Contents

[What is version control system? 4](#_Toc477448511)

[Basic Features /terms in VCS 4](#_Toc477448512)

[Basic Setup 4](#_Toc477448513)

[Basic Actions 4](#_Toc477448514)

[Advanced Actions 4](#_Toc477448515)

[Distributed Version Control(Git) 5](#_Toc477448516)

[New Terms in Distributed version control 5](#_Toc477448517)

[Advantages in distributed version control 5](#_Toc477448518)

[Git Architecture 5](#_Toc477448519)

[Installation of git bash on local machine 6](#_Toc477448520)

[Setup and configuration of remote and local git 6](#_Toc477448521)

[Creating remote git repository as github 6](#_Toc477448522)

[Cloning remote git repository. Command is: - git clone <remote url> 7](#_Toc477448523)

[Creating local git repository. Command is: - git init 7](#_Toc477448524)

[Adding remote location. Command is: - git remote add origin <remote\_url> 8](#_Toc477448525)

[Configuring git for username and email. Command is: - git config <parameter> <value> 8](#_Toc477448526)

[Stages of file in Git 9](#_Toc477448527)

[Untracked 9](#_Toc477448528)

[Unmodified 9](#_Toc477448529)

[Modified 9](#_Toc477448530)

[Staged 9](#_Toc477448531)

[Commands for working with files in local git 9](#_Toc477448532)

[Create new directory. command is: - mkdir <dirName> 9](#_Toc477448533)

[Remove empty directory. Command is: - rmdir <directoryName> 10](#_Toc477448534)

[Remove directory, subdirectory and all files under it. Command is: - rm -rf <directoryName> 10](#_Toc477448535)

[Creating new files. Command is: - vim/vi <fileName>. 10](#_Toc477448536)

[Creating file and folder without using command line. 10](#_Toc477448537)

[Deleting files inside a folder. Command is: - rm <fileName> 10](#_Toc477448538)

[Renaming a file or folder. Command is: - mv <oldFileName> <newFileName> 10](#_Toc477448539)

[Displaying content of a file. Command is: - cat <fileName> 11](#_Toc477448540)

[Listing the files and directory inside a directory. Command is:- ls, ls -a, ls -l 11](#_Toc477448541)

[Command for working with staging 11](#_Toc477448542)

[Current status of staging area. Command is: - git status 11](#_Toc477448543)

[Adding single new file for commit. Command is: - git add <fileName> 12](#_Toc477448544)

[Adding multiple files in staging area. Command is: - git add .(dot character after space) 12](#_Toc477448545)

[Adding only the modified files in staging area. Command is: - git add -u 13](#_Toc477448546)

[Deleting files for commit. Command is: - git rm <fileName> 13](#_Toc477448547)

[Ignore files from being tracked: - .gitignore file(dot gitignore file) 14](#_Toc477448548)

[Command for committing 15](#_Toc477448549)

[Committing single file. Command is: - git commit <fileName> 15](#_Toc477448550)

[Committing multiple items. Command is: - git commit 15](#_Toc477448551)

[Committing without opening the editor. Command is: - Git commit -m “user comment” 16](#_Toc477448552)

[Commands related to viewing commit history in local git repository 16](#_Toc477448553)

[Displays all commits. Command is:- git log 16](#_Toc477448554)

[Displaying commit history in one line. Command is: - git log –oneline(double minus) 17](#_Toc477448555)

[Counting all commits. Command is: - git log –oneline | wc – 1 18](#_Toc477448556)

[Displaying short logs. Command is: - git shortlog 18](#_Toc477448557)

[Displaying no of commits short by author. Command is: - git shortlog -sne 18](#_Toc477448558)

[Viewing specific commit. Command is: - git show HEAD / git show <5 digit sha> 18](#_Toc477448559)

[Displaying difference between two commits. Command is git diff <sha 1>..<sha 2> 19](#_Toc477448560)

[Filtering commit history on text in commit message. Command is: - git log –grep <text> 20](#_Toc477448561)

[Other switches for displaying commit history 20](#_Toc477448562)

[Link for better understanding of logs 20](#_Toc477448563)

[Recovering Old files 21](#_Toc477448564)

[File updated in working directory but not added in staging. Git checkout <filename> 21](#_Toc477448565)

[Files added in staging area. Command is: - git reset HEAD <fileName> 21](#_Toc477448566)

[Recovering deleted files not staged. Command is: - git checkout <fileName> 22](#_Toc477448567)

[Recovering deleted file from staging area. Git reset HEAD <fileName> 22](#_Toc477448568)

[Understanding git diff, git diff –cached and git diff HEAD 23](#_Toc477448569)

[Git Reset. Command is: - git reset HEAD~<# from last commit> --soft/hard 24](#_Toc477448570)

[Git Revert. Command is: - git revert HEAD/HEAD~<#>/sha no of commit 25](#_Toc477448571)

[Working with branches. 27](#_Toc477448572)

[Displaying all branches in a repository. Command is: - git branch -r/--all/--list 27](#_Toc477448573)

[Creating new branch, Command is: - git branch <branchname> [<startpoint>] 27](#_Toc477448574)

[Checking out a new local branch. Git checkout <branchName> <startpoint> 28](#_Toc477448575)

