**GIT Basics**

Before starting with Git, let us know what is Version Control.

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

VCS – is a software that helps software developers to work together to maintain a complete history of their work.

VCS functions.

1. Allows developers to work simultaneously.
2. Doesn’t allow overwriting each other’s changes.
3. Maintains a history of every session.

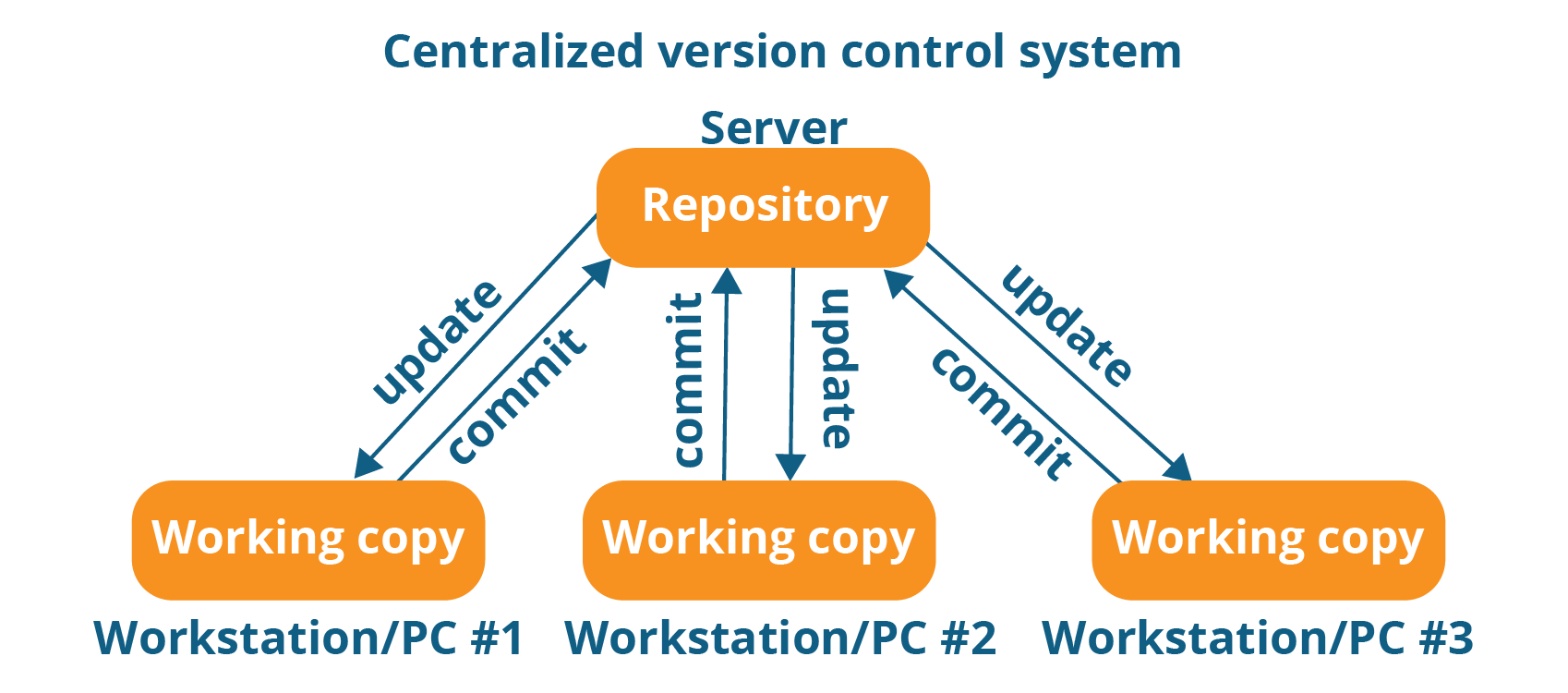
There are two types of VCS:

* Centralized Version Control System (CVCS)
* Distributed Version Control System (DVCS)

**Centralized VCS**

Centralized version control system (CVCS) 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.

Please refer to the diagram below to get a better idea of CVCS:



The repository in the above diagram indicates a central server that could be local or remote which is directly connected to each of the programmer’s workstation.

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

Even though it seems pretty convenient to maintain a single repository, it has some major drawbacks. Some of them are:

1. It is not locally available; meaning you always need to be connected 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.

