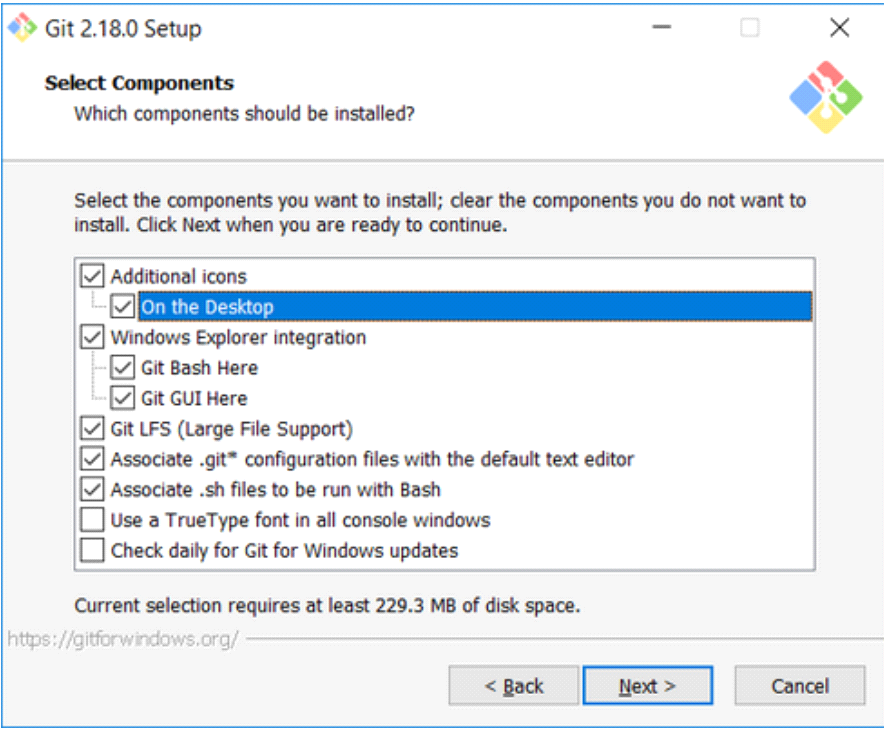


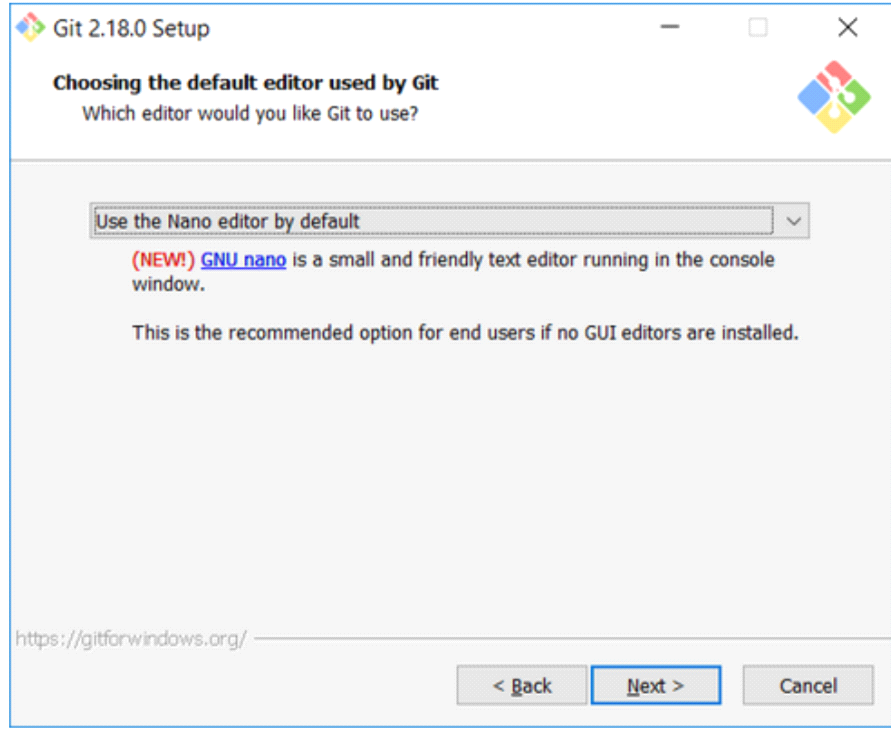
**Aim:** Setting up the git client.Git Installation: Download the Git installation program (Windows, Mac, or Linux) from gitgigit