[Creating branch and checking out. Git checkout -b <branchname> [<startpoint>] 28](#_Toc477448576)

[Deleting branch. Command is:- git branch -d/D <branchName> 28](#_Toc477448577)

[Renaming a branch. Command is :- git branch -m <oldBranchName> <newName> 28](#_Toc477448578)

[Filtering branches. Git branch –merged/--no-merged [<branchName>] 28](#_Toc477448579)

[Merging branches with no conflicts. Command is :- git merge <branchName> 29](#_Toc477448580)

[Merging branches with conflicts with command line editors. 30](#_Toc477448581)

[Merge Tool 30](#_Toc477448582)

[Default editor such as notepad or vim editor. 32](#_Toc477448583)

[Git Working with Remote 34](#_Toc477448584)

[Creating local repository and pushing 34](#_Toc477448585)

[Cloning from remote. Command is: - git clone <url> 36](#_Toc477448586)

[Displaying remote tracking branches. Command is: - Git remote -v 36](#_Toc477448587)

[Publishing local branch in remote. Command is: - Git push -u origin <local branch name> 36](#_Toc477448588)

[Inspecting a remote repository. Command is: - Git remote show origin 37](#_Toc477448589)

[Pushing changes in tracked branches. Command is: - Git push origin/origin HEAD/origin <name> 37](#_Toc477448590)

[Deleting remote branch. Command is: - Git push origin –delete <branchName> 38](#_Toc477448591)

[Renaming remote branch name 38](#_Toc477448592)

[Fetching remote branch without updating local. Command is: -Git fetch/git fetch origin <branch name> 38](#_Toc477448593)

[Pulling all changes from remote in local. Command is: - Git pull origin /git pull origin <branchname> 39](#_Toc477448594)

[Git Pull with rebasing. Command is: - Git pull -rebase origin 40](#_Toc477448595)

[Git Tagging 41](#_Toc477448596)

[Git Tag 41](#_Toc477448597)

[Git annotated tags. Git tag -a <verison no> / git tag -a <verison no > -m <message> 41](#_Toc477448598)

[Git lightweight tags. Git tag <versionNo> 42](#_Toc477448599)

[Git tagging later. Git tag <tagName> <5,6 digit of sha # of commit> 42](#_Toc477448600)

[Displaying information of tag. Git show <tagNo> 42](#_Toc477448601)

[Pushing tags to remote. Git push origin –tags 43](#_Toc477448602)

[Git checkout specific tag. Git checkout -b <newBranchName> <tagNo> 43](#_Toc477448603)

[Git Stash 44](#_Toc477448604)

[Git stash / git stash -u 44](#_Toc477448605)

[Git stash list 44](#_Toc477448606)

[Git stash show [<stash>]/git stash show [<stash>] -p 44](#_Toc477448607)

[Git stash pop [--index] [<stash>] 45](#_Toc477448608)

[Removing single temporary stash. Git stash drop [<Stash>] 45](#_Toc477448609)

[Removing all temporary stash. Git stash clear 45](#_Toc477448610)

[Adding stash entry without removing it. Git stash apply [--indes] [<stash>] 45](#_Toc477448611)

[Creating branch from stashed changes. Git stash branch <branchname> <stashNo> 45](#_Toc477448612)

# What is version control system?

Helps you track your files over time. When you mess up you can easily get back to a previous working version.

Following are its advantages.

* **Backup and Restore.** Files are saved as they are edited, and you can jump to any moment in time.
* **Synchronization.** Let people share files and stay up-to-date with the latest version.
* **Undo changes.** Throw away your changes and go back to the “last known good” version in the database.
* **Track Changes**. As files are updated, you can leave messages explaining why the change happened (stored in the VCS, not the file).

# Basic Features /terms in VCS

## Basic Setup

* **Repository (repo)**: The database storing the files.
* **Server**: The computer storing the repo.
* **Client**: The computer connecting to the repo.
* **Working Set/Working Copy**: Your local directory of files, where you make changes.
* **Trunk/Main**: The primary location for code in the repo. Think of code as a family tree — the trunk is the main line.

## Basic Actions

* **Add**: Put a file into the repo for the first time, i.e. begin tracking it with Version Control.
* **Revision**: What version a file is on (v1, v2, v3, etc.).
* **Head**: The latest revision in the repo.
* **Check out**: Download a file from the repo.
* **Check in**: Upload a file to the repository (if it has changed). The file gets a new revision number, and people can “check out” the latest one.
* **Check in Message**: A short message describing what was changed.
* **Changelog/History**: A list of changes made to a file since it was created.
* **Update/Sync**: Synchronize your files with the latest from the repository. This lets you grab the latest revisions of all files.
* **Revert**: Throw away your local changes and reload the latest version from the repository.

