Subject Name: **Source Code Management**

Subject Code: **CS181**

Cluster: **beta** Group: **G8**

Department: **DCSE**

CHITKARA UNIVERSITY

Submitted By:

Devwani thakur

2110990430

G8

Submitted To:

Dr. Monit kapoor

List of Programs

CHITKARA UNIVERSITY

EXPERIMENT NO. 1

Aim: Setting up of Git Client.

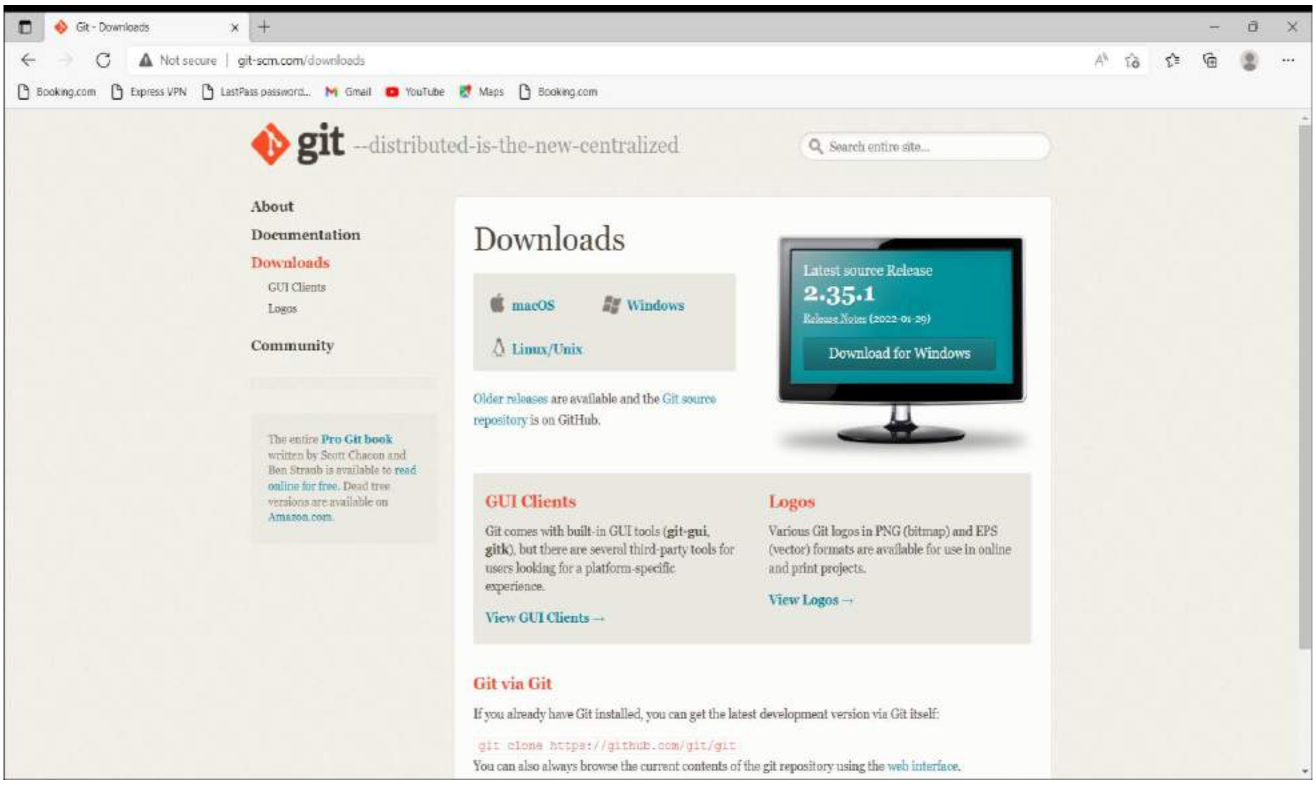
Installing and Configuring the Git client

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 (Window, Mac, Linux) from http://git-scm.com/downloads.



The entire Pro Git book written by Scott Chacon and Ben Straub is available to read online for free. Dead tree versions are available on Amazon.com.

Older releases are available and the Git source repository is on GitHub.

Git comes with built-in GUI tools (git-gui, gitk), but there are several third-party tools for users looking for a platform-specific experience.

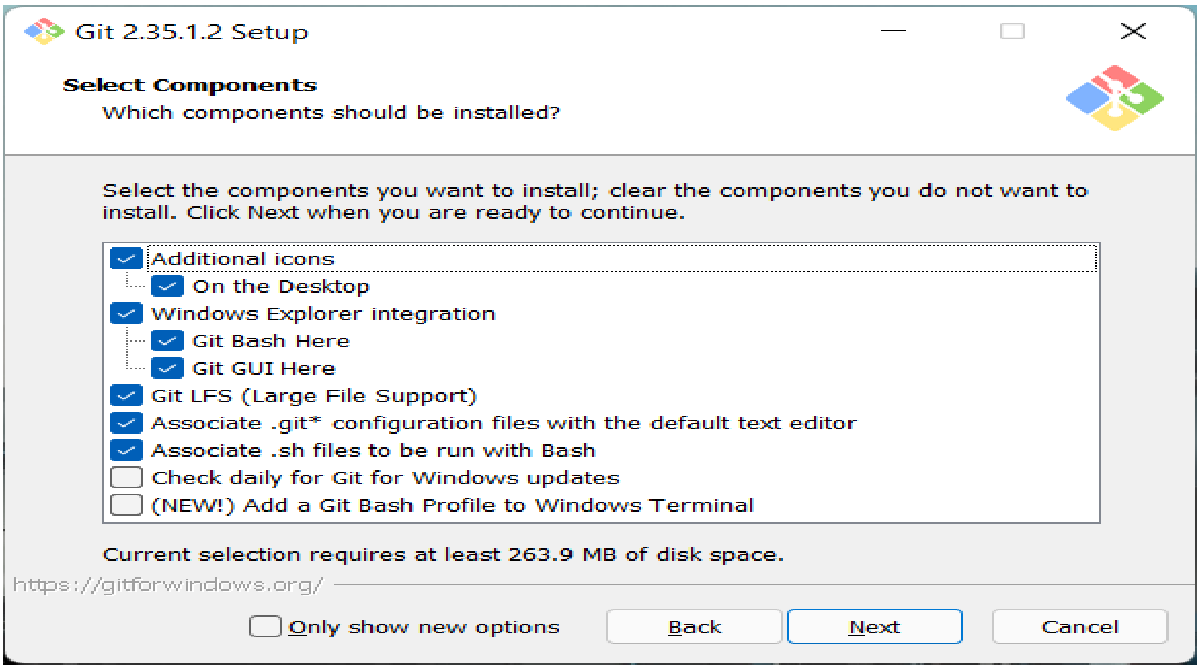
Various Git logos in PNG (bitmap) and EPS (vector) formats are available for use in online and print projects.

