a specified state. You can reset the state of specific files as well as an entire branch. This is useful if you haven't pushed your commit up to GitHub or another remote repository yet.

#### Reset a file or set of files

The following command lets you selectively choose chunks of content and revert or unstage it.

```
git reset (--patch | -p) [tree-ish] [--] [paths]
```

#### Unstage a file

If you moved a file into the staging area with git add, but no longer want it to be part of a commit, you can use git reset to unstage that file:

```
git reset HEAD FILE-TO-UNSTAGE
```

The changes you made will still be in the file, this command just removes that file from your staging area.

#### Reset a branch to a prior commit



The following command resets your current branch's HEAD to the given COMMIT and updates the index. It basically rewinds the state of your branch, then all commits you make going forward write over anything that came after the reset point. If you omit the MODE, it defaults to --mixed:

`git reset MODE COMMIT`

The options for MODE are:

- --soft: does not reset the index file or working tree, but resets HEAD to commit. Changes all files to "Changes to be committed"
- --mixed: resets the index but not the working tree and reports what has not been updated
- --hard: resets the index and working tree. Any changes to tracked files in the working tree since commit are discarded
- --merge: resets the index and updates the files in the working tree that are different between commit and HEAD, but keeps those which are different between the index and working tree
- --keep: resets index entries and updates files in the working tree that are different between commit and HEAD. If a file that is different between commit and HEAD has local changes, the reset is aborted

Be very careful when using the --hard option with git reset since it resets your commit, staging area and your working directory. If this option is not used properly then one can end up losing the code that is written.

## Git Revert

Both the git revert and git reset commands undo previous commits. But if you've already pushed your commit to a remote repository, it is recommended that you do not use git reset since it rewrites the history of commits. This can make working on a repository with other developers and maintaining a consistent history of commits very difficult.

Instead, it is better to use git revert, which undoes the changes made by a previous commit by creating an entirely new commit, all without altering the history of commits.

## Revert a commit or set of commits

The following command lets you revert changes from a previous commit or commits and create a new commit.

```
git revert [--no-edit] [-n] [-m parent-number] [-s] [-S[<keyid>]] <commit>...
```

```
git revert --continue
```

```
git revert --quit
```

```
git revert --abort
```

## Common options:

-e

--edit

- This is the default option and doesn't need to be explicitly set. It opens your system's default text editor and lets you edit the new commit message before commit the revert.
- This option does the opposite of -e, and git revert will not open the text editor.
- This option prevents git revert from undoing a previous commit and creating a new one. Rather than creating a new commit, -n will undo the changes from the previous commit and add them to the Staging Index and Working Directory.

--no-edit

-n

-no-commit

## Example.

Let's imagine the following situation: 1.) You are working on a file and you add and commit your changes. 2.) You then work on a few other things, and make some more commits. 3.) Now you realize, three or four commits ago, you did something that you would like to undo - how can you do this?

You might be thinking, just use git reset, but this will remove all of the commits after the one you would like to change - git revert to the rescue! Let's walk through this example:

```
mkdir learn_revert # Create a folder called `learn_revert`  
cd learn_revert # `cd` into the folder `learn_revert`  
git init # Initialize a git repository
```

```
touch first.txt # Create a file called `first.txt`  
echo Start >> first.txt # Add the text "Start" to `first.txt`
```

```
git add . # Add the `first.txt` file  
git commit -m "adding first" # Commit with the message "Adding first.txt"
```

```
echo WRONG > wrong.txt # Add the text "WRONG" to `wrong.txt`  
git add . # Add the `wrong.txt` file  
git commit -m "adding WRONG to wrong.txt" # Commit with the message "Adding  
WRONG to wrong.txt"
```

```
echo More >> first.txt # Add the text "More" to `first.txt`  
git add . # Add the `first.txt` file  
git commit -m "adding More to first.txt" # Commit with the message "Adding More to  
first.txt"
```

```
echo Even More >> first.txt # Add the text "Even More" to `first.txt`  
git add . # Add the `first.txt` file  
git commit -m "adding Even More to First.txt" # Commit with the message "Adding More to  
first.txt"
```

# OH NO! We want to undo the commit with the text "WRONG" - let's revert! Since this commit was 2 from where we are not we can use git revert HEAD~2 (or we can use git log and find the SHA of that commit)

```
git revert HEAD~2 # this will put us in a text editor where we can modify the commit  
message.
```

```
ls # wrong.txt is not there any more!
```

```
git log --oneline # note that the commit history hasn't been altered, we've just added a new  
commit reflecting the removal of the `wrong.txt`
```

And with that you're one step closer to getting your black belt in Git.

A Project report  
on  
**“Edunation”**  
with  
**Source Code Management**  
(CS181)

Submitted by

Bhaway Khattar      2110992055

Himanshu Garg      2110992044

Udit      2110992026

Vasu Khattar      2110992054

**CHITKARA**  
UNIVERSITY





## Department of Computer Science & Engineering

Chitkara University Institute of Engineering and Technology, Punjab

Jan- June  
(2021-22)

Institute/School Name	<b>Chitkara University Institute of Engineering and Technology</b>		
Department Name	<b>Department of Computer Science &amp; Engineering</b>		
Programme Name	<b>Bachelor of Engineering (B.E.), Computer Science &amp; Engineering</b>		
Course Name	<b>Source Code Management</b>	Session	<b>2021-22</b>
Course Code	<b>CS181</b>	Semester/Batch	<b>2<sup>nd</sup>/2021</b>
Vertical Name	<b>Zeta</b>	Group No	<b>G27</b>
Course Coordinator	<b>Dr. Neeraj Singla</b>		
Faculty Name	<b>Dr. Sachendra Singh Chauhan</b>		

Submission

Name:

Signature:

Date:

## Table of Content

<b>S. No.</b>	<b>Title</b>	<b>Page No.</b>
1	Version control with Git	4-9
2	Problem Statement	10
3	Objective	11
4	Resources Requirements – Frontend	12-13
5	Concepts and commands	14-26
6	Workflow and Discussion	27-30
7	Reference	31

# Version Control With Git

## ❖ What is the Version Control System?

A **Version Control System (VCS)** is a tool that helps software developers keep track of how their software development projects desktop applications, websites, mobile apps, etc - change over time.

Each snapshot or state of the files and folders in a codebase at a given time can be called a "version." Version control systems were created to allow developers a convenient way to create, manage, and share those versions. It allows them to have *control* over managing the versions of their code as it evolves over time.

Version control systems also enable collaboration within a team of software developers, without losing or overwriting anyone's work. After a developer makes a set of code changes to one or more files, they tell the version control system to save a representation of those changes.

A version control system can also be referred to as a source control system, source code management, version control software, version control tools, or other combinations of these terms.

## ❖ Benefits of the Version Control System

The Version Control System is very helpful and beneficial in software development; developing software without using version control is unsafe. It provides backups for uncertainty. Version control systems offer a speedy interface to developers. It also allows software teams to preserve efficiency and agility according to the team scales to include more developers.

Some key benefits of having a version control system are as follows.

- Complete change history of the file
- Simultaneously working
- Branching and merging

## ❖ History of VCS

There are 3 types of VCS:

### 1. Local Version Control System:

In a local version control system, files are simply copied into a separate directory locally. Versions of the same file are stored so as to allow the easy retrieval of any particular version at any point in time. This system is commonly used for small personal projects or files as it provides the facility of versioning your project in an easy manner locally.

#### **Advantages:**

- Easy to set up
- No internet needed
- Cheap to run

#### **Disadvantages:**

- Error prone
- Unsafe (stored locally)
- Not suitable for team projects
- As data is stored in local machine. If the local machine crashes, it would not be possible to retrieve the files, and all the information will be lost.

### 2. Centralized Version Control System:

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

#### **Advantages:**

- Reasonably easy to set up
- Various options (proprietary and open source)
- Allows for file sharing amongst team members
- Project is stored on a more reliable server (possibly cloud)
- Admin can control the use and structure of the repository



### **Disadvantages:**

- Single point of failure (if server fails then changes will not be available)
- File conflicts due to updates from different people

### **3. Distributed Version Control system:**

In a distributed version control system, there will be one or more servers and many collaborators similar to the centralized system. But the difference is, not only do they check out the latest version, but each collaborator will have an exact copy of the main repository on their local machines.

Each user has their own repository and a working copy. This is very useful because even if the server crashes we would not lose everything as several copies are residing in several other computers.

### **Advantages:**

- Reliable (everyone has a copy of all versions)
  - Allows for file share amongst team members
  - Various Options available
- ### **Disadvantages:**
- More complex to use/set up
  - Heavy on Local Storage

## **❖ What is Git?**

**Git** is an **open-source distributed version control system**. It is designed to handle minor to major projects with high speed and efficiency. It is developed to coordinate the work among the developers. The version control allows us to track and work together with our team members at the same workspace.

Git was created by **Linus Torvalds** in **2005** to develop Linux Kernel. It is also used as an important distributed version-control tool for **the DevOps**.

## ❖ Features of Git

Some remarkable features of Git are as follows:

- **Open Source**

Git is an **open-source tool**. It is released under the **GPL** (General Public License) license.

- **Scalable**

Git is **scalable**, which means when the number of users increases, the Git can easily handle such situations.

- **Distributed**

One of Git's great features is that it is **distributed**. Distributed means that instead of switching the project to another machine, we can create a "clone" of the entire repository. Also, instead of just having one central repository that you send changes to, every user has their own repository that contains the entire commit history of the project. We do not need to connect to the remote repository; the change is just stored on our local repository. If necessary, we can push these changes to a remote repository.

- **Security**

Git is secure. It uses the **SHA1 (Secure Hash Function)** to name and identify objects within its repository. Files and commits are checked and retrieved by its checksum at the time of checkout. It stores its history in such a way that the ID of particular commits depends upon the complete development history leading up to that commit. Once it is published, one cannot make changes to its old version.

- **Speed**

Git is very **fast**, so it can complete all the tasks in a while. Most of the git operations are done on the local repository, so it provides a **huge speed**. Also, a centralized version control system continually communicates with a server somewhere.

