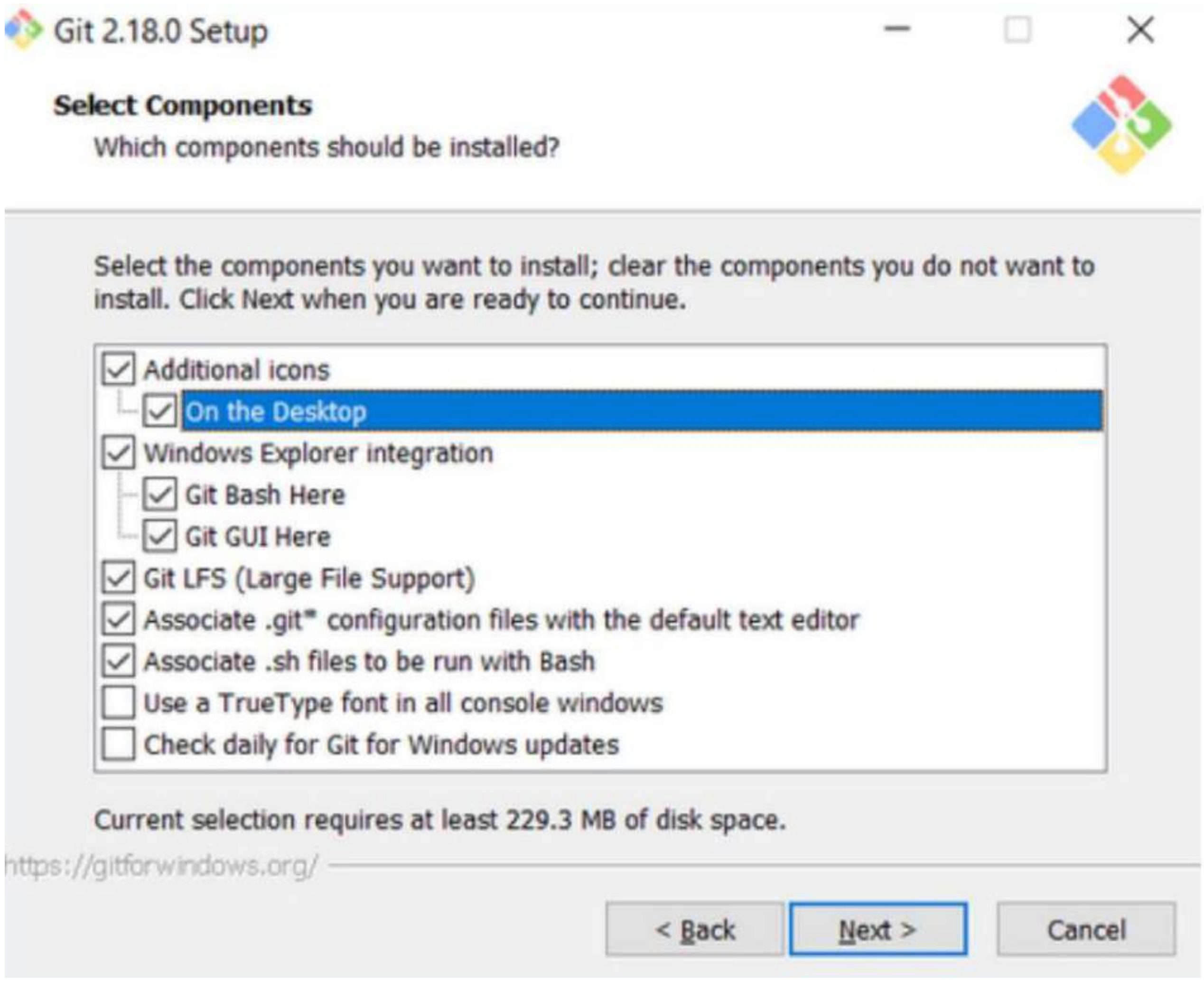
Experiment No. 01

Aim: Setting up the git client.

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

At first, a dialogue box will appear which will ask for the permission to make changes on your computer and will be required to click on yes. After that the installer will start but before that a few more dialogue boxes will appear, 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 to select the most preferable option as shown:



Git 2.18.0 Setup

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
- [ ] Use a TrueType font in all console windows
- [ ] Check daily for Git for Windows updates