If you already have Git installed, you can get the latest development version via Git itself:

```
git clone https://github.com/git/git
```

You can also always browse the current contents of the git repository using the web interface.

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:



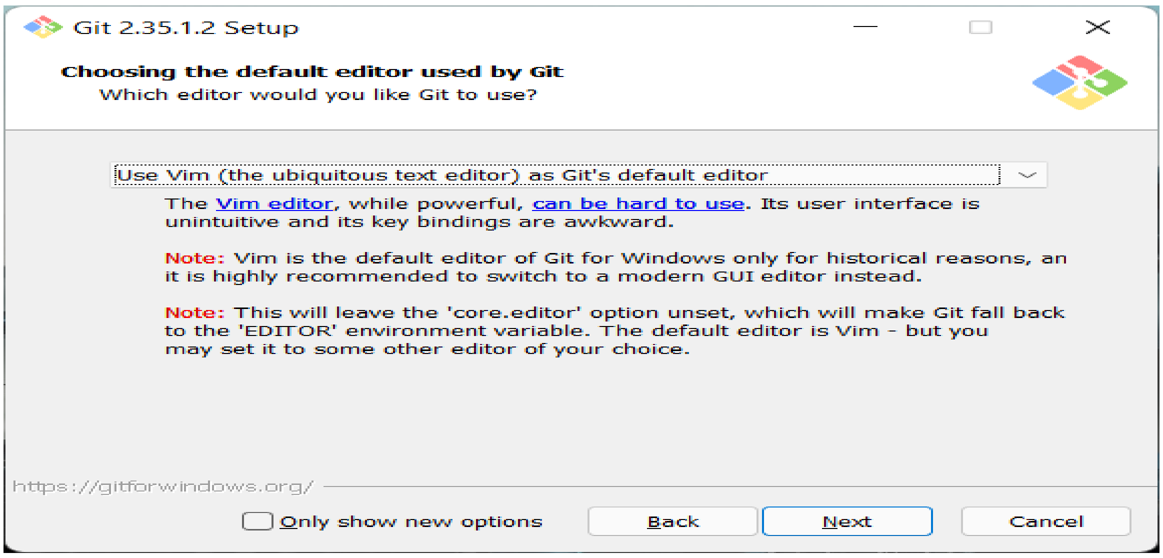
Select Components
Which components should be installed?

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

- [✓] Additional icons
  - [✓] On the Desktop
- [✓] Windows Explorer integration
  - [✓] Git Bash Here
  - [✓] Git GUI Here
- [✓] Git LFS (Large File Support)
- [✓] Associate .git* configuration files with the default text editor
- [✓] Associate .sh files to be run with Bash
- [ ] Check daily for Git for Windows updates
- [ ] (NEW!) Add a Git Bash Profile to Windows Terminal

Current selection requires at least 263.9 MB of disk space.

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.



Choosing the default editor used by Git
Which editor would you like Git to use?

Use Vim (the ubiquitous text editor) as Git's default editor

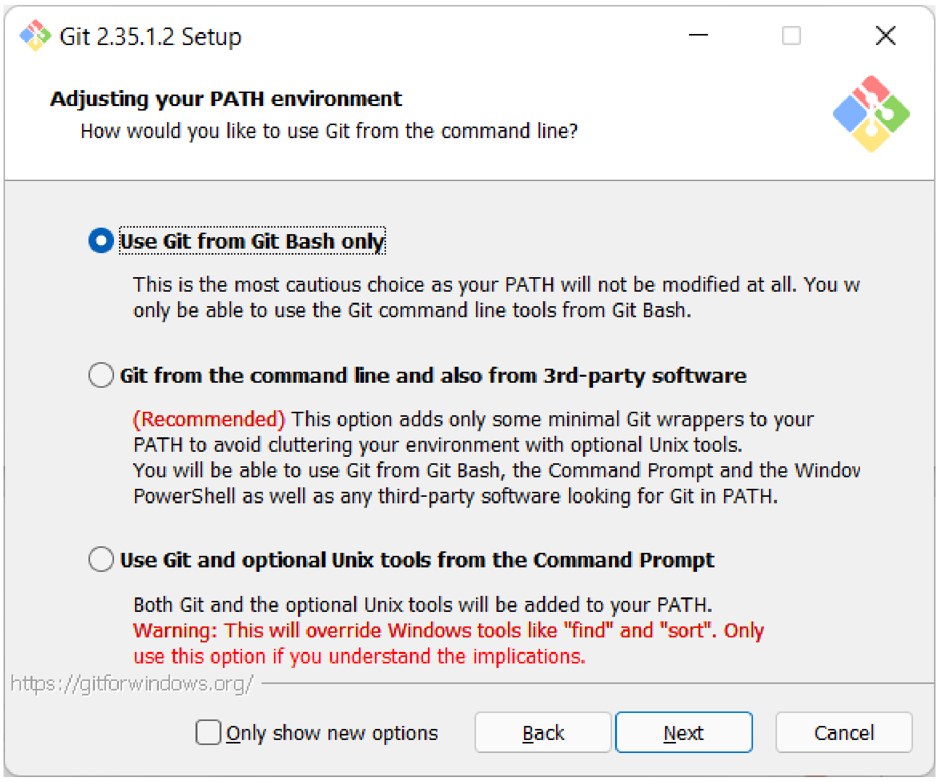
The Vim editor, while powerful, can be hard to use. Its user interface is unintuitive and its key bindings are awkward.

Note: Vim is the default editor of Git for Windows only for historical reasons, an it is highly recommended to switch to a modern GUI editor instead.

Note: This will leave the 'core.editor' option unset, which will make Git fall back to the 'EDITOR' environment variable. The default editor is Vim - but you may set it to some other editor of your choice.

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.


