**What is git**

GIT is a distributed version control system and source code management (SCM) system with an emphasis to handle small and large projects with speed and efficiency.

Git is a free and open source.

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

Git is a distributed version control system (DVCS), while SVN is a centralized version control system.

**Distributed Development**: In SVN, each developer gets a working copy that points back to a single central repository. Git, however, is a distributed v ersion control system. Instead of a working copy, each developer gets their own local repository, complete with a full history of commits.

* Free and open source.
* Easier branching.
* High availability.
* Faciliates team collaoration.

**What is head**

HEAD is a reference to the last commit in the currently checked-out branch.

Latest commits are point to the head

**What is a repository in GIT?**

Git stores project details and files information in a data structure called a repository. The Git repository is stored in the same directory as the project itself, in a subdirectory called .git.

A git repository contains, among other things, the following:

* A set of commit objects.
* A set of references to commit objects, called heads.

**What do you mean by git add?**

git add .’ command adds all modified and new files in the current directory and all subdirectories to the staging area (index), thus preparing them to be included in the next git commit. Any files matching the patterns in the .got ignore file will be ignored by git add.

**Explain git status command.**

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.

**Function of git clone command.**

The git clone command creates a copy of an existing Git repository to the local repository. To get the copy of a central repository, “cloning” is the most common way used by developers.

**How do I integrate changes from GIT one branch into another?**

in Git, there are two main ways to integrate changes from one branch into another: the **merge** and the **rebase**.

**Difference between git merge vs rebase.**

Both does the same thing (integrate branches) in slightly different way.

The major benefit of rebasing is that you get a much cleaner project history. First, it eliminates the unnecessary merge commits required by git merge.

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:



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

* It is not locally available; meaning you always need to be connected to a network to perform any action.
* 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****

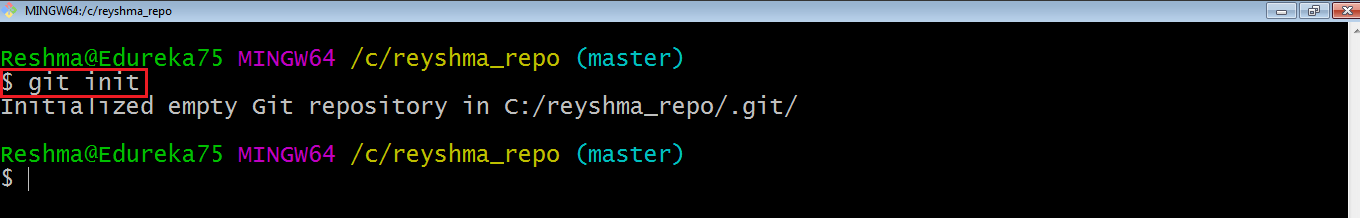
These systems do not necessarily rely on a central server to store all the versions of a project file.

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.

**Commands:**

## ****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.

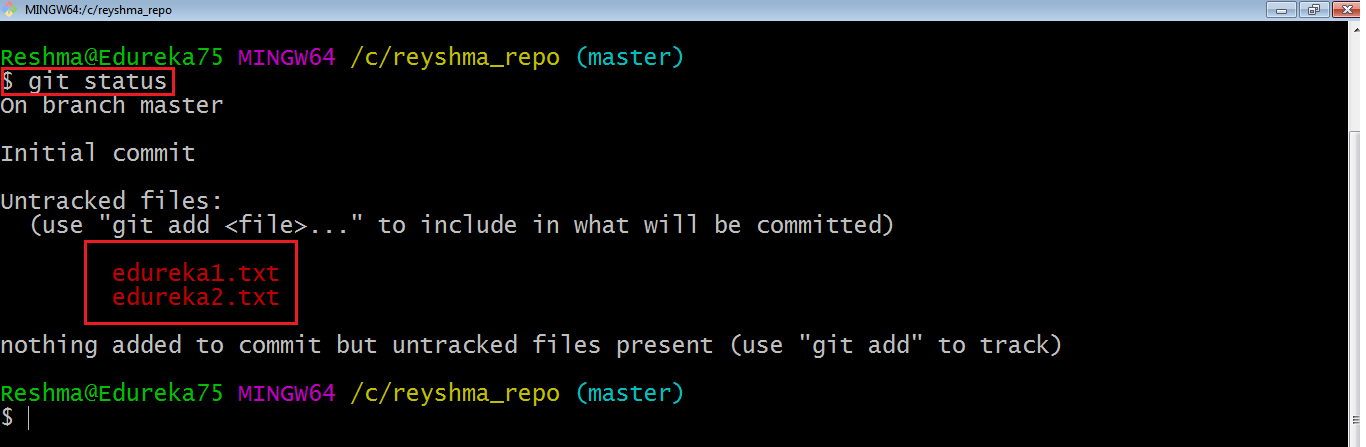
Now that my repository is initialized, let me create some files in the directory/repository. For e.g. I have created two text files namely edureka1.txt and edureka2.txt.

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:



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.

**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:

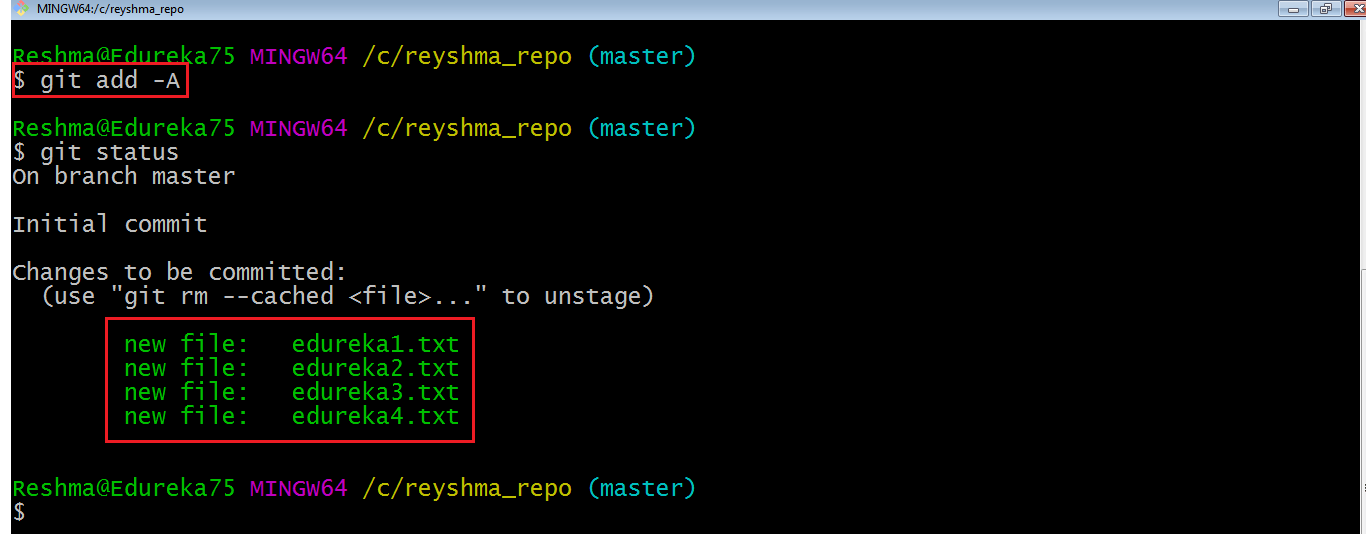
**git add <directory>**

or

**git add <file>**

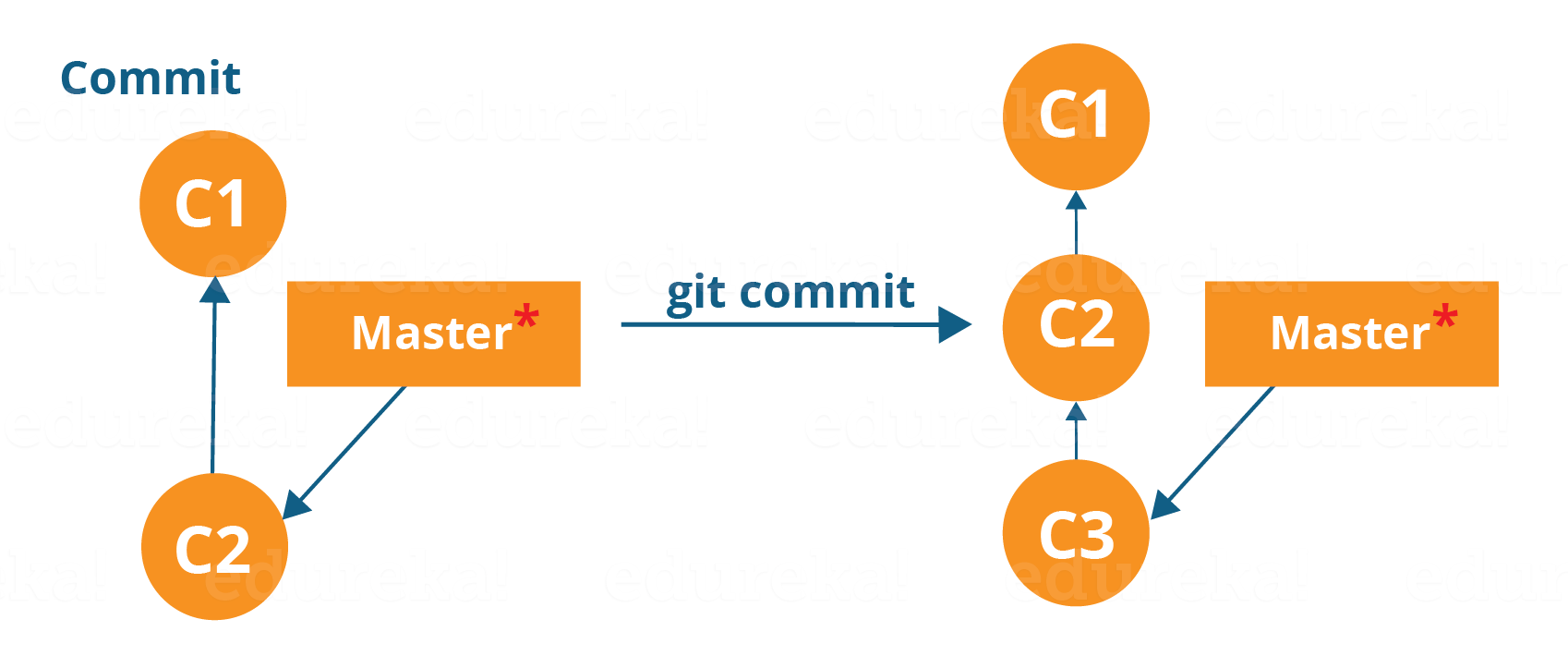
Let me demonstrate the **git add** for you so that you can understand it better.