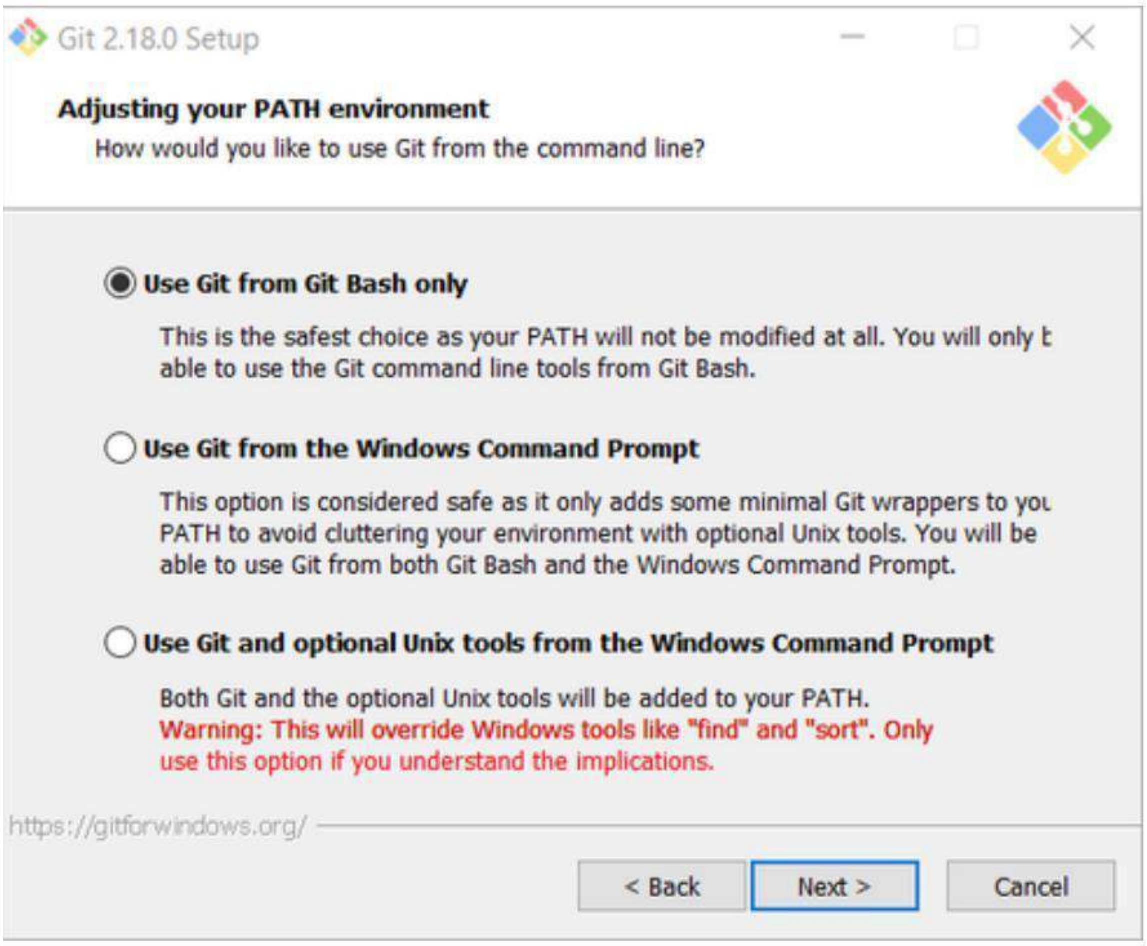
Current selection requires at least 229.3 MB of disk space.

https://gitforwindows.org/

In the choosing the default editor is used by Git dialog, it is strongly recommended that you DO NOT select the default VIM editor. It is better to use Notepad++ or Nano.

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.


Git 2.18.0 Setup

Adjusting your PATH environment

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

(●) **Use Git from Git Bash only**

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

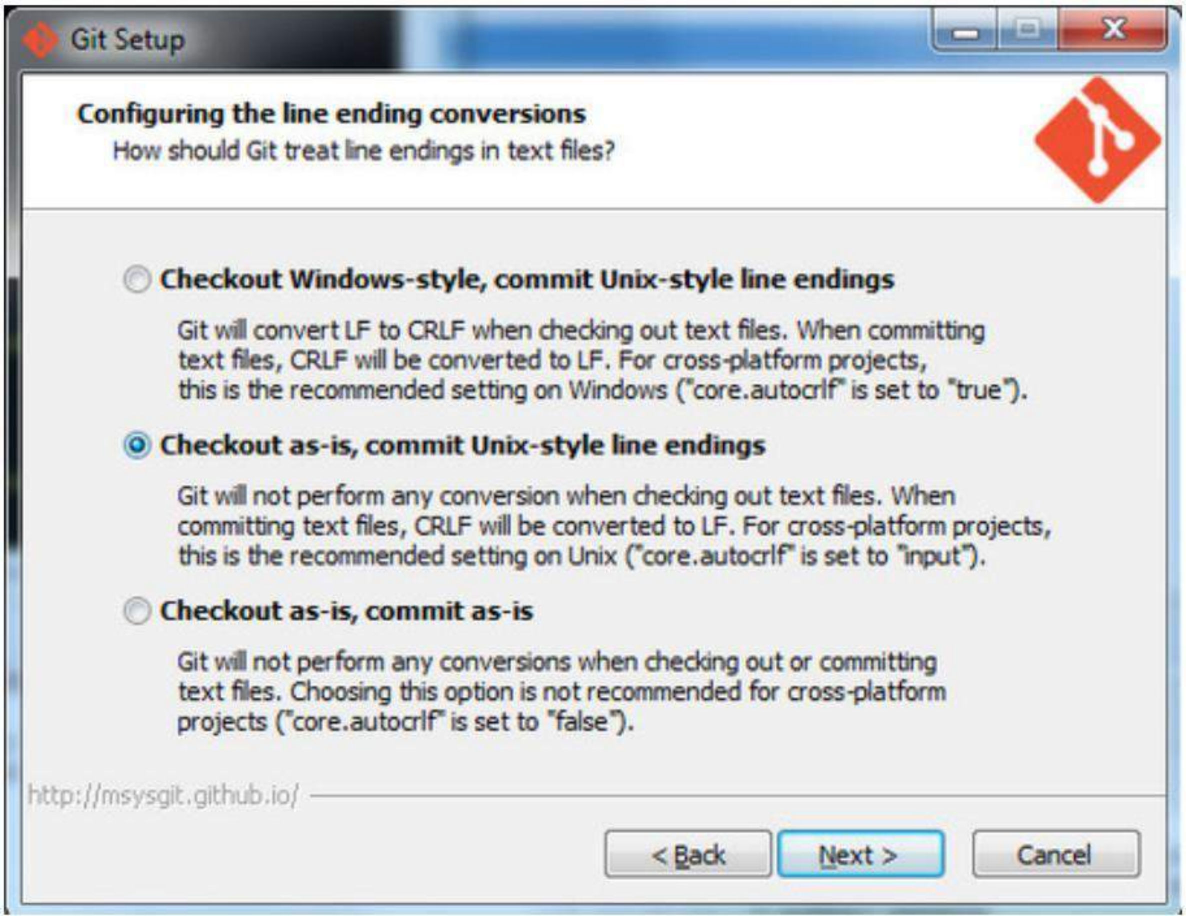
( ) **Use Git from the Windows Command Prompt**

