# Version Control

Version Control is the management of changes to documents, computer programs, large websites and other collection of information.

There are two types of VCs:

1. Centralized Version Control System (CVCS)
2. Distributed Version Control System (DVCS)

## Centralized VCS

Centralized version control system uses a central server to store all files and enables team collaboration. It works on a single repository to which users can directly access a central server.



Figure Centralized Version Control System

There is only one centralized repository connected to each of the programmer’s workstation.

Every programmer can extract or update their workstation with the data present in the repository or can make changes to the data or commit in the repository Every operation performed directly on centralized repository.

**Drawbacks:**

1. It is not locally available; we always need to connect to a network to perform any action.
2. Since everything is centralized, in any case of the central server getting crashed or corrupted will result in losing the entire data of the project.

## Distributed VCS

These systems do not rely on a central server to store all the versions of a project file.

In Distributed VCS, every contributor has a local copy or clone of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository.



Figure Distributed Version Control Systems

As shown in figure 2, every programmer maintains a local repository on its own which is actually the copy or clone of the central repository on their hard drive. They can commit and update their local repository without any interference.

Developers can perform below two operations:

1. Update their local repositories with new data from the central server by **pull** operations.
2. Commit or affect changes to the main repository by **push operations.**

The cloning of entire repository into local workstation has below advantages:

1. All operations (except pull and push) are very fast because the tool only needs to access the hard drive, not the remote server. Hence there is no need of active internet connection.
2. Committing new change-sets can be done locally without manipulating the data on the main repository. Once group of change-sets ready, we can push them all at once.
3. Every contributor has a full copy of the project repository, they can share changes with one another if they want to get some feedback before affecting changes in the main repository.
4. If the central server gets crashed at any point of time, the lost data can be easily recovered from any of the contributor’s machine

# GIT

1. Git is a free, open source distributed version control system tool designed to handle everything from small to very large projects with speed and efficiency.
2. It was created by Linus Torvalds in 2005 to develop Linux Kernel.
3. Git has the functionality, performance, security and flexibility that most teams and individual developers need.

## Features of GIT

1. **Free and Open Source**
2. **Speed**

* No need to connect to any network for performing operation

1. **Scalable**

* Git is very scalable. So, if in future , the number of collaborators increase Git can easily handle this change. Though Git represents an entire repository, the data stored on the client’s side is very small as Git compresses all the huge data through a lossless compression technique.

1. **Reliable**: Since every contributor has its own local repository , on the events of a system crash, lost data can be recovered from any of the local repositories. We have backup of all the files at every point of time.
2. **Secure:**

* GIT uses SHA1(Secure Hash Functions) to name and identify objects within its repository
* The Git history is stored in such a way that the ID of a particular version (a *commit*in Git terms) depends upon the complete development history leading up to that commit.
* Once changes are published , it is not possible to change the version number without being noticed.

1. **Economical:**

* In case of CVCS, the central server needs to be powerful enough to serve requests of the  entire team. For smaller teams, it is not an issue, but as the team size grows, the hardware  limitations of the server can be a performance bottleneck.
* In case of DVCS, developers don’t interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side, so the server hardware can be very simple indeed.

1. **Support non linear development**

* Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history.
* Branches in Git are very lightweight. A branch in Git is only a reference to a single commit. With its parental commits, the full branch structure can be constructed.

1. **Easy Branching**

* Branch management with Git is very simple. It takes only few seconds to create, delete, and merge branches. Feature branches provide an isolated environment for every change to your codebase.
* When a developer wants to start working on something, no matter how big or small, they create a new branch. This ensures that the master branch always contains production-quality code.

1. **Distributed Development**

* Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another.
* These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch.

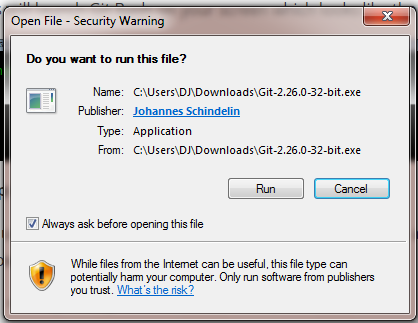
1. **Compatibility with exisiting systems/protocols**

* Repositories can be published via http, ftp or a Git protocol over either a plain socket, or ssh. Git also has a Concurrent Version Systems (CVS) server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories. Apache SubVersion (SVN) and SVK repositories can be used directly with Git-SVN.