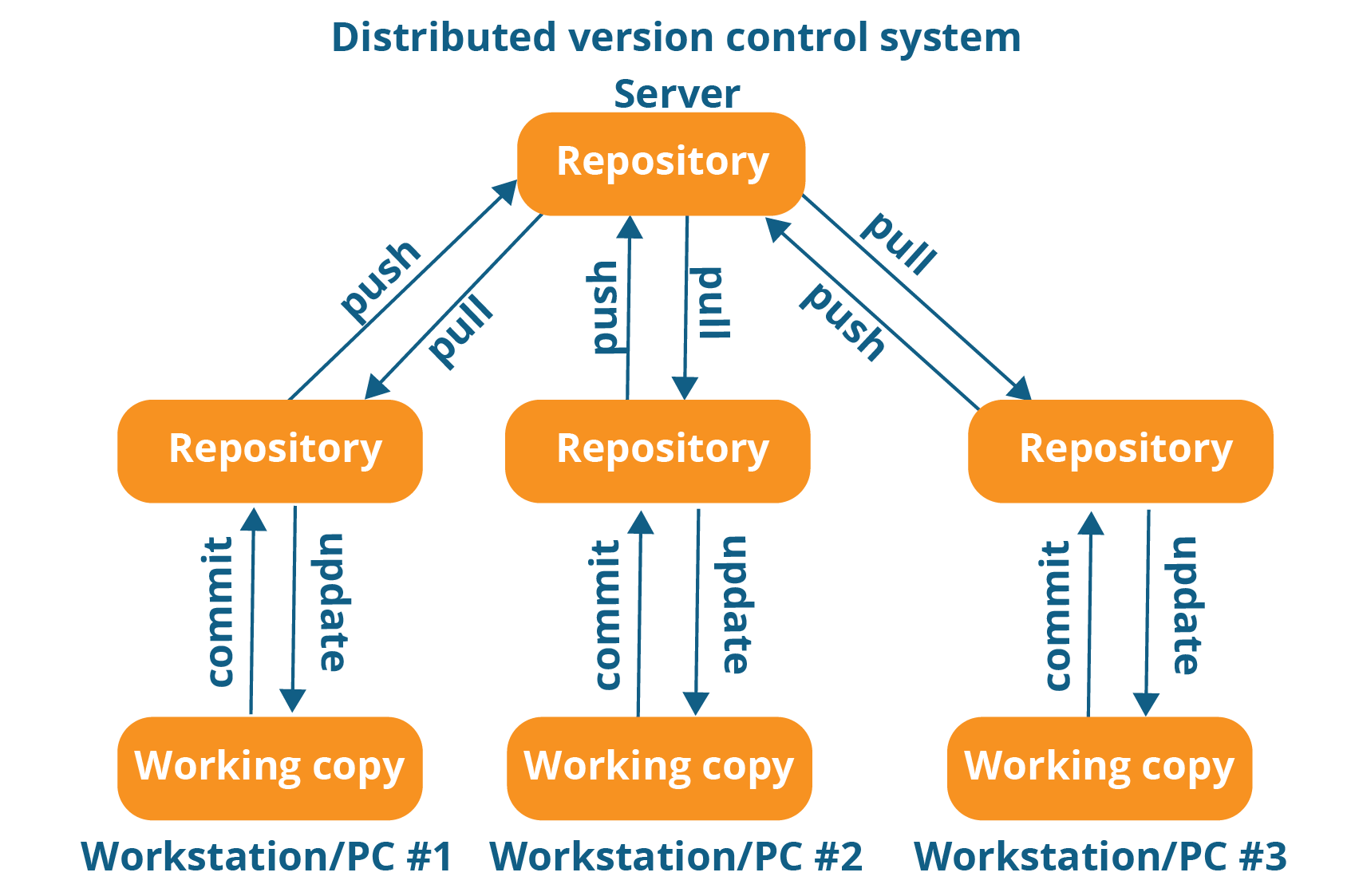
This is when Distributed VCS comes to the rescue.

**Distributed VCS**

These systems do not necessarily 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.

You will understand it better by referring to the diagram below:



As you can see in the above diagram, 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.

They can update their local repositories with new data from the central server by an operation called “**pull**” and affect changes to the main repository by an operation called “**push**” from their local repository.

The act of cloning an entire repository into your workstation to get a local repository gives you the following advantages:

1. All operations (except push & pull) are very fast because the tool only needs to access the hard drive, not a remote server. Hence, you do not always need an internet connection.
2. Committing new change-sets can be done locally without manipulating the data on the main repository. Once you have a group of change-sets ready, you can push them all at once.
3. Since 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 one of the contributor’s local repositories.

After knowing Distributed VCS, its time we take a dive into what is Git.

Advantages of GIT:

1. Free and Open source – can use to manage data freely and also can download its source code and perform changes according to our requirement.
2. Fast and small - As most of the operations are performed locally, it gives a huge benefit in terms of speed. Git does not rely on the central server; that is why, there is no need to interact with the remote server for every operation. The core part of Git is written in C, which avoids runtime overheads associated with other high-level languages. Though Git mirrors entire repository, the size of the data on the client side is small. This illustrates the efficiency of Git at compressing and storing data on the client side.
3. Implicit Backup - The chances of losing data are very rare when there are multiple copies of it. Data present on any client side mirrors the repository, hence it can be used in the event of a crash or disk corruption.
4. Security - Git uses a common cryptographic hash function called secure hash function (SHA1), to name and identify objects within its database. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. It implies that, it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git.
5. No need of powerful hardware - 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.
6. Easier branching **-** CVCS uses cheap copy mechanism, If we create a new branch, it will copy all the codes to the new branch, so it is time-consuming and not efficient. Also, deletion and merging of branches in CVCS is complicated and time-consuming. But branch management with Git is very simple. It takes only a few seconds to create, delete, and merge branches.

Users can perform many operations with the local repository such as add file, remove file, rename file, move file, commit changes, and many more.

**What Is Git?**

Git is a Distributed Version Control tool that supports distributed non-linear workflows by providing data assurance for developing quality software. Before you go ahead, check out this video on GIT which will give you better in-sight.