Git 2.35.1.2 Setup

Adjusting your PATH environment

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

[✓ Use Git from Git Bash only]

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

Git from the command line and also from 3rd-party software

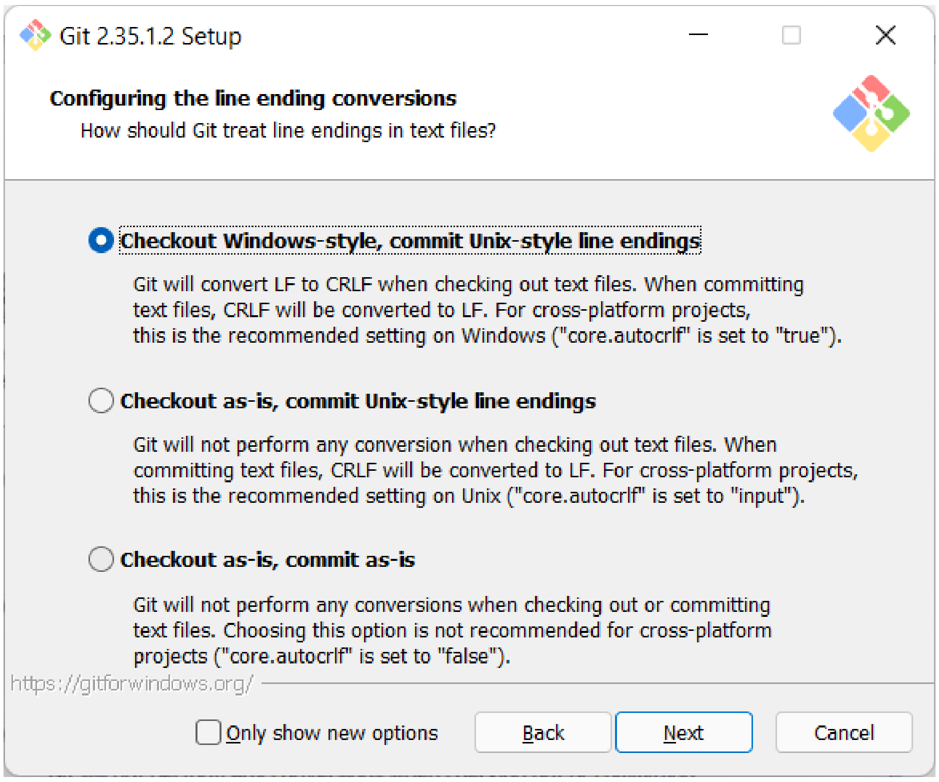
(Recommended) This option adds only some minimal Git wrappers to your
PATH to avoid cluttering your environment with optional Unix tools.
You will be able to use Git from Git Bash, the Command Prompt and the Windov
PowerShell as well as any third-party software looking for Git in PATH.

Use Git and optional Unix tools from the Command Prompt

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

https://gitforwindows.org/

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


Git 2.35.1.2 Setup

Configuring the line ending conversions

How should Git treat line endings in text files?

[✓ Checkout Windows-style, commit Unix-style line endings]

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

Checkout as-is, commit Unix-style line endings

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

Checkout as-is, commit as-is

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

https://gitforwindows.org/

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:

# !.classpat

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

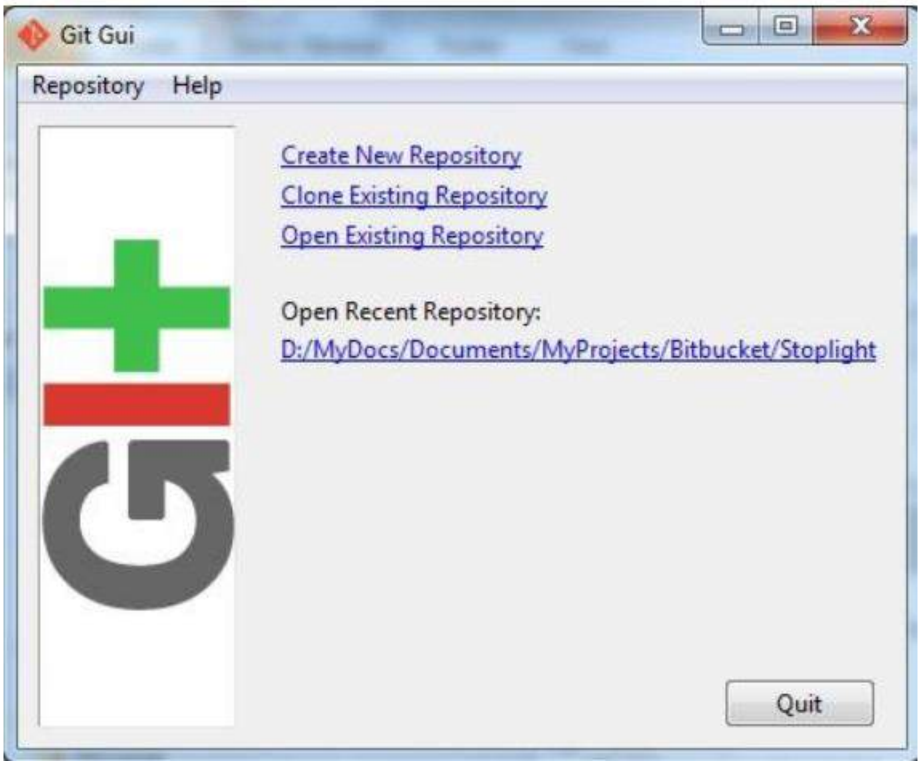
   This ensures that all pushes go back to the branch from which they were pulled. Otherwise pushes will go to the master branch, forcing a merge.

4. Generating public/private key pairs for authentication

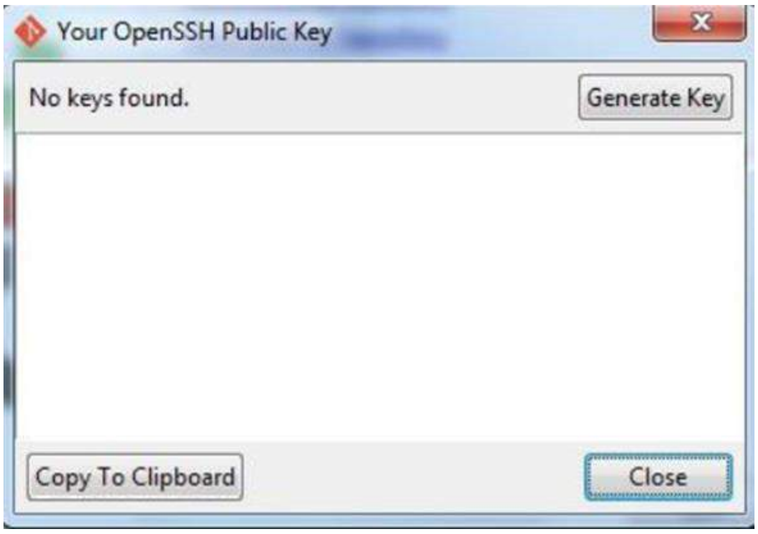
This part is critical and used to authenticate your access to the repository.

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 user 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).