## Advanced Actions

* **Branch**: Create a separate copy of a file/folder for private use (bug fixing, testing, etc.). Branch is both a verb (“branch the code”) and a noun (“Which branch is it in?”).
* **Diff/Change/Delta**: Finding the differences between two files. Useful for seeing what changed between revisions.
* **Merge (or patch)**: Apply the changes from one file to another, to bring it up-to-date. For example, you can merge features from one branch into another.
* **Conflict**: When pending changes to a file contradict each other (both changes cannot be applied).
* **Resolve**: Fixing the changes that contradict each other and checking in the correct version.
* **Locking**: Taking control of a file so nobody else can edit it until you unlock it. Some version control systems use this to avoid conflicts.
* **Breaking the lock**: Forcibly unlocking a file so you can edit it. It may be needed if someone locks a file and goes on vacation (or “calls in sick” the day Halo 3 comes out).
* **Check out for edit**: Checking out an “editable” version of a file. Some VCSs have editable files by default, others require an explicit command.

# Distributed Version Control(Git)

Traditional version control helps you backup, track and synchronize files. Distributed version control makes it easy to share changes. Done right, you can get the best of both worlds: simple merging and centralized releases.

In a distributed model, every developer has their own repo but If desired, everyone can push changes into a common repo.

## New Terms in Distributed version control

* **push**: send a change to another repository (may require permission)
* **pull**: grab a change from a repository

## Advantages in distributed version control

* **Everyone has a local sandbox.** You can make changes and roll back, all on your local machine.
* **It works offline.** You only need to be online to share changes.
* **It handles changes well.** Distributed version control systems were built around sharing changes.
* **Branching and merging is easy.** Because every developer “has their own branch”, every shared change is like reverse integration.
* **Git has integrity.** Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it’s impossible to change the contents of any file or directory without Git knowing about it.

# Git Architecture

Below is the architecture of git. It gives a detail about the flow of files in Git.

There is working directory in our local file system. We make some changes i.e. add/modify or deleting files here. Our changes are pushed in staging area. We commit the changes in local repository. Finally, we push the changes to remote for other to access it.



# Installation of git bash on local machine

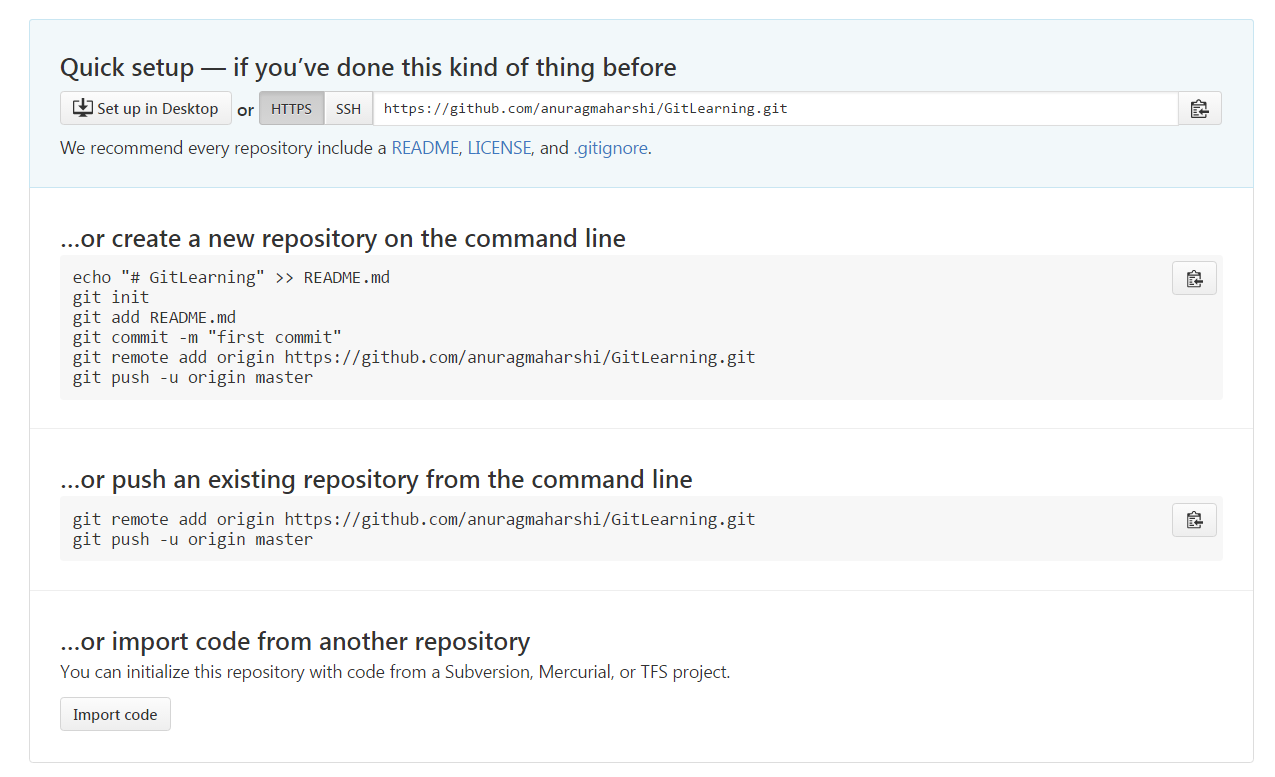
* Download git bash exe from <https://git-scm.com/downloads>
* Install the software

# Setup and configuration of remote and local git

Below is the list of commands for working with Git in our local system.

## Creating remote git repository as github

* Navigate to <https://github.com/>
* Click on Sign up to create a new account or sign in if you have an account.
* Click start a new Project and enter repository name and click on create repository.
* Below will be the page after that.

