**Subject Name:** Source code management

**Subject Code:** CS181

**Cluster:** Beta

**Department:** DCSE

**Submitted By:**                                   **Submitted To:**

**Name:** Amarbir Singh                           Dr. Monit Kapoor

**Roll No.** 2110990159
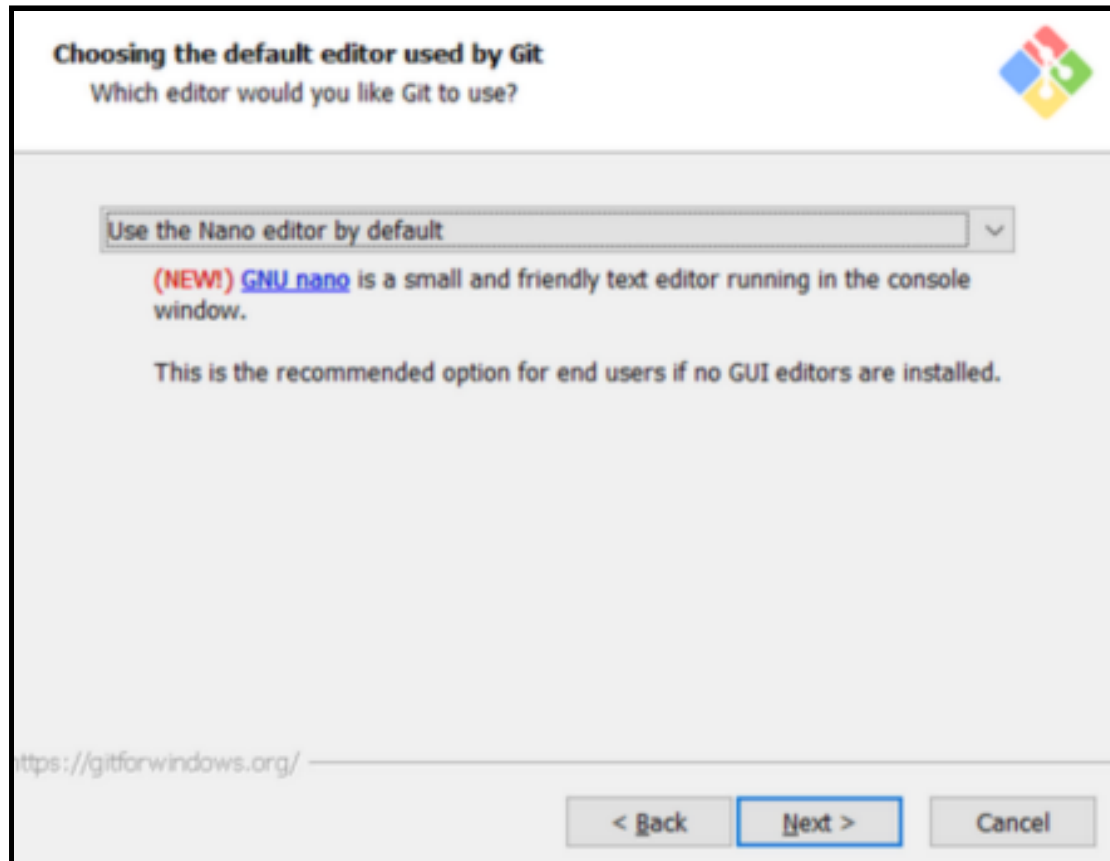
**Aim: Setting up the git client.**

**Git Installation:** **Download the Git installation program (Windows, Mac, or Linux) from Git - Downloads (git-scm.com).**