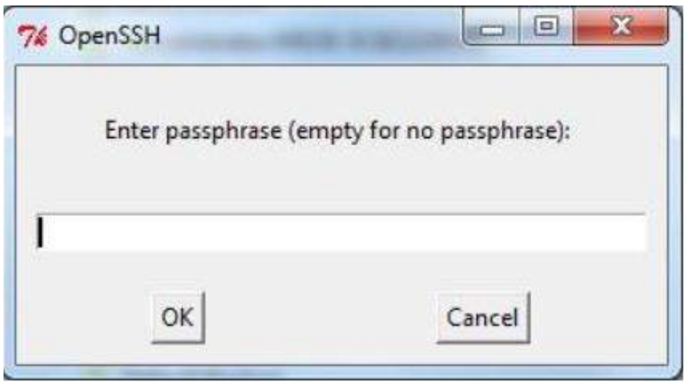
Generating the key pair is easy: From within File Explorer, right-click on any folder. From the context menu, select **Git GUI Here.** The following appears:



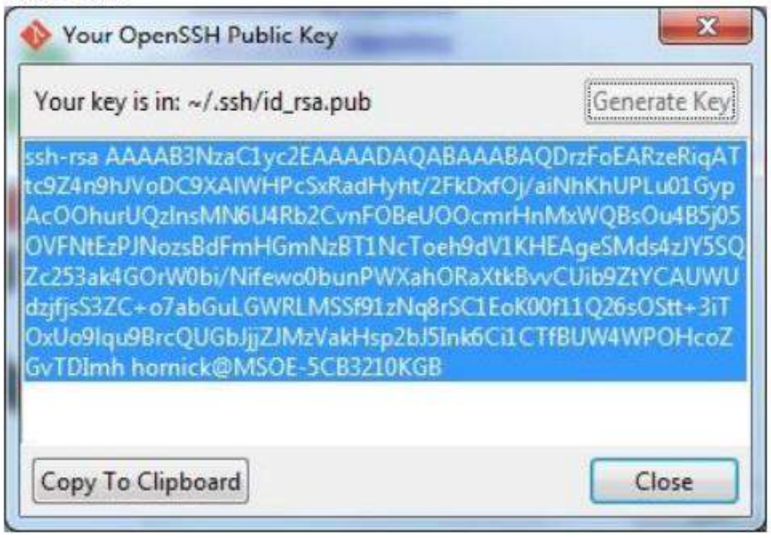
From the **Help** menu, select the **Show SSH Key** command. The following pup-up dialog appears:



Initially, you have no public/private key pair; thus the message **"No keys found"** appears withing 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:



Your key is in: ~/.ssh/id_rsa.pub

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDrzFoEARzeRiqAT
tc9Z4n9hJVoDC9XAIWHPcSxRadHyht/2FkDxfOj/aiNhKhUPLu01Gyp
AcOOhurUQzlnsMN6U4Rb2CvnFOBeUOOcmrHnMxWQBsOu4B5j05
OVFNtEzPJNozsBdFmHGmNzBT1NcToeh9dV1KHEAgeSMds4zJY5SQ
Zc253ak4GOrW0bi/Nifewo0bunPWXahORaXtkBvvCUib9ZtYCAUWU
dzjfjsS3ZC+o7abGuLGWRLMSSf91zNq8rSC1EoK00f11Q26sOStt+3iT
OxUo9lqu9BrcQUGbJjjZJMzVakHsp2bJ5Ink6Ci1CTfBUW4WPOHcoZ
GvTDImh hornick@MSOE-5CB3210KGB
```

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

EXPERIMENT NO. 2

Aim: Setting up GitHub Account

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, GitHub lets you complete most Git-related actions directly in the browser, from creating and forking repositories to editing files and opening pull requests.	This method is useful if you want a visual interface and need to do quick, simple changes that don't require working locally
GitHub Desktop	GitHub Desktop extends and simplifies your GitHub.com workflow, using a visual interface instead of text commands on the command line. For more information on getting started with GitHub Desktop, see "Getting started with GitHub Desktop."	This method is best if you need or want to work with files locally, but prefer using a visual interface to use Git and interact with GitHub.
IDE or text editor	You can set a default text editor, like Atom or Visual Studio Code to open and edit your files with Git, use extensions, and view the project structure. For more information, see "Associating text editors with Git."	This is convenient if you are working with more complex files and projects and want everything in one place, since text editors or IDEs often allow you to directly access the command line in the editor.
Command line, with or without 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 "Git cheatsheet." 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 "GitHub CLI."	This is most convenient if you are already working from the command line, allowing you to avoid switching context, or if you are more comfortable using the command line.
GitHub API	GitHub has a REST API and GraphQL API that you can use to interact with GitHub. For more information, see	The GitHub API would be most helpful if you wanted to automate common tasks, back up your data, or create

	"Getting started with the API."	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."

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

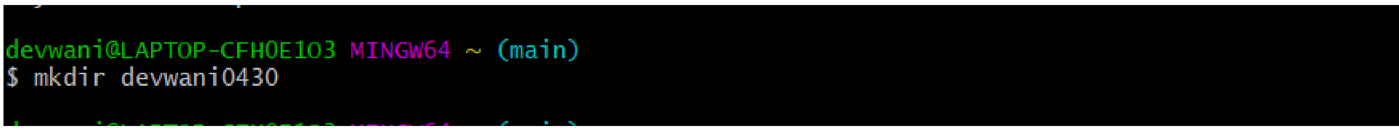
EXPERIMENT NO. 3

Aim: Generate logs

Commands -:

mkdir <name of Directory>

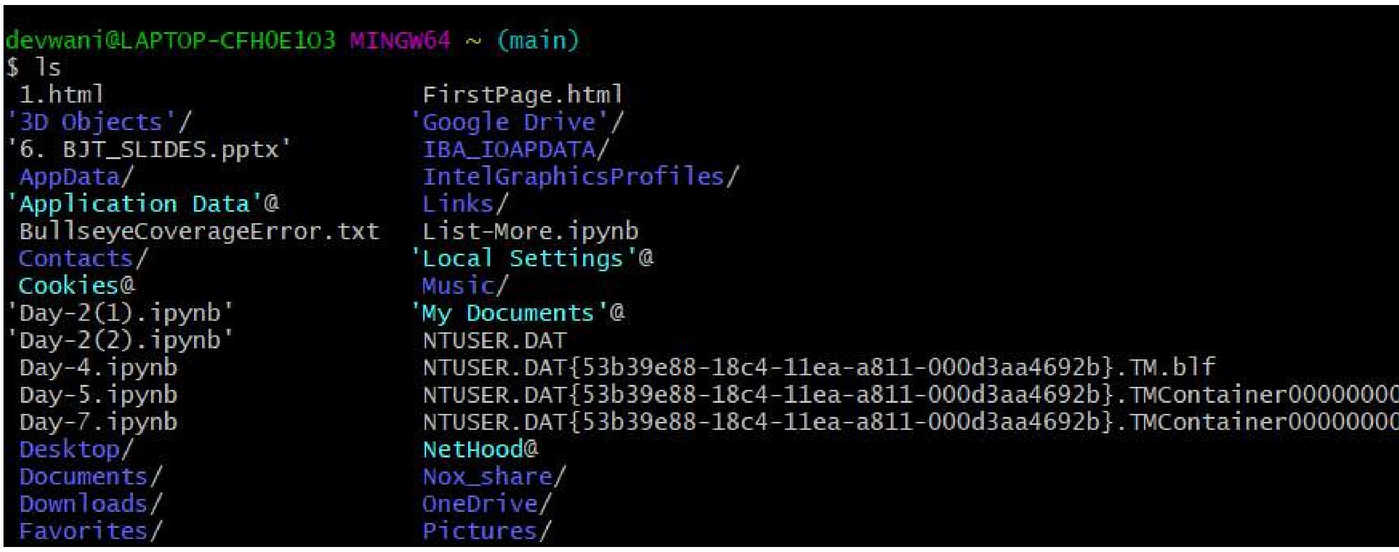
Allows users to create or make new directories