Performance tests conducted by Mozilla showed that it was **extremely fast compared to other VCSs**. Fetching version history from a locally

stored repository is much faster than fetching it from the remote server.

The **core part of Git** is **written in C**, which **ignores** runtime overheads associated with other high-level languages.

Git was developed to work on the Linux kernel; therefore, it is

**capable** enough to **handle large repositories** effectively. From the beginning, **speed** and **performance** have been Git's primary goals.

- **Supports non-linear development**

Git supports **seamless branching and merging**, which helps in visualizing and navigating a non-linear development. A branch in Git represents a single commit. We can construct the full branch structure with the help of its parental commit.

- **Branching and Merging**

**Branching and merging** are the **great features** of Git, which makes it different from the other SCM tools. Git allows the **creation of multiple branches** without affecting each other. We can perform tasks like **creation, deletion, and merging** on branches, and these tasks take a few seconds only. Below are some features that can be achieved by branching:

- We can **create a separate branch** for a new module of the project, commit and delete it whenever we want.
- We can have a **production branch**, which always has what goes into production and can be merged for testing in the test branch.
- We can create a **demo branch** for the experiment and check if it is working. We can also remove it if needed.
- The core benefit of branching is if we want to push something to a remote repository, we do not have to push all of our branches. We can select a few of our branches, or all of them together.

- **Data Assurance**

The Git data model ensures the **cryptographic integrity** of every unit of our project. It provides a **unique commit ID** to every commit through a **SHA algorithm**. We can **retrieve and update** the commit by commit ID. Most of the centralized version control systems do not provide such integrity by default.

- **Staging Area**

The **Staging area** is also a **unique functionality** of Git. It can be

considered as a **preview of our next commit**, moreover, an **intermediate area** where commits can be formatted and reviewed before completion. When you make a commit, Git takes changes that are in the staging area and make them as a new commit. We are allowed to add and remove changes from the staging area. The staging area can be considered as a place where Git stores the changes.

However, Git doesn't have a dedicated staging directory where it can store some objects representing file changes (blobs). Instead of this, it uses a file called index.

Another feature of Git that makes it apart from other SCM tools is that **it is possible to quickly stage some of our files and commit them without committing other modified files in our working directory.**

- **Maintain the clean history**

Git facilitates with Git Rebase; It is one of the most helpful features of Git. It fetches the latest commits from the master branch and puts our code on top of that. Thus, it maintains a clean history of the project.

# Problem Statement

Let's take a scenario to know the importance of git and to know the real power and the importance of git in everyday life of a coder...

Just Imagine you are working on a website. You completed the first version of the project. You decided to modify it. You made a copy of your project folder and started making some changes without risking the integrity of your first version of the project. You made your second version of the project. Similarly, you made many versions of your project just to make it better and better.

Now the problem with this is that copying or making duplicate folders of your project is not an optimized approach. We have to keep in mind the space and time complexities. But with git it becomes simpler to manage space and time as github is like a cloud where you will upload your data

If we are going to host our Education website Website over. So, we don't need to make copies of the project and imagine we made a mistake and we want to go to the previous versions of the project. How will we roll back to a previous version?

Then, we will realize the power and importance of Git and github.

# Objective

Following points are the Objectives of Git:

1. **To perform collaborations:** Git keeps track of changes to files and allows multiple users to coordinate updates to those files.
2. **To Track Histories:** Git is used to track changes in the source code.
3. **To enact distributed development:** Git enables the developers to manage the changes offline and allows you to branch and merge whenever required, giving them full control over the local code base.
4. **To perform branching:** Git allows you to develop features, fix bugs, or safely experiment with new ideas in a contained area of your repository.
5. **To Speed up Tasks:** Git tries to minimize latency, which means good perceived performance. Which means it will be easier and faster to enact changes.

# Resources Requirements – Frontend

## HTML: The Building Blocks of the Internet

HTML stands for HyperText Markup Language. It is a relatively simple language that allows developers to create the basic structure of a website. Even the most complex websites have HTML at their core. It's also the second-most-used programming language by developers, according to a recent Stack Overflow survey.

You may be asking yourself why HTML is called a “markup language.” The reason is that instead of using a programming language to perform the desired functions, HTML (like other markup languages) uses tags to annotate, or “mark up,” different types of content on a web page and identify the purposes they each serve to the page's overall design. You likely see snippets of HTML more than you even realize. Have you ever noticed text at the bottom of a printed-out email that reads something like “”? That's HTML. A markup language also helps web developers avoid formatting every instance of an item category separately (e.g., bolding the headlines on a website), which saves time and avoids errors.

HTML uses “elements,” or tags, to denote things like the beginning of a paragraph, the bolding of a font, or the addition of a photo caption. In this way, it controls how a webpage looks, how the text is separated and formatted, and what the user sees. For people who have never used programming languages before, HTML is an excellent place to start.

## CSS

If HTML represents the building blocks of a website, CSS is a way to shape and enhance those blocks. CSS is a style sheet language used to specify the way different parts of a webpage appear to users. In other words, it's a way to add some style and additional formatting to what you've already built with HTML.

For example, perhaps you've used HTML to add header text, and now you want that header to have a more pleasant font, a background color, or other formatting elements that make it more sleek, professional, and stylish. That's where CSS comes in. CSS also helps websites adapt to different device types and screen sizes so that your pages render equally well on smartphones, tablets, or desktop computers.

To understand the difference between HTML and CSS, it's important to understand their histories. When HTML was invented in 1990, it was only designed to inform a document's structural content (e.g., separating headlines from body text). However, when stylistic elements like fonts and colors were developed, HTML wasn't able to adapt. To solve this issue, CSS was invented as a set of rules that can assign properties to HTML elements, building off of the existing markup language to create a more complex webpage.

## JavaScript

JavaScript is the most complex of the three front end languages discussed in this article, building on top of both HTML and CSS. If you're trying to compare the languages, think of it like this: While HTML creates the basic structure for a website, CSS adds style to that structure, and JavaScript takes all of that work and makes it interactive and more functionally complex.

A classic example of how JavaScript works is the menu button that you're used to seeing on the top corner of most websites. You know the one — the three stacked lines that show a list of website sections you can visit when clicked. These buttons and their functionality are all present thanks to JavaScript. It can also help you develop keyboard shortcuts or change the color of a button when a cursor hovers over it.

JavaScript is crucial to all web development. It's supported by all of the modern web browsers, and it is used on almost every site on the web. According to a recent Stack Overflow survey, JavaScript is the most commonly used programming language by developers around the world, with 67.7 percent of developers putting it to use in their work. So, if you're interested in learning web development — whether professionally or even just as a hobby — you'd be smart to learn JavaScript.





# Concepts and commands

## Task 1 - Add collaborators on GitHub Repository

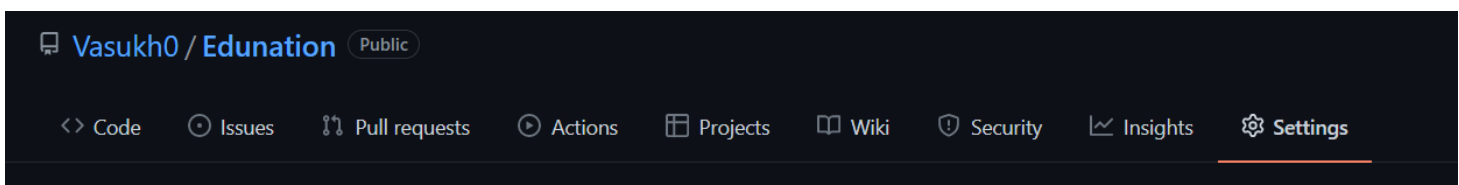
You can invite users to become collaborators to your personal repository. If you're using GitHub Free, you can add unlimited collaborators on public and private repositories. Repositories owned by an organization can grant more granular access.

For more information, see "Access permissions on GitHub."

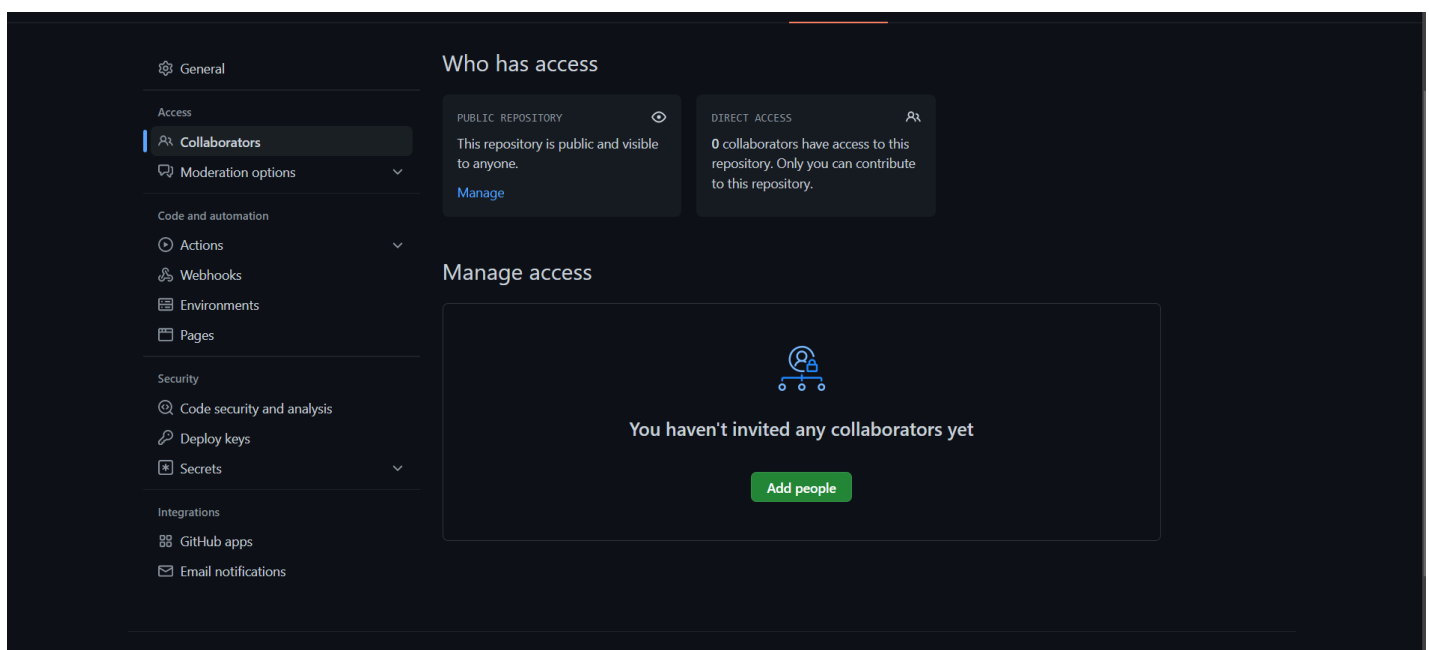
Pending invitations will expire after 7 days, restoring any unclaimed licenses. If you're a member of an enterprise with managed users, you can only invite other members of your enterprise to collaborate with you.