This option is considered safe as it only adds some minimal Git wrappers to you PATH to avoid cluttering your environment with optional Unix tools. You will be able to use Git from both Git Bash and the Windows Command Prompt.

( ) **Use Git and optional Unix tools from the Windows Command Prompt**

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

In the Configure 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 editor's 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").

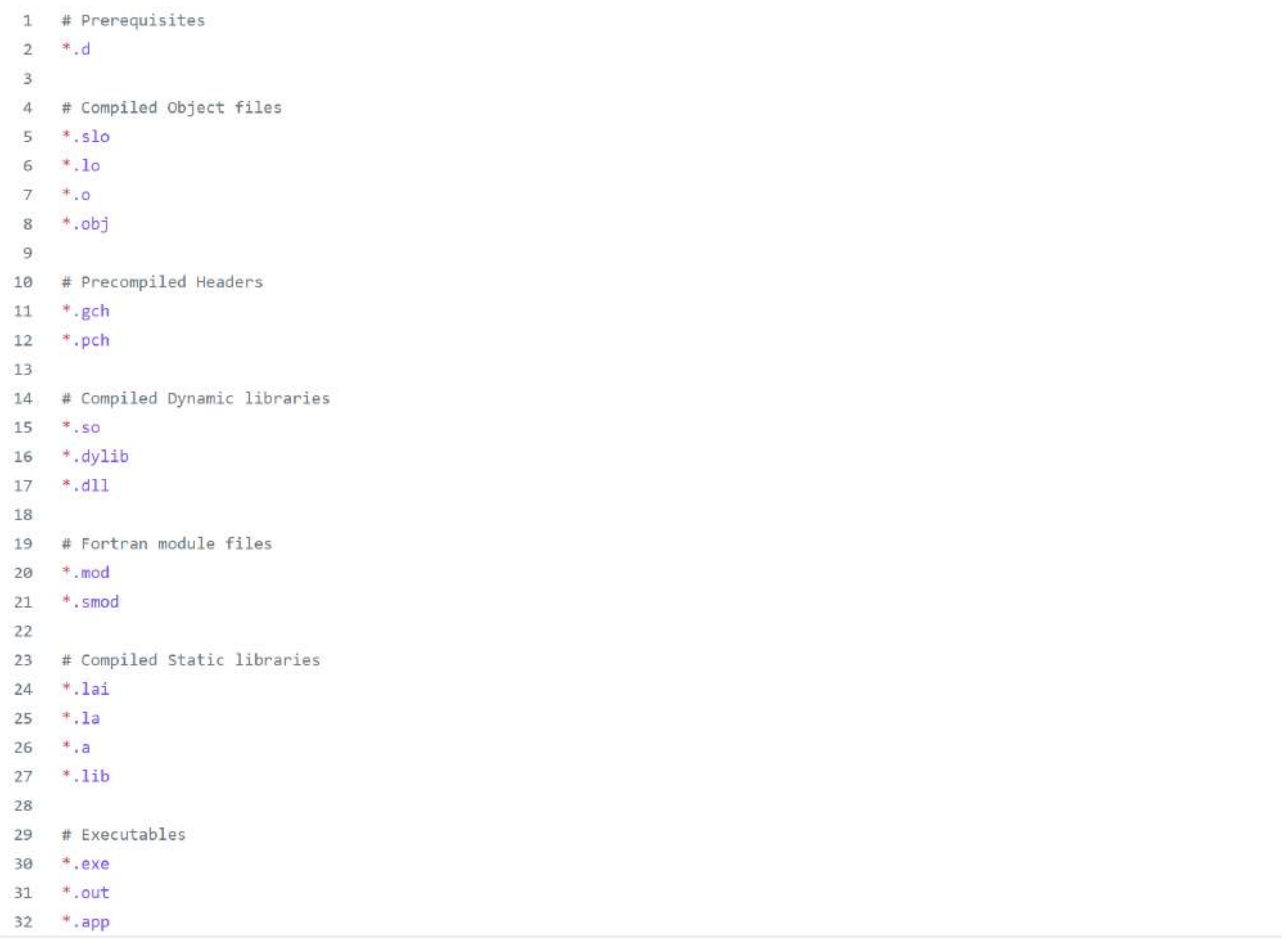
http://msysgit.github.io/

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 but you want Git to ignore certain files such as .exe 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).

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.)



```
# Prerequisites
*.d

# Compiled Object files
*.slo
*.lo
*.o
*.obj

# Precompiled Headers
*.gch
*.pch

# Compiled Dynamic libraries
*.so
*.dylib
*.dll

# Fortran module files
*.mod
*.smod

# Compiled Static libraries
*.lai
*.la
*.a
*.lib

# Executables
*.exe
*.out
*.app
```

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. (Where name is your MOSE as login name)

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

c. Enter the command `git config --global user. Email "name@mose.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 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.