Some things to know:

HEAD is a pointer, which always points to the latest commit in the branch.

atahseen@CSCINDAI424207 MINGW64 /d/Devops/DevOpsTools/GIT (master)

$ ls -1 .git/refs/heads/

master

atahseen@CSCINDAI424207 MINGW64 /d/Devops/DevOpsTools/GIT (master)

$ cat .git/config

[core]

repositoryformatversion = 0

filemode = false

bare = false

logallrefupdates = true

symlinks = false

ignorecase = true

[gui]

wmstate = normal

geometry = 1362x685+-85+53 175 196

[remote "origin"]

url = https://github.com/atahseen1/mygit.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

To view git configurations:

* > git config –list

To set user name and email:

* > git config –global user.name “First Last”
* > git config –global user.email “email id”

Terminology:

Git Bash

Git Hub

Tag

Trunk

Branch

Master

Merge

SSH Key

Pull

Push

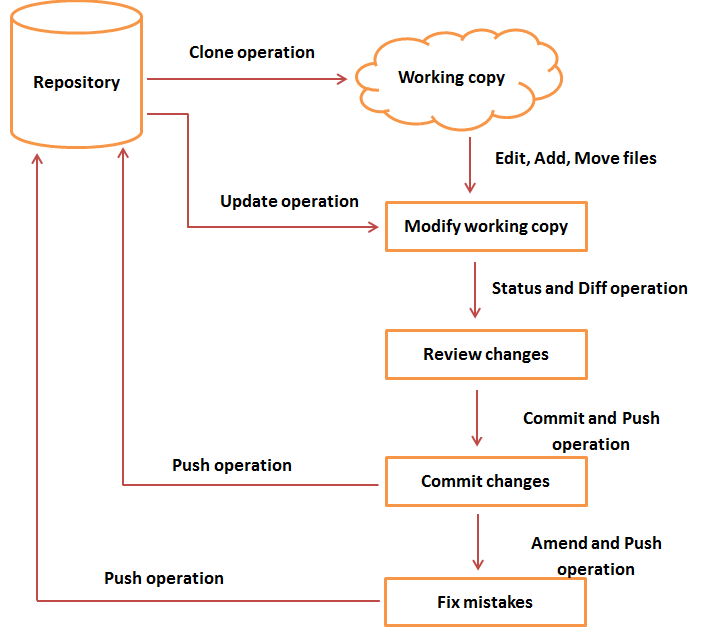
Repository

Clone

General workflow is as follows:

* You clone the Git repository as a working copy.
* You modify the working copy by adding/editing files.
* If necessary, you also update the working copy by taking other developer's changes.
* You review the changes before commit.
* You commit changes. If everything is fine, then you push the changes to the repository.
* After committing, if you realize something is wrong, then you correct the last commit and push the changes to the repository.

Shown below is the pictorial representation of the work-flow.



**Git Tutorial – Operations & Commands**

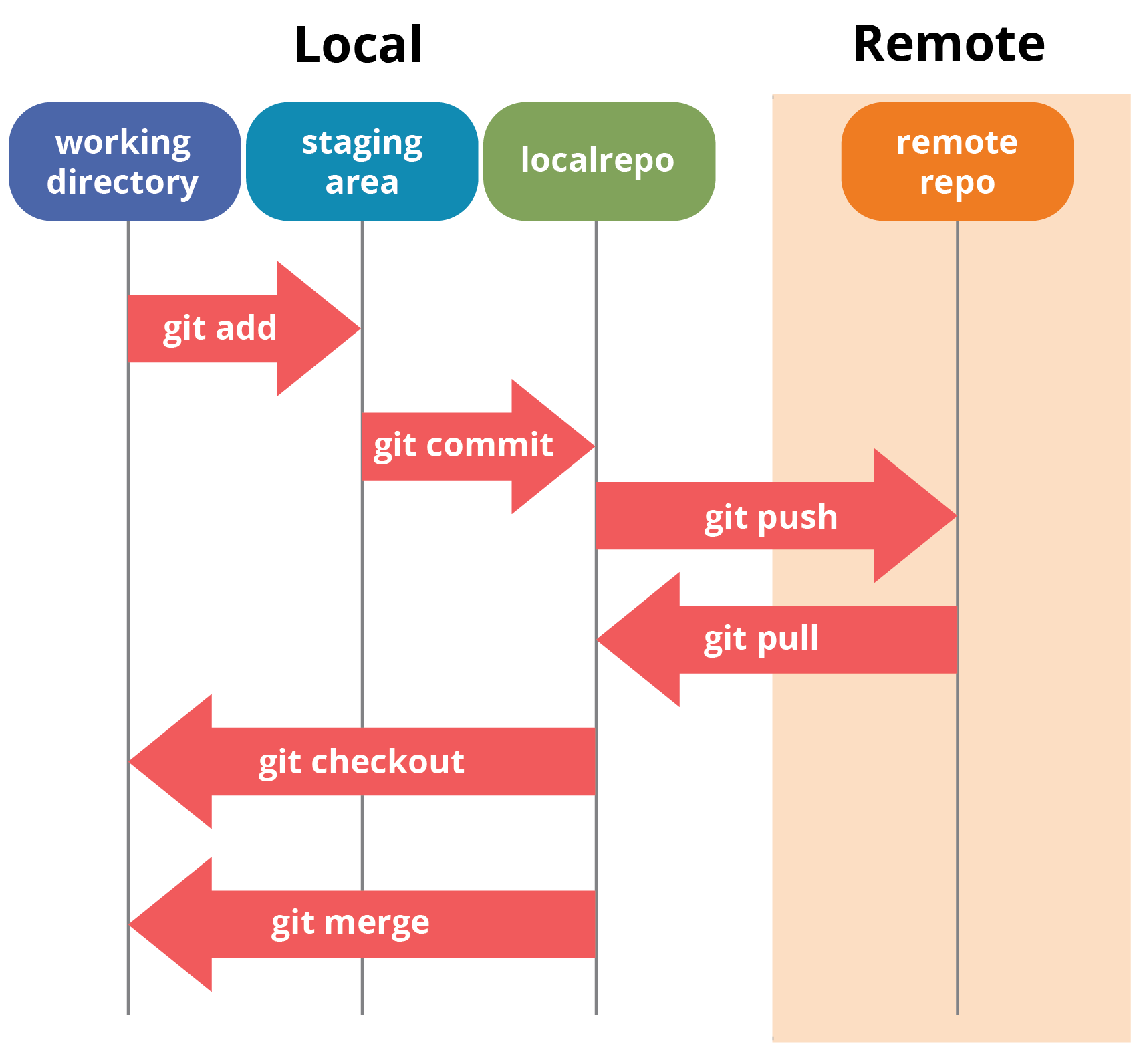
Some of the basic operations in Git are:

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

Some advanced Git operations are:

1. Branching
2. Merging
3. Rebasing

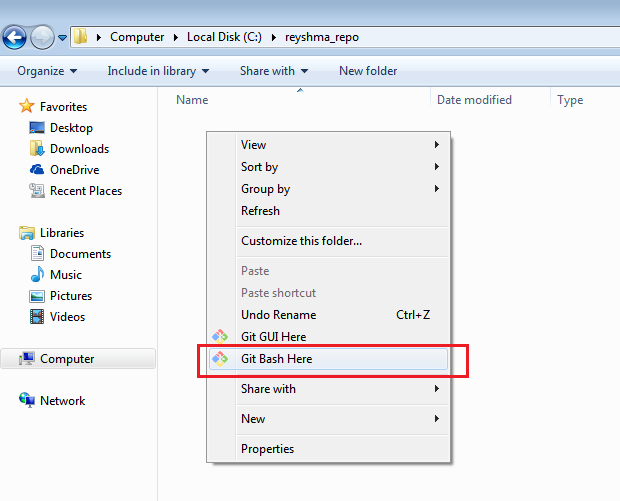
Let me first give you a brief idea about how these operations work with the Git repositories. Take a look at the architecture of Git below:



If you understand the above diagram well and good, but if you don’t, you need not worry, I will be explaining these operations in this Git Tutorial one by one. Let us begin with the basic operations.

I will show you the commands and the operations using Git Bash. 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***’.

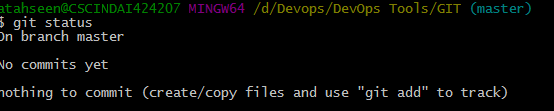


This will open up Git Bash terminal where you can enter commands to perform various Git operations.

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.



**git init** creates an empty Git repository or re-initializes an existing one. It basically creates a**.git** directory with sub directories and template files. Running a **git init** in an existing repository will not overwrite things that are already there. It rather picks up the newly added templates.

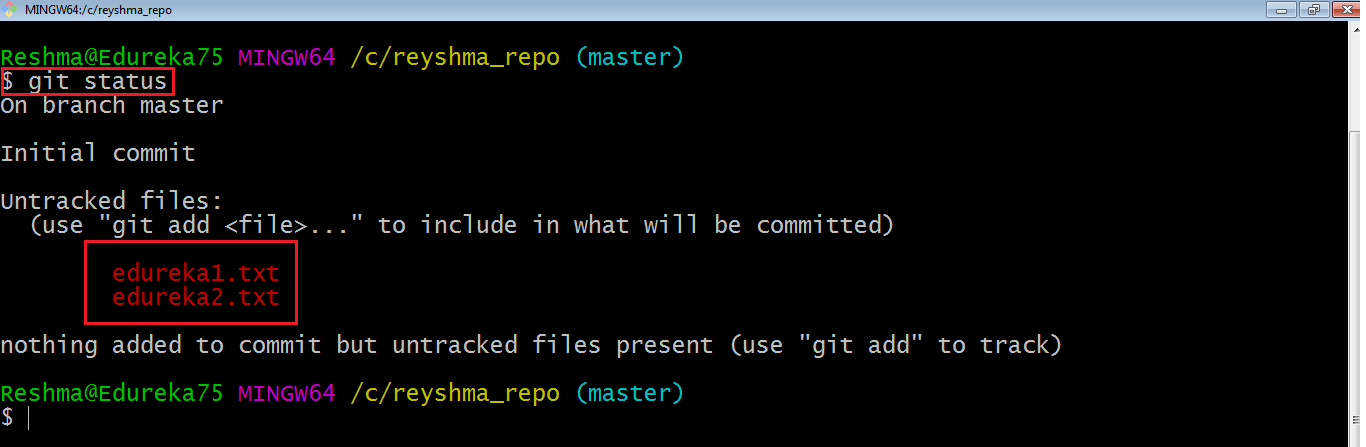
Now that my repository is initialized, let me create some files in the directory/repository. For e.g. I have created two text files namely *edureka1.txt*and *edureka2.txt*.