For more information, see "Types of GitHub accounts." Note: GitHub limits the number of people who can be invited to a repository within a 24-hour period. If you exceed this limit, either wait 24 hours or create an organization to collaborate with more people.

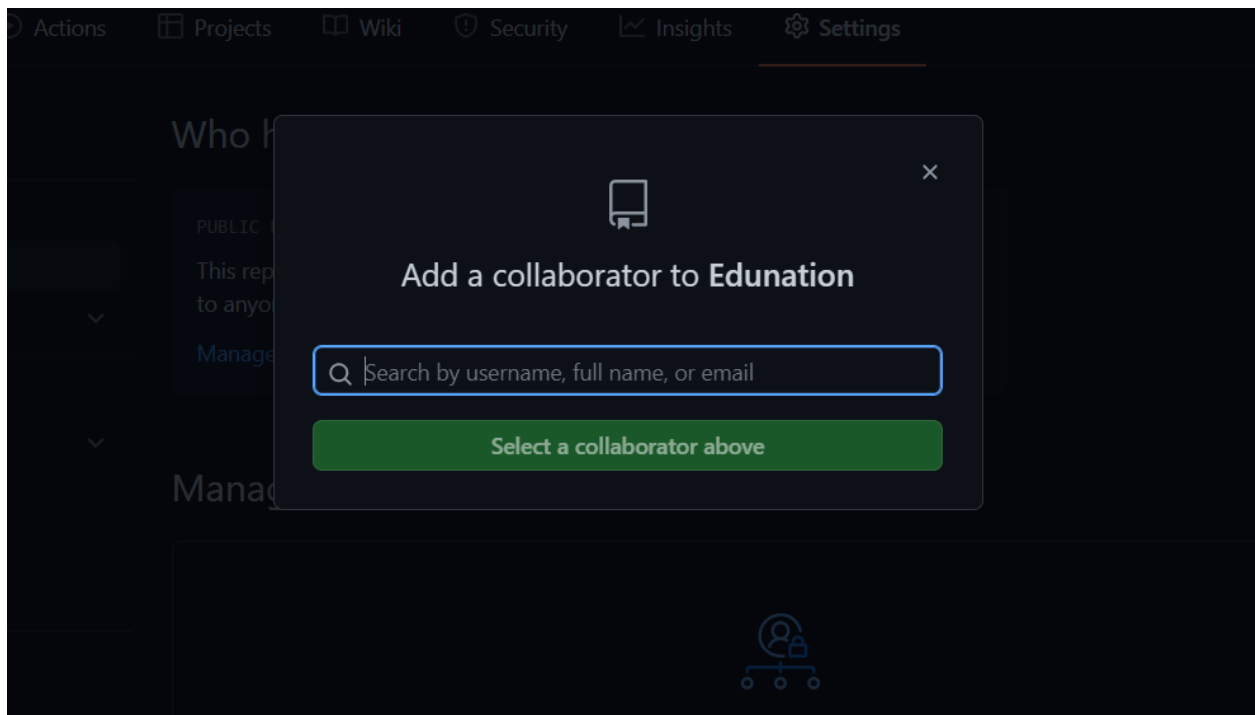
1. Ask for the username of the person you're inviting as a collaborator. If they don't have a username yet, they can sign up for GitHub For more information, see "[Signing up for a new GitHub account](#)".
2. On GitHub.com, navigate to the main page of the repository.
3. Under your repository name, click Settings.



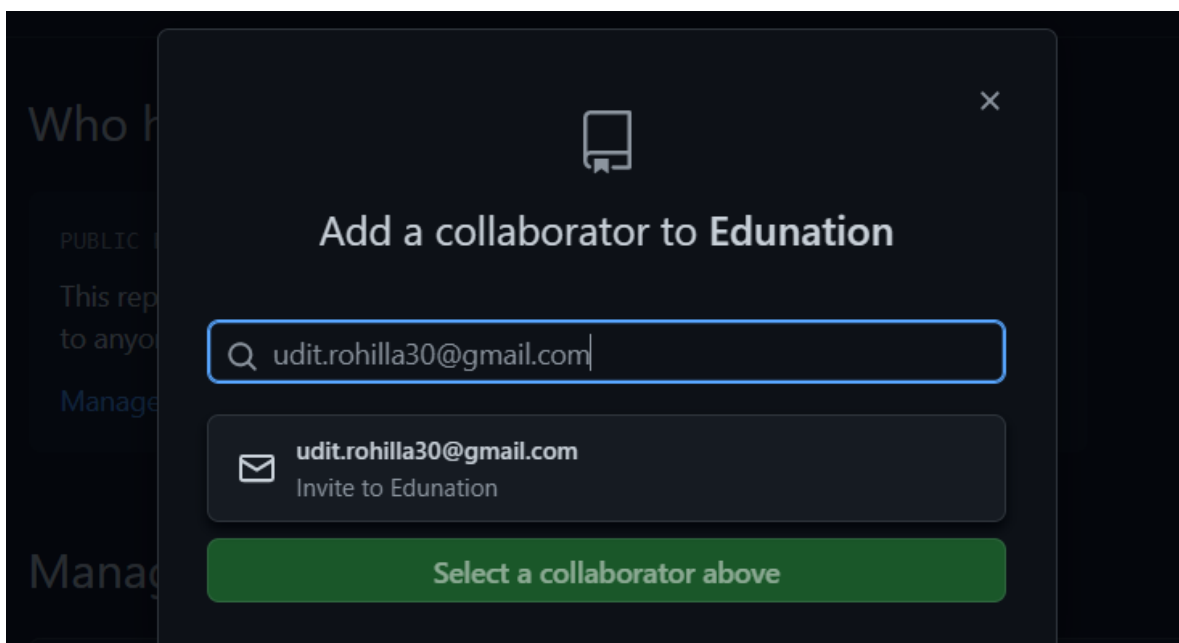
4. In the "Access" section of the sidebar, click Collaborators & teams.
5. Click Invite a collaborator.

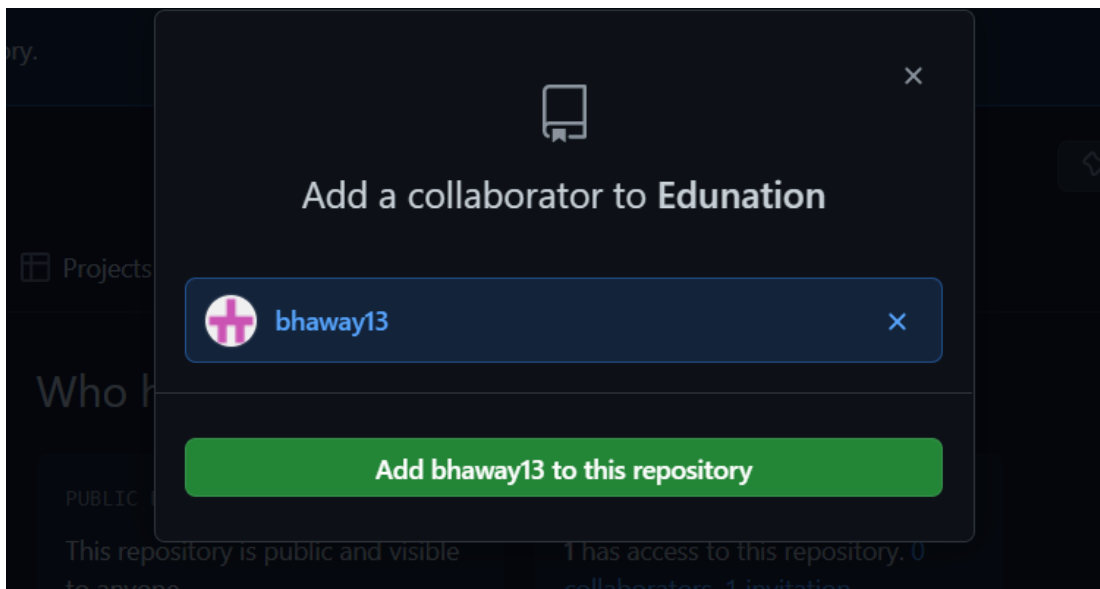
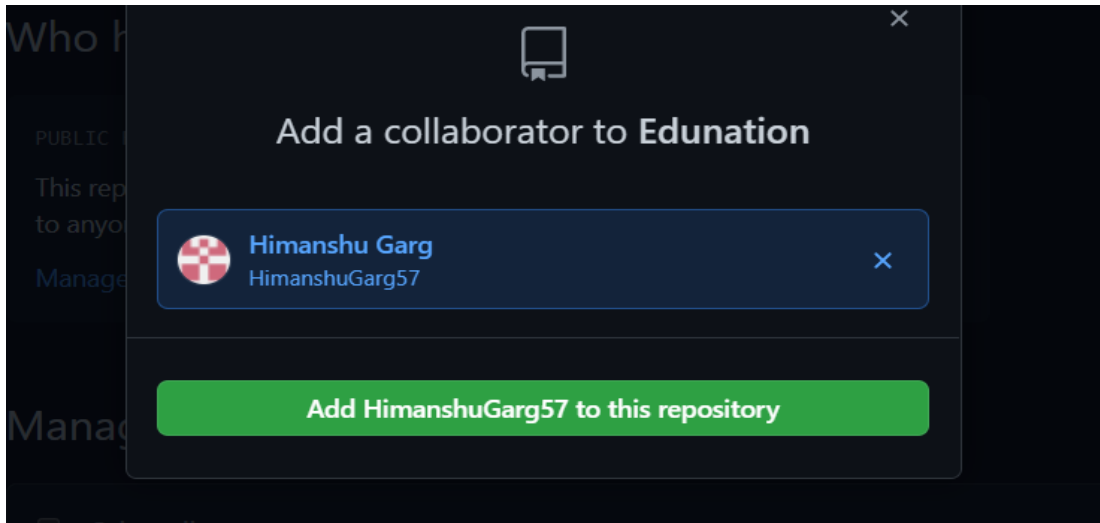


6. In the search field, start typing the name of the person you want to invite, then click a name in the list of matches.