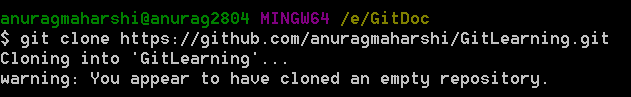


## Cloning remote git repository. Command is: - git clone <remote url>

We need to clone a remote repository in our local to work it. This will download all the files, commits meta data related to repository in our local.

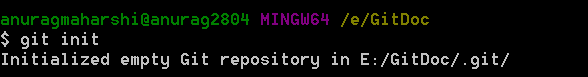
Eg: git clone https://github.com/anuragmaharshi/TestCentral.git

* Open git bash (C:\Program Files\Git\git-bash.exe)
* Goto folder in your local driver where you need to create repository. Eg E:\GitLearning
* Use cd command to navigate eg cd e:
* Cd gitlearning
* Enter git command to clone. git clone https://github.com/anuragmaharshi/TestCentral.git



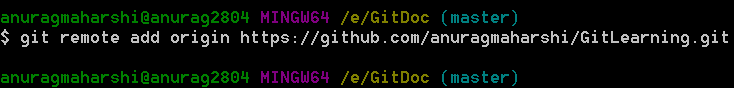
## Creating local git repository. Command is: - git init

Git init command will create a new empty repository on our local. We can now work with git locally.



## Adding remote location. Command is: - git remote add origin <remote\_url>

This command is used to for setting up remote url. Local git will coordinate with this url for pushing, pulling, fetching etc. This will be the central repository for storing local repository.

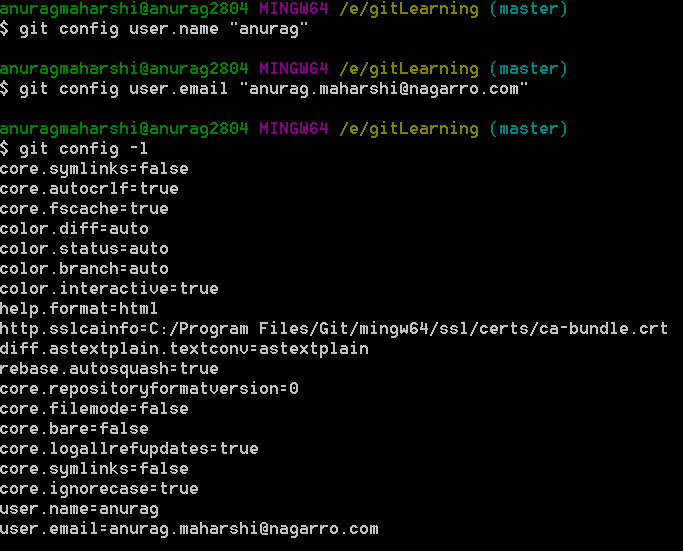


## Configuring git for username and email. Command is: - git config <parameter> <value>

We need to configure user name and user email id in Git. This information is published in commit and other activities. So before working on git this environment variables need to be configured.

Git config user.name “userName”

Git config user.email “user@test.com”

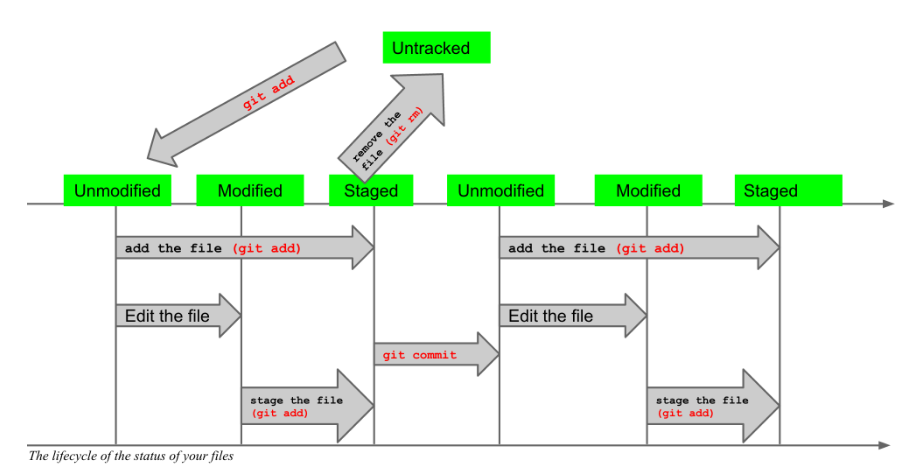


Git config –list and git config -l displays all the settings.

This setting is local to the repository. If user have multiple git system either he needs to set in each. There is also an option for global and system level config.

# Stages of file in Git

Every files and directory in git follow below life cycle.



## Untracked

When files are added for first time in git they are untracked. Git keeps track of all the files after they are committed once. So after new files are created, they must be committed for tracking changes.

## Unmodified

When user don’t update a file, its status in unmodified. Unmodified files are not committed.

## Modified

When user changes the name or content of a file, it is modified. Only tracked files can come in this status. Modified files are staged before it is committed.