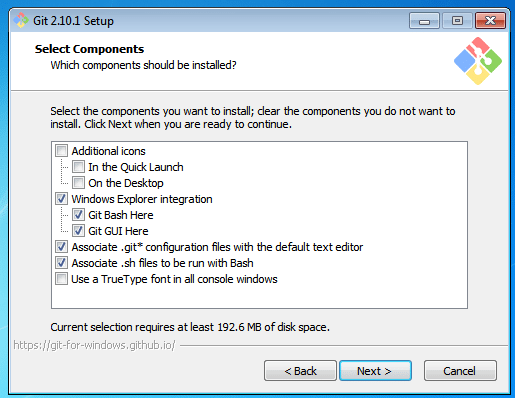
To download the GIT

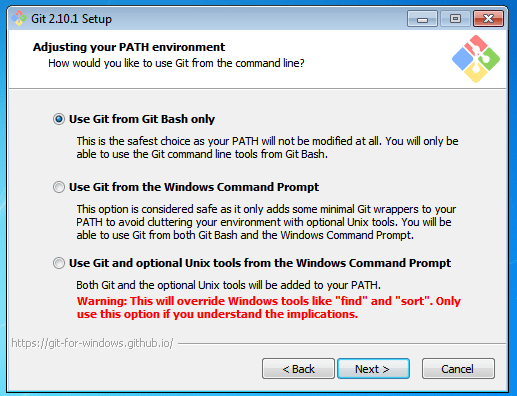
<https://git-scm.com/download/win/>

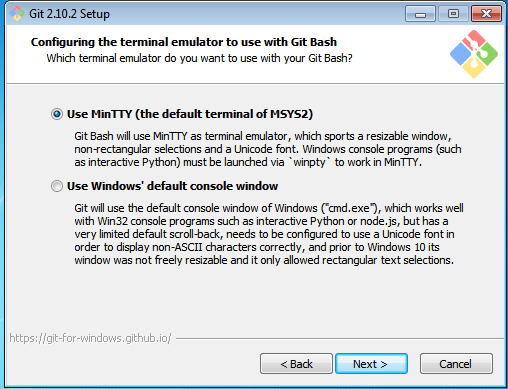
1. Click on the downloaded file

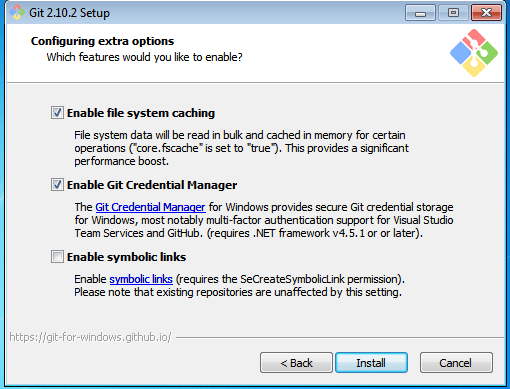


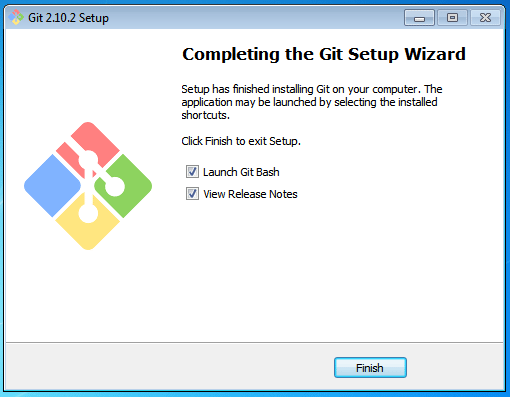
1. Click on run



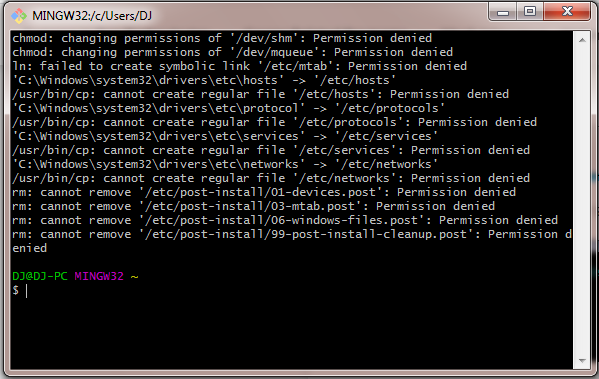