```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ mkdir devwani0430
```

ls

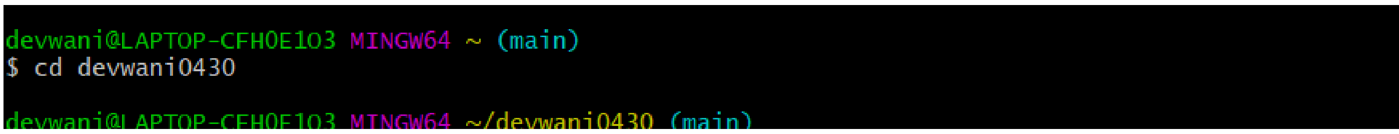
It is used **to list information about files and directories within the file system**.



```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ ls
 1.html                        FirstPage.html
'3D Objects'/                 'Google Drive'/
'6. BJT_SLIDES.pptx'           IBA_IOAPDATA/
 AppData/                      IntelGraphicsProfiles/
'Application Data'@            Links/
 BullseyeCoverageError.txt     List-More.ipynb
 Contacts/                    'Local Settings'@
 Cookies@                      Music/
'Day-2(1).ipynb'              'My Documents'@
'Day-2(2).ipynb'               NTUSER.DAT
 Day-4.ipynb                   NTUSER.DAT{53b39e88-18c4-11ea-a811-000d3aa4692b}.TM.blf
 Day-5.ipynb                   NTUSER.DAT{53b39e88-18c4-11ea-a811-000d3aa4692b}.TMContainer0000000
 Day-7.ipynb                   NTUSER.DAT{53b39e88-18c4-11ea-a811-000d3aa4692b}.TMContainer0000000
 Desktop/                      NetHood@
 Documents/                    Nox_share/
 Downloads/                    OneDrive/
 Favorites/                    Pictures/
```

cd

change the current working directory in various operating systems.

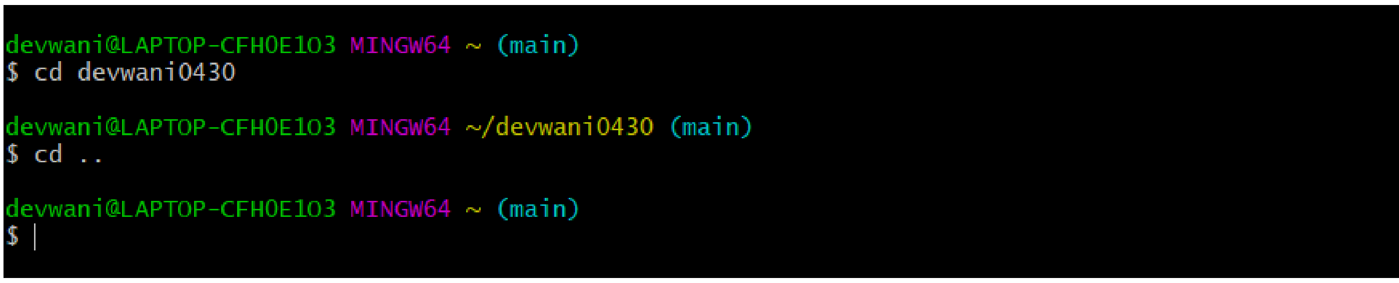


```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ cd devwani0430

devwani@LAPTOP-CFH0E103 MINGW64 ~/devwani0430 (main)
```

Cd ..

Use to exit from current directory.



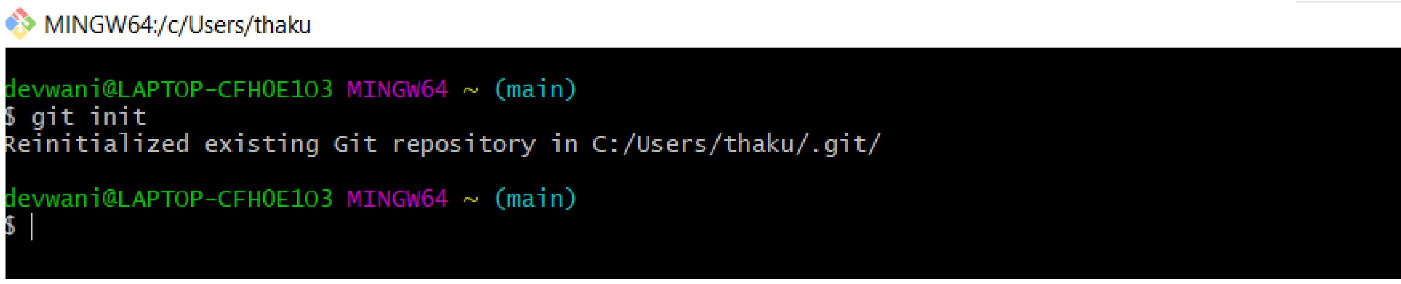
```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ cd devwani0430

devwani@LAPTOP-CFH0E103 MINGW64 ~/devwani0430 (main)
$ cd ..

devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ |
```

git init

The git init command **creates a new Git repository**.



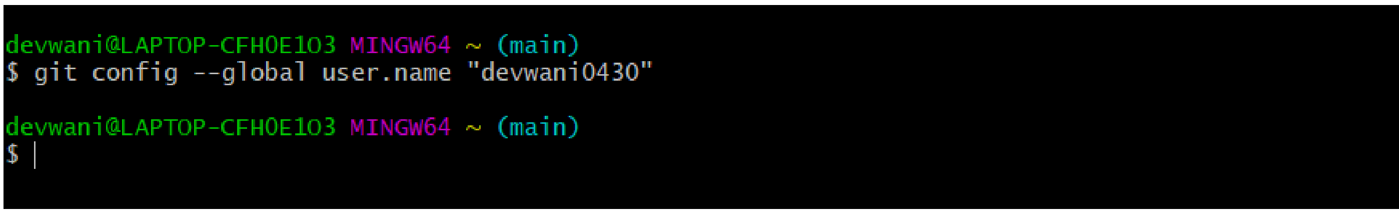
MINGW64:/c/Users/thaku

```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git init
Reinitialized existing Git repository in C:/Users/thaku/.git/

devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ |
```

git config --global user.name

To set your Git username, run the **git config –global user.name command**.

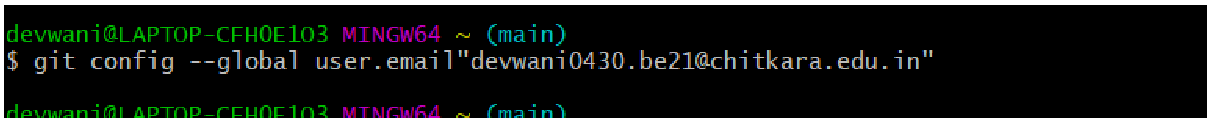


```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git config --global user.name "devwani0430"

devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ |
```

git config –global email id

Git allows you to set a global and per-project username and email address.

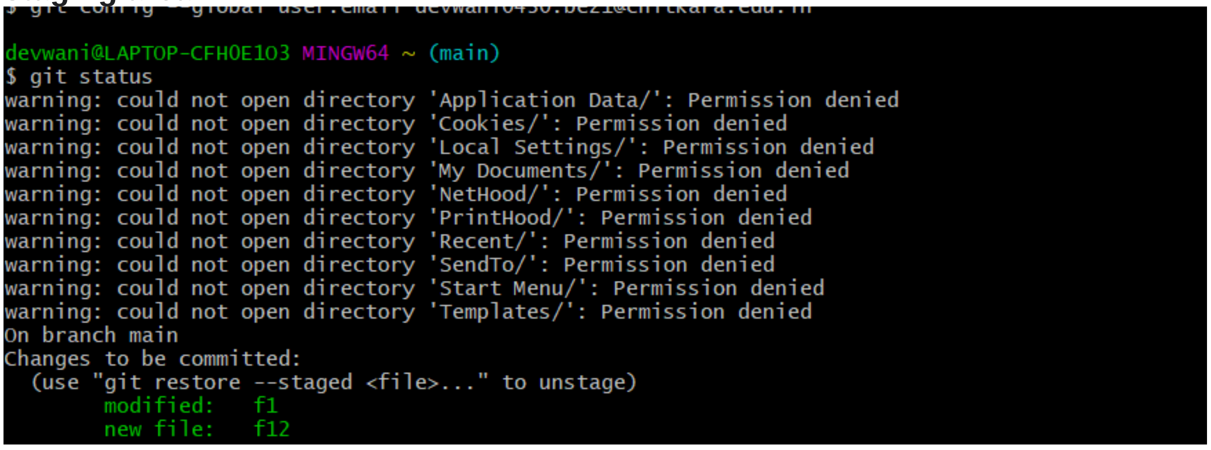


```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git config --global user.email"devwani0430.be21@chitkara.edu.in"

devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
```

git status

The git status command **displays the state of the working directory and the staging area.**

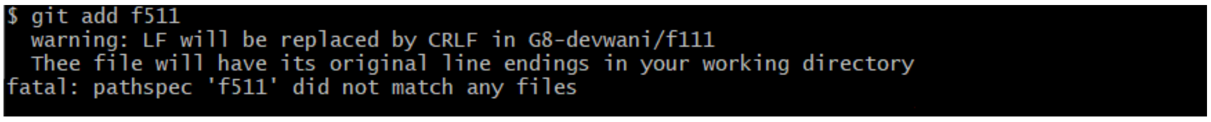


```
$ git config --global user.email devwani0430.be21@chitkara.edu.in

devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git status
warning: could not open directory 'Application Data/': Permission denied
warning: could not open directory 'Cookies/': Permission denied
warning: could not open directory 'Local Settings/': Permission denied
warning: could not open directory 'My Documents/': Permission denied
warning: could not open directory 'NetHood/': Permission denied
warning: could not open directory 'PrintHood/': Permission denied
warning: could not open directory 'Recent/': Permission denied
warning: could not open directory 'SendTo/': Permission denied
warning: could not open directory 'Start Menu/': Permission denied
warning: could not open directory 'Templates/': Permission denied
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   f1
        new file:   f12
```

git add

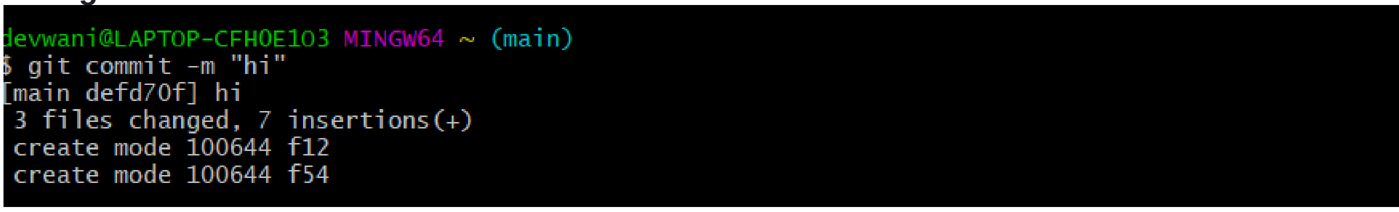
selects that file, and moves it to the staging area, marking it for inclusion in the next commit.