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:



his 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.

**Add**

This command updates the index using the current content found in the working tree and then prepares the content in the staging area for the next commit.

Thus, after making changes to the working tree, and before running the **commit**command, you must use the**add** command to add any new or modified files to the index. For that, use the commands below:

**git add <directory>**

or

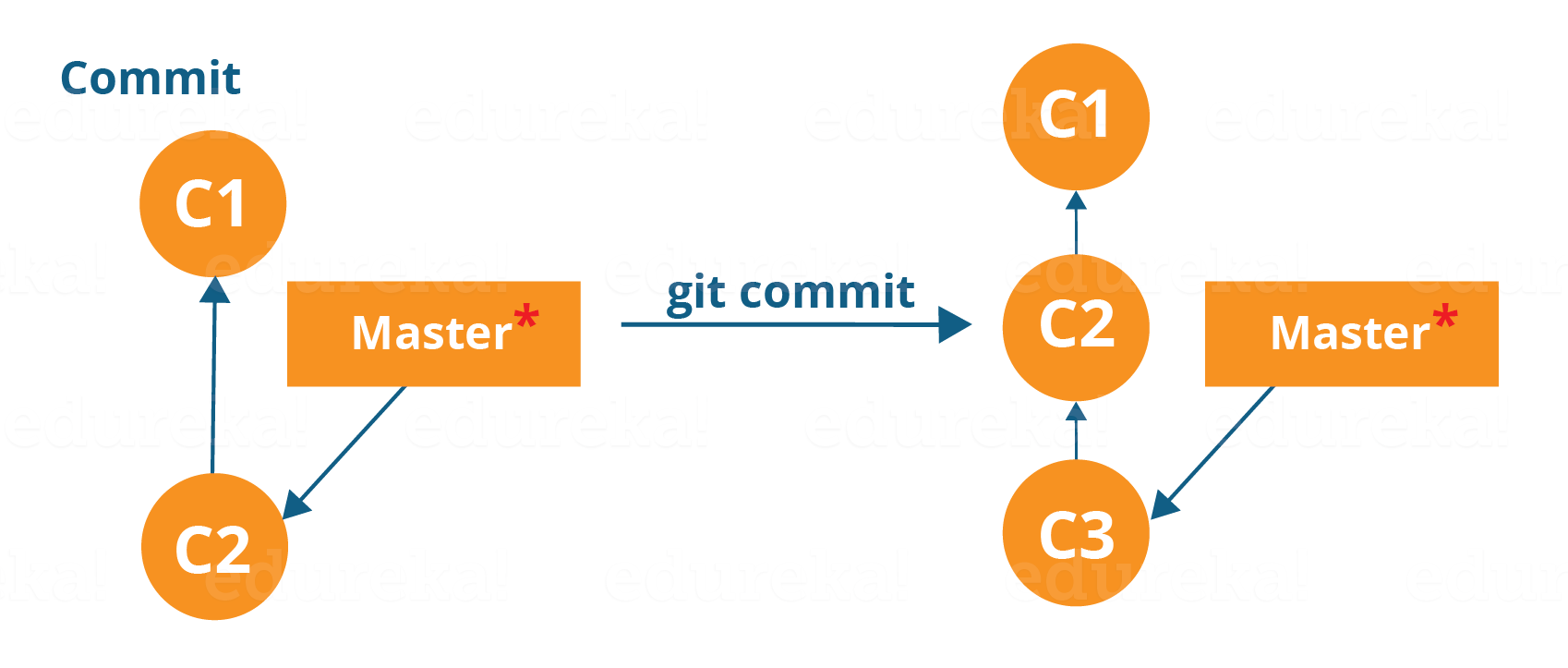
**git add <file>**

Let me demonstrate the **git add** for you so that you can understand it better.

I have created two more files *edureka3.txt* and *edureka4.txt*. Let us add the files using the command **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. Now that the new files are added to the index, you are ready to commit them.