When running the installer, various screens appear (Windows screens shown). Generally, you can accept the default selections, except in the screens below where you do not want the default selections:In the Select Components screen, make sure Windows Explorer Integration is selected as shown:
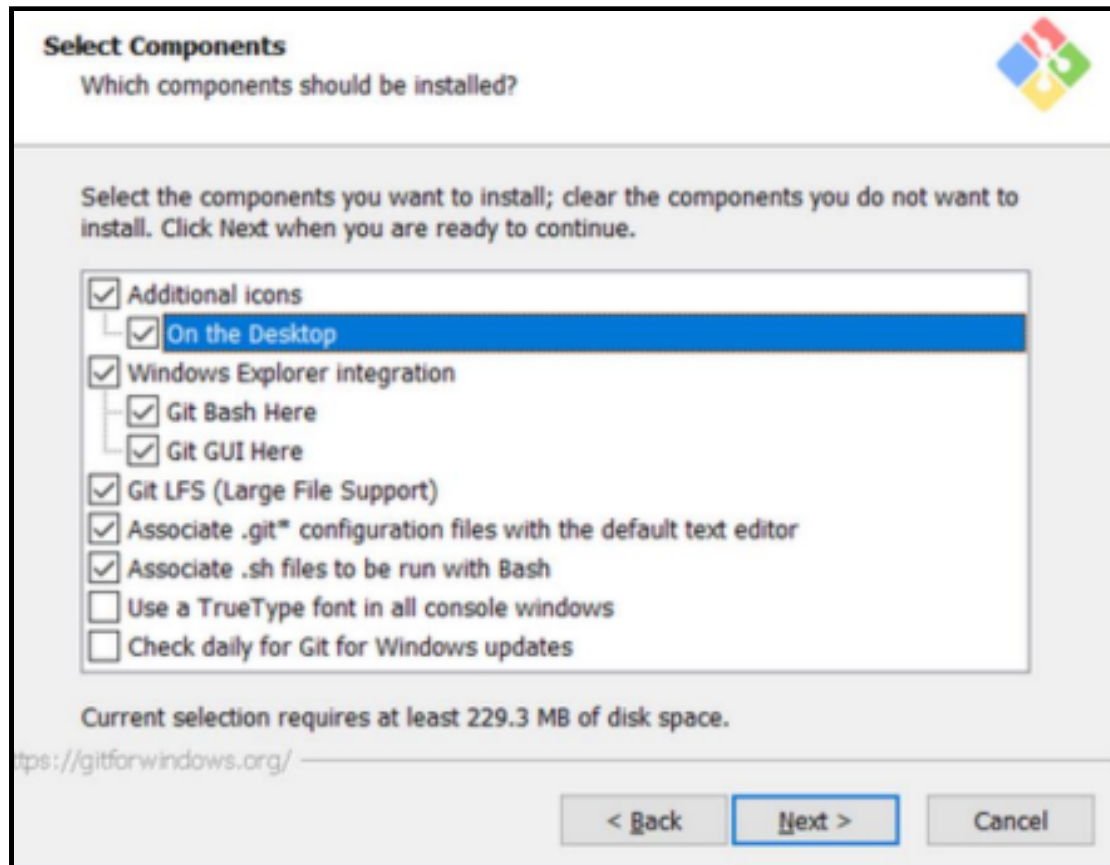


In the choosing the default editor is used by Git dialog, it is strongly recommended that you DO NOT select default VIM editor- it is challenging to learn how to use it, and there are better

modern editors available. Instead, choose Notepad++ or Nano – either of those is much easier to use. It is strongly recommended that you select Notepad++.