```
$ git add f511
  warning: LF will be replaced by CRLF in G8-devwani/f111
  Thee file will have its original line endings in your working directory
fatal: pathspec 'f511' did not match any files
```

git commt –m <any comment>

The git commit command **captures a snapshot of the project's currently staged changes**.

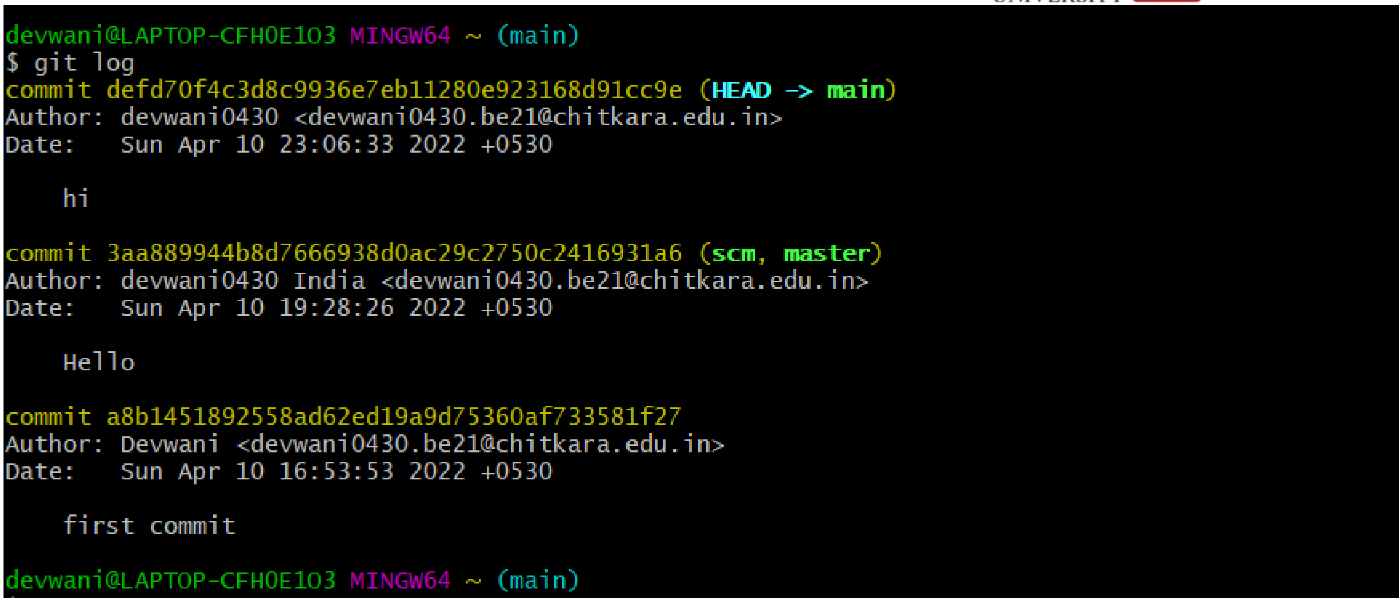


```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git commit -m "hi"
[main defd70f] hi
 3 files changed, 7 insertions(+)
 create mode 100644 f12
 create mode 100644 f54
```

git

git log

Git log is **a utility tool to review and read a history of everything that happens to a repository**.


```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git log
commit defd70f4c3d8c9936e7eb11280e923168d91cc9e (HEAD -> main)
Author: devwani0430 <devwani0430.be21@chitkara.edu.in>
Date:   Sun Apr 10 23:06:33 2022 +0530

    hi

commit 3aa889944b8d7666938d0ac29c2750c2416931a6 (scm, master)
Author: devwani0430 India <devwani0430.be21@chitkara.edu.in>
Date:   Sun Apr 10 19:28:26 2022 +0530

    Hello

commit a8b1451892558ad62ed19a9d75360af733581f27
Author: Devwani <devwani0430.be21@chitkara.edu.in>
Date:   Sun Apr 10 16:53:53 2022 +0530

    first commit

devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
```

EXPERIMENT NO. 4

Aim: Create and visualize branches

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 large-scale development effort, Git branches are an integral part of your everyday workflow. The following content will expand on the internal Git branching architecture.

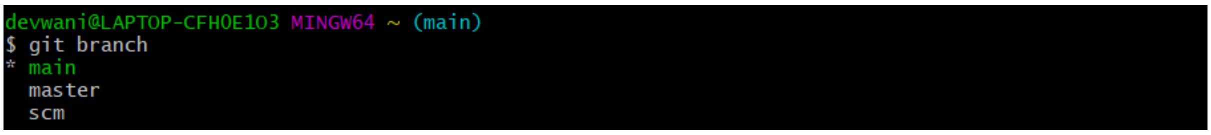
How it works

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

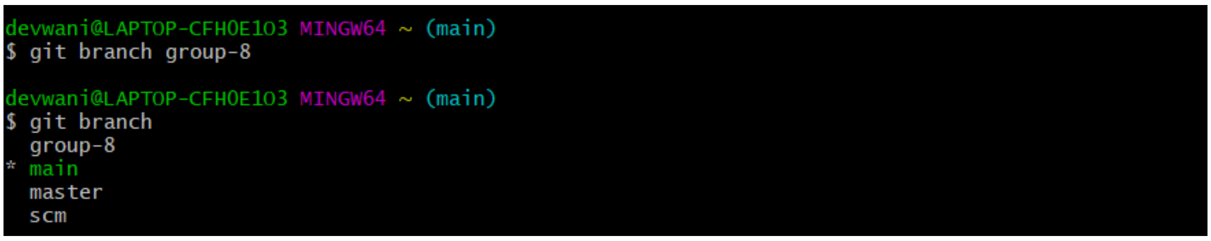


```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git branch
* main
  master
  scm
```

gi

Git branch <branch>

Create a new branch called ＜branch＞. This does not check out the new branch

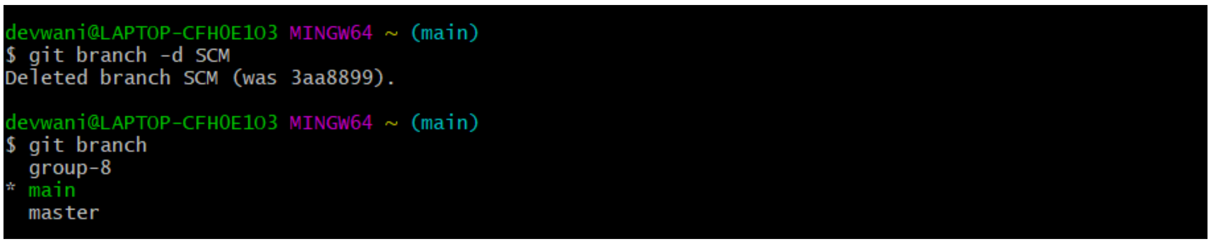