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:

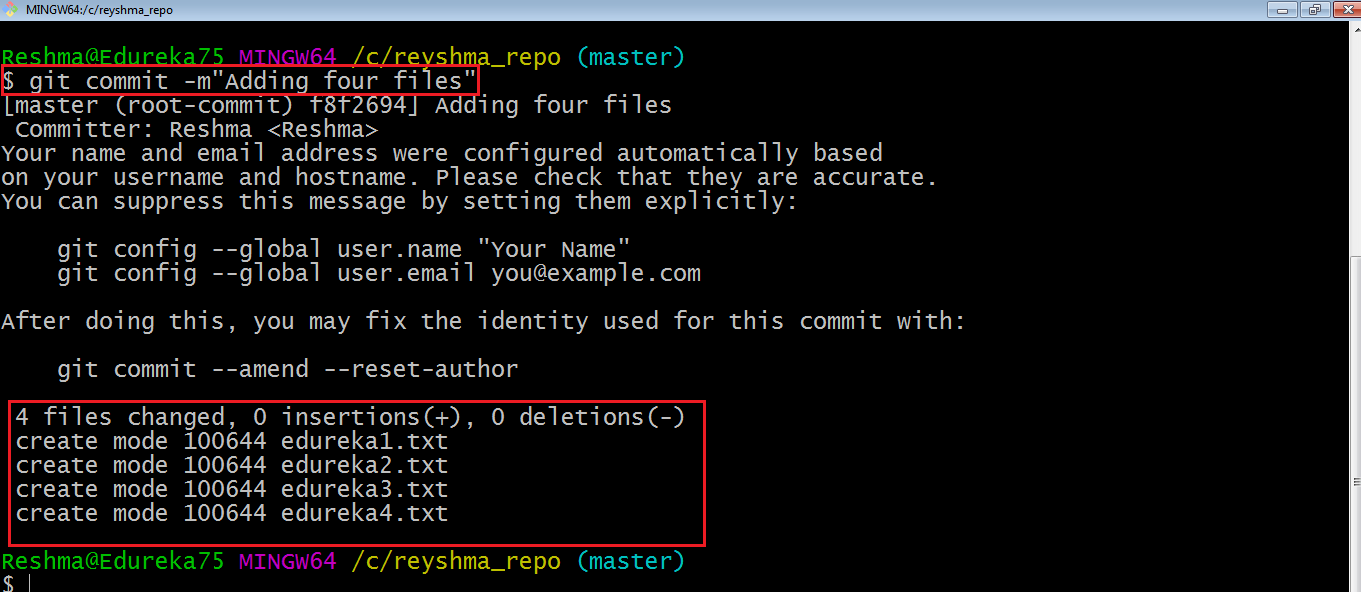
**git commit**

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.

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

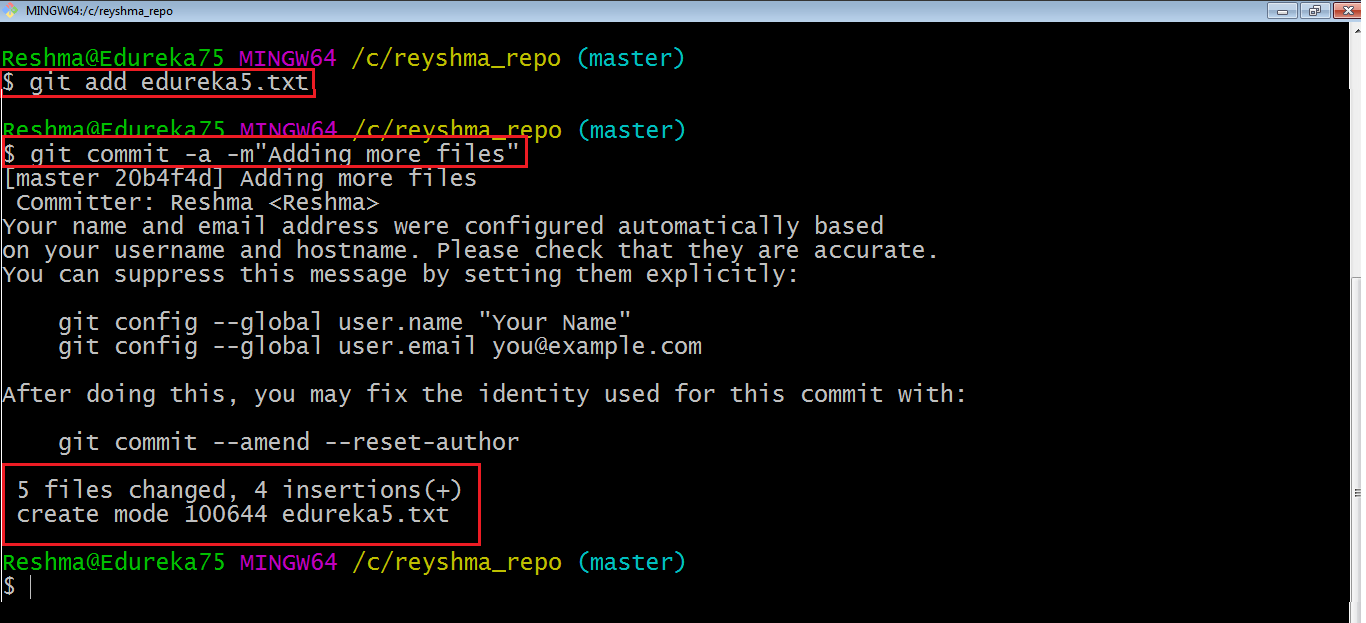
**git commit -a**

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.

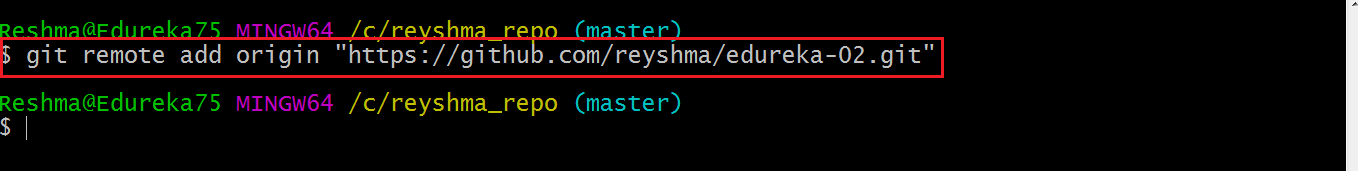
**add -A**. This command will add all the files to the index which are in the directory but not updated in the index yet.

**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:

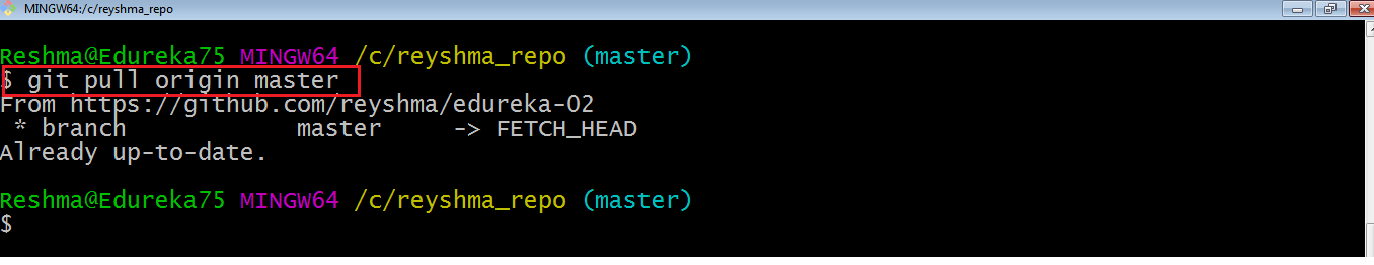
**git remote add origin <link of your central repository>**



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:*

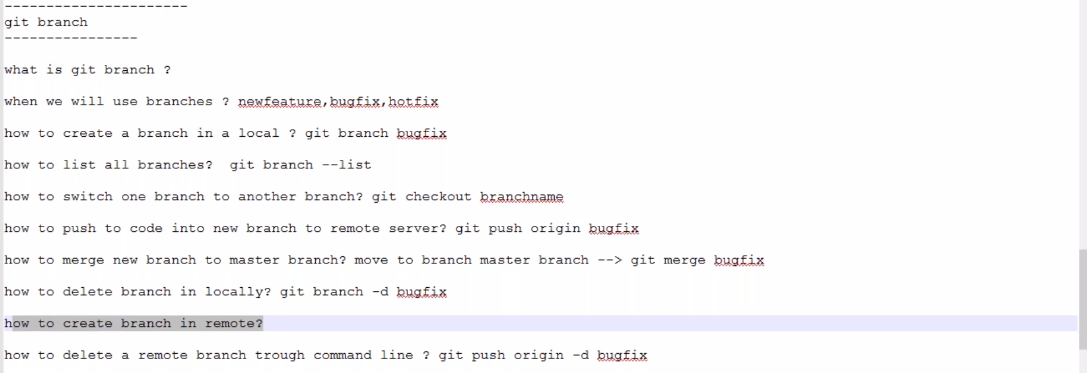
***git pull origin <branch-name>***

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.