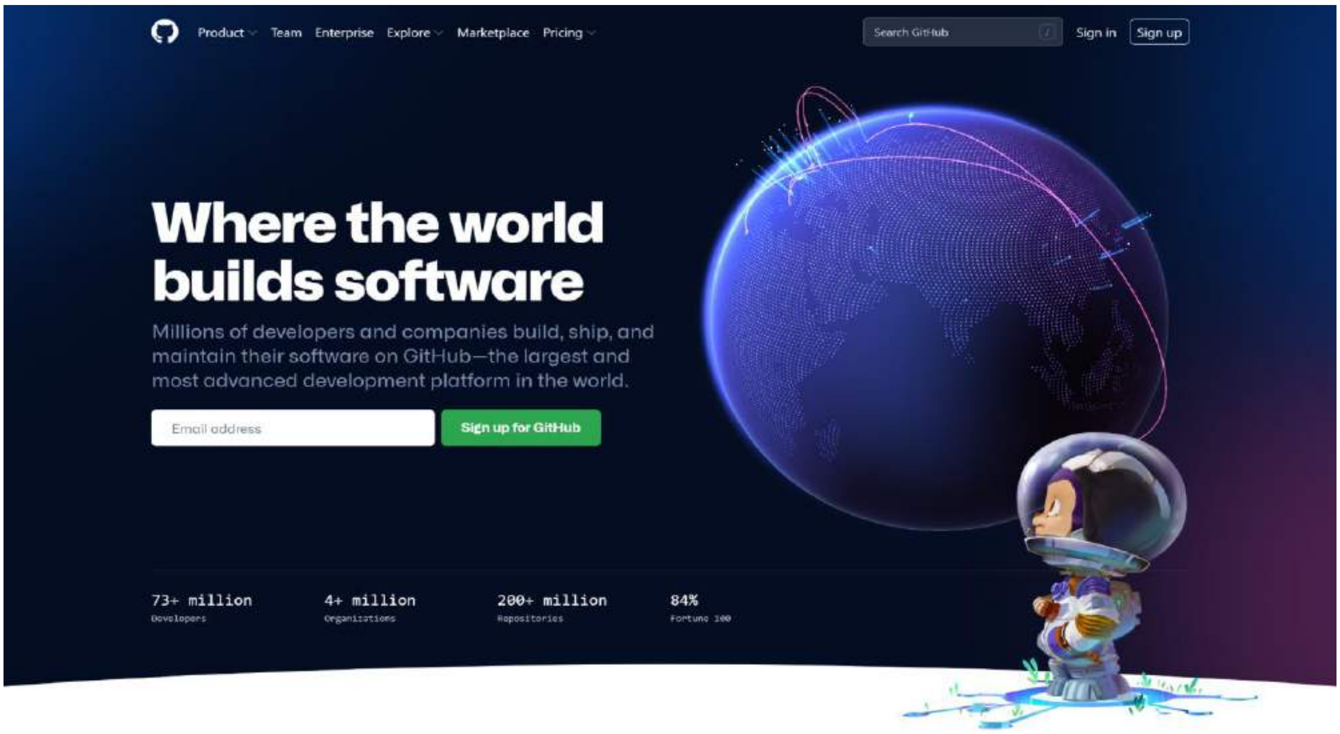
Experiment No. 02

Aim: Setting up GitHub Account

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

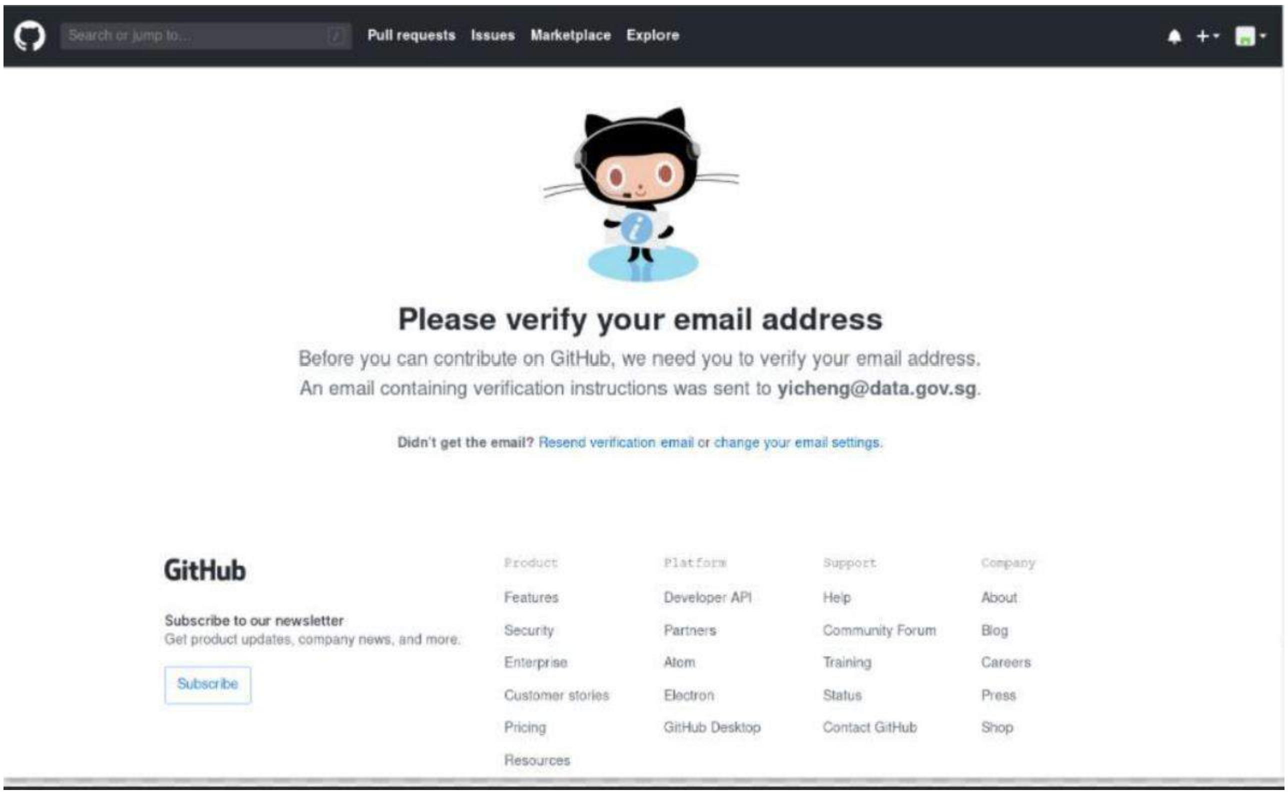
There are several types of accounts on GitHub. Every person has their own user name, which can be part of multiple organizations and teams. Your user account is your identity on GitHub.com and represents you as an individual.