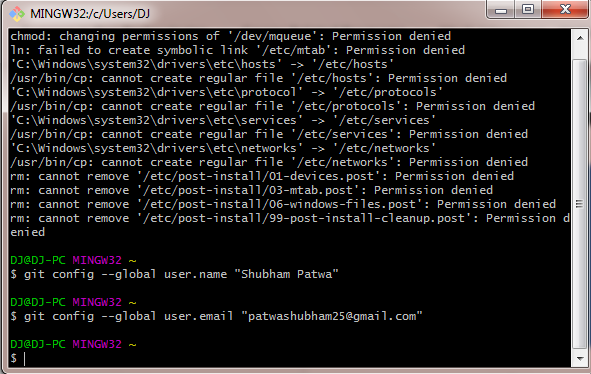
This will launch the Git Bash screen once we click on finish button:



Let us proceed with configuring Git with your username and email. In order to do that, type the following commands in your Git Bash:

git config - - global user.name "<your name>"

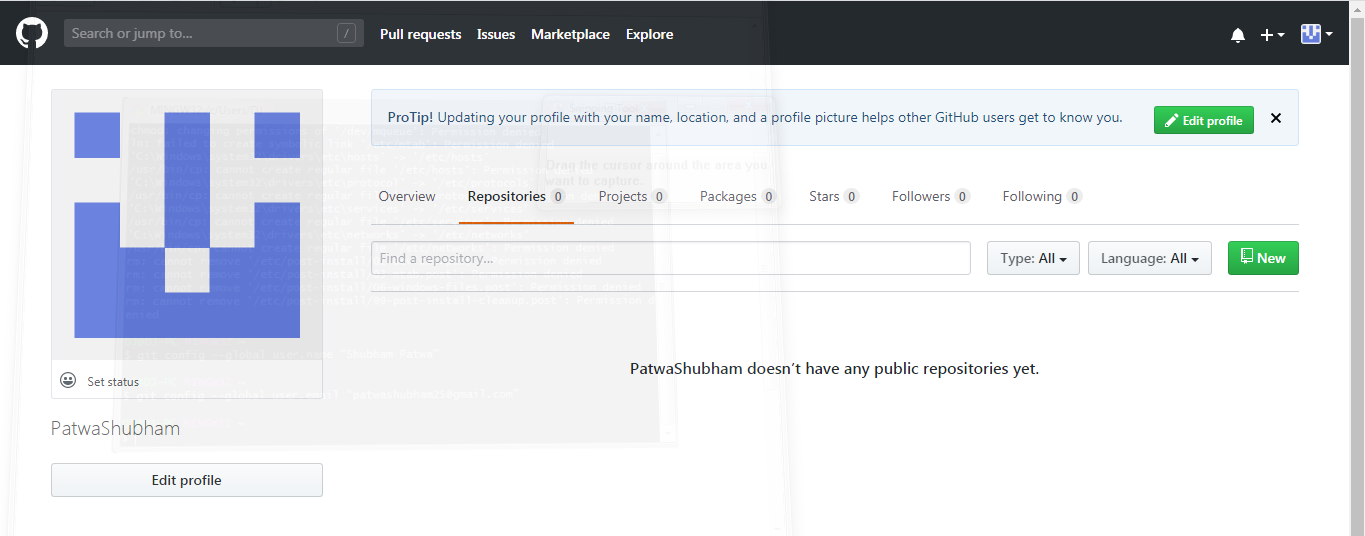
git config - - global user.email "<your email>"