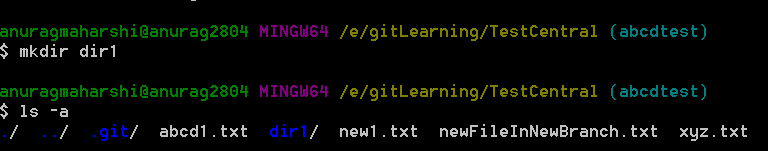
## Staged

After files are committed their status become Staged. It means the file in working directory is same as that in local repository.

# Commands for working with files in local git

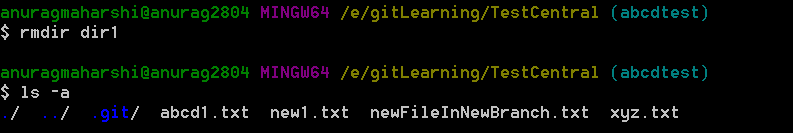
## Create new directory. command is: - mkdir <dirName>

Mkdir dir1



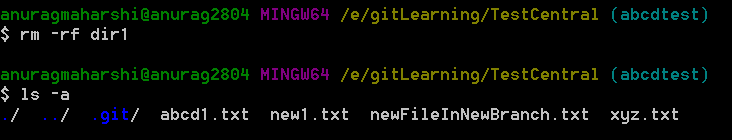
## Remove empty directory. Command is: - rmdir <directoryName>

Rmdir dir1



## Remove directory, subdirectory and all files under it. Command is: - rm -rf <directoryName>

Rm dir1



## Creating new files. Command is: - vim/vi <fileName>.

Vim or vi is the default editor of git bash. Files can be created using this editor.

To write contents in this editor use [Esc] key, [Insert Key].

After contents are written in the file, press esc and enter :wq to save and exit the editor.

Same editor can be used to editing the content of files.

## Creating file and folder without using command line.

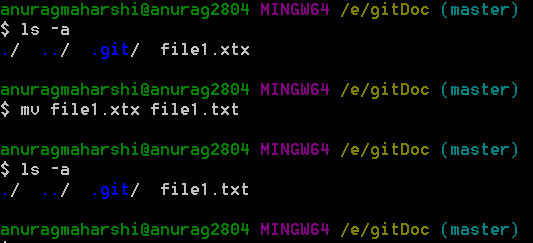
Directory and files can be created directly in the working directory. These files and folder will also be tracked.

## Deleting files inside a folder. Command is: - rm <fileName>

Rm file1.txt

## Renaming a file or folder. Command is: - mv <oldFileName> <newFileName>

Mv file1.txt file2.txt



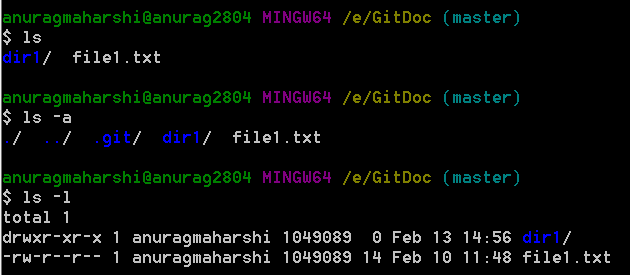
## Displaying content of a file. Command is: - cat <fileName>

Cat file1.txt



## Listing the files and directory inside a directory. Command is:- ls, ls -a, ls -l

* ls : It displays the list of files and directory
* ls -a : It displays all files and directory including with .(dot)
* ls -l : it displays files and directory along with different parameters

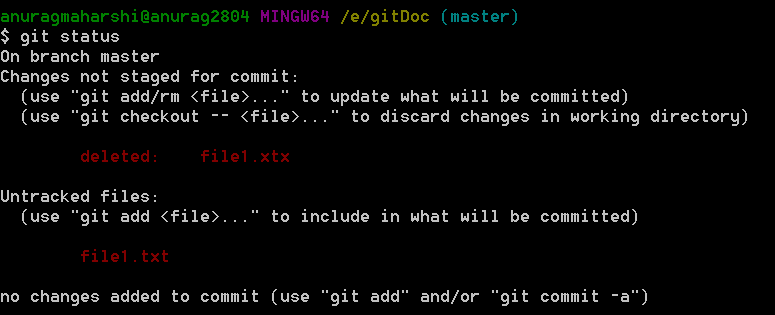


# Command for working with staging

After files, folders are added or removed we need it to add in staging area.

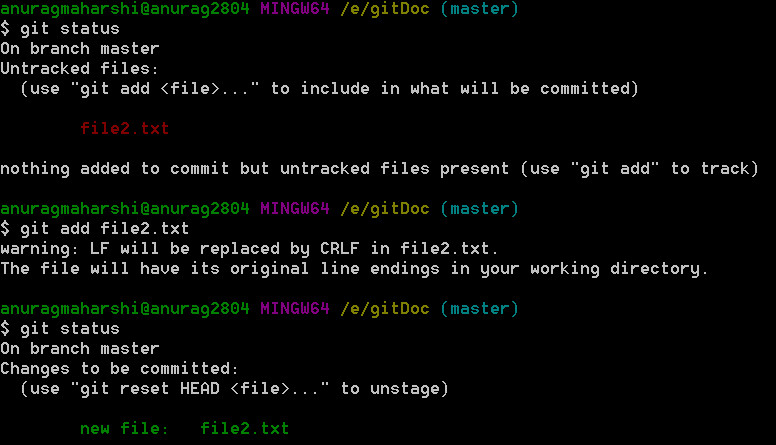
## Current status of staging area. Command is: - git status

Git status list the status of file system which will be committed in local repository. It displays all the changes which we have made.