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



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 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".

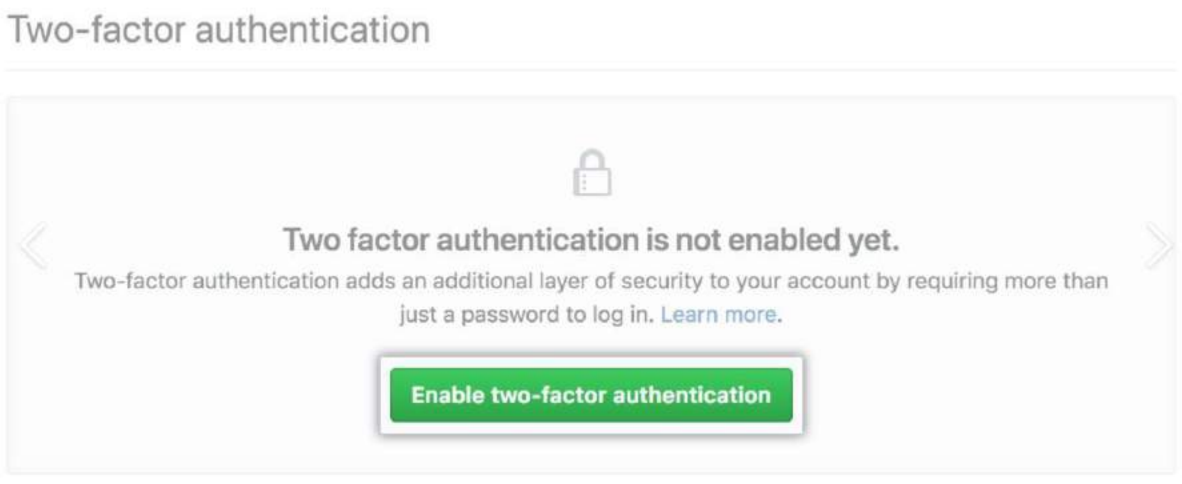


Please verify your email address

Before you can contribute on GitHub, we need you to verify your email address.
An email containing verification instructions was sent to **yicheng@data.gov.sg**.

Didn't get the email? Resend verification email or change your email settings.

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."

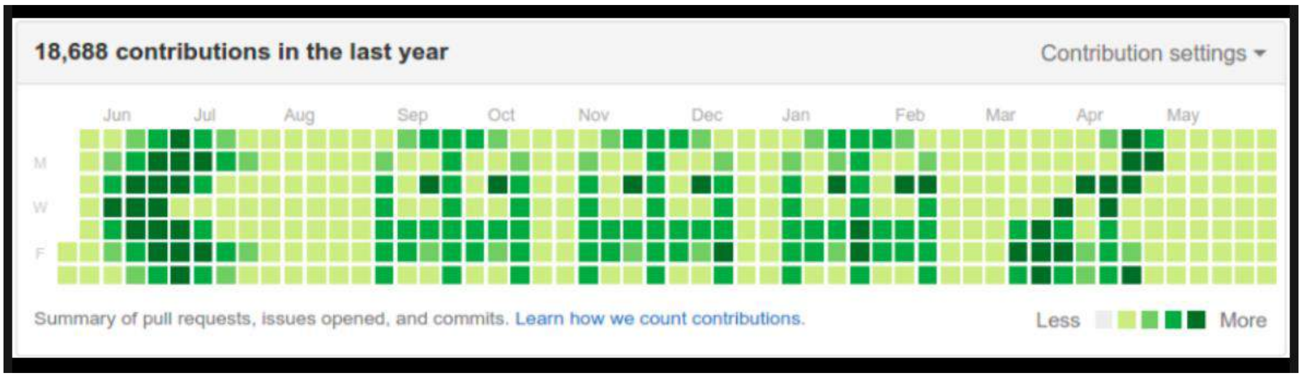


Two-factor authentication

Two factor authentication is not enabled yet.

Two-factor authentication adds an additional layer of security to your account by requiring more than just a password to log in. Learn more.