**Merge Conflicts:**

Two members are working same file, at same time one person modified and committed, after another person modified and trying to pushing same file. In this merge conflicts occurred, git is inform to take the latest file using pull command, after that open merge conflict file in editor and removed un wanted character and save it, after push it will push without any error.

Branching:



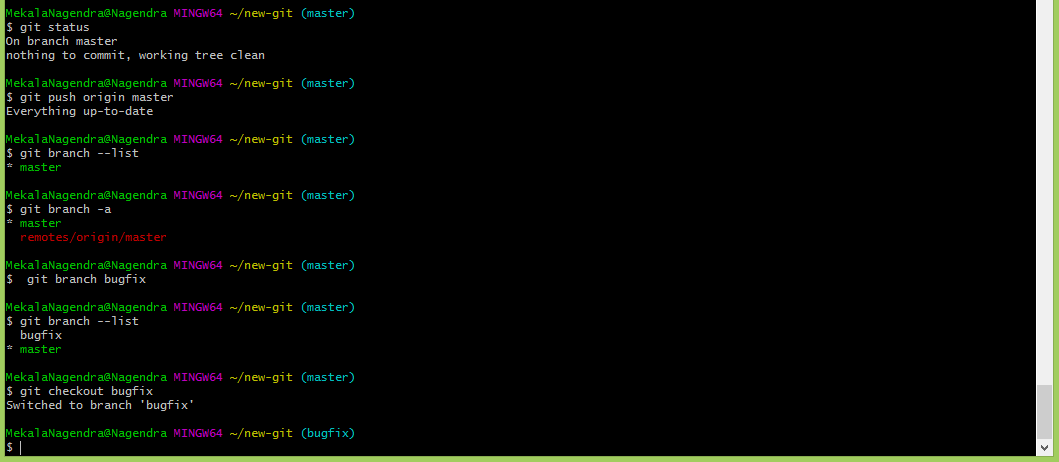
Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes

Git branch is basically and independent line of development, we can take advanges of branching when working on new feature or bug fixes.

git branch –list // it will show only local branches

git branch –a // it will show local and remote branches

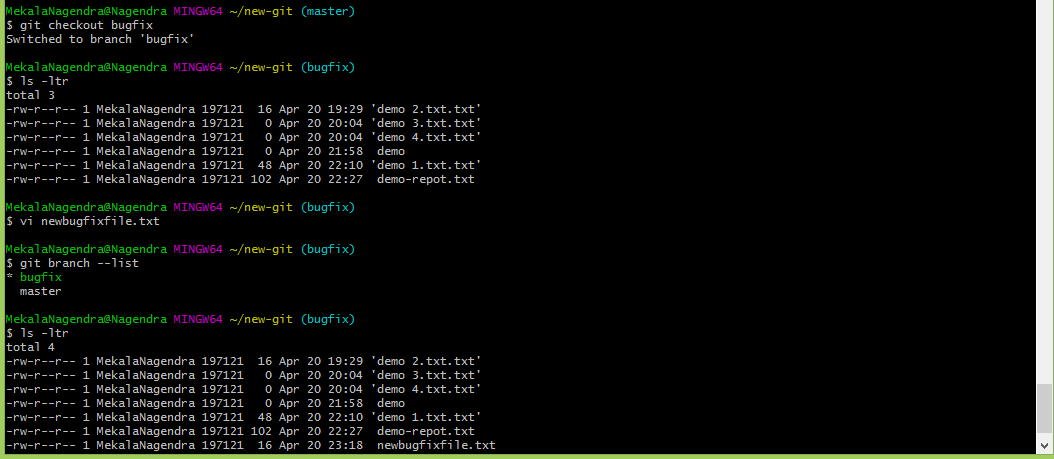
git branch bugfix // creating new branch name bugfix

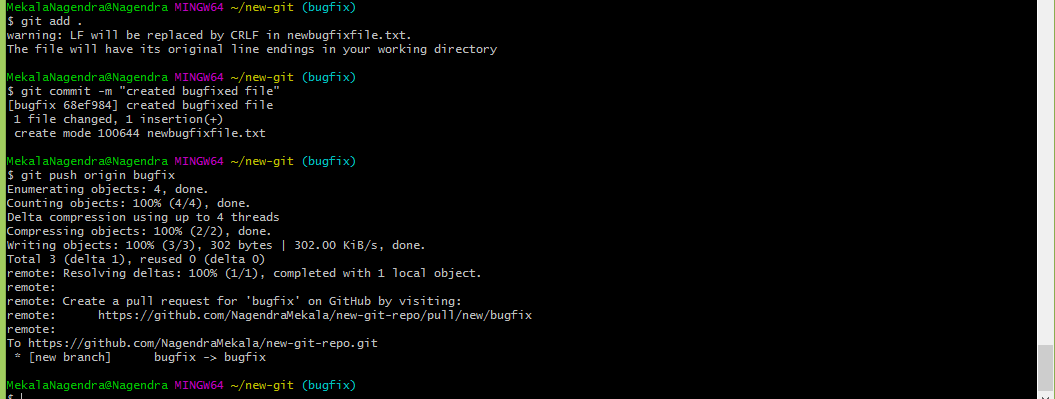


git checkout bugfix // switch to new branch (bugfix)

Created new file in bugfix branch and pushed to remote repository

git push origin bugfix // push the chages into remote

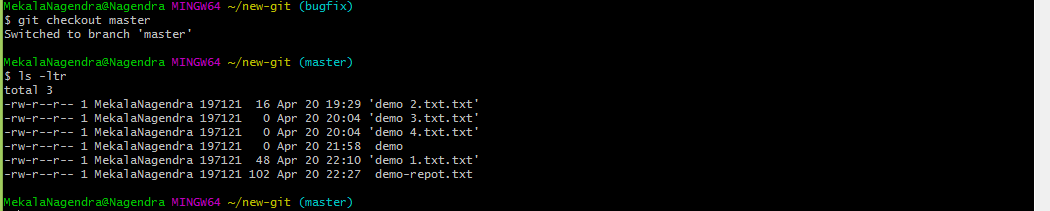




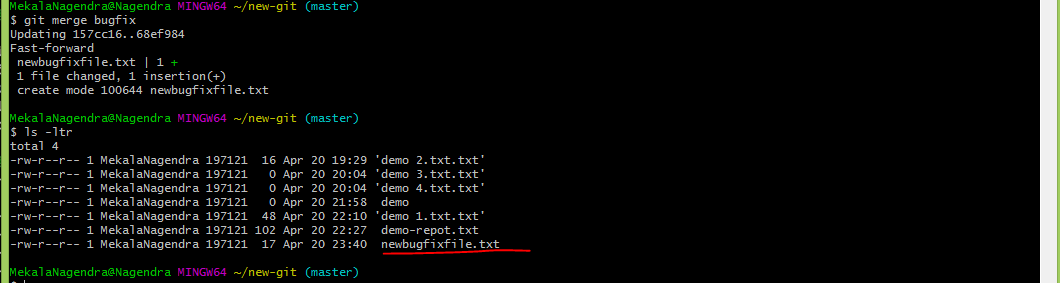
**Merger to master branch**

git checkout master // **Switched to master branch**

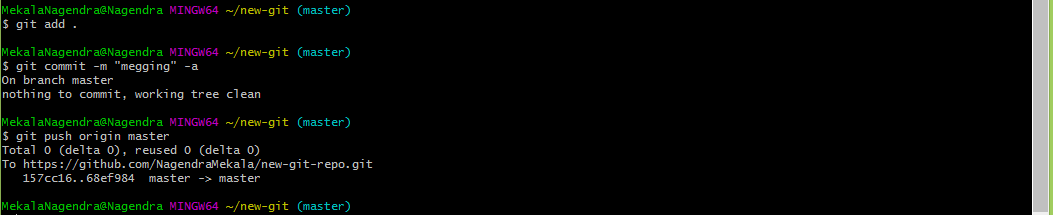
ls -ltr



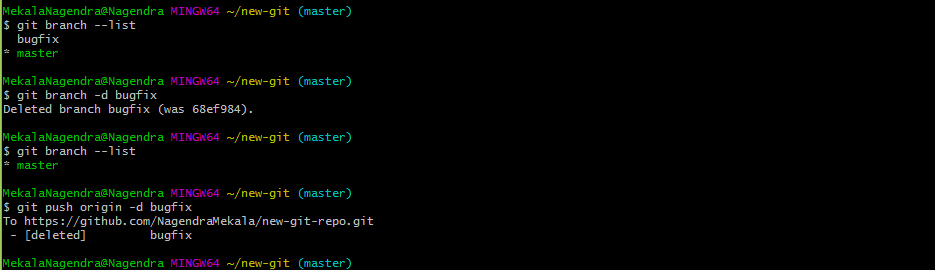
git merge bugfix // merged bug fixed branch to master branch