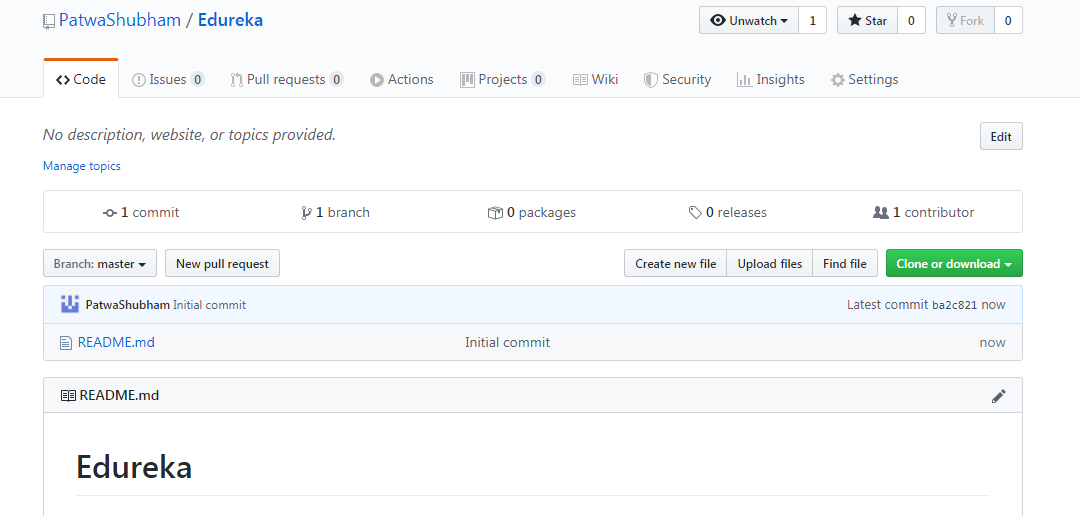
To view all the configuration files please use below command:

git config --list

Create a new account (if not exists) on [www.github.com](http://www.github.com)



Click on New Button and fill in the necessary details for creating new Repository

The motive of Git is to manage a project or a set of files as they change over time. Git stores this information in a data structure called a Git repository. The repository is the core of Git.

To be very clear, a Git repository is the directory where all of your project files and the related metadata resides.

Git records the current state of the project by creating a tree graph from the index. It is usually in the form of a Directed Acyclic Graph (DAG).

## GIT Commands and Operations

Some of the basic operations in GIT are:

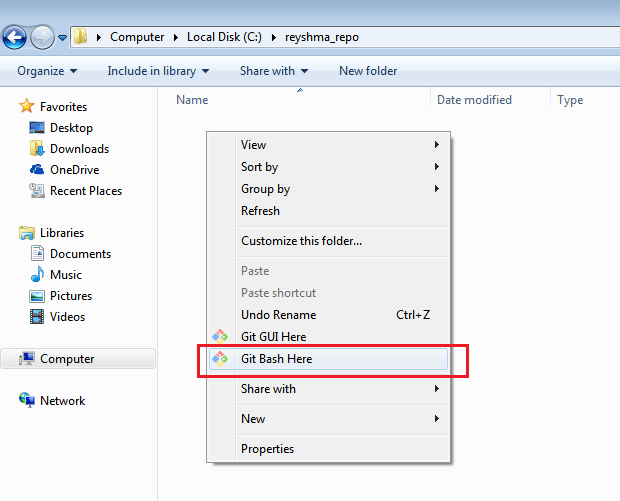
1. Initialize
2. Add
3. Commit
4. Pull
5. Push

Few advanced operations like Branching ,merging, rebashing etc.



 Git Bash is a text-only command line interface for using Git on Windows which provides features to run automated scripts.

After installing Git in your Windows system, just open your folder/directory where you want to store all your project files; right click and select ‘Git Bash here’.

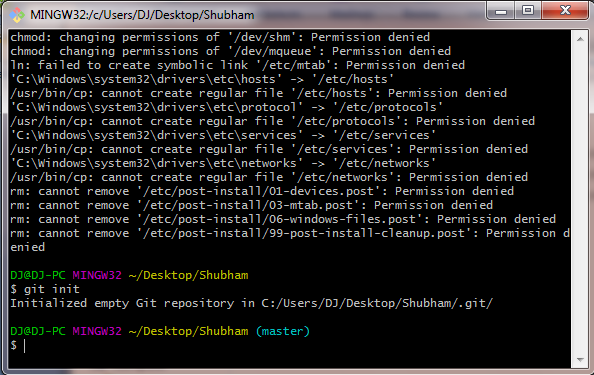


This will open up Git Bash terminal where you can enter commands to perform various Git operations.