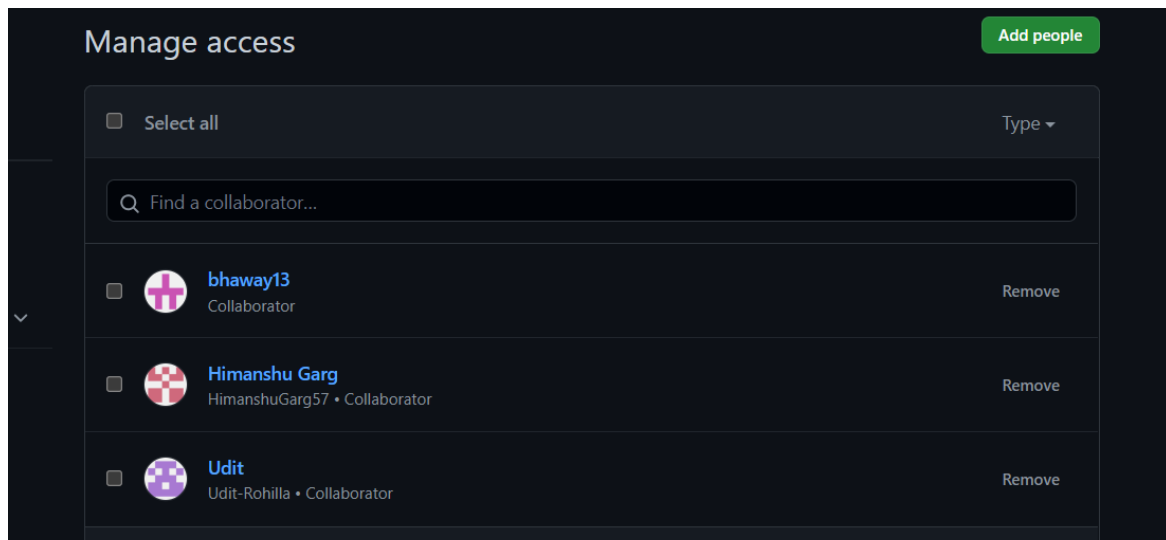


7. Click Add NAME to REPOSITORY.





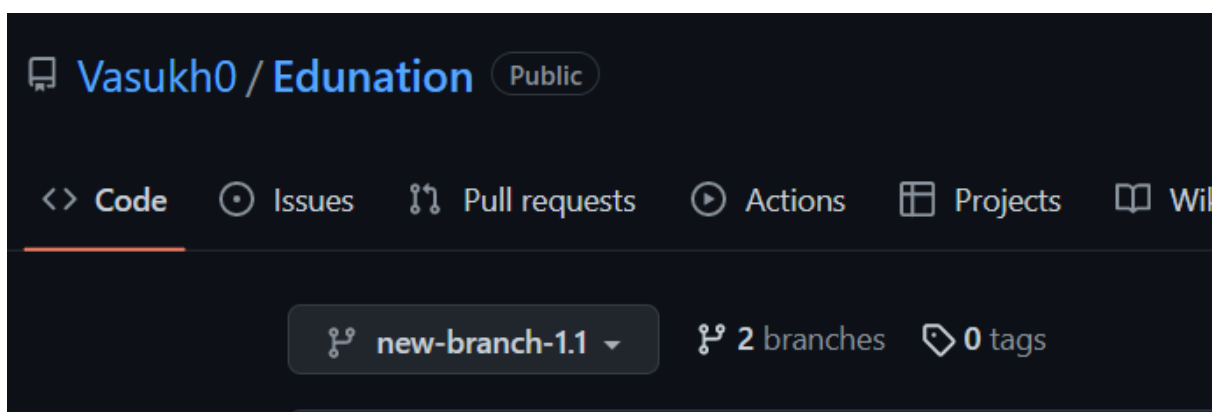
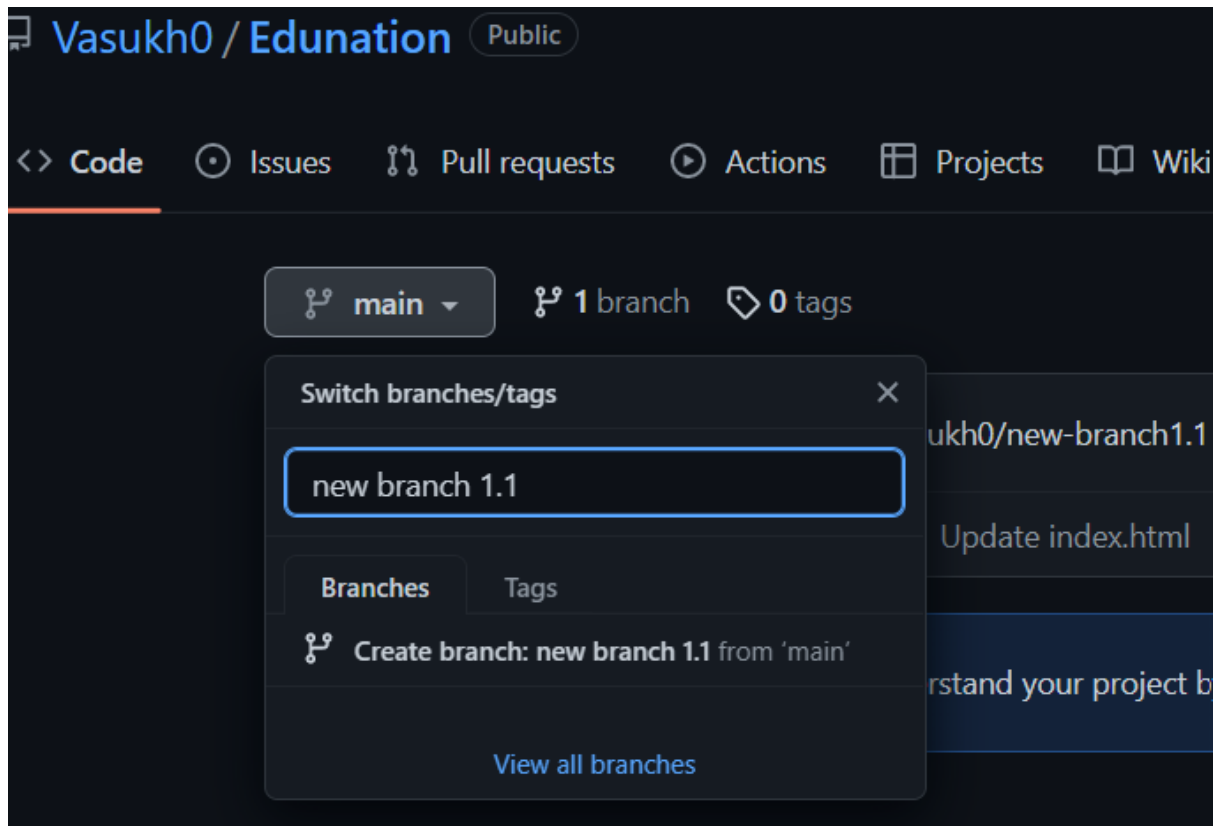
- The user will receive an email inviting them to the repository. Once they accept your invitation, they will have collaborator access to your repository and after accepting you will see the collaborators name as seen in the image below.



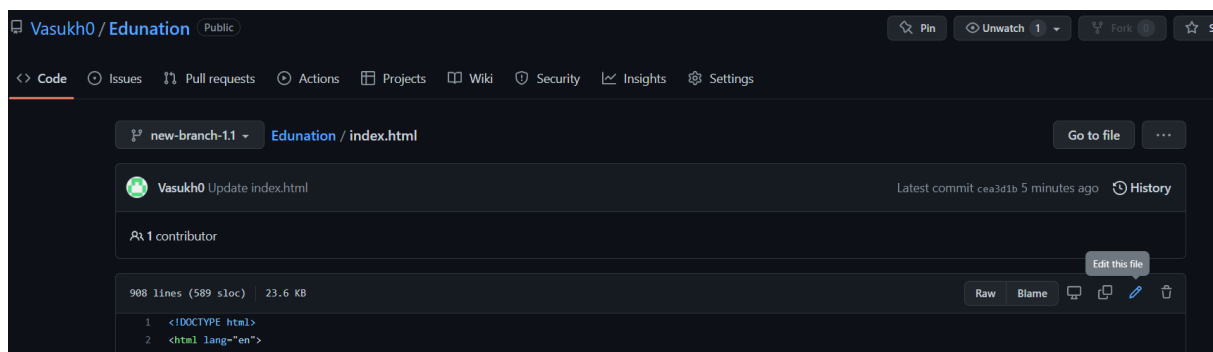
- Now all members are ready to contribute to the project.