**Pushing changes to master branch after changes done**

****

**Deleting the branch after merged from remote/local repository**

****

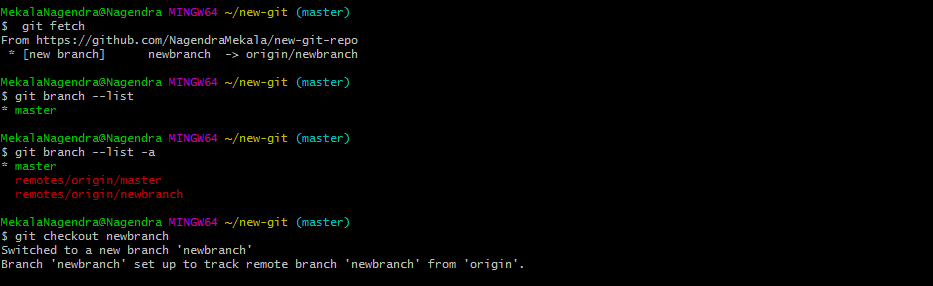
**How to pull remote branch already created**

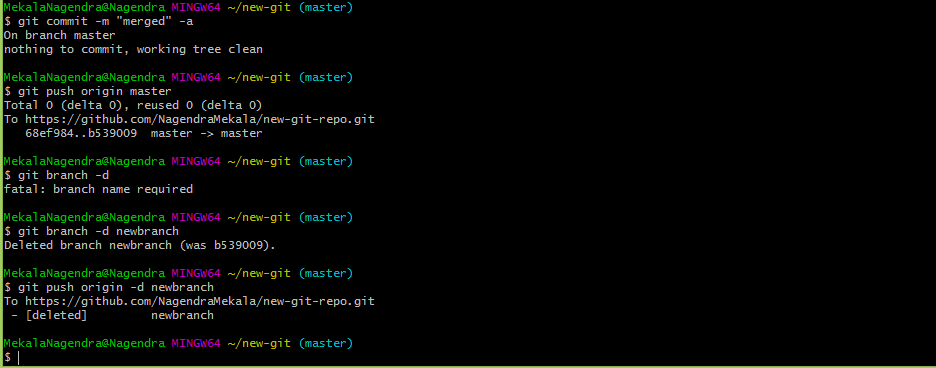
Create one branch on github

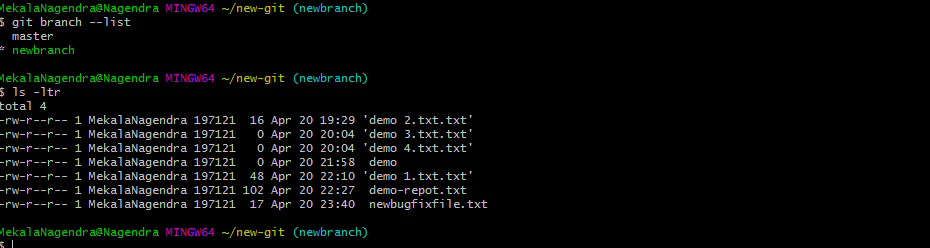
git fetch // fetch the remote branches to local

git branch –list

git checkout newbranch

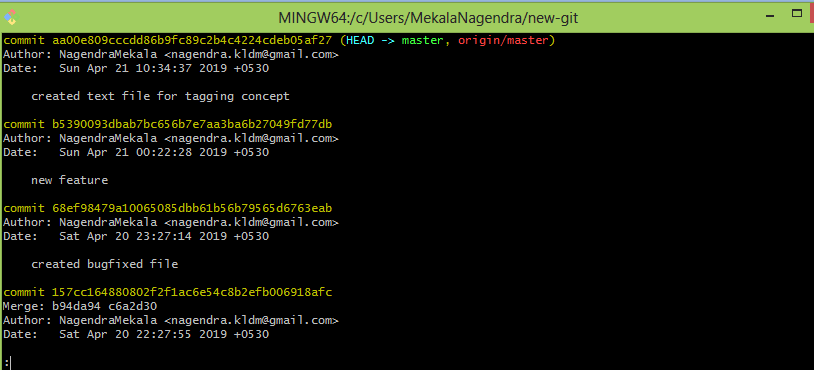
****

****

****

**Git log**

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

****

**Tags**

Tag operation allows giving meaningful names, allow you to identify specific release versions of your code. You can think of a tag as a branch that doesn't change

Git tag mainly for to release particular version of code (project) after completion development.

Tags two types

* Lightweight:

Lightweight tags are essentially 'bookmarks' to a commit, they are just a name and a pointer to a commit, useful for creating quick links to relevant commits

* Annotated :

Annotated tags store extra meta data such as: the tagger name, email, and date. This is important data for a public release

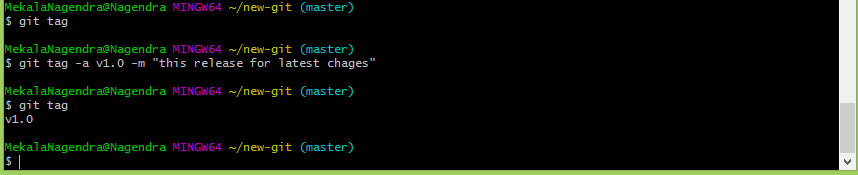
Annotated tags are stored as full objects in the Git database. To reiterate, They store extra meta data such as: the tagger name, email, and date. Similar to commits and commit message

**Annotated tag**

By default current committed will be taken for tagging.

git tag -a v1.0 -m "this release for latest chages"