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:

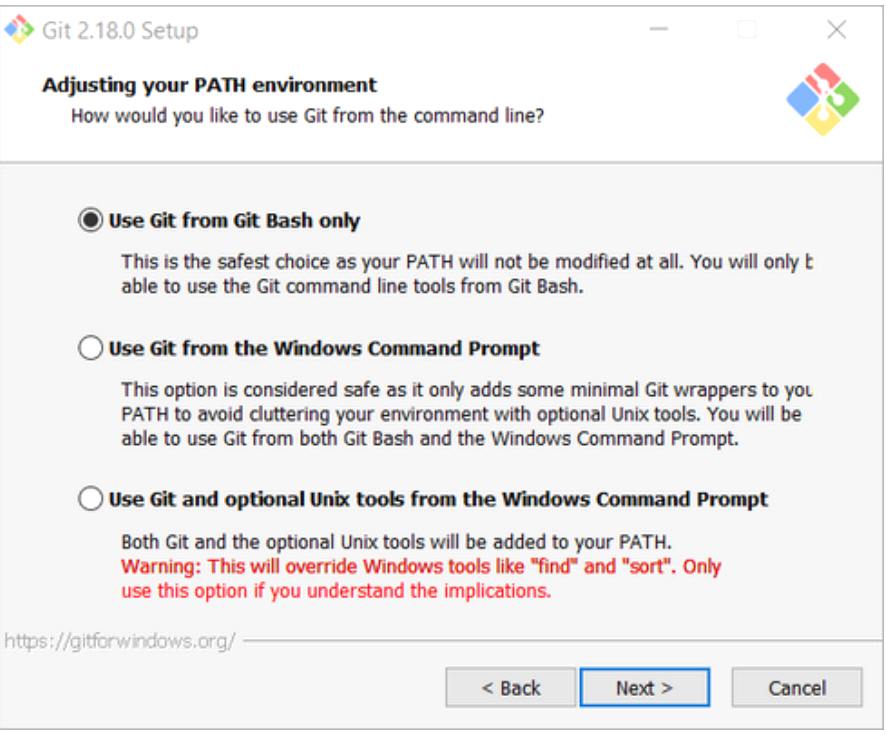


In the choosing the default editor is used by Git dialog, it is strongly recommended that you DO NOT select default VIM editor- it is challenging to learn how to use it, and there are better modern editors available. Instead, choose Notepad++ or Nano – either of those is much easier to use. It is strongly recommended that you select Notepad++.