Now, the next task is to initialize your repository.

**Initialize**

In order to do that, we use the command **git init.**Please refer to the below screenshot.



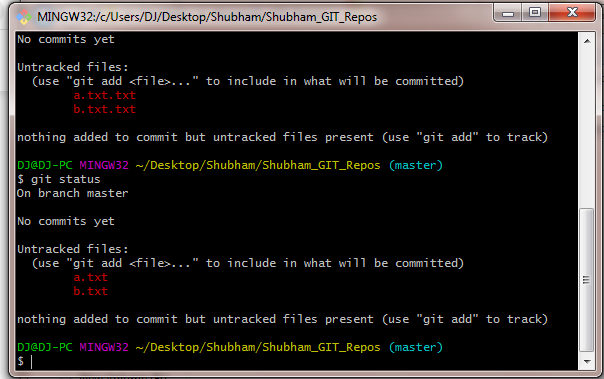
Now that my repository is initialized, let me create some files in the directory/repository. For e.g. I have created two text files namely edureka1.txt and edureka2.txt.

Let’s see if these files are in my index or not using the command git status. The index holds a snapshot of the content of the working tree/directory, and this snapshot is taken as the contents for the next change to be made in the local repository.

**Git status**

The git status command lists all the modified files which are ready to be added to the local repository.

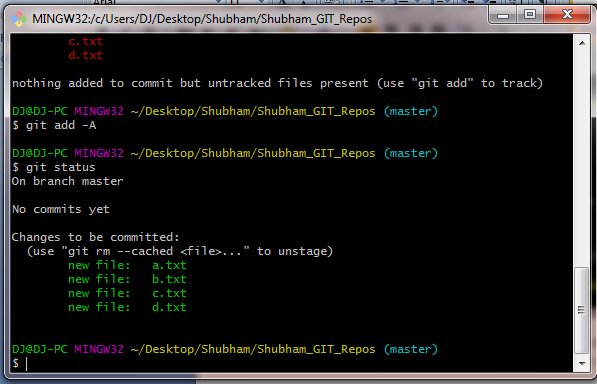
Let us type in the command to see what happens:



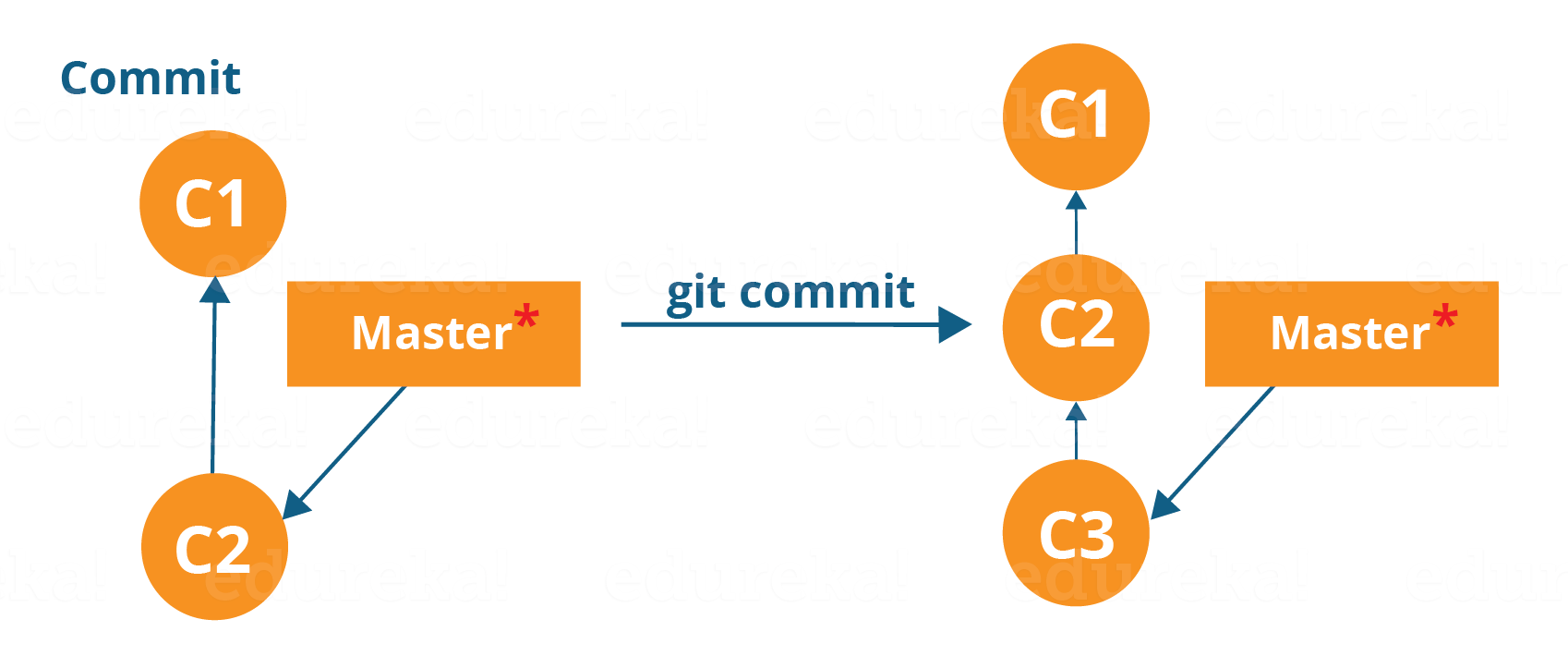
This shows that I have two files which are not added to the index yet. This means I cannot commit changes with these files unless I have added them explicitly in the index.