## Task 2 - Open and close a pull request.

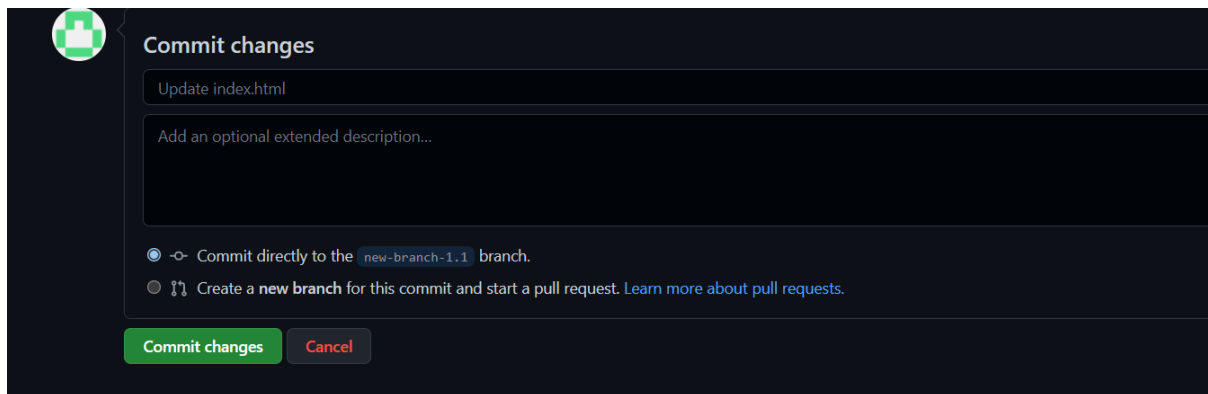
1. To open a pull request we first have to make a new branch, by using git branch *branchname* option.



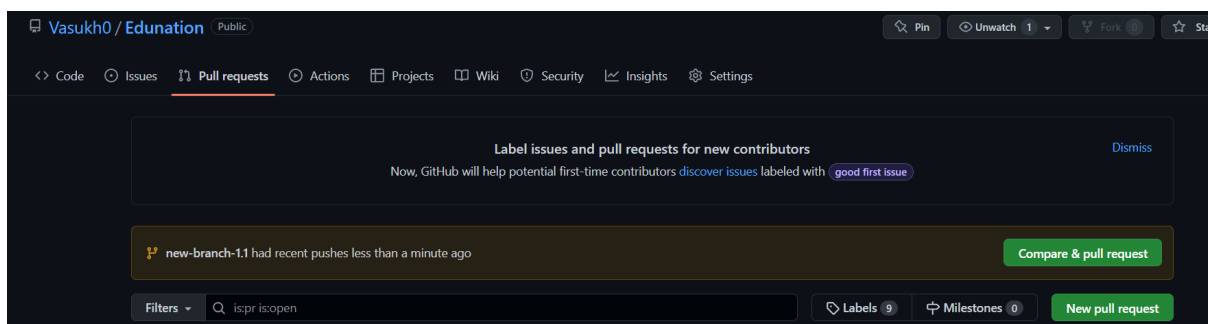
2. After making a new branch we add a file to the branch or make changes in the existing file.



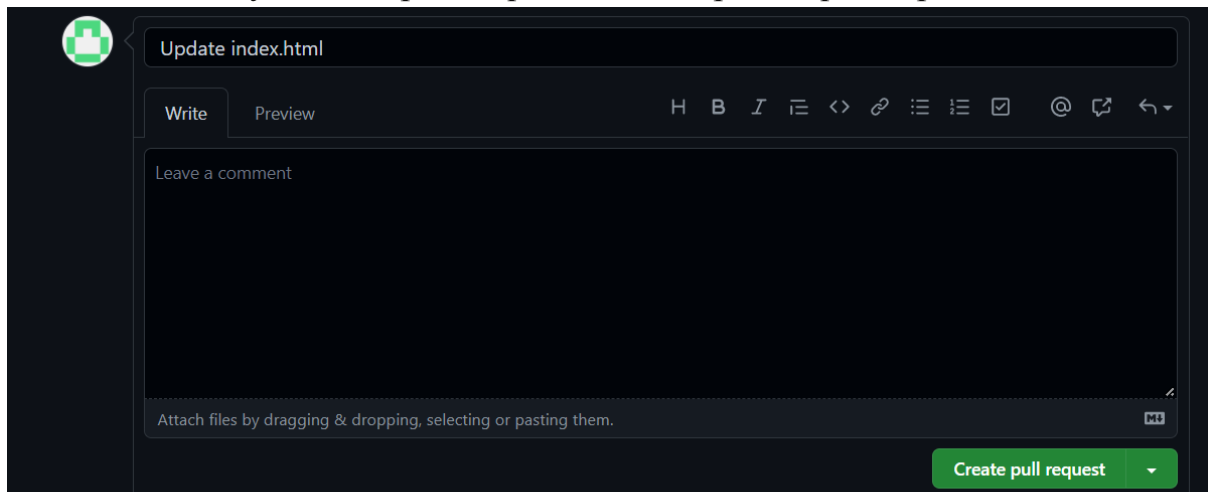
3. Add and commit the changes to the local repository.



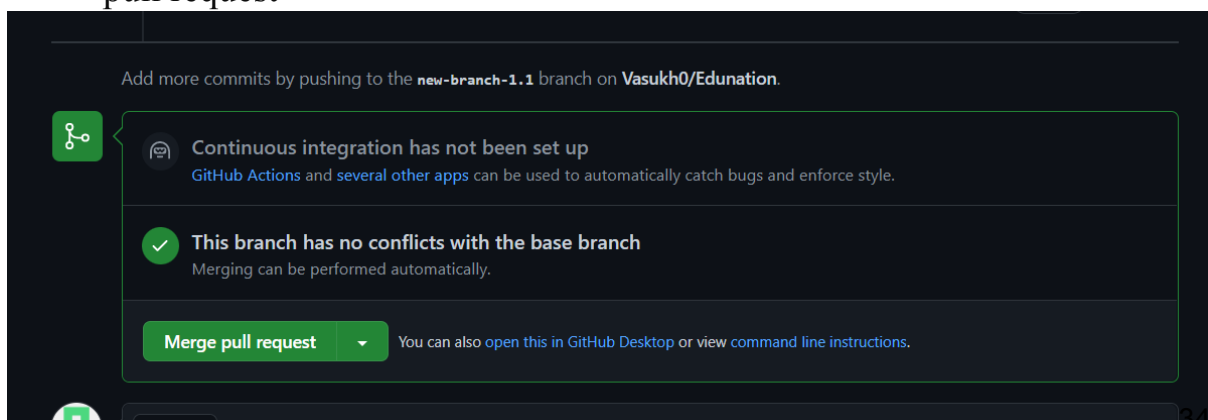
4. After pushing a new branch Github will either automatically ask you to create a pull request or you can create your own pull request.



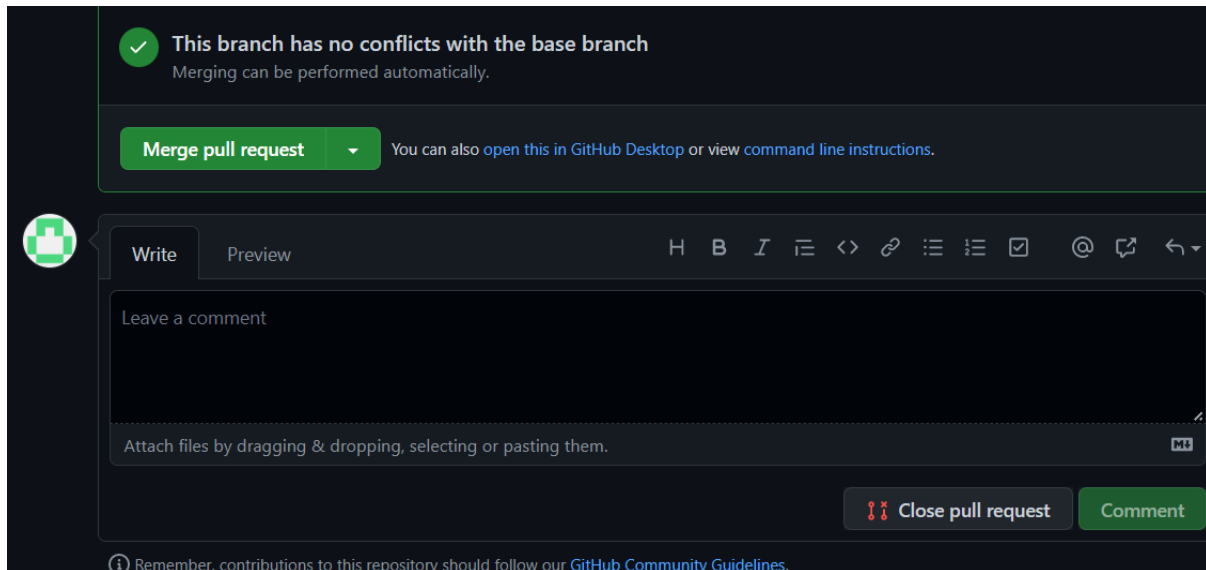
5. To create your own pull request click on pull request option.