5.Viewing your GitHub profile and contribution graph: Your GitHub profile tells what you have done or have made changes to which repositories and gits you've pinned, the organization memberships you've chosen to publicize, the contributions and the projects you've made. For more information, see "Viewing contributions on your profile."


18,688 contributions in the last year

Summary of pull requests, issues opened, and commits. Learn how we count contributions.

Experiment No. 03

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 initialized repository, so this is usually the first command you'll run in a new project.

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:

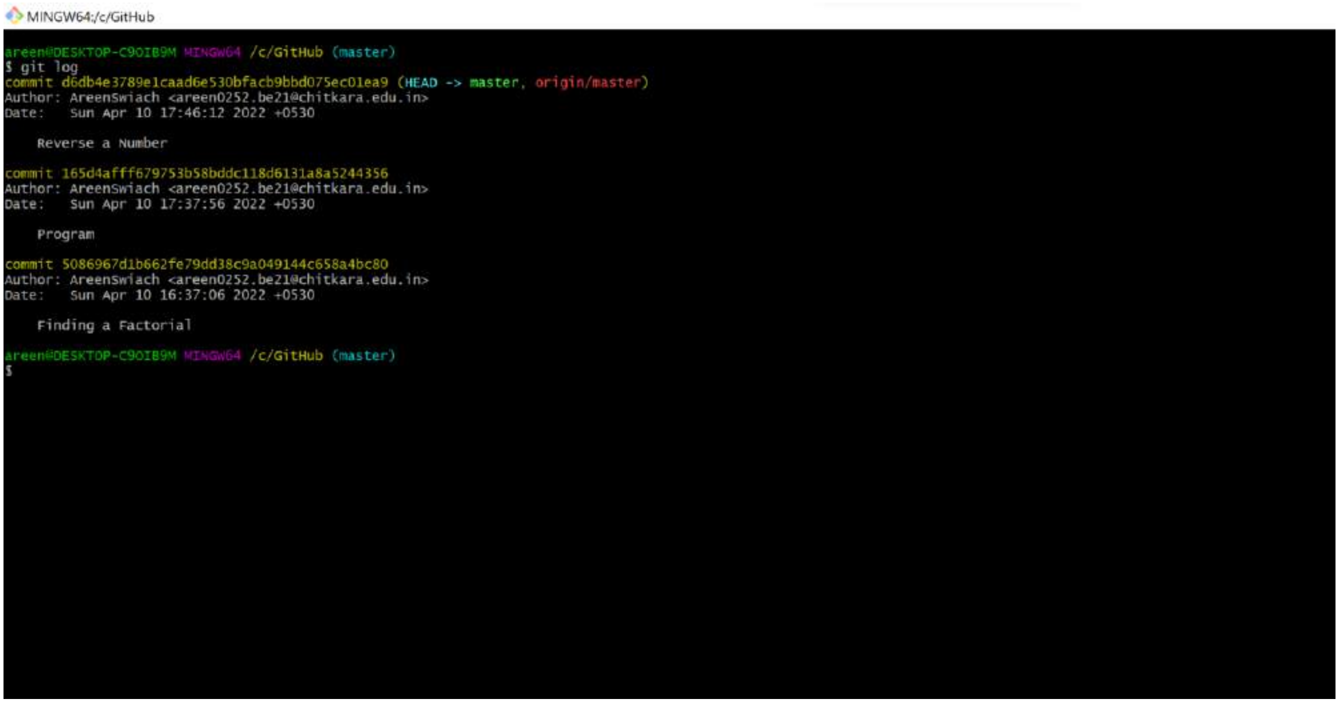
The git commit command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as "safe" versions of a project -Git will never change them unless you explicitly ask it to.

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 common commands of git. It is the most useful command for Git. Every time you need to check the history of commits, 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:



```
areen@DESKTOP-C9OIB9M MINGW64 /c/GitHub (master)
$ git log
commit d6db4e3789e1caad6e530bfacb9bbd075ec01ea9 (HEAD -> master, origin/master)
Author: AreenSwiach <areen0252.be21@chitkara.edu.in>
Date:   Sun Apr 10 17:46:12 2022 +0530

    Reverse a Number

commit 165d4afff679753b58bddc118d6131a8a5244356
Author: AreenSwiach <areen0252.be21@chitkara.edu.in>
Date:   Sun Apr 10 17:37:56 2022 +0530

    Program

commit 5086967d1b662fe79dd38c9a049144c658a4bc80
Author: AreenSwiach <areen0252.be21@chitkara.edu.in>
Date:   Sun Apr 10 16:37:06 2022 +0530

    Finding a Factorial

areen@DESKTOP-C9OIB9M MINGW64 /c/GitHub (master)
$
```

Experiment No. 04

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"