**git add –A**

This command will add all the files to the index which are in the directory but not updated in the index yet.



**Commit**

It refers to recording snapshots of the repository at a given time. Committed snapshots will never change unless done explicitly. Let me explain how commit works with the diagram below:

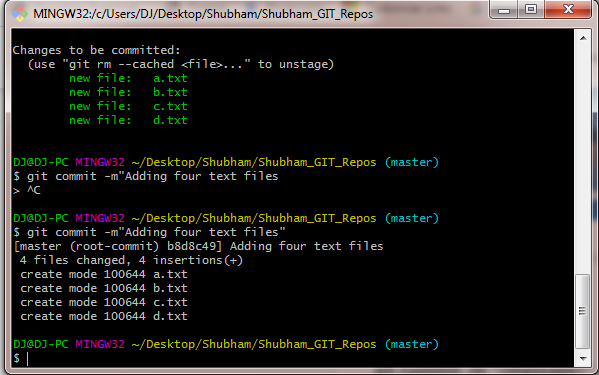
Here, C1 is the initial commit, i.e. the snapshot of the first change from which another snapshot is created with changes named C2. Note that the master points to the latest commit.

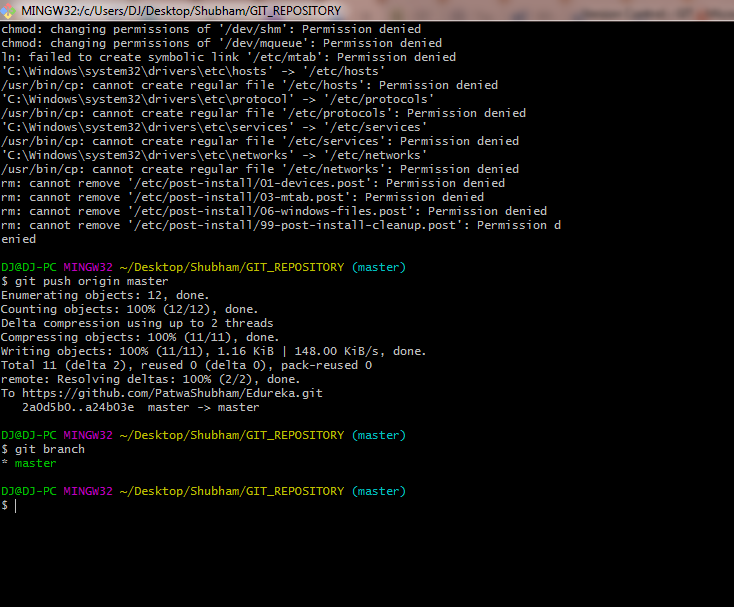
Now, when I commit again, another snapshot C3 is created and now the master points to C3 instead of C2.