git tag



git show

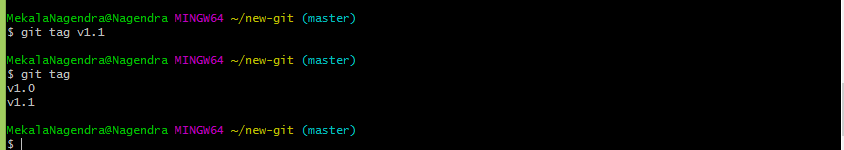


**Lightweight tag:**

git tag v1.1

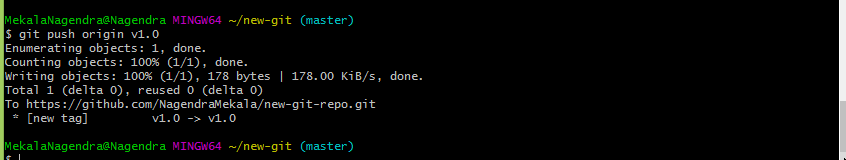
git tag

git tag



**Push tags:**

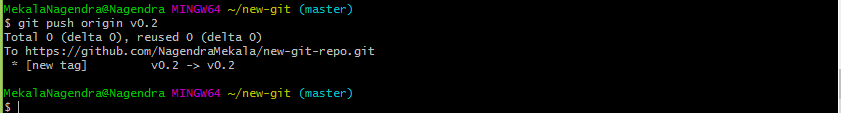
git push origin v1.0

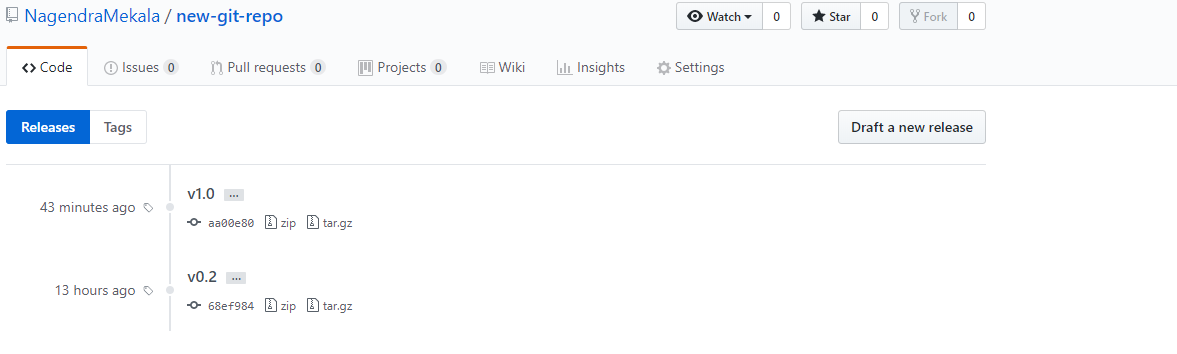


**Tagging particular commit**

git tag v0.2 68ef98479a10065085dbb61b56b79565d6763eab

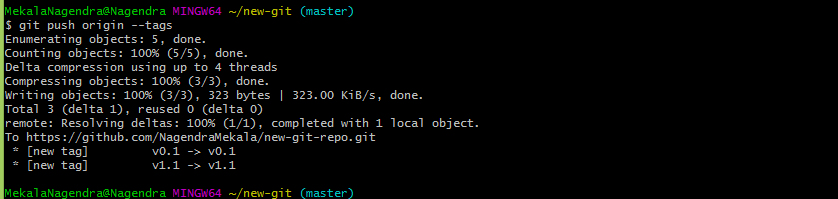
git push origin v0.2

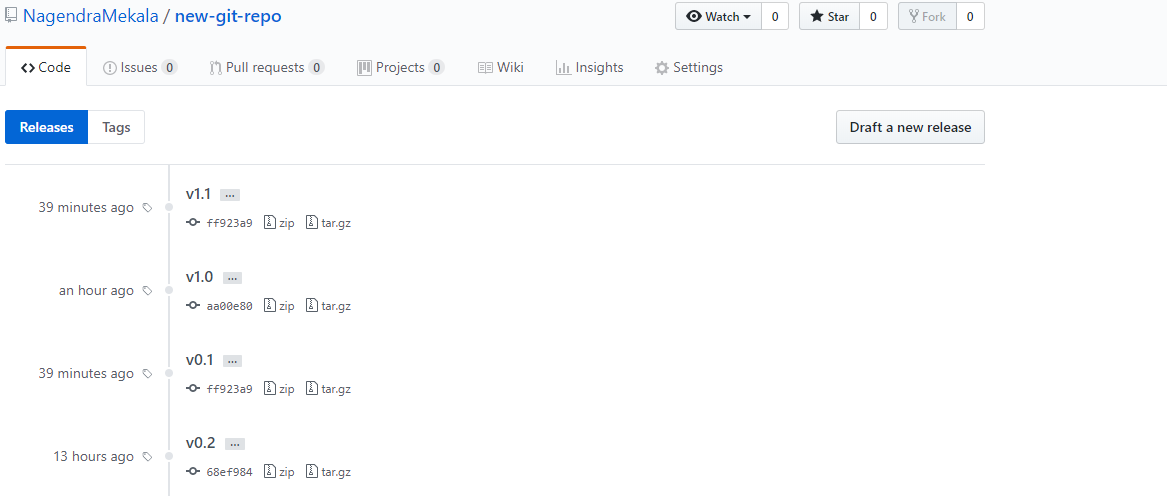




**Push all tags at time:**

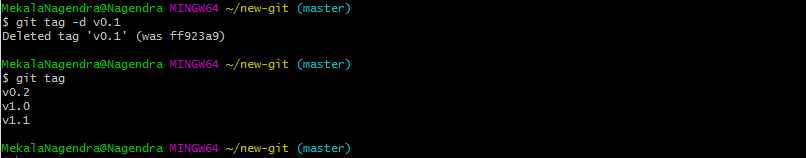
git push origin --tags





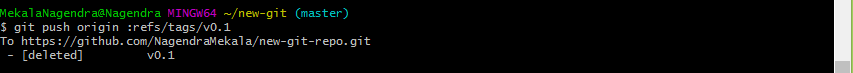
**Delete tags from local:**

git tag –d v0.1



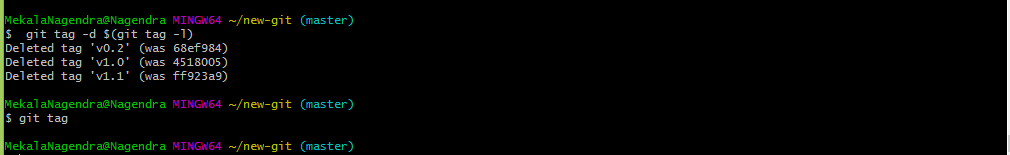
**Delete tag from repository:**

git push origin :refs/tags/v0.1

****

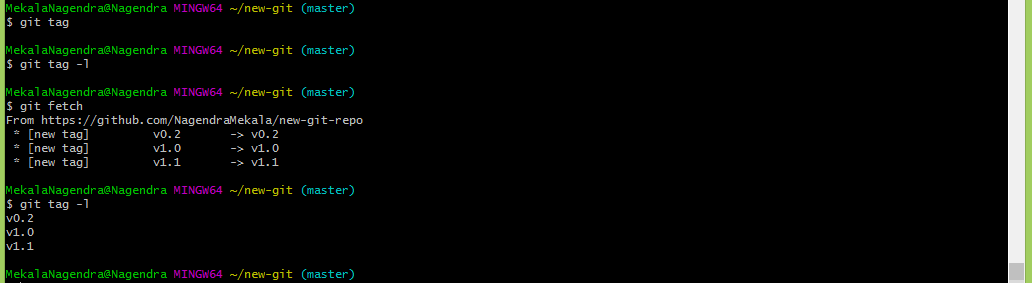
**Delete all tags from local:**

git tag -d $(git tag -l)



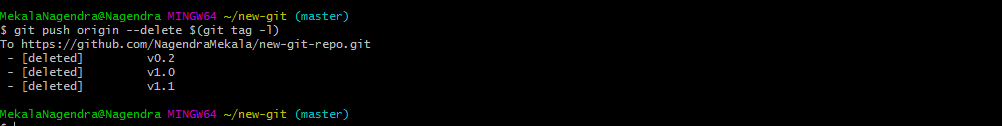
Revert back tags from remote, deleting tags in local

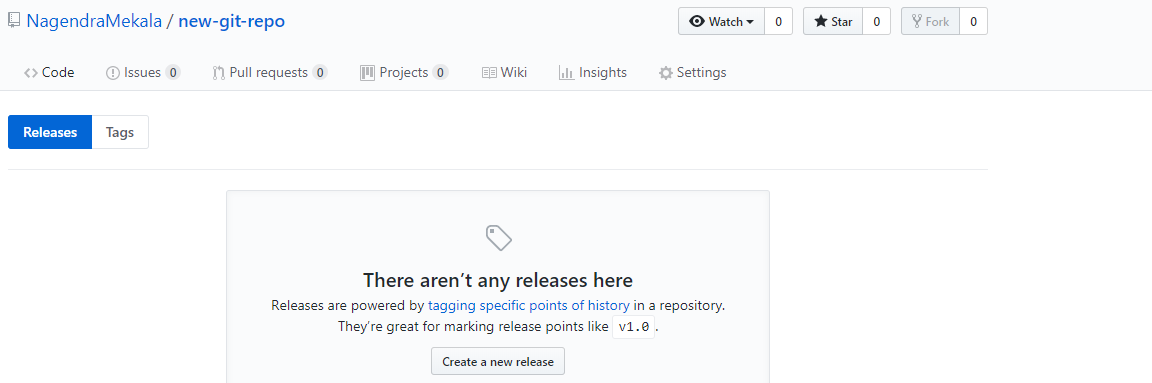
git fetch



**Delete all tags from remote**

git push origin --delete $(git tag -l)

****

****

**Difference between git merge vs rebase.**

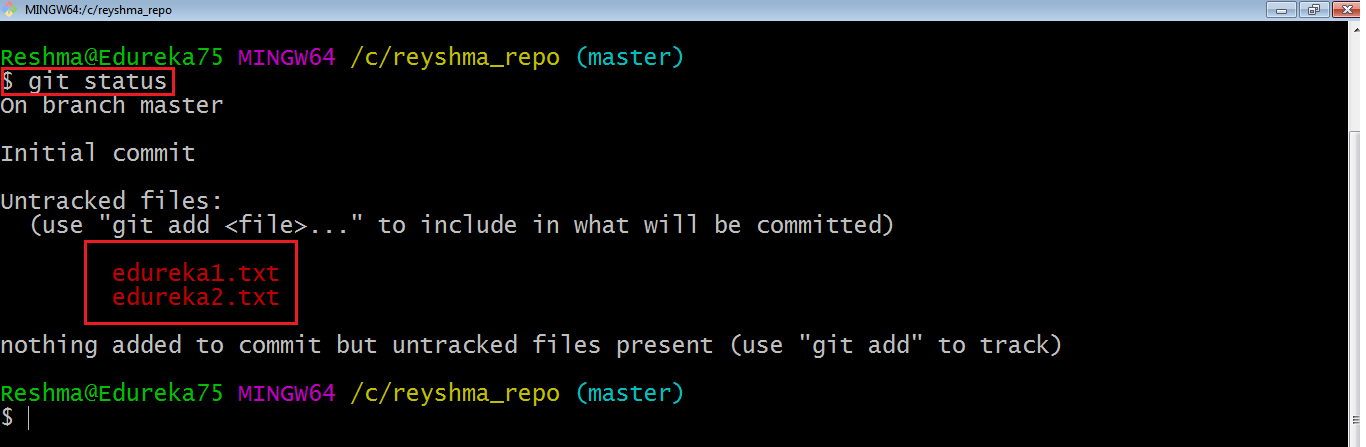
Both does the same thing (integrate branches) in slightly different way.

The major benefit of rebasing is that you get a much cleaner project history. First, it eliminates the unnecessary merge commits required by git merge.

**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:

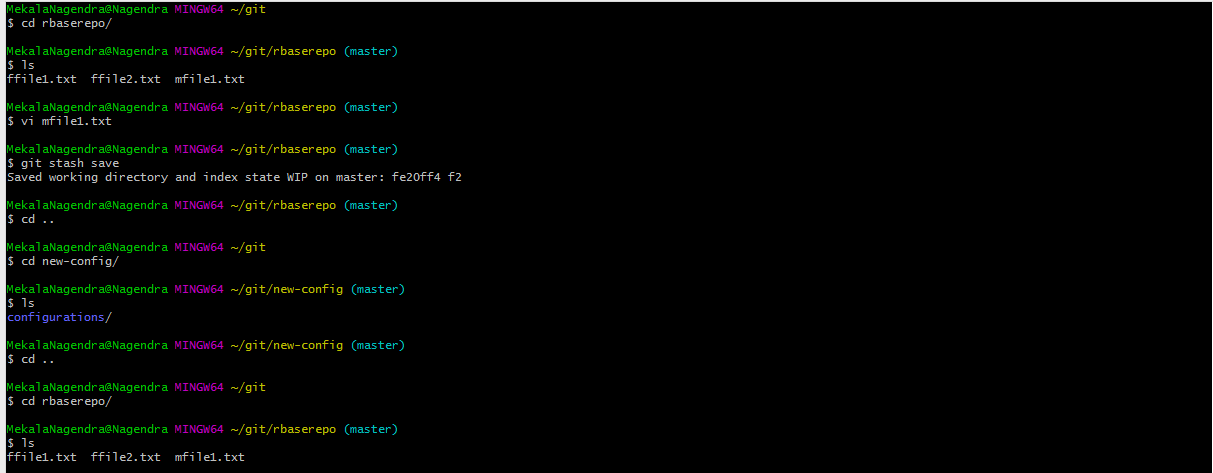


This shows that I have two files which are not added to the index yet. This means I cannot commit changes with these files unless I have added them explicitly in the index.