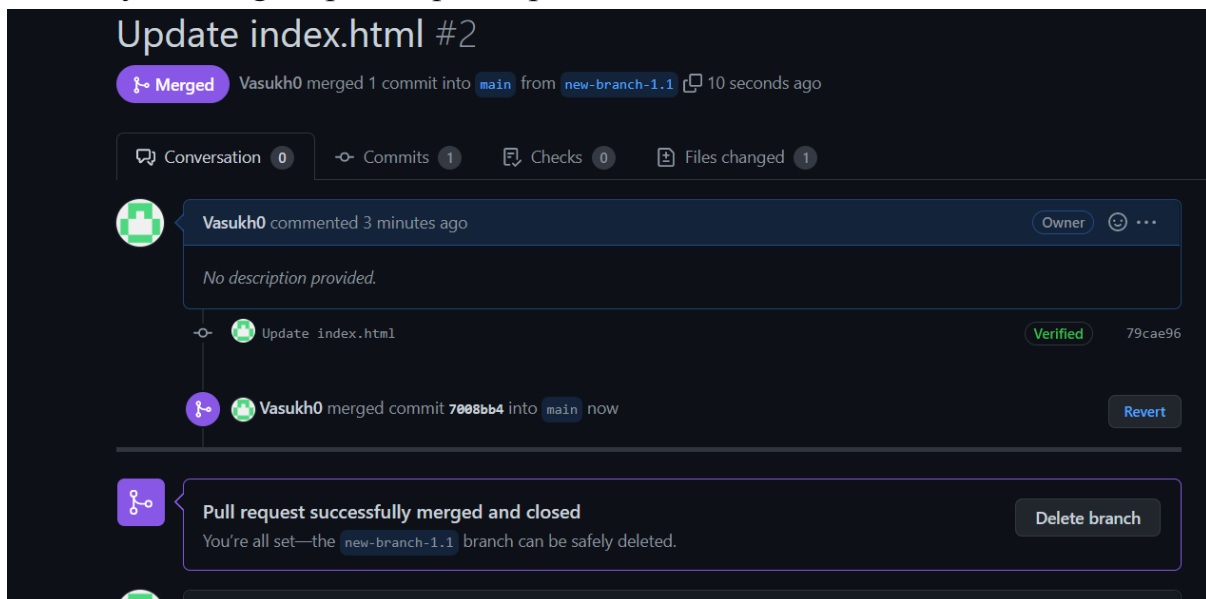
6. Github will detect any conflicts and ask you to enter a description of your pull request



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



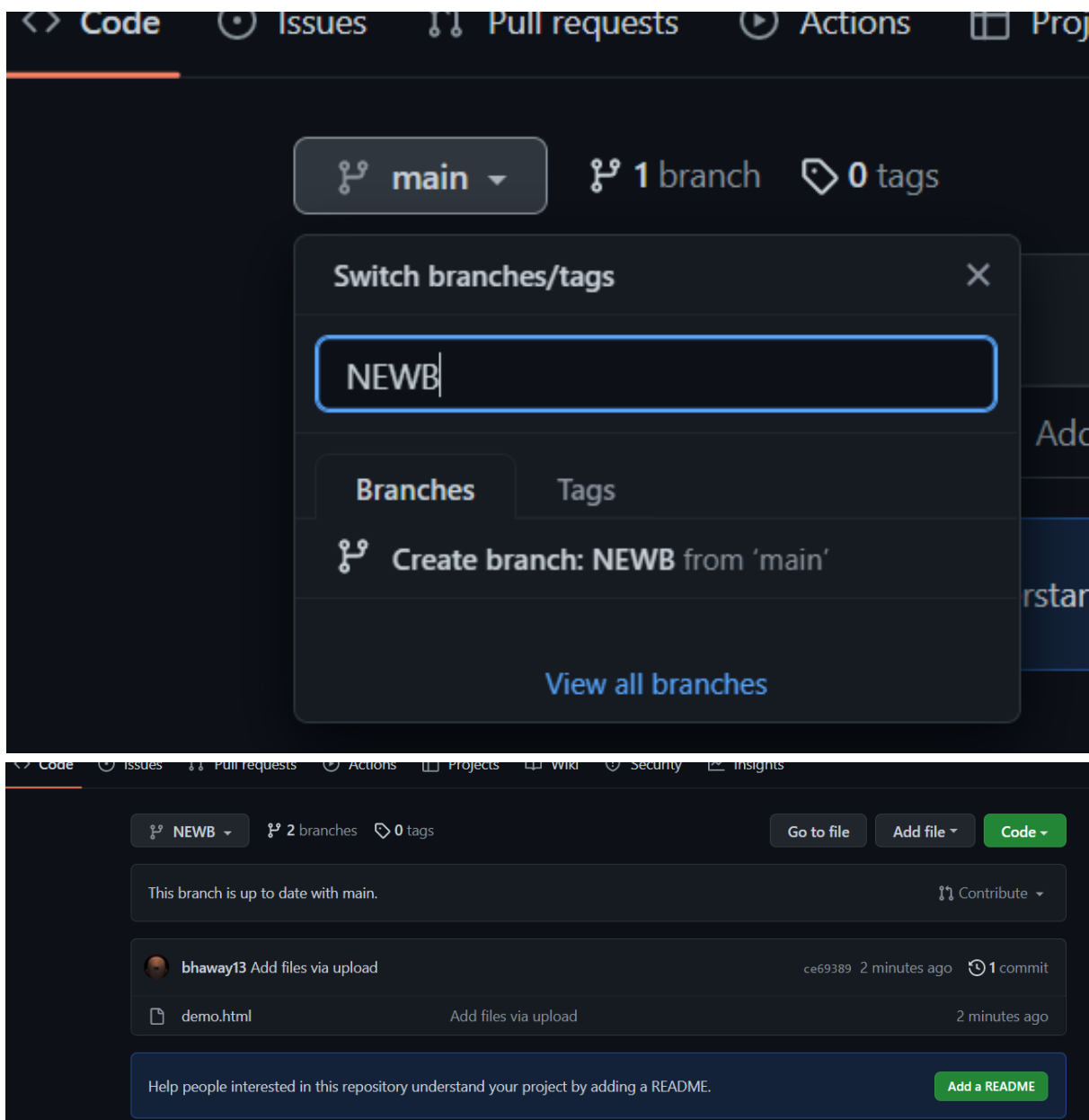
8. If the team member chooses not to merge your pull request they will close your pull request.
9. To close the pull request simply click on close pull request and add comment/ reason why you closed the pull request.
10. You can see all the pull requests generated and how they were dealt with by clicking on pull request option.



**Task 3** - Create a pull request on a team member's repo and Close pull requests generated by team members on your Repo as a maintainer.

### Pull request on a team member's Repo

1. To open a pull request we first have to make a new branch, by using git branch *branchname* option.

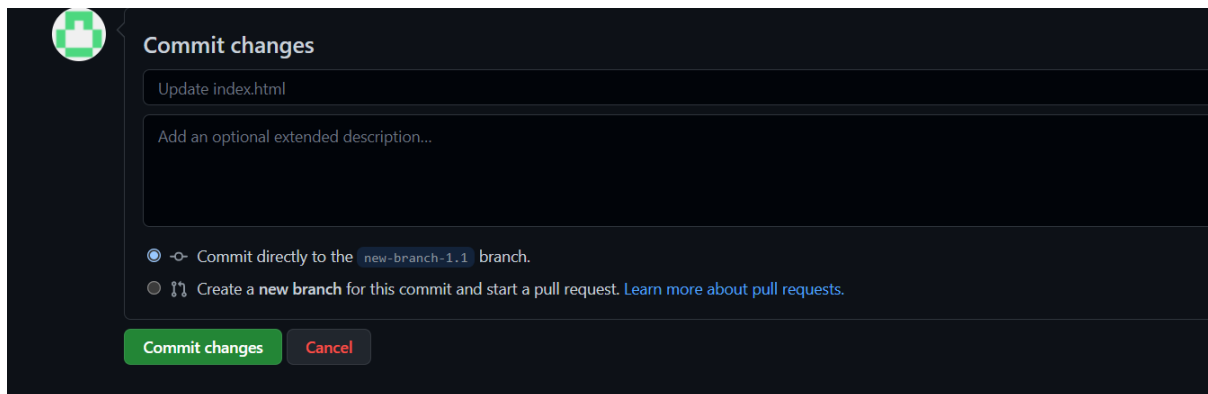




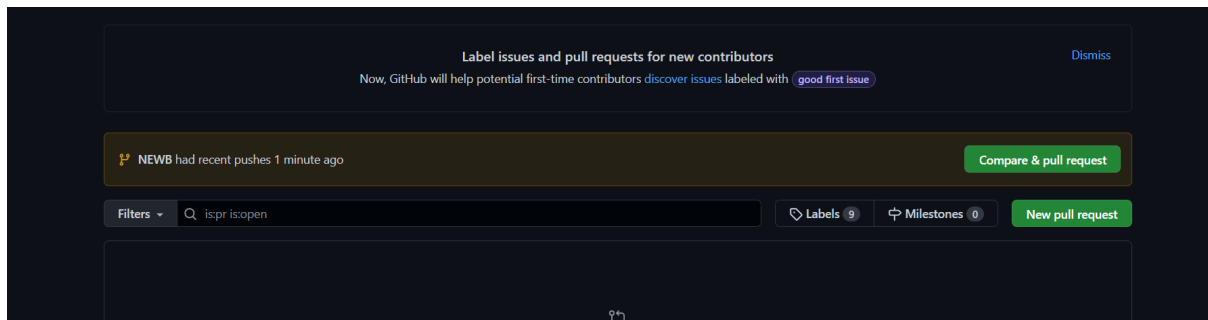
2. After making a new branch we add a file to the branch or make changes in the existing file.