## Adding single new file for commit. Command is: - git add <fileName>

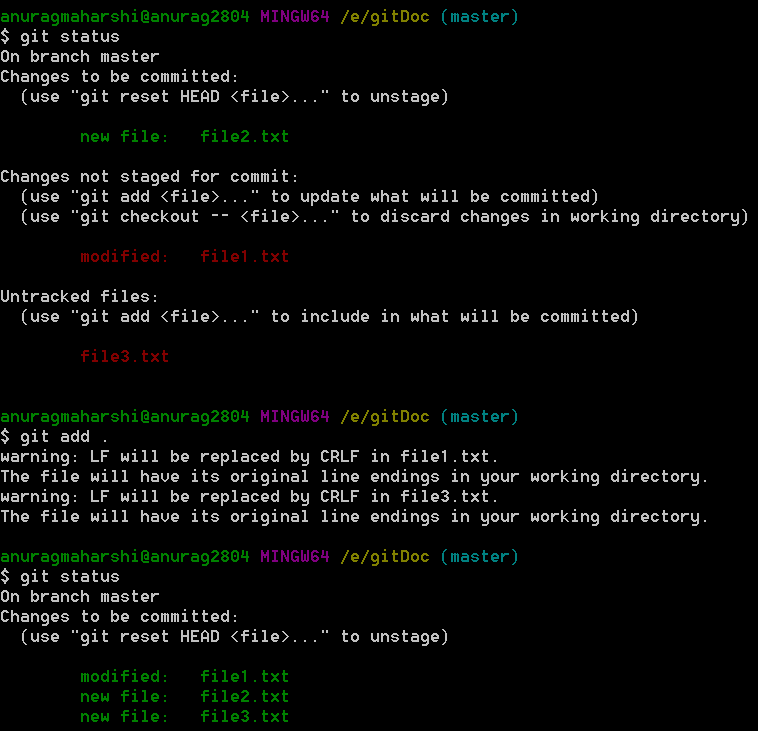
Git add file2.txt. This will add the file in staging area. Once file is added in staging area only then it can be tracked and committed.



## Adding multiple files in staging area. Command is: - git add .(dot character after space)

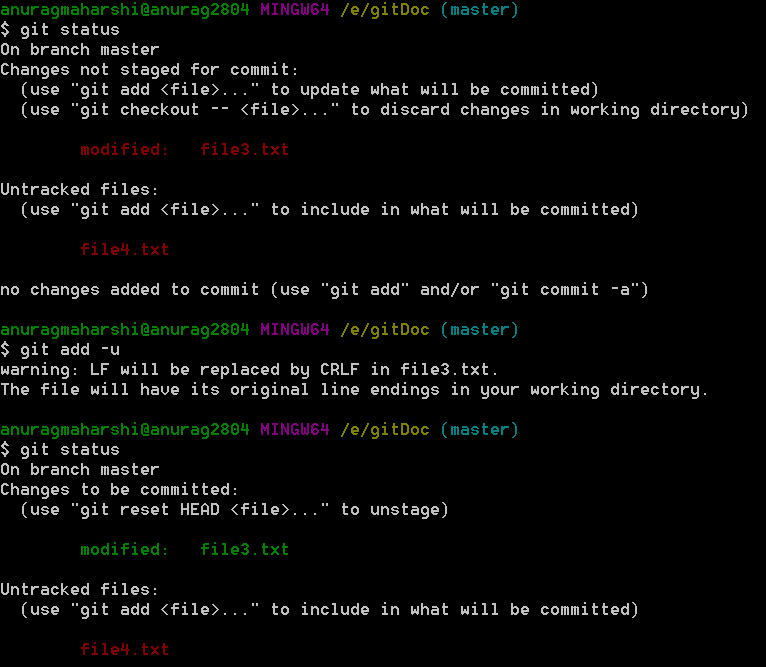
When we need to add multiple files both untracked (new files) and modified files we can use the command git add .(dot).

We can also use git add -A



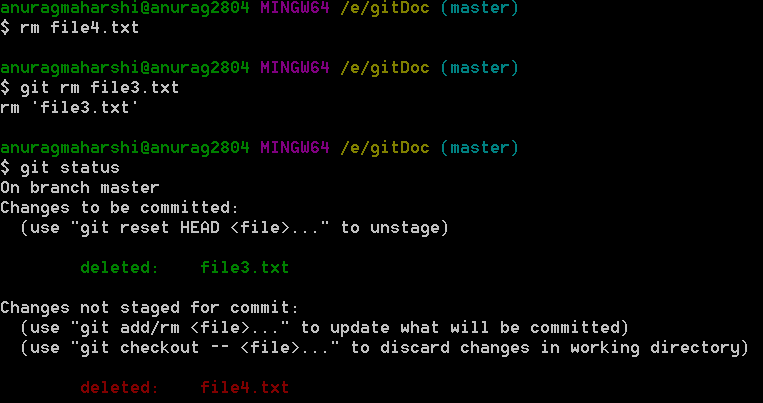
## Adding only the modified files in staging area. Command is: - git add -u

Git add -u will only add the tracked files which have been modified, to the staging area. It will not add new files in staging area.