**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:

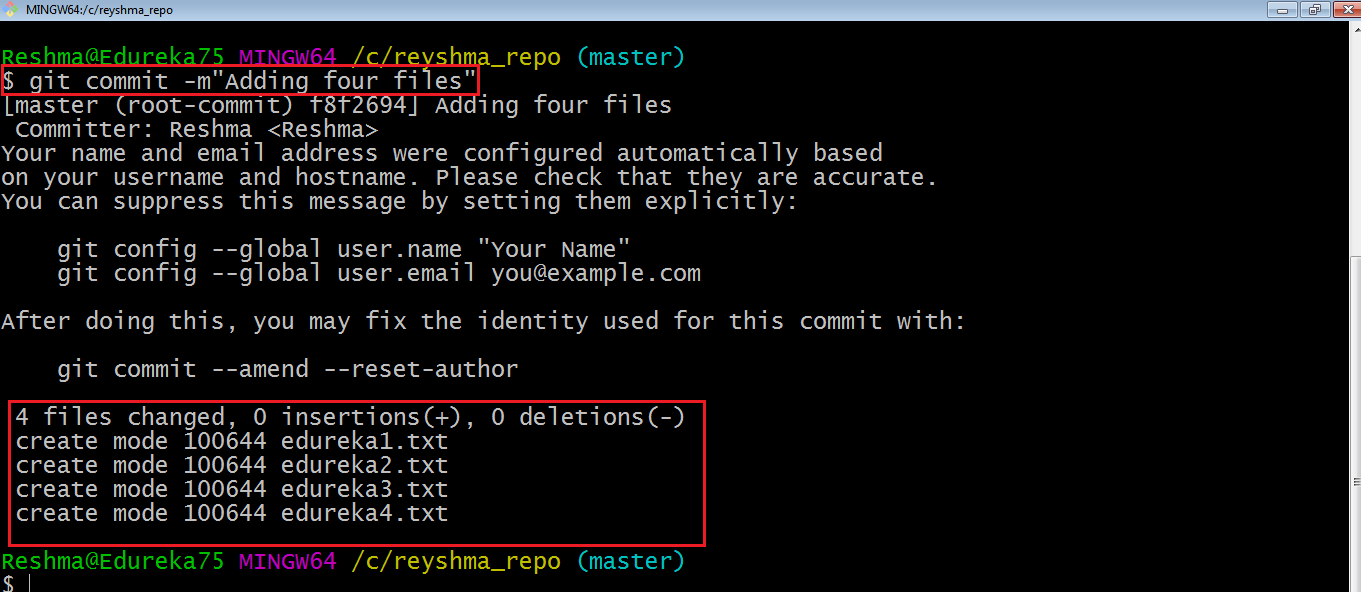
**git commit**

This will commit the staged snapshot and will launch a text editor prompting you for a commit message.

Or you can use:

**git commit -m “<message>”**

Let’s try it out.



As you can see above, the **git commit** command has committed the changes in the four files in the local repository.

* **> git log**

After commit to view log details, he runs the git log command. It will display the information of all the commits with their commit ID, commit author, commit date and **SHA-1** hash of commit.

Now, if you want to commit a snapshot of all the changes in the working directory at once, you can use the command below:

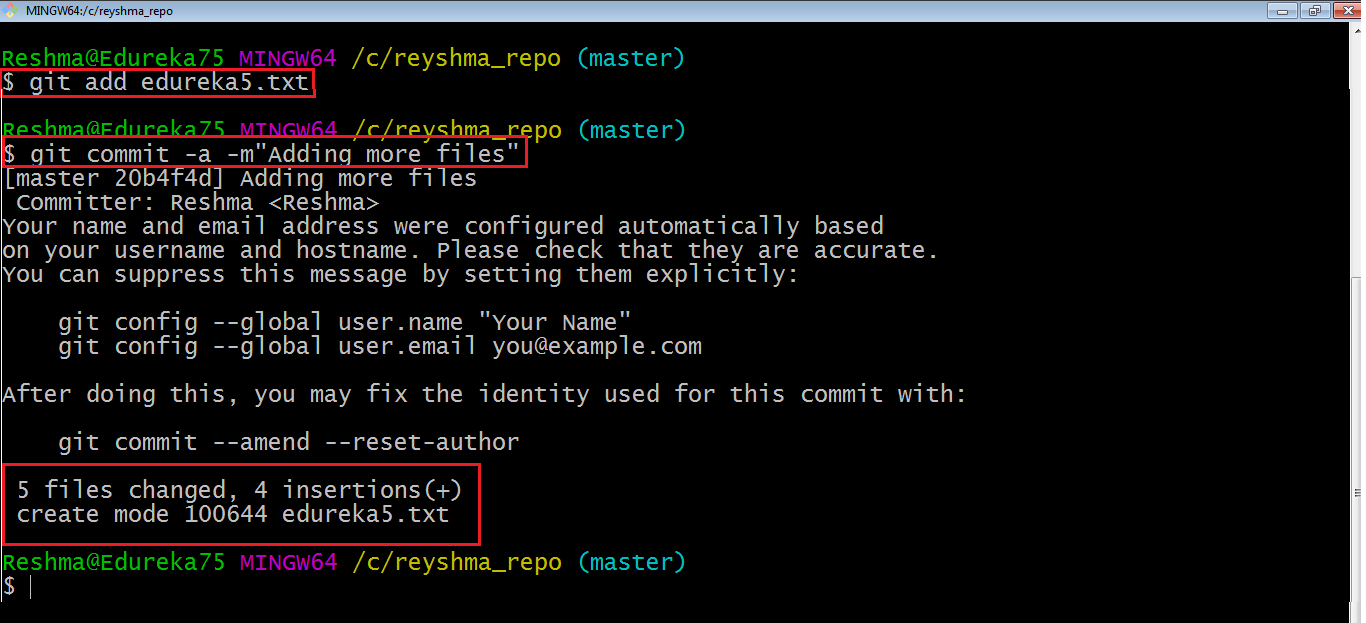
**git commit -a**

I have created two more text files in my working directory viz. *edureka5.txt* and *edureka6.txt* but they are not added to the index yet.