3. Add and commit the changes to the local repository.

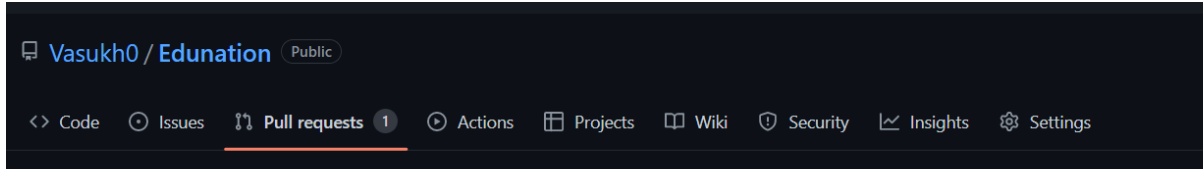


4. Pull request created.

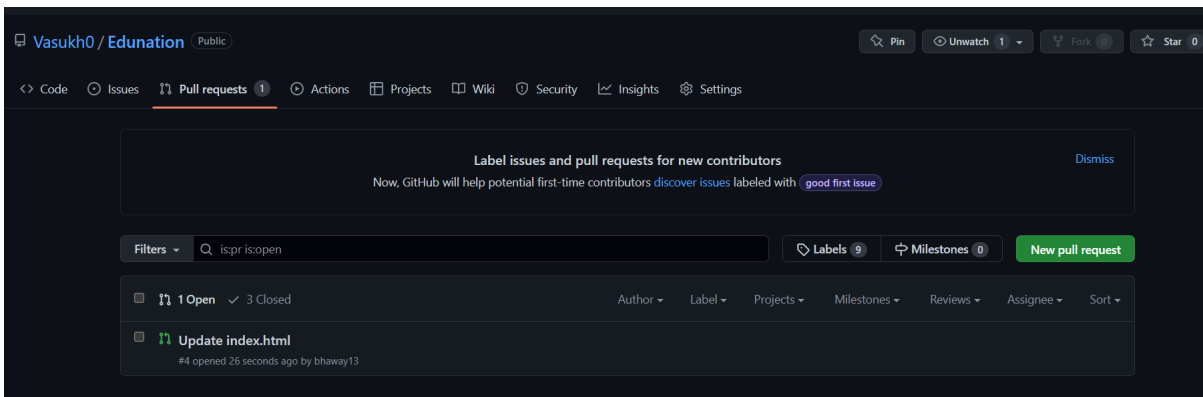


## Close requests generated by team members on your Repo

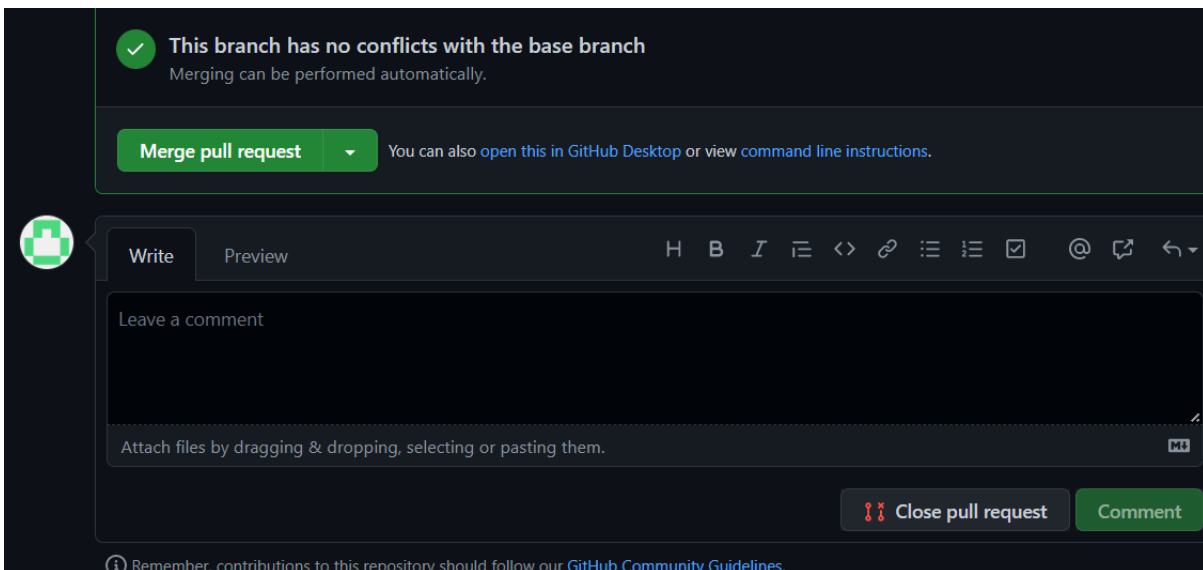
1. Ask your team member to login to his/her Github account.
2. They will notice a new notification in the pull request menu



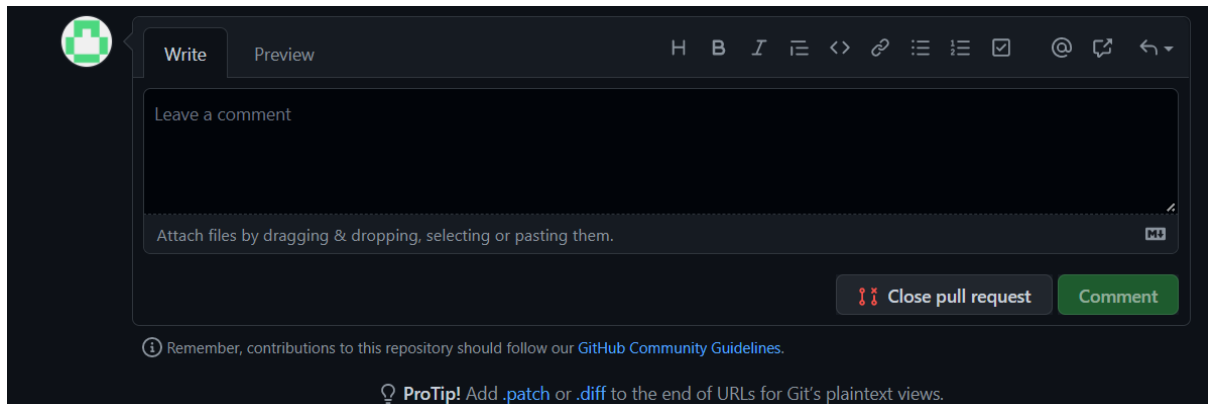
3. Click on it. The pull request generated by you will be visible to them.



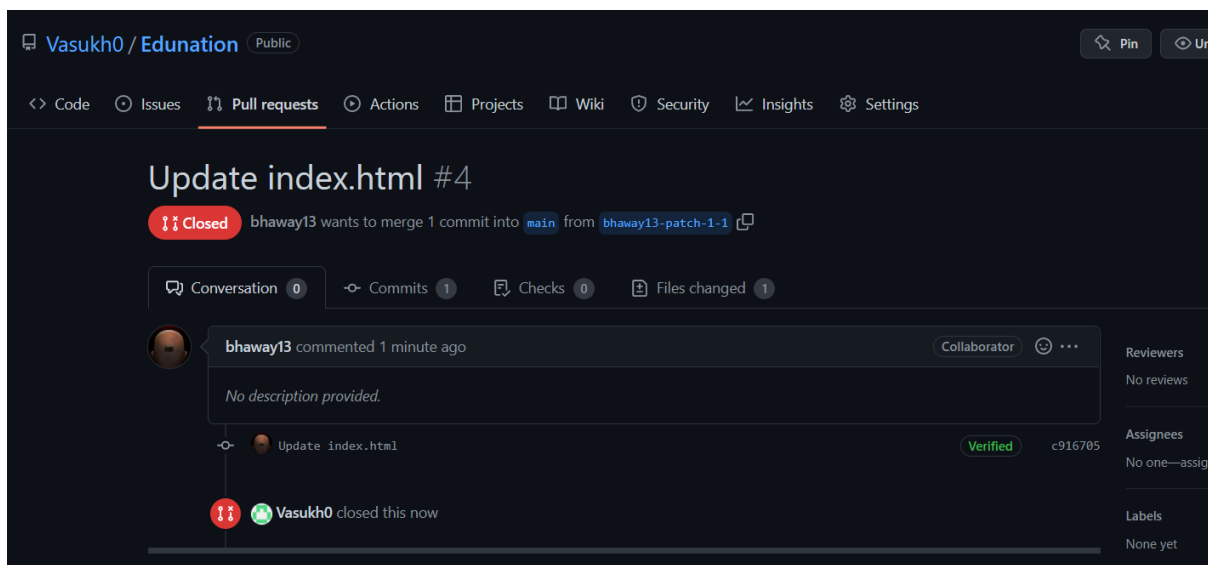
4. Click on it. The pull request generated by you will be visible to them.
5. By selecting the merge branch option the main branch will get updated for all the team members.



6. By selecting Close the pull request the pull request is not accepted and not merged with the main branch.



7. The result of closing the request is shown below.



## Task 4 - Publish and print network graphs.

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

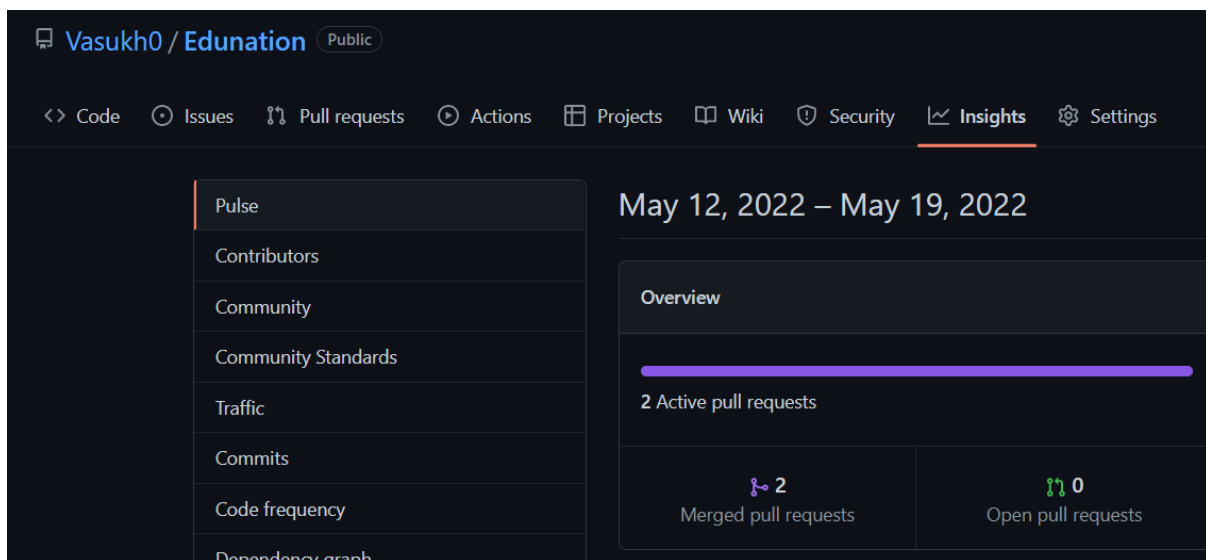
A repository's graphs give you information on traffic, projects that depend on the repository, contributors and commits to the repository, and a repository's forks and network. If you maintain a repository, you can use this data to get a better understanding of who's using your repository and why they're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