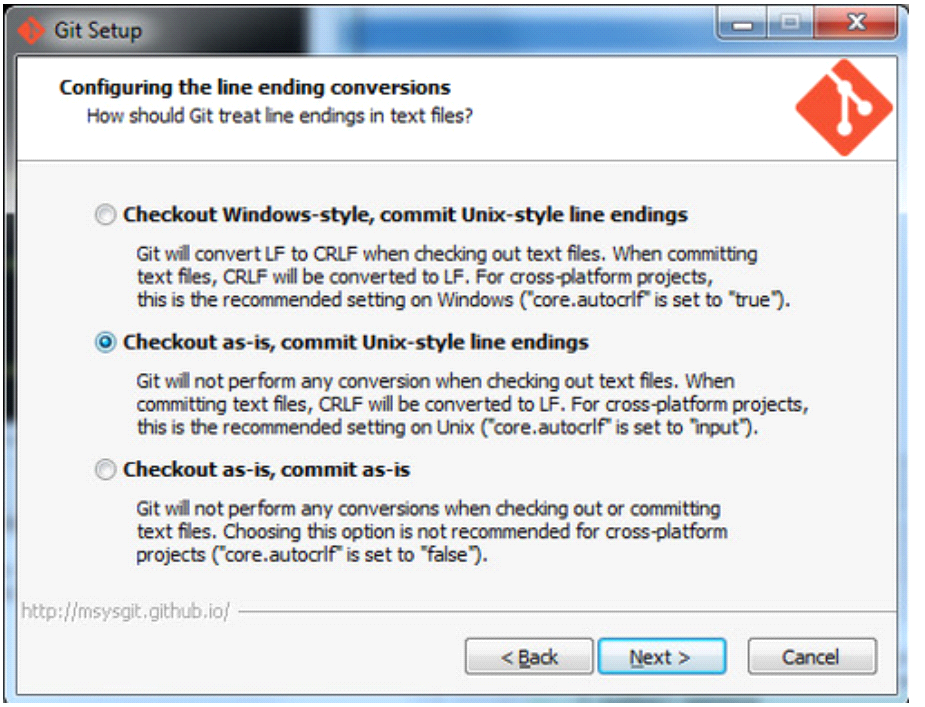


In the Adjusting your PATH screen, all three options are acceptable:

* Use Git from Git Bash only: no integration, and no extra command in your command path.
* 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.
* 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 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 allfiles in a project. Typically, this is notwhat you want; rather, you want Git to ignore certain files such as .bakfiles created by an editor or .classfiles 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\namefolder (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>.)

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:

* 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.
* Enter the command (replacing name as appropriate) git config --global core.excludesfile c:/users/name/. gitignore

This tells Git to use the .gitignorefile 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.

* Enter the command git config --global user.Email "[name@msoe.edu](mailto: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.