```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git branch group-8

devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git branch
  group-8
* main
  master
  scm
```

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.

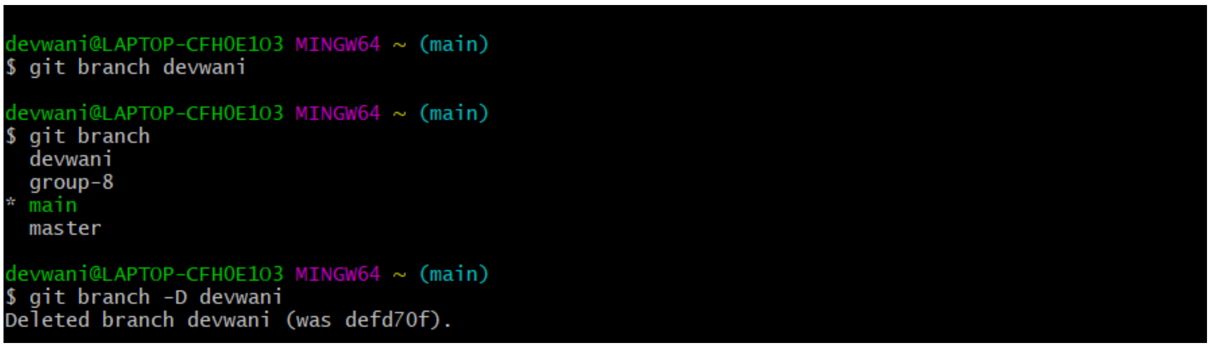


```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git branch -d SCM
Deleted branch SCM (was 3aa8899).

devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git branch
  group-8
* main
  master
```

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.



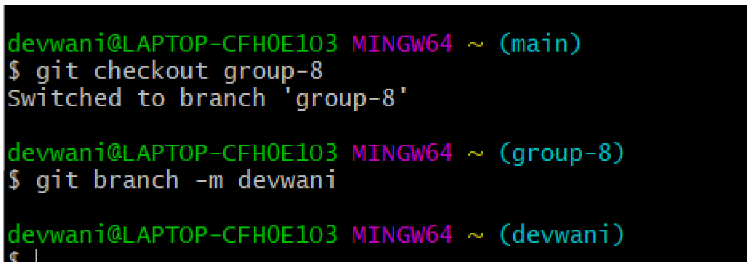
```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git branch devwani

devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git branch
  devwani
  group-8
* main
  master

devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git branch -D devwani
Deleted branch devwani (was defd70f).
```

Git branch –m <branch>

Rename the current branch to ＜branch＞.



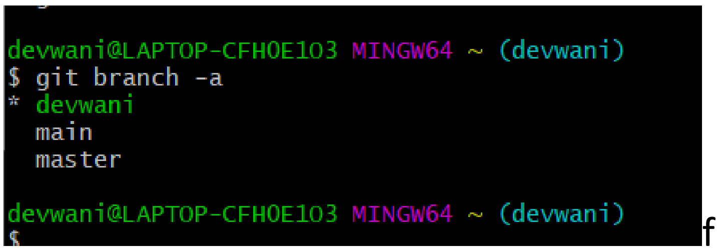
```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (main)
$ git checkout group-8
Switched to branch 'group-8'

devwani@LAPTOP-CFH0E103 MINGW64 ~ (group-8)
$ git branch -m devwani

devwani@LAPTOP-CFH0E103 MINGW64 ~ (devwani)
```

Git branch –a

List all remote branches



```
devwani@LAPTOP-CFH0E103 MINGW64 ~ (devwani)
$ git branch -a
* devwani
  main
  master

devwani@LAPTOP-CFH0E103 MINGW64 ~ (devwani)
$
```

f

Summary

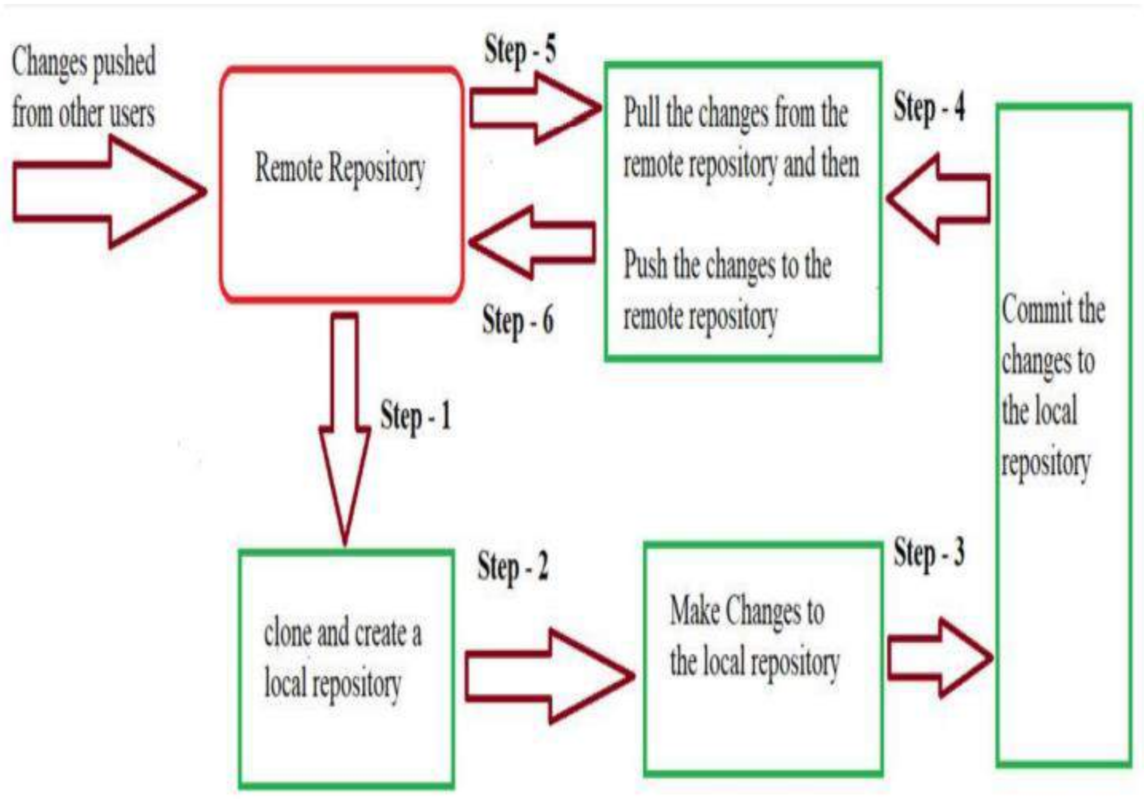
In this document we discussed Git's branching behavior and the git branch command. The git branch commands primary functions are to create, list, rename and delete branches. To operate further on the resulting branches the command is commonly used with other commands like git checkout. Learn more about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

EXPERIMENT NO. 5

Aim: Git lifecycle description.

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 –


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

- **In Step – 1**, We first clone any of the code residing in the remote repository to mak e our won local repository.
- **In Step-2** we edit the files that we have cloned in our local repository and make th e necessary changes in it.
- **In Step-3** we commit our changes by first adding them to our staging area and co mmitting 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 that changes.
- If there are no changes we directly proceed with **Step – 6** in which we push our ch anges 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 <file name>

// 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>