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

    Use the Nano editor by default                                    ∨

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

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




ttps://gitforwindows.org/

                                    < Back        Next >          Cancel
```

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

1. Use Git from Git Bash only: no integration, and no extra command in your command path.
2. Use Git from the windows Command Prompt: add flexibility – you can simply run git from a windows command prompt, and is often the setting for people in industry – but this does add some extra commands.
3. Use Git and optional Unix tools from the Windows Command Prompt: this is also a robust choice and useful if you like to use Unix like commands like grep.
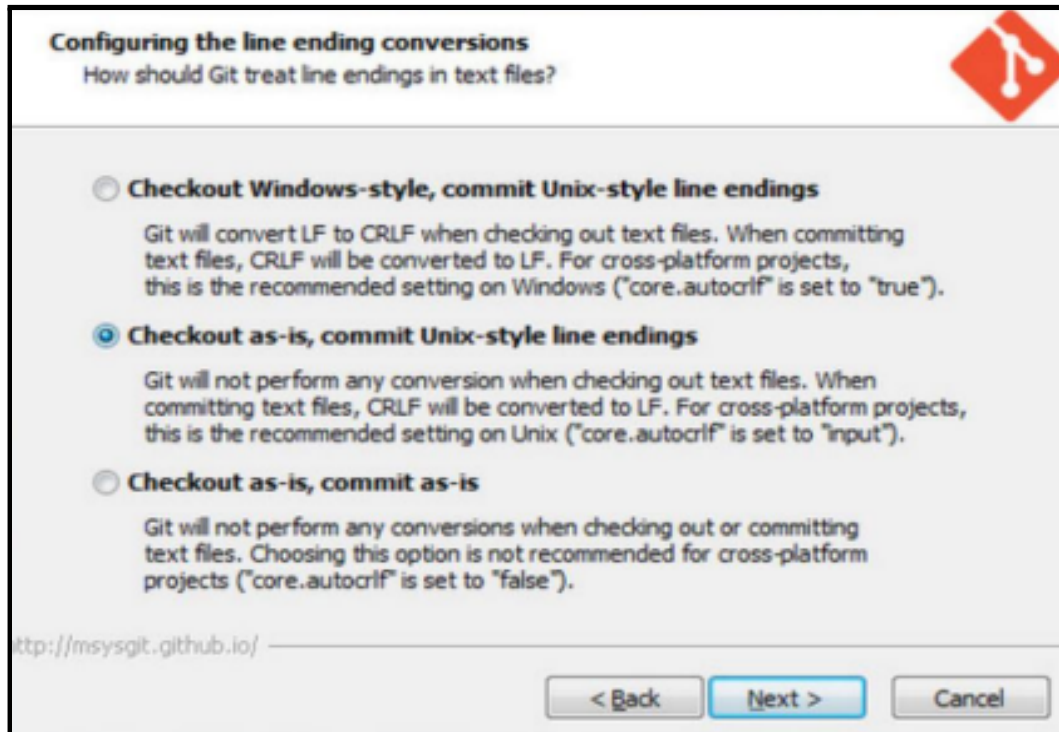
In the Configuring the line ending screen, select the middle option (Checkout-as-is, commit Unix-style line endings) as shown. This helps migrate files towards the Unix-style (LF) terminators that most modern IDE's and editors support. The Windows convention (CR-LF line termination) is only important for Notepad.

**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").

ttp://msysgit.github.io/

[< Back]  [Next >]  [Cancel]

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.d
b


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

Note: You can always edit this file and add additional patterns for other types of files you might want to ignore. Note that you can also have a .gitignore files in any folder naming additional files to ignore. This is useful for project specific build products. Once Git is installed, there is some remaining custom configuration you must do. Follow the steps below:

a. From within File Explorer, right-click on any folder. A context menu appears  containing the commands "Git Bash here" and "GitGUI 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 usesthisto 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.