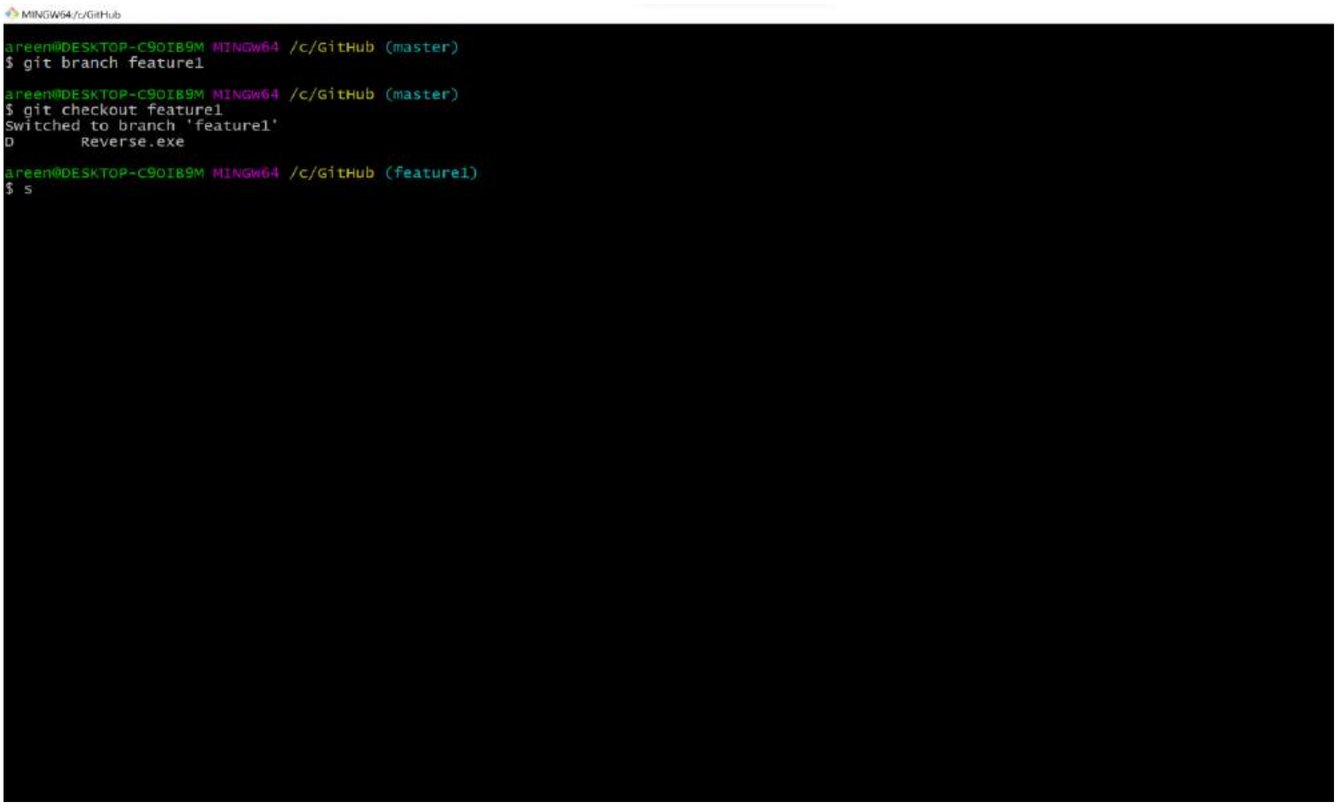
Visualizing Branches:

To visualize, we have to create a new file in the new branch "feature1" 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, sending it to the staging area and finally we can rollback to any previously saved version of this file.

After this we will change the branch from feature1 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 different branches by using the git merge command.