### ****git stash [**** store (something) safely and secretly in a specified place ]

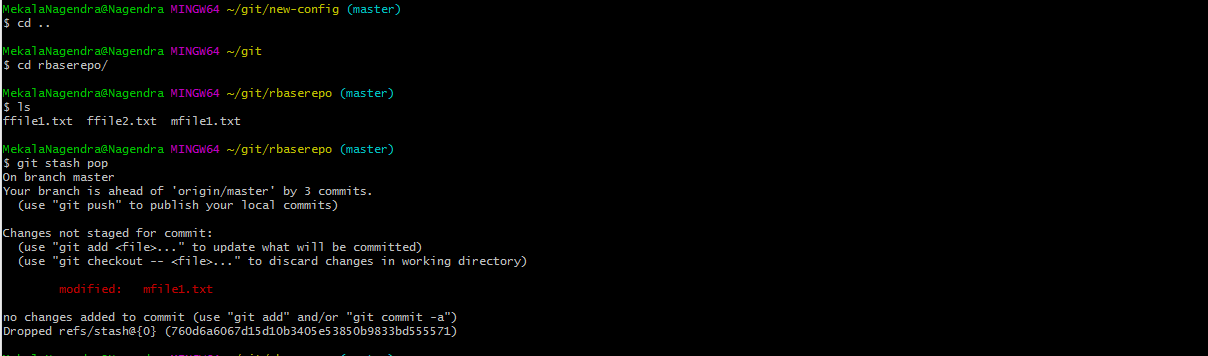
**Usage: git stash save**

This command temporarily stores all the modified tracked files.



**Usage: git stash pop**

This command restores the most recently stashed files.



**Delete specific stash**

To delete a specific stash we use the git stash drop stash@{id} command.

$ git stash save "new feature"

Saved working directory and index state On master: new feature

HEAD is now at f066f07 initial commit

$ git stash list

stash@{0}: On master: new feature

$ git stash drop stash@{0}

Dropped stash@{0} (d1b5d4fbabaf246d8436dd038fdca23cc9a90a29)

**Delete all stashes**

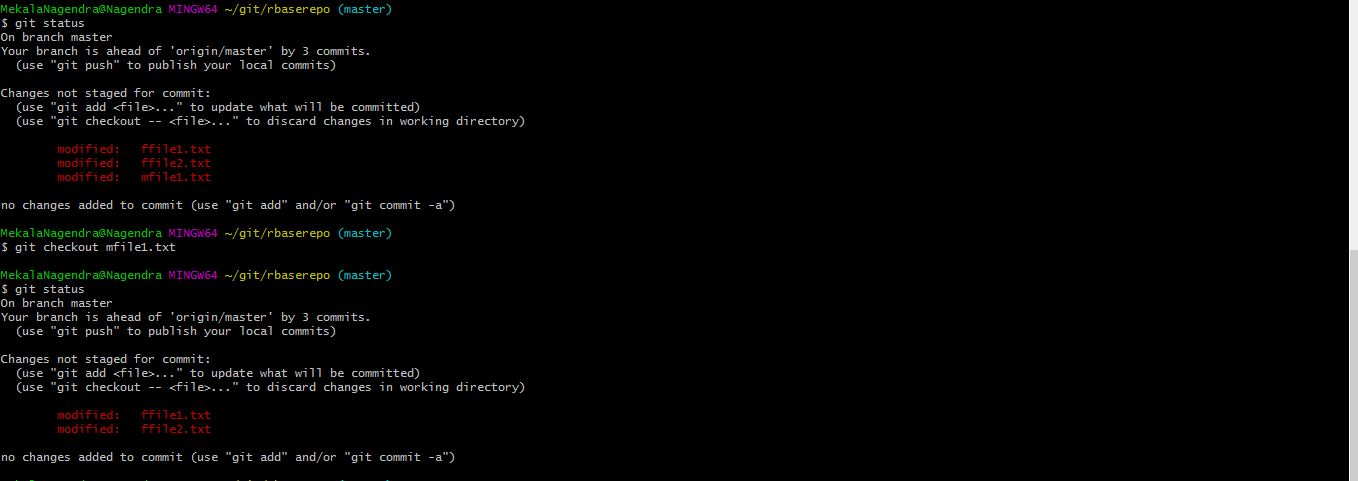
To delete all stashes we use the following command.

$ git stash clear

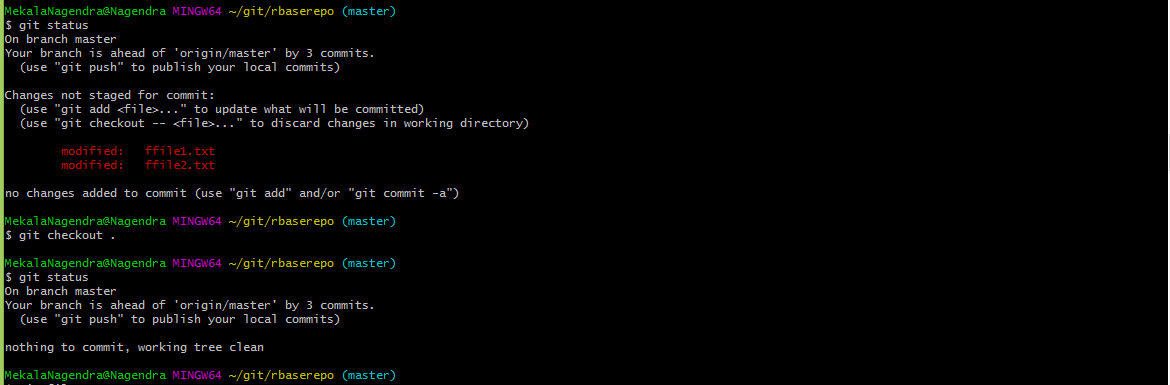
**Git undo operation**

* Git checkout is used for switching one branch to other also local working directory undo changes before commitng
* git checkout -- <file>..." to discard changes in working directory
* git checkout <filename> //for specific file in repo
* git checkout . // for all files undo in current repo
* This will not make changes to the commit history.
* Has potential to overwrite files in the working directory.

**git checkout mfile1.txt**

****

**git checkout .**

****

**Git reset**

'Git Reset' resets your index as well as the working directory to the state of your last commit.

Operation perform: After committing and before pushing command

**soft: only changes HEAD, but doesn’t change staged files in index or working files**

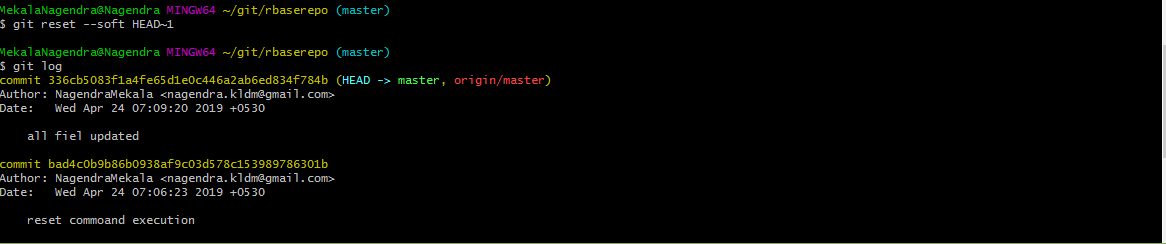
**last commit deleted but file is located index and staging are**

****

**After reset**

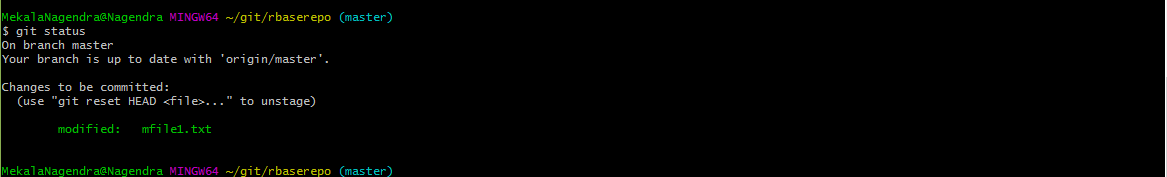
**git reset –soft HEAD~1**

**we can give up to head no 5,**

****

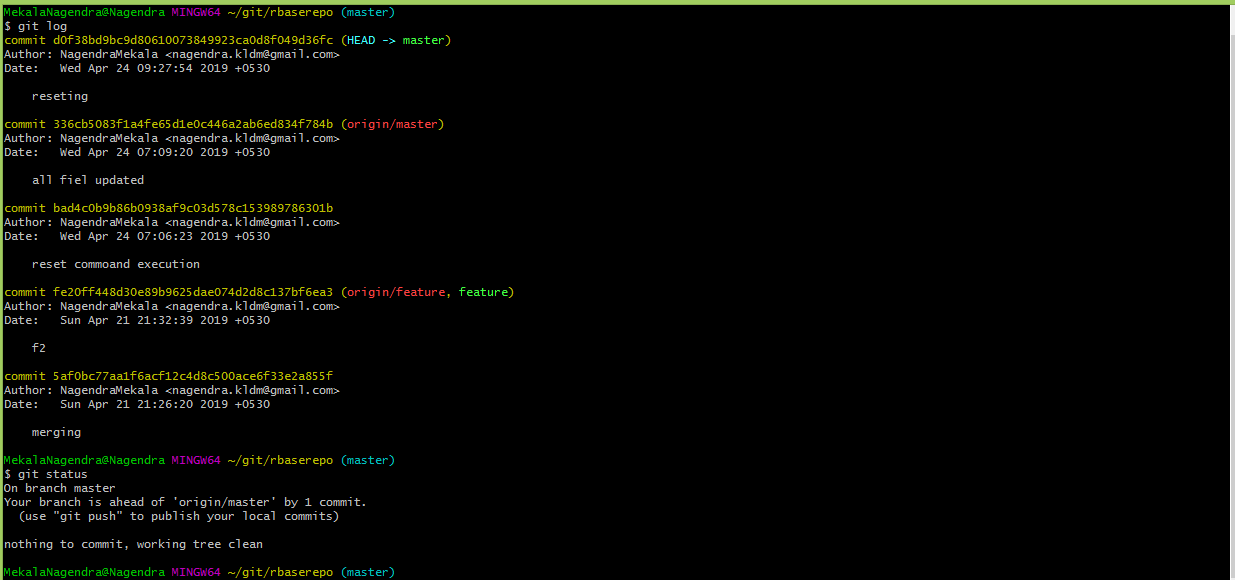
**Staging are not deleted –soft option**