**Aim**

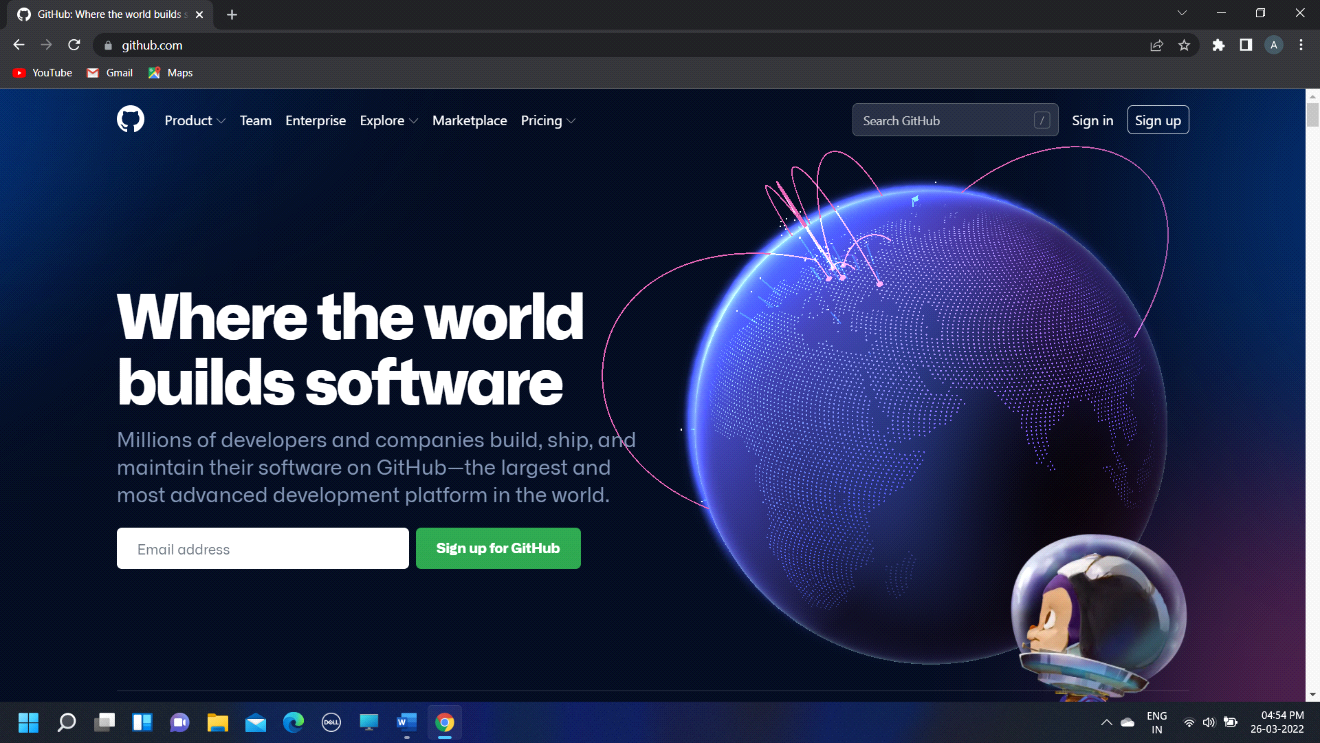
Setting up GitHub Account

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.

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

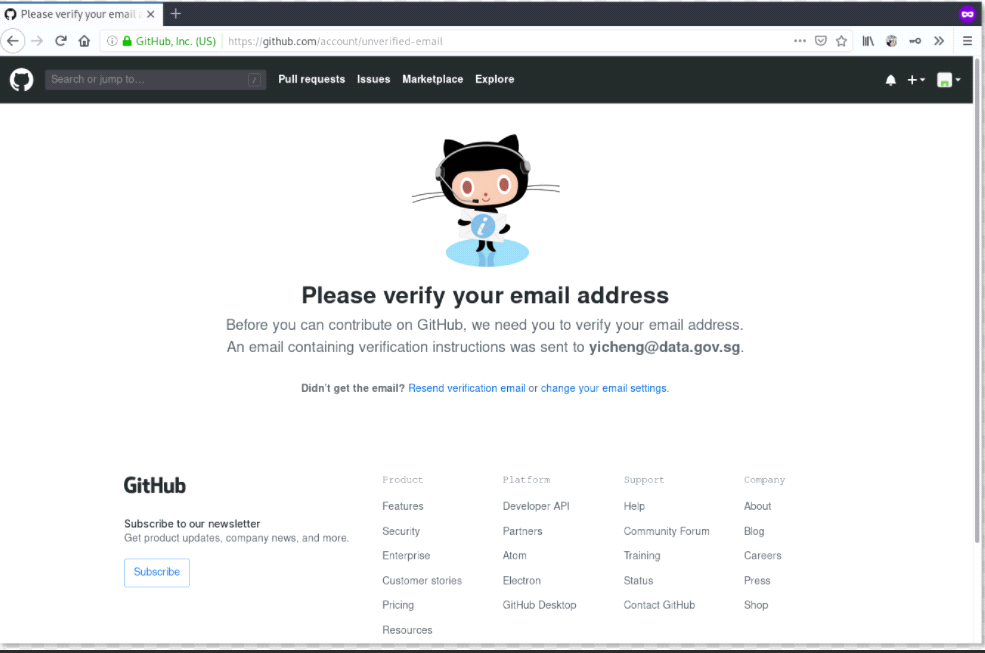
To keep your GitHub account secure you should use a strong and unique password. For more information, see “[Creating a strong password](https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-strong-password)”.