Git aims to keep commits as lightweight as possible. So, it doesn’t blindly copy the entire directory every time you commit; it includes commit as a set of changes, or “delta” from one version of the repository to the other. In easy words, it only copies the changes made in the repository.

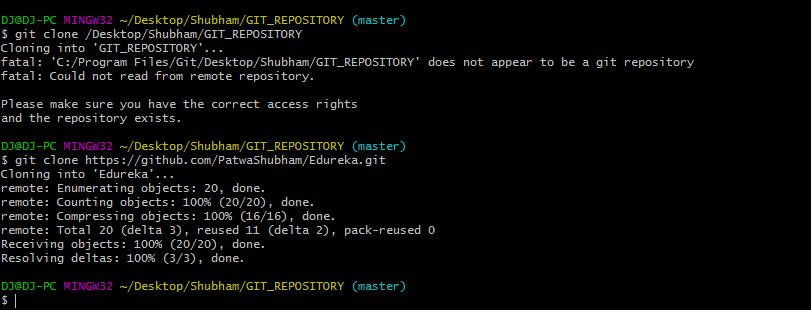
You can commit by using the command below:

**git commit -m”<message>” This**





Git Clone:



## ****Git Commands****

### git config

Usage: git config –global user.name “[name]”

Usage: git config –global user.email “[email address]”

This command sets the author name and email address respectively to be used with your commits.

Git Config Command - Git Commands - Edureka

### git init

Usage: git init [repository name]

This command is used to start a new repository.

GitInit Command - Git Commands - Edureka

### git clone

Usage: git clone [url]

This command is used to obtain a repository from an existing URL.



### git add

Usage: git add [file]

This command adds a file to the staging area.

Git Add Command - Git Commands - Edureka

Usage: git add \*

This command adds one or more to the staging area.

Git Add Command - Git Commands - Edureka

### git commit

Usage: git commit -m “[ Type in the commit message]”

This command records or snapshots the file permanently in the version history.



Usage: git commit -a

This command commits any files you’ve added with the git add command and also commits any files you’ve changed since then.

Git Commit Command - Git Commands - Edureka

### git diff

Usage: git diff

This command shows the file differences which are not yet staged.



 Usage: git diff –staged

This command shows the differences between the files in the staging area and the latest version present.



Usage: git diff [first branch] [second branch]

This command shows the differences between the two branches mentioned.



### git reset

Usage: git reset [file]

This command unstages the file, but it preserves the file contents.



Usage: git reset [commit]

This command undoes all the commits after the specified commit and preserves the changes locally.

Git Reset Command - Git Commands - Edureka

Usage: git reset –hard [commit]  This command discards all history and goes back to the specified commit.

Git Reset Command - Git Commands - Edureka

### git status

Usage: git status

This command lists all the files that have to be committed.



### git rm

Usage: git rm [file]

This command deletes the file from your working directory and stages the deletion.

Git Rm Command - Git Commands - Edureka

### git log

Usage: git log

This command is used to list the version history for the current branch.



Usage: git log –follow[file]

This command lists version history for a file, including the renaming of files also.



### git show

Usage: git show [commit]

This command shows the metadata and content changes of the specified commit.



### git tag

Usage: git tag [commitID]

This command is used to give tags to the specified commit.



### git branch

Usage: git branch

This command lists all the local branches in the current repository.

Git Branch Command - Git Commands - Edureka

Usage: git branch [branch name]

This command creates a new branch.

Git Branch Command - Git Commands - Edureka

Usage: git branch -d [branch name]

This command deletes the feature branch.

Git Branch Command - Git Commands - Edureka

### git checkout

Usage: git checkout [branch name]

This command is used to switch from one branch to another.

Git Checkout Command - Git Commands - Edureka

Usage: git checkout -b [branch name]

This command creates a new branch and also switches to it.

Git Checkout Command - Git Commands - Edureka

### git merge

Usage: git merge [branch name]

This command merges the specified branch’s history into the current branch.

Git Merge Command - Git Commands - Edureka

### git remote

Usage: git remote add [variable name] [Remote Server Link]

This command is used to connect your local repository to the remote server.