**Just we can commit directly**

****

**Git mixed**

**Comment and staging area deleted.**

****

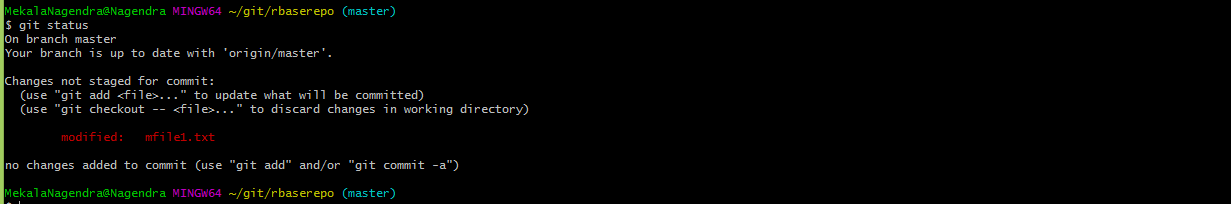
**After**

**git reset --mixed HEAD~1**

****

**After reset git status command**

If in index area the file will be green color. In below file is red color so it is not staging area.

****

**We have to do**

**git add .**

**git commit –m “reset” –a**

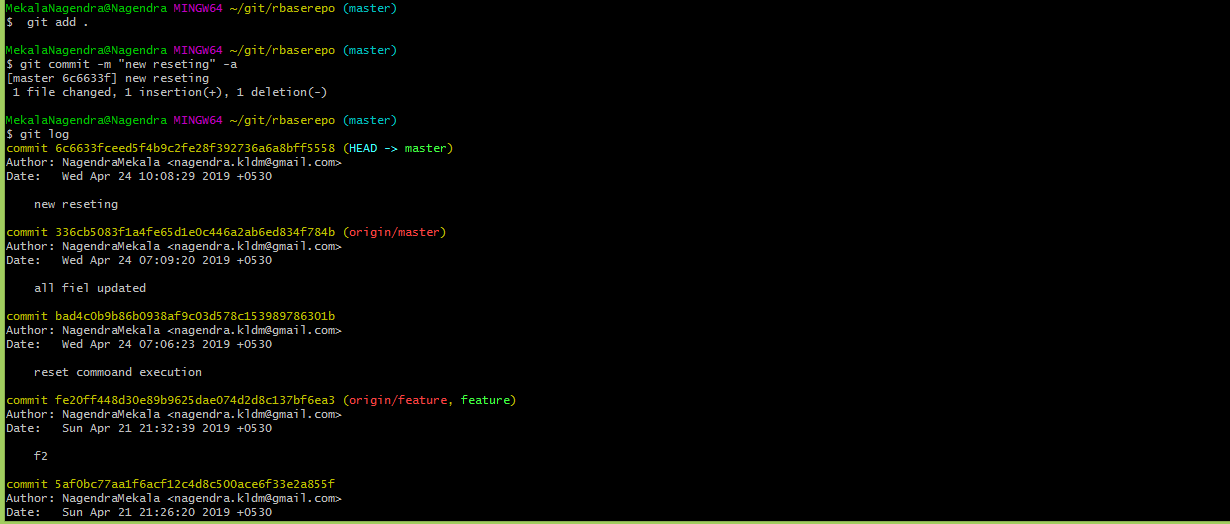
**Note:**

If use –soft, we directly commit the message

If use –mixed , we have to perform add and commit operations

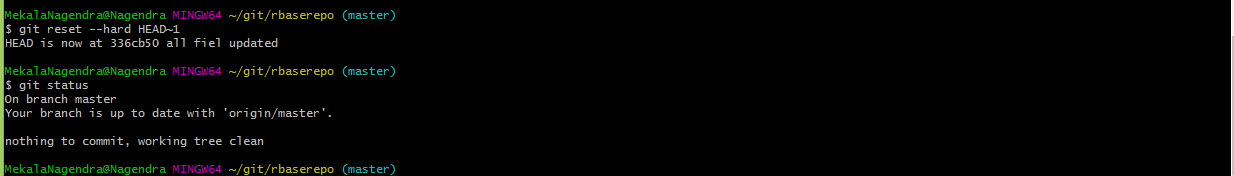
**Git hard**

**Comment message, staging (index) area and file (working copy) will be deleted from local repository also.**



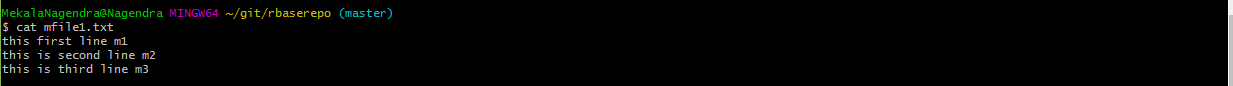
**After reset command**

git reset --hard HEAD~1

****

****

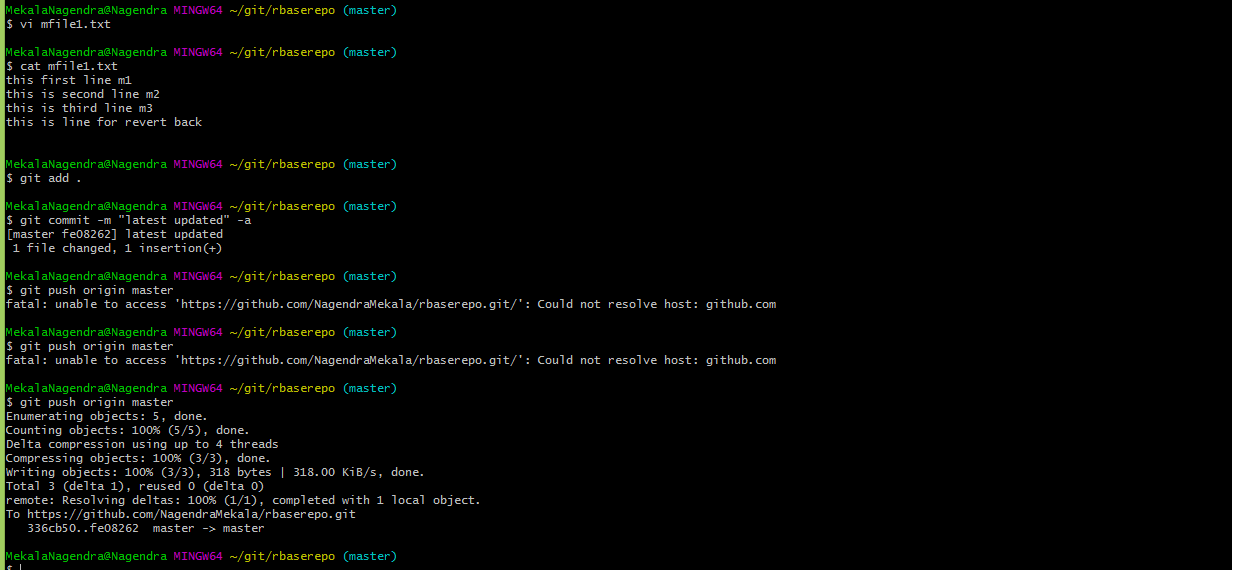
**Updated line to be deleted**

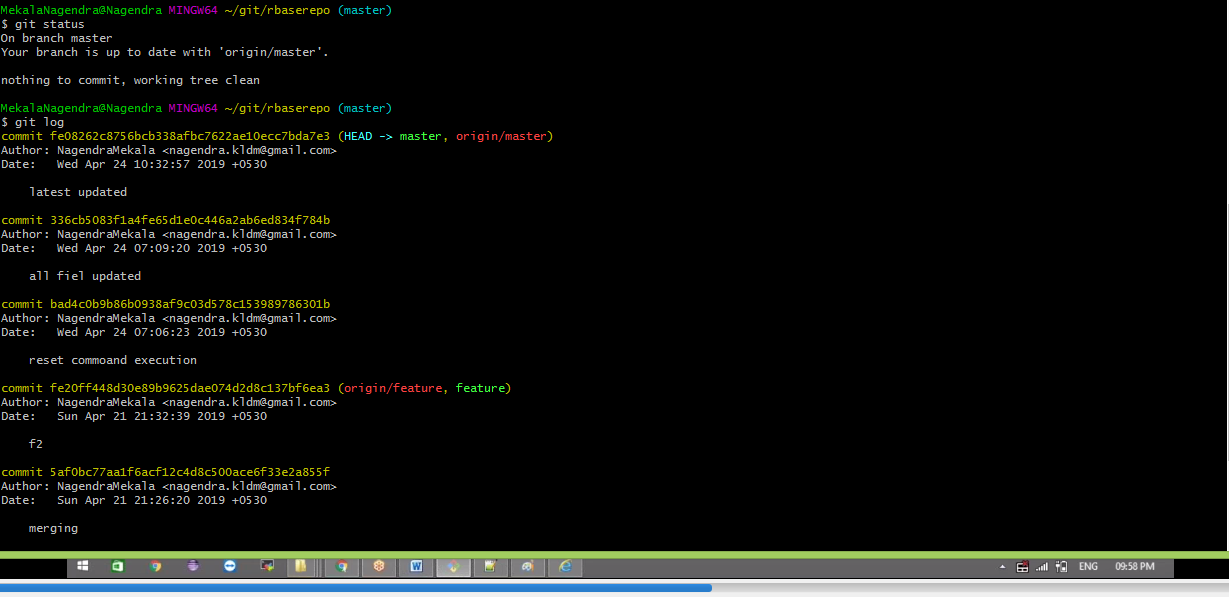
****

**Revert back**

**Operations perform :** After pushing file into central repository

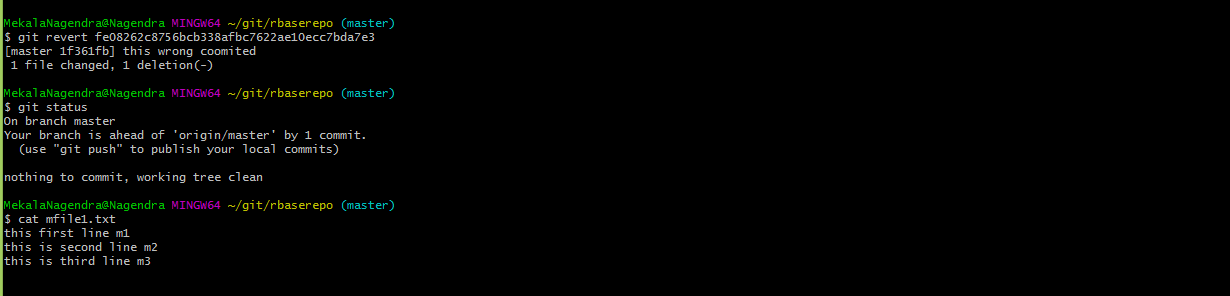
**Before git revert command file had to be modified and pushing central repo**

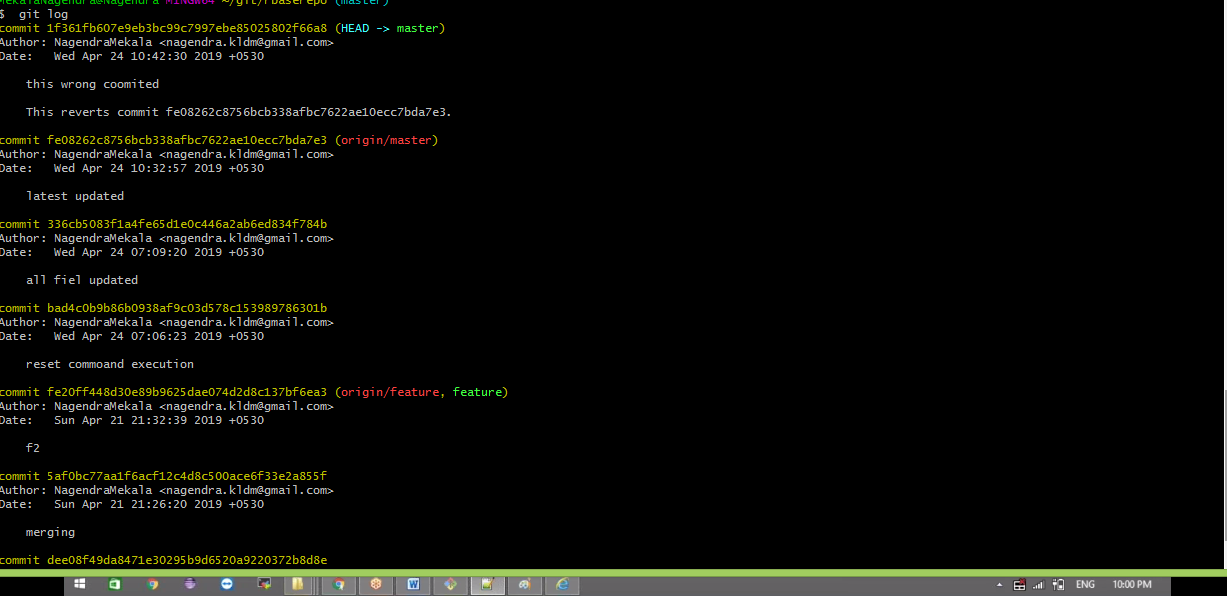
****

****

**Revert changes in central repository**

git revert fe08262c8756bcb338afbc7622ae10ecc7bda7e3