I am adding edureka5.txt using the command:

**git add edureka5.txt**

I have added *edureka5.txt* to the index explicitly but not *edureka6.txt* and made changes in the previous files. I want to commit all changes in the directory at once. Refer to the below snapshot.



This command will commit a snapshot of all changes in the working directory but only includes modifications to tracked files i.e. the files that have been added with **git add** at some point in their history. Hence, *edureka6.txt*was not committed because it was not added to the index yet. But changes in all previous files present in the repository were committed, i.e. *edureka1.txt*, *edureka2.txt*, *edureka3.txt*, *edureka4.txt*and *edureka5.txt*.  
Now I have made my desired commits in my local repository.

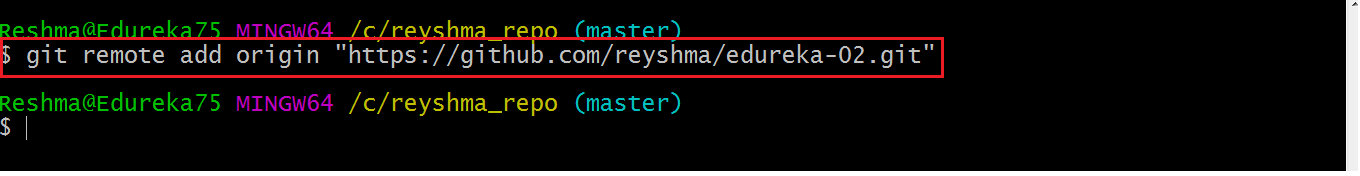
Note that before you affect changes to the central repository you should always pull changes from the central repository to your local repository to get updated with the work of all the collaborators that have been contributing in the central repository. For that we will use the **pull** command.

**Pull**

The **git pull** command fetches changes from a remote repository to a local repository. It merges upstream changes in your local repository, which is a common task in Git based collaborations.

But first, you need to set your central repository as origin using the command:

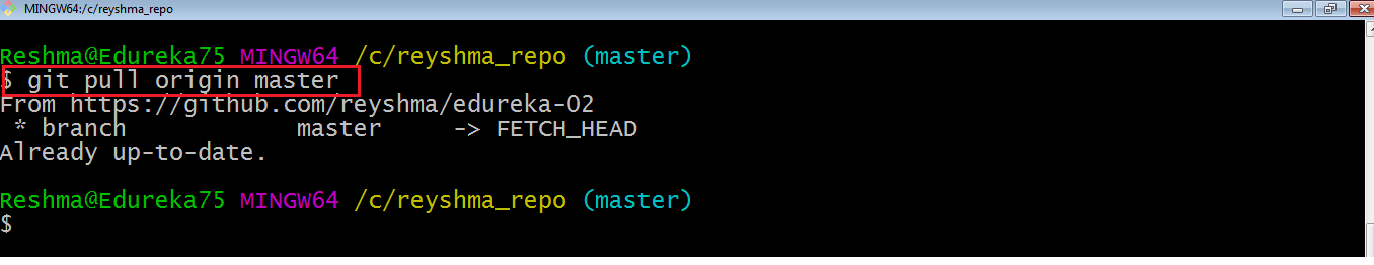
**git remote add origin <link of your central repository>**



Now that my origin is set, let us extract files from the origin using pull. For that use the command:

**git pull origin master**

This command will copy all the files from the master branch of remote repository to your local repository.



Since my local repository was already updated with files from master branch, hence the message is Already up-to-date. Refer to the screen shot above.

***Note:****One can also try pulling files from a different branch using the following command:*

***git pull origin <branch-name>***

Your local Git repository is now updated with all the recent changes. It is time you make changes in the central repository by using the**push** command.

**Push**

This command transfers commits from your local repository to your remote repository. It is the opposite of pull operation.

Pulling imports commits to local repositories whereas pushing exports commits to the remote repositories .

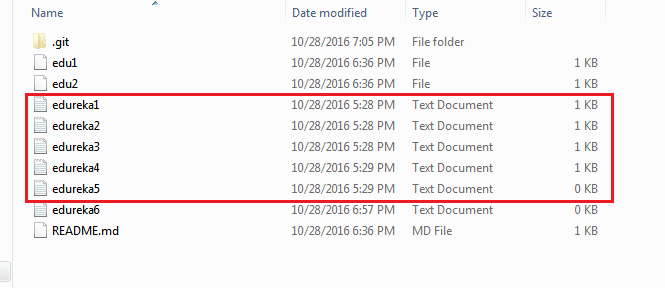
The use of **git push** is to publish your local changes to a central repository. After you’ve accumulated several local commits and are ready to share them with the rest of the team, you can then push them to the central repository by using the following command:

**git push <remote>**

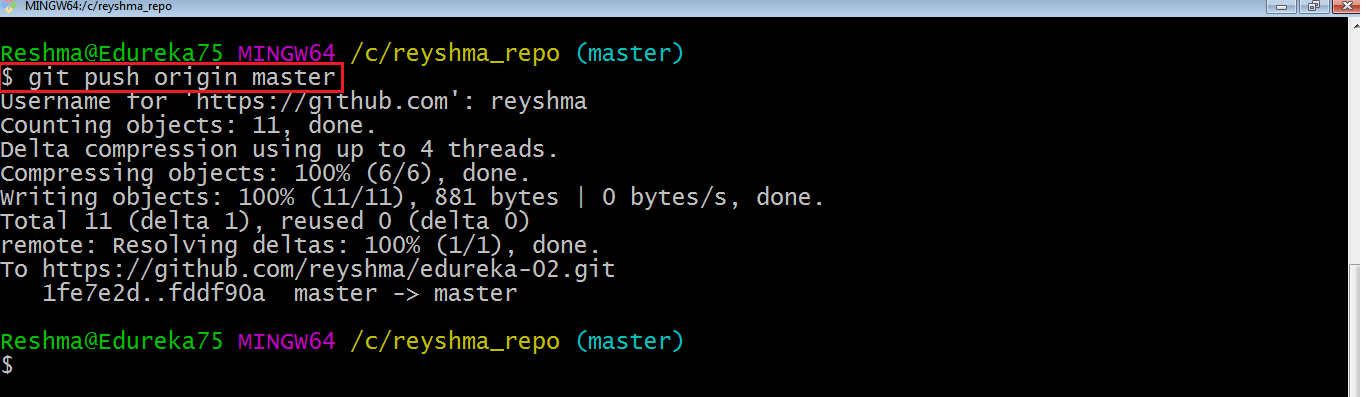
**Note***: This remote refers to the remote repository which had been set before using the pull command*.

This pushes the changes from the local repository to the remote repository along with all the necessary commits and internal objects. This creates a local branch in the destination repository.

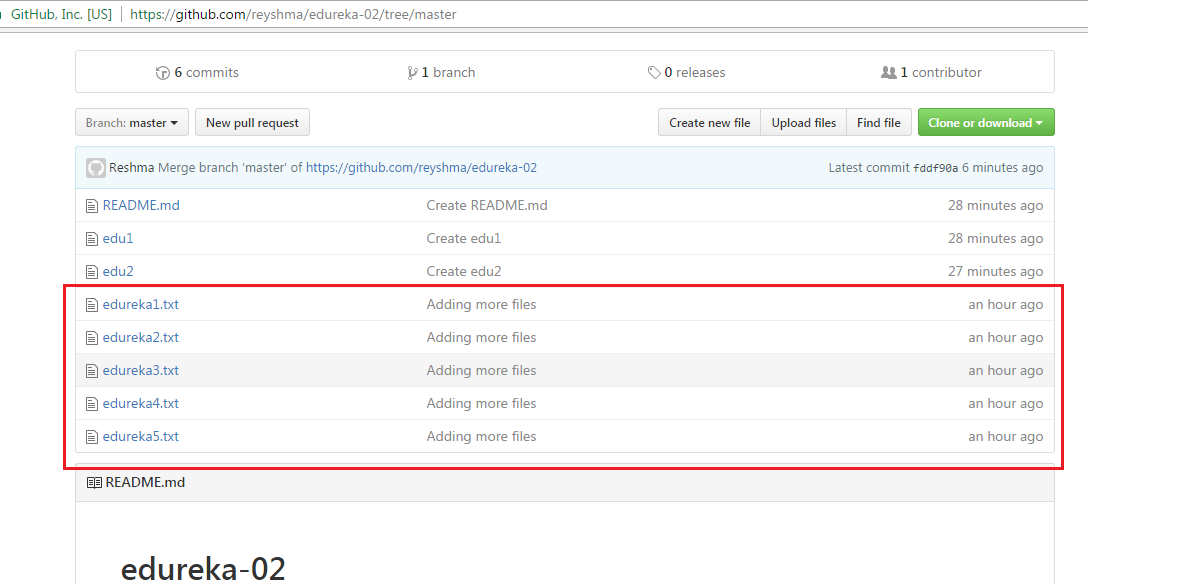
Let me demonstrate it for you.



The above files are the files which we have already committed previously in the commit section and they are all “*push-ready*“. I will use the command **git push origin master** to reflect these files in the master branch of my central repository.



Let us now check if the changes took place in my central repository.



Yes, it did. :-)

To prevent overwriting, Git does not allow push when it results in a non-fast forward merge in the destination repository.

**Note**:*A non-fast forward merge means an upstream merge i.e. merging with ancestor or parent branches from a child branch.*

To enable such merge, use the command below:

**git push <remote> –force**

The above command forces the push operation even if it results in a non-fast forward merge.

If you want to learn branching, merging and other operations, read the full blog here: [Git Tutorial | Commands And Operations In Git | Edureka](https://www.edureka.co/blog/git-tutorial/?utm_campaign=social-media-september-ra&utm_medium=crosspost&utm_source=quora)

git remote -v

git remote -l

**Why do we use VCS?**

[**https://mikemcquaid.com/2014/01/18/why-use-version-control/**](https://mikemcquaid.com/2014/01/18/why-use-version-control/)

[**https://www2.le.ac.uk/services/research-data/organise-data/version-control**](https://www2.le.ac.uk/services/research-data/organise-data/version-control)

[**https://www.atlassian.com/blog/software-teams/version-control-centralized-dvcs**](https://www.atlassian.com/blog/software-teams/version-control-centralized-dvcs)

[**https://dont-be-afraid-to-commit.readthedocs.io/en/latest/cheatsheet.html**](https://dont-be-afraid-to-commit.readthedocs.io/en/latest/cheatsheet.html)