**Aim:**
**Setting up GitHub Account**

The first steps in starting with GitHub are to create an account, choose a product that fits your needs best, verify your email, set up two-factor authentication, and view your profile. There are several types of accounts on GitHub. Every person who uses GitHub has their own user account, which can be part of multiple organisations and teams. Your user account is your identity on GitHub.com and represents you as an individual.

1. Creating an account: To sign up for an account on GitHub.com, navigate to https://github.com/ and follow the prompts. To keep your GitHub account secure you should use a strong and unique password.
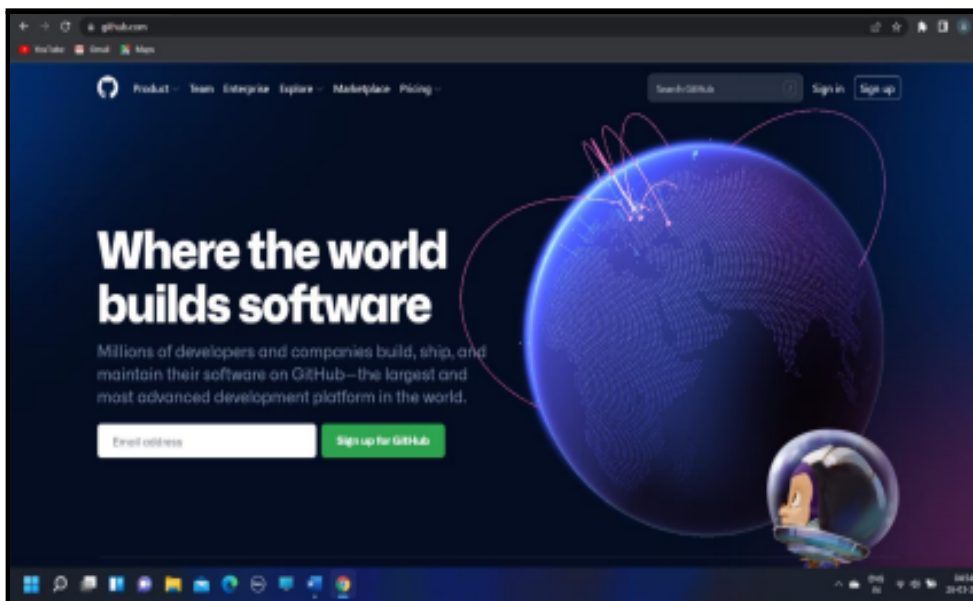


2. **Choosing your GitHub product:** You can choose GitHub Free or GitHub Pro to get access to different features for your personal account. You can upgrade at any time if you are unsure at first which product you want.

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.
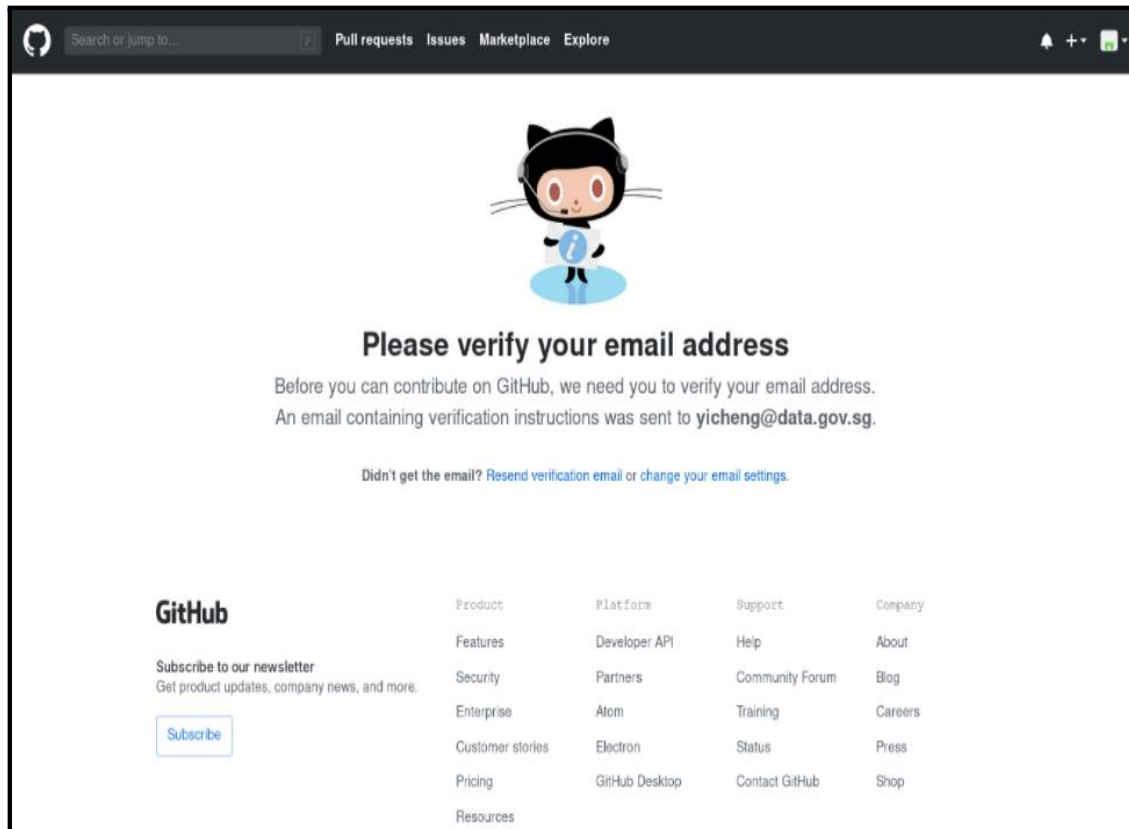
**4. Configuring two-factor authentication:** Two-factor authentication, or 2FA, is an extra  layer of security used when logging into websites or apps.
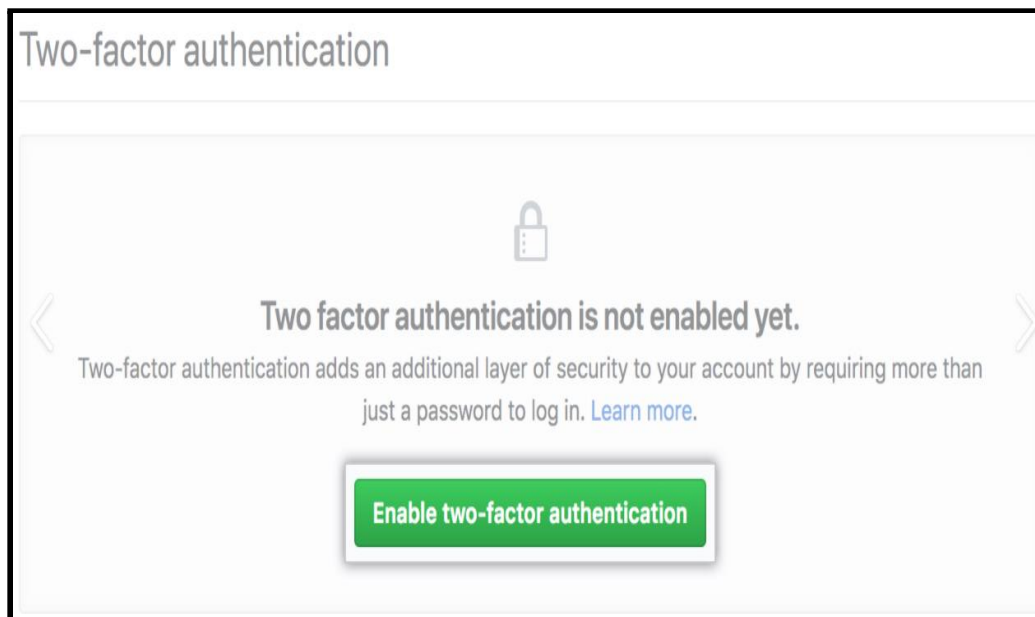
**5. Viewing your GitHub profile and contribution graph:** Your GitHub profile tells people the story of your work through the repositories and gists you've pinned, the organisation memberships you've chosen to publicize, the contributions you've made, and the projects you've created
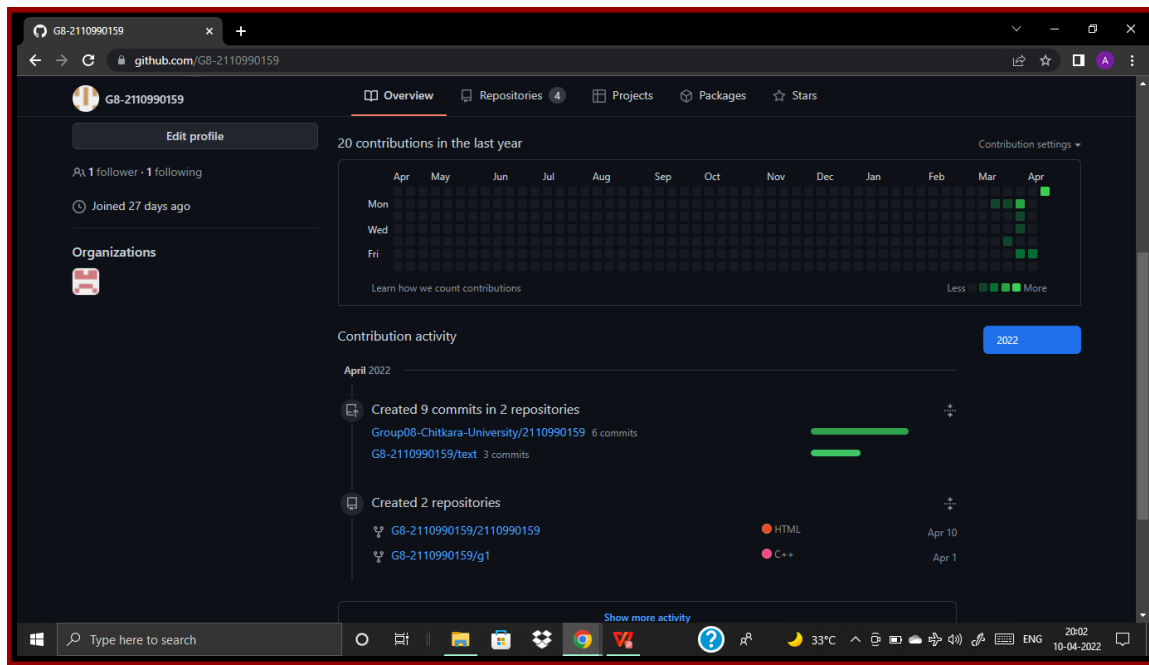
**Aim: Program to generate logs**

**Basic Git init :**

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

**Basic Git status :**

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

**Basic Git commit :**

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

**Basic Git add command :**

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

## Basic Git log

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

**Aim:**

 **Create and visualize branches in Git**

**How to create branches:**

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