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

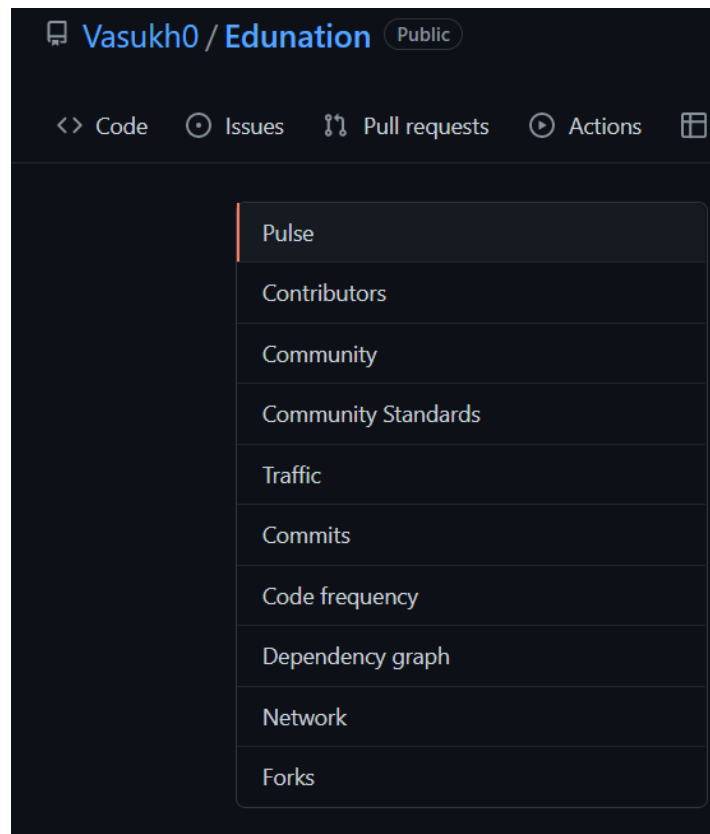
- Pulse
- Contributors
- Traffic
- Commits
- Code frequency
- Network

### Steps to access network graphs of respective repository

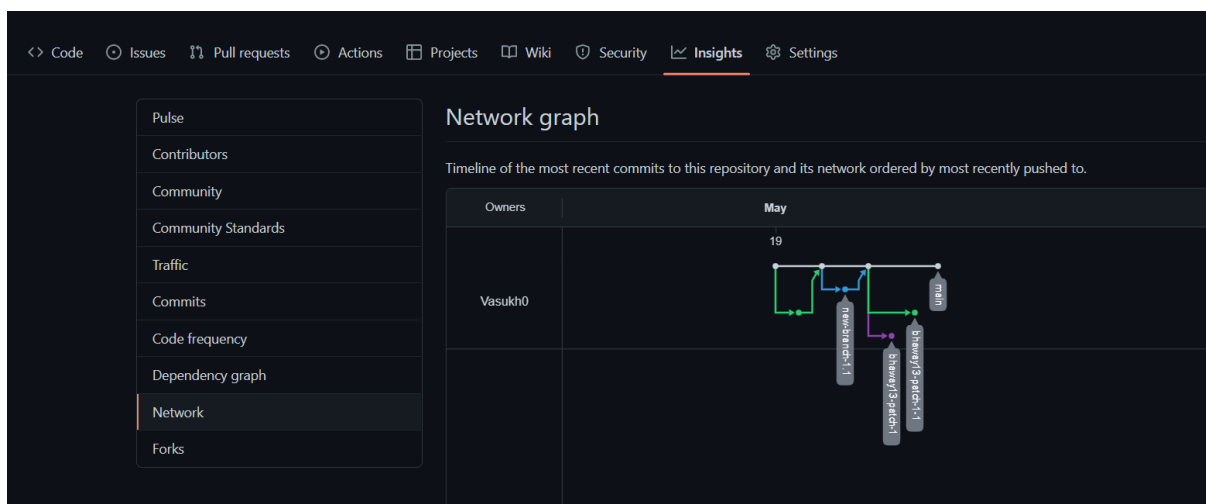
1. On GitHub.com, navigate to the main page of the repository.
2. Under your repository name, click **Insights**.



3. At the left sidebar, click on **Network**.



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



## Workflow and Discussion

1. I created a repository by the name Edunation and added all my teammates into the repository ( Bhaway , Udit , Himanshu ) by clicking on settings < Collaborators < Add a collaborator , then by typing their username into the search box and clicking on add collaborator.
2. Everyone accepted the request on their mail and were added as collaborators in the repository.
3. After that we added the files in the repository and then committed all the files.
4. Then I made a branch and made some changes in the file “index” and committed it and created a pull request.
5. After that a pull request was created and Himanshu got 2 options to merge or to close the pull request. He merged the pull request and code in the index file was changed.
6. Then after discussion Bhaway made changes in the project and added files to the repository. Udit merged the pull request and all the code was saved.
7. After that Himanshu created a repository and added me , Bhaway and Udit as the collaborators. He added a txt file and added his name , roll no. and group to it.
8. Then I also added my name , roll no. and group to it and created a pull request and merged it and the same was done by Bhaway and Udit.
9. After that Bhaway made changes to the css file and this created a conflict as I made changes to the file also.
10. Udit made some changes to the code and merged the pull request.
11. After the merge conflicts got resolved I got the mail regarding it.
12. After that I clicked on Insights < Network to print and publish the network graph of our project repository



## Manage access

[Add people](#)

☐ Select all Type ▾

<input type="checkbox"/>	<b>bhaway13</b> Collaborator	<a href="#">Remove</a>
<input type="checkbox"/>	<b>Himanshu Garg</b> HimanshuGarg57 • Collaborator	<a href="#">Remove</a>
<input type="checkbox"/>	<b>Udit</b> Udit-Rohilla • Collaborator	<a href="#">Remove</a>

Get team access controls and discussions for your contributors in an organization.  
**NEW** Private repos and unlimited members are free. [Create an organization](#)

**Vasukh0 / Edunation** Public

[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#)

**main** ▾ 1 branch 0 tags

**Switch branches/tags** ✕

**Branches** **Tags**

**Create branch: new branch 1.1** from 'main'

[View all branches](#)

[Vasukh0/new-branch1.1](#)

[Update index.html](#)

[Understand your project b](#)

**Commit changes**

☒ Commit directly to the **new-branch-1.1** branch.

☐ Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)


[Commit changes](#) [Cancel](#)


✓ **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 H B I ≡ <> 🔗 ≡ ≡ ☑ @ ↻ ↶


Leave a comment





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


 Close pull request **Comment**

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).



## Update index.html #2



 Merged Vasukh0 merged 1 commit into `main` from `new-branch-1.1` 10 seconds ago


 Conversation 0  Commits 1  Checks 0  Files changed 1

 Vasukh0 commented 3 minutes ago Owner ⌵

No description provided.

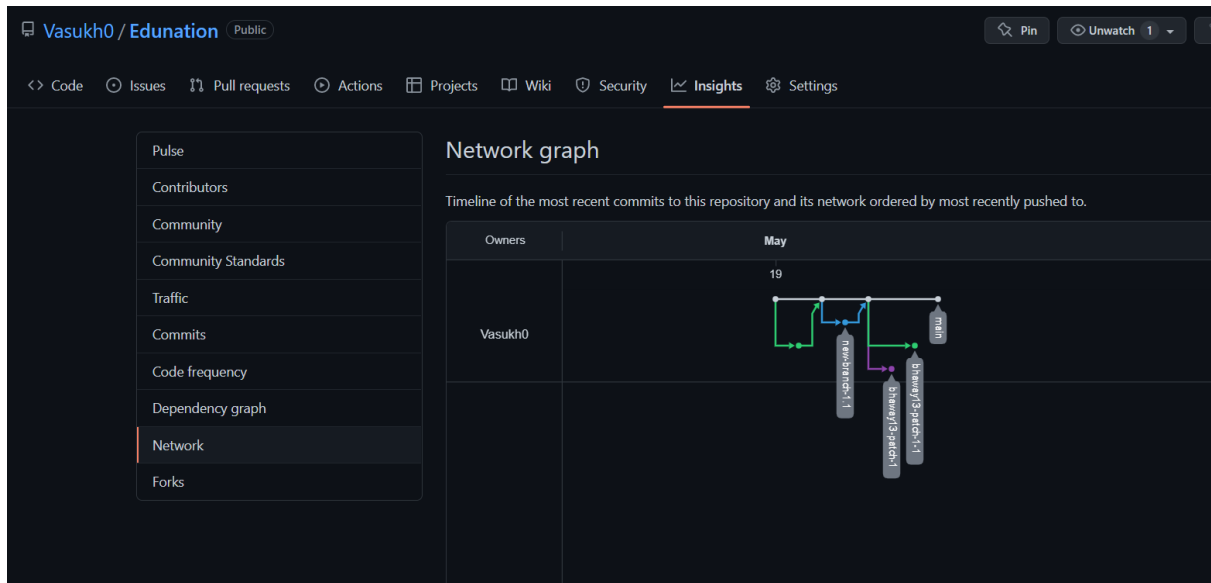
  Update index.html Verified 79cae96

  Vasukh0 merged commit 7008bb4 into `main` now Revert

 Pull request successfully merged and closed  
You're all set—the `new-branch-1.1` branch can be safely deleted. Delete branch

1	NAME	ROLL NO.	GROUP NO.
2			
3	1.HIMANSHU GARG	2110992044	G27
4	2.VASU KHATTAR	2110992054	G27
5	3.BHAWAY KHATTAR	2110992055	G27-A
6	4.Udit	2110992026	G27-A





# Reference

Git Version Control System :

<https://www.freecodecamp.org/news/what-is-git-and-how-to-use-it-c341b049ae61/>

<https://phoenixnap.com/kb/how-git-works>

<https://bodhizazen.net/git-advantages-and-disadvantages/>

<https://www.toolsqa.com/git/what-is-git/>

Resources Requirements – Frontend:

<https://techbootcamps.utexas.edu/blog/html-css-javascript/>