Git Remote Command - Git Commands - Edureka

### git push

Usage: git push [variable name] master

This command sends the committed changes of master branch to your remote repository.



Usage: git push [variable name] [branch]

This command sends the branch commits to your remote repository.



Usage: git push –all [variable name]

This command pushes all branches to your remote repository.



Usage: git push [variable name] :[branch name]

This command deletes a branch on your remote repository.



### git pull

Usage: git pull [Repository Link]

This command fetches and merges changes on the remote server to your working directory.



### git stash

Usage: git stash save

This command temporarily stores all the modified tracked files.

Git Stash Command - Git Commands - Edureka

Usage: git stash pop

This command restores the most recently stashed files.



Usage: git stash list

This command lists all stashed changesets.

Git Stash Command - Git Commands - Edureka

Usage: git stash drop

This command discards the most recently stashed changeset.

|  |  |  |
| --- | --- | --- |
| Git task | Notes | Git commands |
| [**Tell Git who you are**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) | Configure the author name and email address to be used with your commits.  Note that Git [strips some characters](http://stackoverflow.com/questions/26159274/is-it-possible-to-have-a-trailing-period-in-user-name-in-git/26219423#26219423) (for example trailing periods) from user.name. | git config --global user.name "Sam Smith"  git config --global user.email sam@example.com |
| [**Create a new local repository**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) |  | git init |
| [**Check out a repository**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) | Create a working copy of a local repository: | git clone /path/to/repository |
| For a remote server, use: | git clone username@host:/path/to/repository |
| [**Add files**](https://www.atlassian.com/git/tutorials/saving-changes#git-add) | Add one or more files to staging (index): | git add <filename>  git add \* |
| [**Commit**](https://www.atlassian.com/git/tutorials/saving-changes#git-commit) | Commit changes to head (but not yet to the remote repository): | git commit -m "Commit message" |
| Commit any files you've added with git add, and also commit any files you've changed since then: | git commit -a |
| [**Push**](https://www.atlassian.com/git/tutorials/syncing#git-push) | Send changes to the master branch of your remote repository: | git push origin master |
| [**Status**](https://www.atlassian.com/git/tutorials/inspecting-a-repository#git-status) | List the files you've changed and those you still need to add or commit: | git status |
| [**Connect to a remote repository**](https://www.atlassian.com/git/tutorials/syncing#git-remote) | If you haven't connected your local repository to a remote server, add the server to be able to push to it: | git remote add origin <server> |
| List all currently configured remote repositories: | git remote -v |
| [**Branches**](https://www.atlassian.com/git/tutorials/using-branches) | Create a new branch and switch to it: | git checkout -b <branchname> |
| Switch from one branch to another: | git checkout <branchname> |
| List all the branches in your repo, and also tell you what branch you're currently in: | git branch |
| Delete the feature branch: | git branch -d <branchname> |
| Push the branch to your remote repository, so others can use it: | git push origin <branchname> |
| Push all branches to your remote repository: | git push --all origin |
| Delete a branch on your remote repository: | git push origin :<branchname> |
| [**Update from the remote repository**](https://www.atlassian.com/git/tutorials/syncing) | Fetch and merge changes on the remote server to your working directory: | git pull |
| To merge a different branch into your active branch: | git merge <branchname> |
| View all the merge conflicts:  View the conflicts against the base file:  Preview changes, before merging: | git diff  git diff --base <filename>  git diff <sourcebranch> <targetbranch> |
| After you have manually resolved any conflicts, you mark the changed file: | git add <filename> |
| **Tags** | You can use tagging to mark a significant changeset, such as a release: | git tag 1.0.0 <commitID> |
| CommitId is the leading characters of the changeset ID, up to 10, but must be unique. Get the ID using: | git log |
| Push all tags to remote repository: | git push --tags origin |
| [**Undo local changes**](https://www.atlassian.com/git/tutorials/undoing-changes) | If you mess up, you can replace the changes in your working tree with the last content in head:  Changes already added to the index, as well as new files, will be kept. | git checkout -- <filename> |
| Instead, to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it, do this: | git fetch origin  git reset --hard origin/master |
| **Search** | Search the working directory for foo(): | git grep "foo()" |