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

**Visualizing Branches:**

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

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

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

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

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (master)
$ git branch activity1

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (master)
$ git checkout activity1
Switched to branch 'activity1'

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (activity1)
$ git status
On branch activity1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   New Text Document.txt

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

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (activity1)
$ git add --a

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (activity1)
$ git branch
* activity1
  master

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (activity1)
$ git commit -m"I am in actuvuty1"
[activity1 4f1e419] I am in actuvuty1
 1 file changed, 1 insertion(+)

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (activity1)
$ git log
commit 4f1e419fd817ee540ee496c853d59e6f0af3bd1b (HEAD -> activity1)
Author: ishayadav3499 <isha2064.be21@chitkara.edu.in>
Date:   Thu Mar 31 11:07:22 2022 -0700

    I am in actuvuty1

commit 8e52366825f4e80f872178825066ed0d6a108320 (master)
Author: ishayadav3499 <isha2064.be21@chitkara.edu.in>
Date:   Thu Mar 31 11:02:27 2022 -0700

    I am changing the New Text document

DELL@DESKTOP-JSI5681 MINGW64 ~/Desktop/SCM (activity1)
```

```
MINGW64:/c/Users/amar/Desktop
  293  pwd
  294  cd D:
  295  git clone https://github.com/MonitKapoor/g1
  296  ls
  297  ls g1
  298  git log
  299  cd g1
  300  pwd
  301  git log --oneline
  302  ls -ah
  303  cat first.cpp
  304  vi first.cpp
  305  cat first.cpp
  306  git status
  307  git add.
  308  git add .
  309  git status
  310  git commit -m "added headings to first.cpp"
  311  git log --oneline
  312  git remote
  313  git push -u origin master
  314  pwd
  315  cd D:
  316  pwd
```