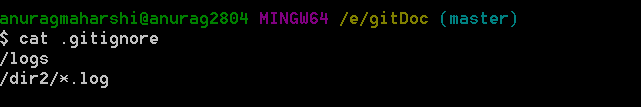
## Deleting files for commit. Command is: - git rm <fileName>

Git rm filename will remove the file from the file system. And we don’t need to use git add command to stage it. However if we use just rm <filename> command we need to use git add command.





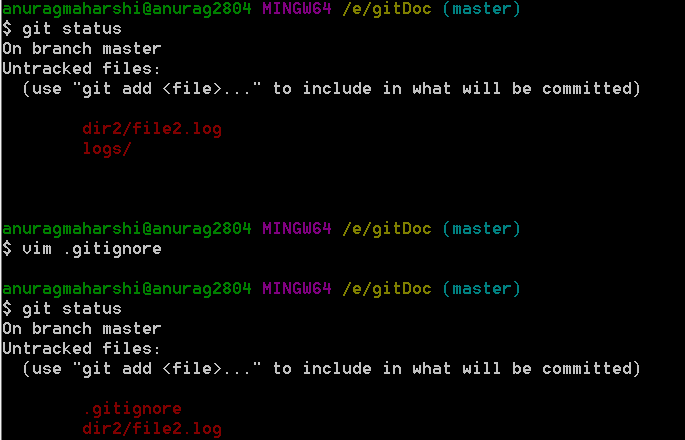
## Ignore files from being tracked: - .gitignore file(dot gitignore file)

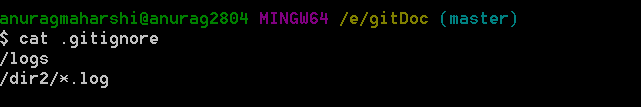


Add the folders and / or files that we don’t need to commit and being tracked. It should start from / (forward slash) to denote the root of the working directory.

Eg /Dir1/\*.txt it will ignore only txt files in Dir1 folder

/logs will ignore all the files in the folder





# Command for committing

## Committing single file. Command is: - git commit <fileName>

Git commit <filename> will commit a single file in local in repository.



Below editor will open for writing the comment associated with commit. To add comment press [Esc], [Insert].

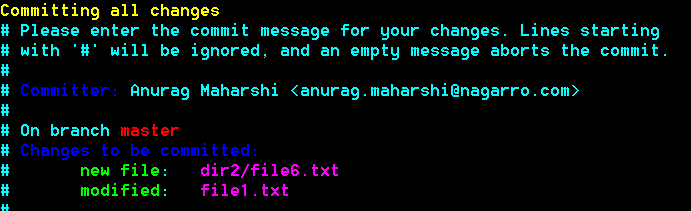
After adding the comment press: wq

Line in yellow the comment.



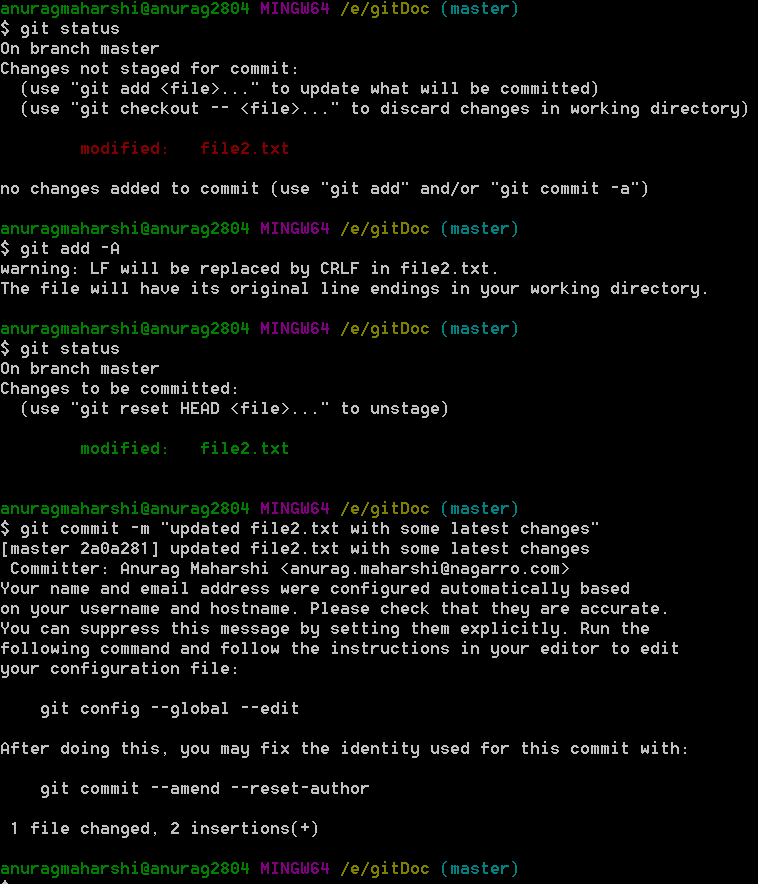
## Committing multiple items. Command is: - git commit

Git commit will add everything which is present in staging area into local repository.