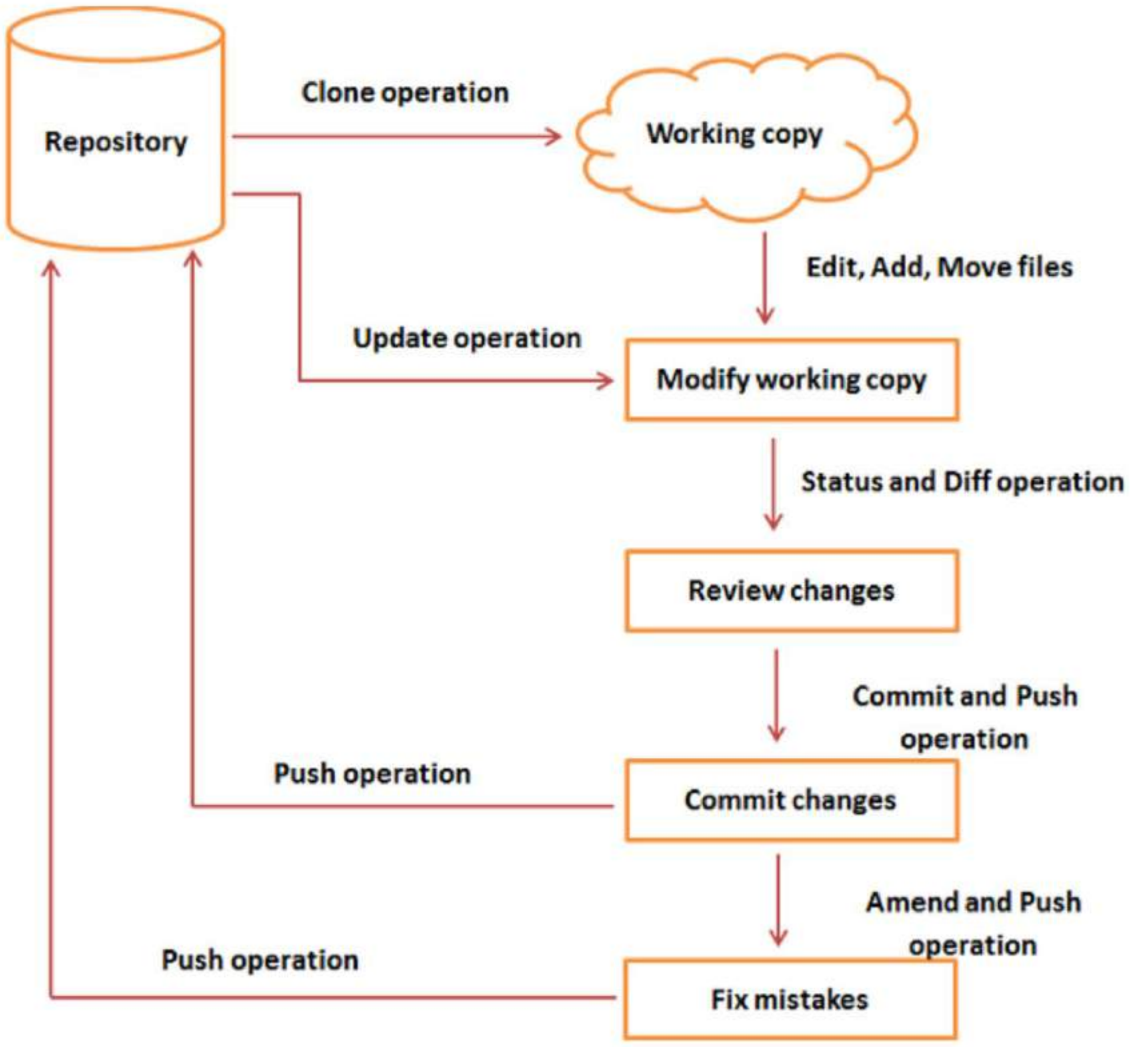

MINGW64:/c/GitHub

```
areen@DESKTOP-C9OIB9M MINGW64 /c/GitHub (master)
$ git branch feature1

areen@DESKTOP-C9OIB9M MINGW64 /c/GitHub (master)
$ git checkout feature1
Switched to branch 'feature1'
D       Reverse.exe

areen@DESKTOP-C9OIB9M MINGW64 /c/GitHub (feature1)
$ s
```

Experiment No. 05

Aim: Git lifecycle description

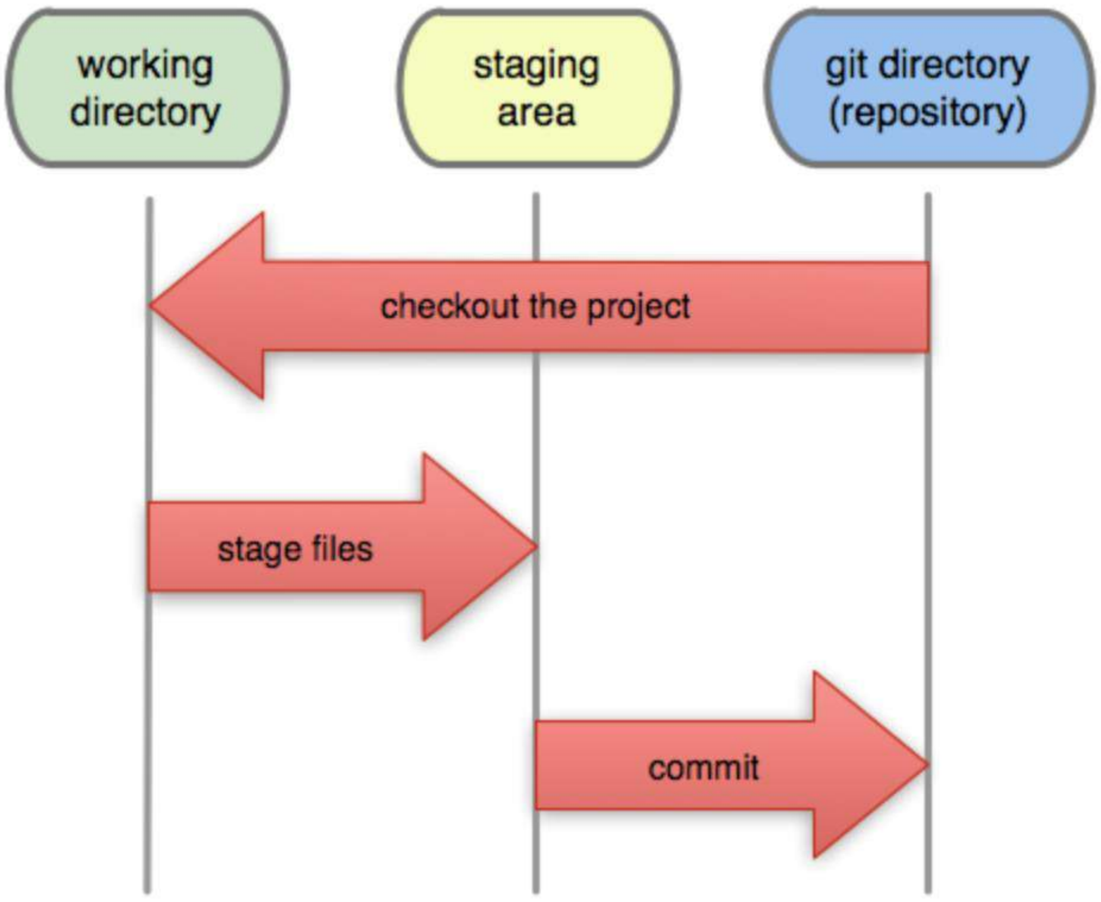
Git is used in our day-to-day work, we use Git for keeping a track of our files, working in a collaboration with our team, to go back to our previous code versions if we face some error. Git helps us in many ways. Let us look at the Lifecycle description that git has and understand more about its life 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 own 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 those 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 the Git version Control System. The three states are-



1. **Working Directory:**

Whenever we want to initialize the 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. 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, 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 the 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>. Then the git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repository. Then, it uploads or we can also say it is added to our account on GitHub.