**Aim: Git lifecycle description**

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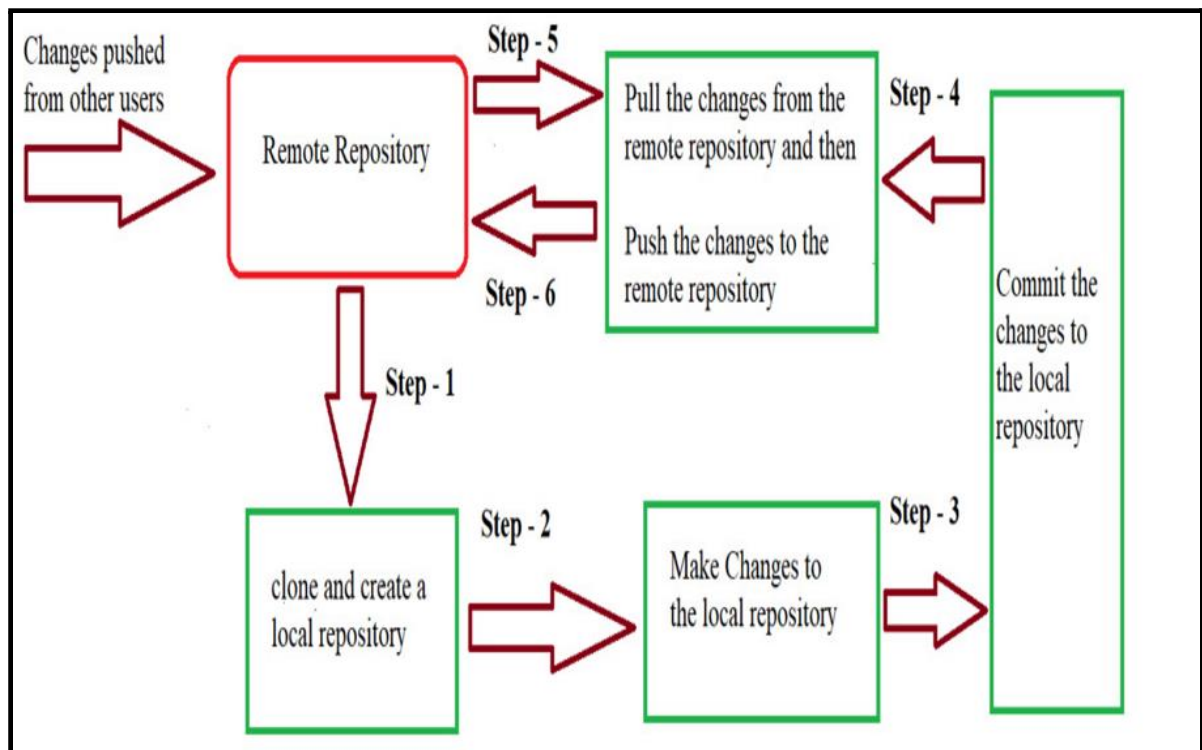