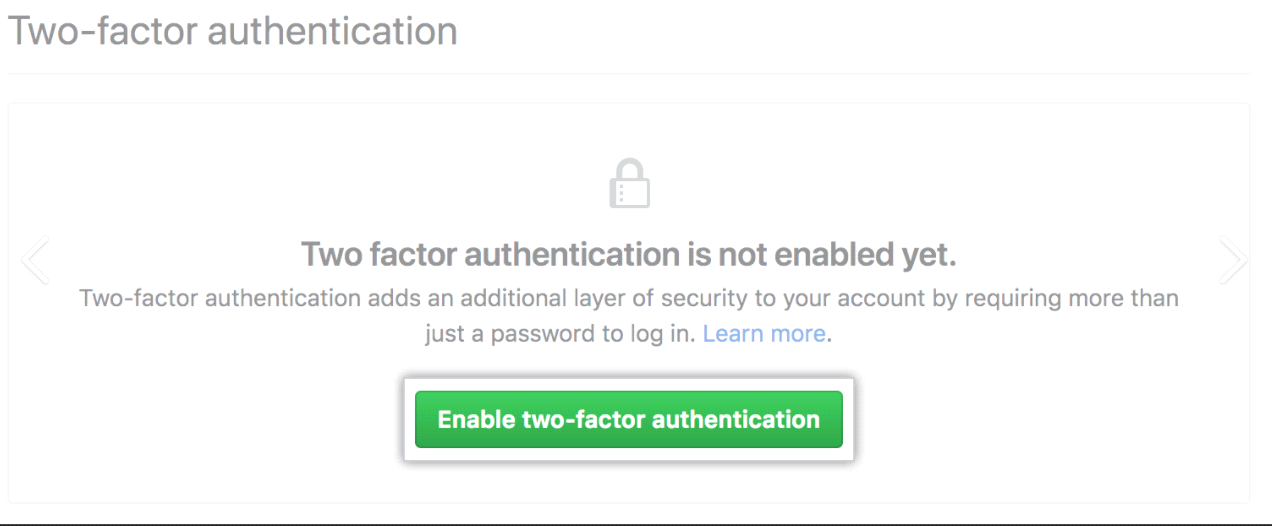
* **Choosing your GitHub product:** You can choose GitHub Free or GitHub Pro to get access to different features for your personal account. You can upgrade at any time if you are unsure at first which product you want.

For more information on all GitHub’s plans, see “[GitHub's products](https://docs.github.com/en/get-started/learning-about-github/githubs-products)”.

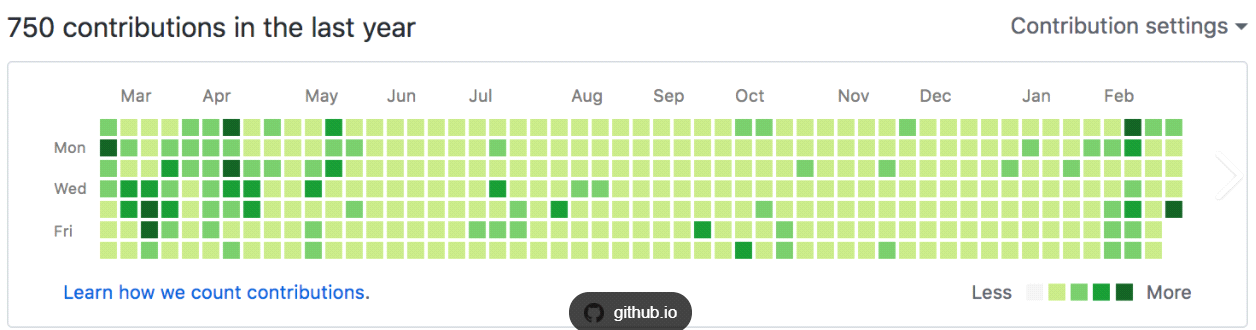
* **Verifying your email address:** To ensure you can use all the features in your GitHub plan, verify your email address after signing up for a new account. For more information, see “[Verifying your email address](https://docs.github.com/en/github/getting-started-with-github/signing-up-for-github/verifying-your-email-address)”.



* **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 safety of your account. For more information, see “[About two-factor authentication](https://docs.github.com/en/github/authenticating-to-github/securing-your-account-with-two-factor-authentication-2fa/about-two-factor-authentication)."



* **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. For more information, see “[About your profile](https://docs.github.com/en/github/setting-up-and-managing-your-github-profile/customizing-your-profile/about-your-profile)” and “[Viewing contributions on your profile](https://docs.github.com/en/github/setting-up-and-managing-your-github-profile/managing-contribution-graphs-on-your-profile/viewing-contributions-on-your-profile).”