****

****

A git pull is what you would do to bring a local branch up-to-date with its remote version, while also updating your other remote-tracking branches.

git pull = git fetch + git merge.

Git pull command

Git pull basically equal to git fetch and git merge command.

When git pull command executed actually two commands are executed

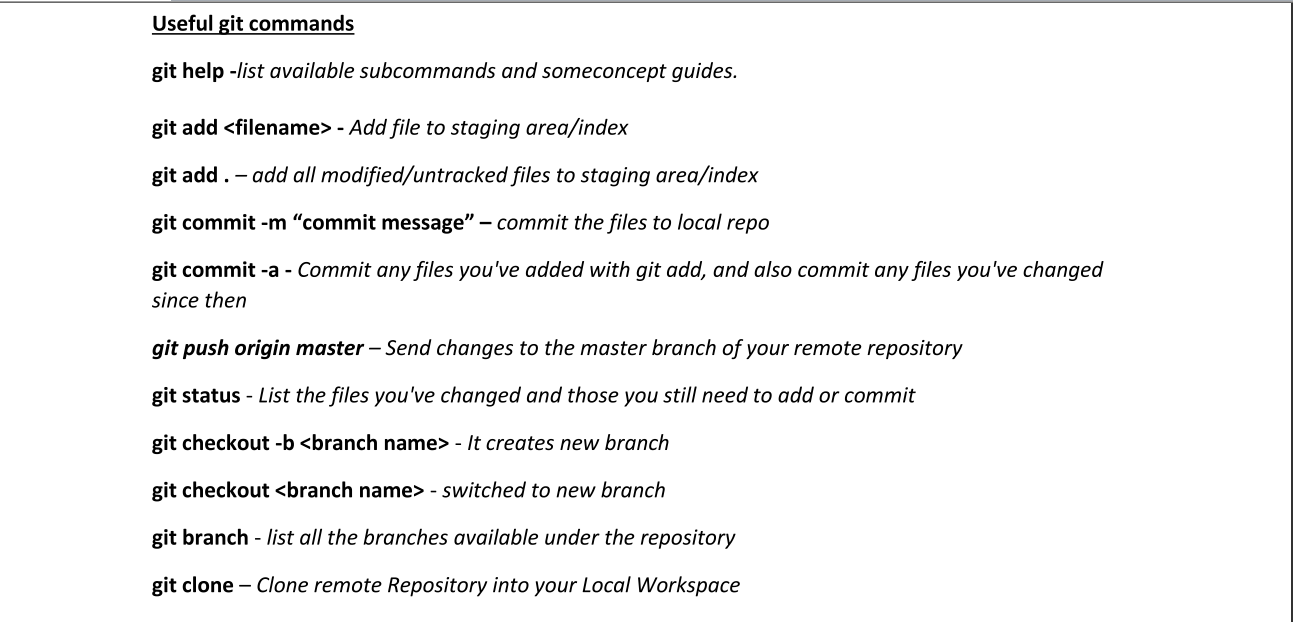
Git pull merge conflicts my occurred.

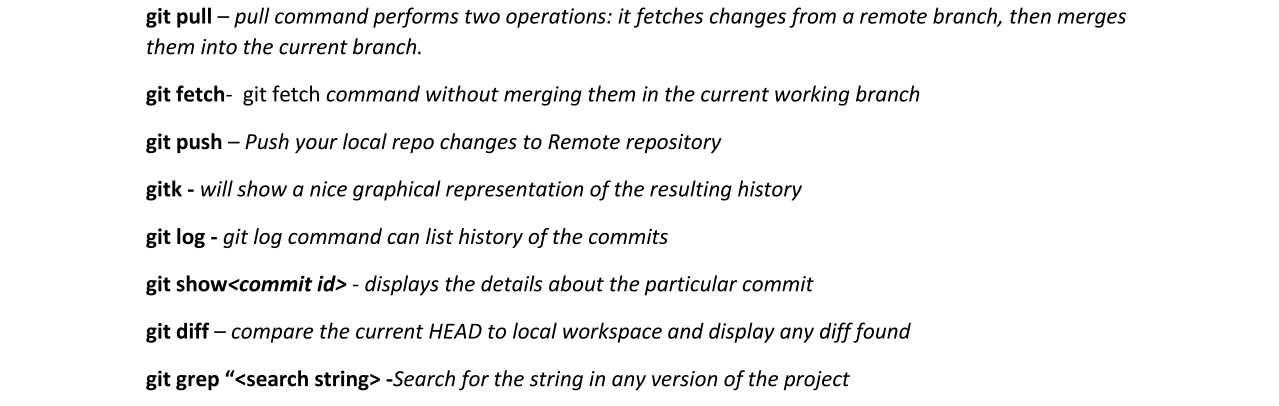
Git fetch there is no merge conflicts occurred.

git pull downloads the changes and merges them into your current branch

**git fetch** really only downloads new data from a remote repository - but it doesn't integrate any of this new data into your working files. Fetch is great for getting a fresh view on all the things that happened in a remote repository.

Fetch only downloads the data from the repository it does not integrate with the files you're currently working with. It only gives a view of all the things happened in the remote repository. Fetch is considered harmless as it never manipulates or spoils anything. You can fetch as many times as you want without disturbing your current working files.

****

****

[**https://www.javapedia.net/GIT-Interview-Questions#qanda2170**](https://www.javapedia.net/GIT-Interview-Questions#qanda2170)