• **Step 1**- We first clone any of the code residing in the remote repository to make our won local repository.

• **Step 2**- We edit the files that we have cloned in our local repository and make the necessary changes in it.

• **Step 3**- We commit our changes by first adding them to our staging area and committing them with a commit message.

• **Step 4 and Step 5-** We first check whether there are any of the changes done in the  remote repository by some other users and we first pull that changes.

• **Step 6-** If there are no changes we push our changes to the remote repository and we  are done with our work.

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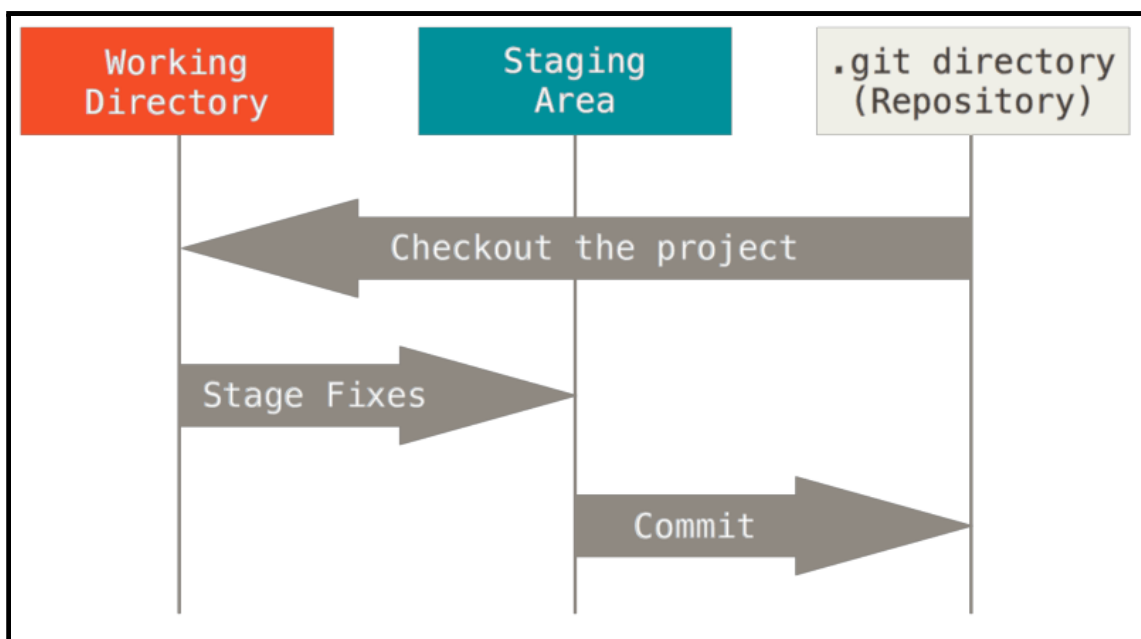


### 1. Working Directory

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

### 2. Staging Area

Now, to track files the different versions of our files we use the command git add. git  add command copies the version of your file from your working directory to the staging  area. We can, however, choose which files we need to add to the staging area because  in our working directory there are some files that we don't want to get tracked,   examples include node modules, temporary files, etc.

Indexing in Git is the one that helps Git in understanding which files need to be added or sent. You can find your staging area in the .git folder inside the index file. git add<filename> git add.

## 3. Git Directory:

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