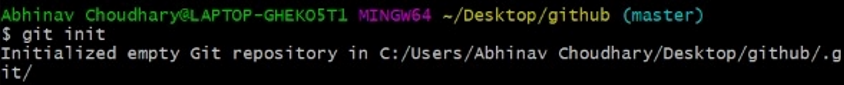


**Aim:** Program to generate logs

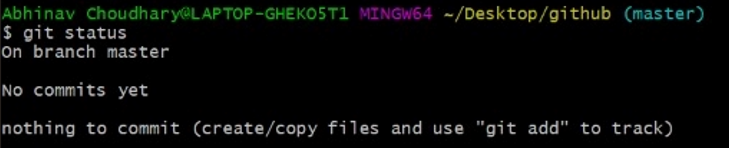
Basic Git init

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

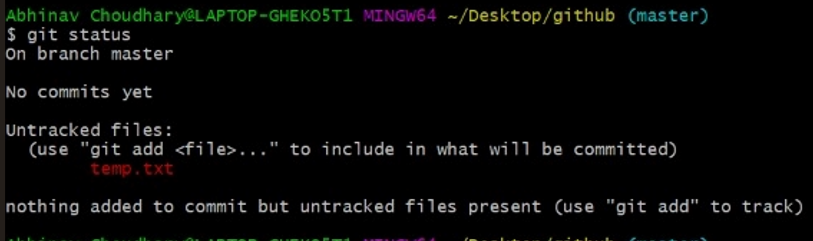


gtracked by Git. Status output does not show you any information regarding the committed project history.



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](https://www.atlassian.com/git/tutorials/saving-changes) 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



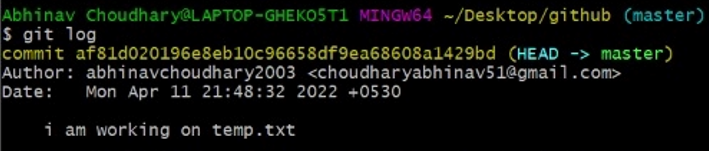
Basic Git add command

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

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.

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](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) and [git merge](https://www.atlassian.com/git/tutorials/using-branches/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 crazy-experiment 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

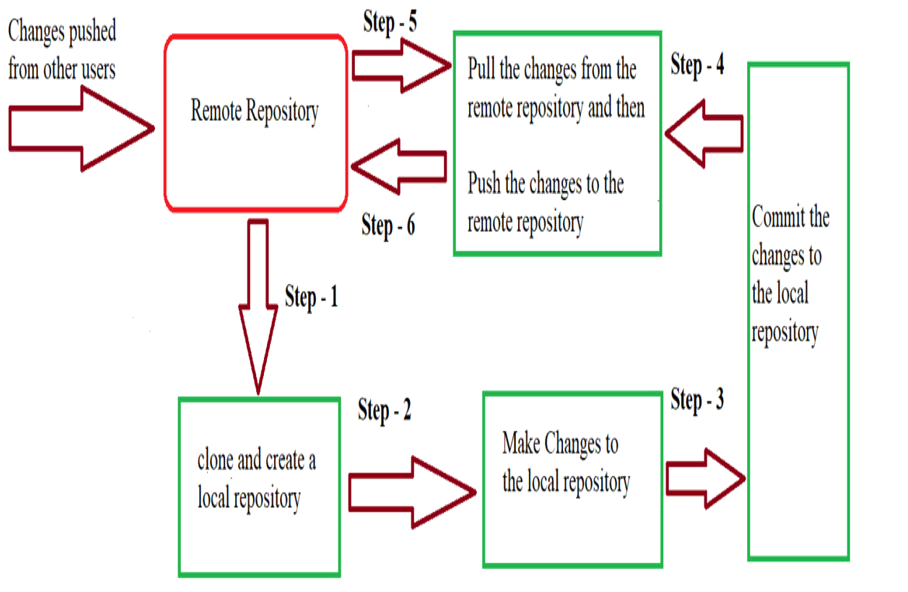
git push origin :crazy-experiment

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



**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-

* **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.

* **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.

* **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>