## Committing without opening the editor. Command is: - Git commit -m “user comment”

If we don’t want to open the above window and directly commit it with simple and small comment then we can use the -m switch.

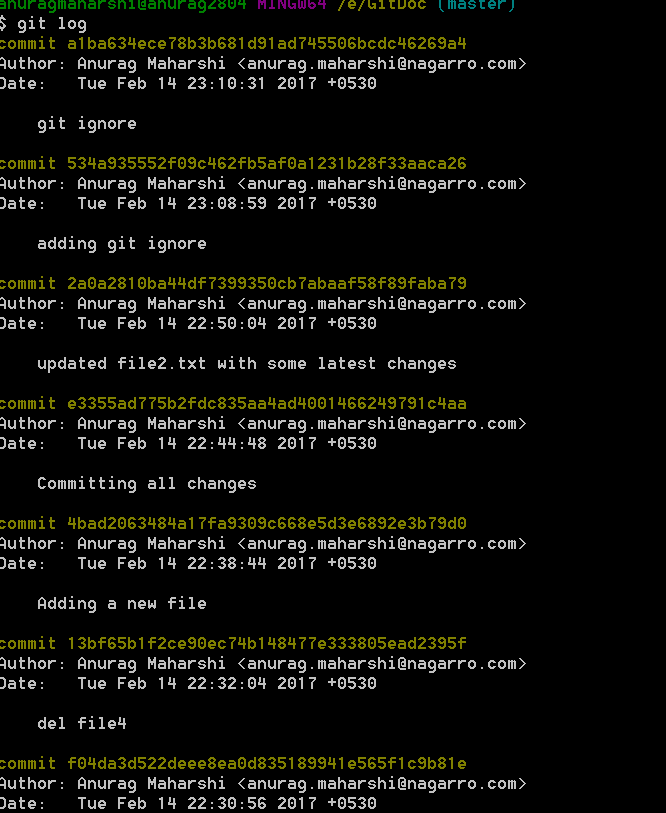


# Commands related to viewing commit history in local git repository

## Displays all commits. Command is:- git log

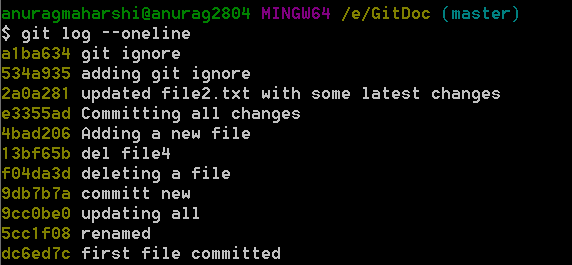
Git log displays the commits made in that repository in reverse chronological order – that is, the most recent commits show up first. If the no of commits is large press [ESC],[: colon] and q to exit the display.

Every commit is associated with a sha code. Git uses this unique sha code to keep track of the changes.



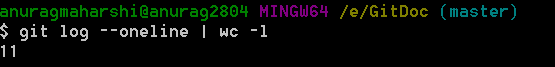
## Displaying commit history in one line. Command is: - git log –oneline(double minus)

Git log –oneline displays the commit history in single line



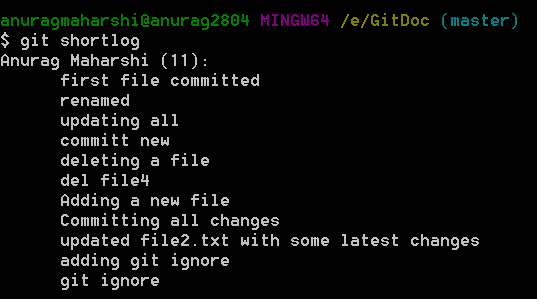
## Counting all commits. Command is: - git log –oneline | wc – 1

Git log –oneline | wc – l displays the no of commits



## Displaying short logs. Command is: - git shortlog

Git shortlog displays the authors and commit messages by them. Total no of commits made by them is in bracket



## Displaying no of commits short by author. Command is: - git shortlog -sne

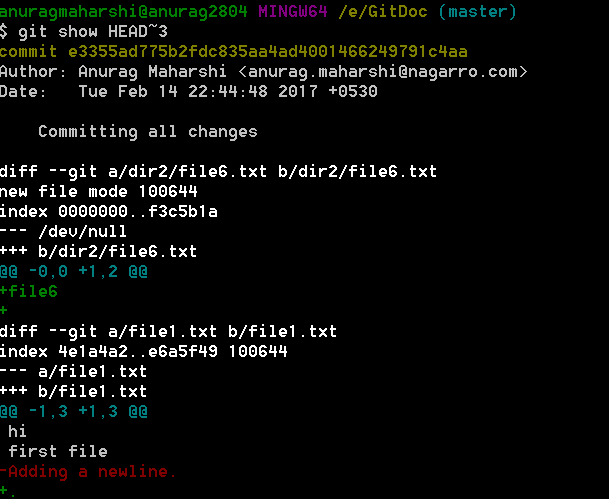
git shortlog -sne displays all commits grouped by author. N denotes display with decreasing no of commits.E denotes displayed email



## Viewing specific commit. Command is: - git show HEAD / git show <5 digit sha>

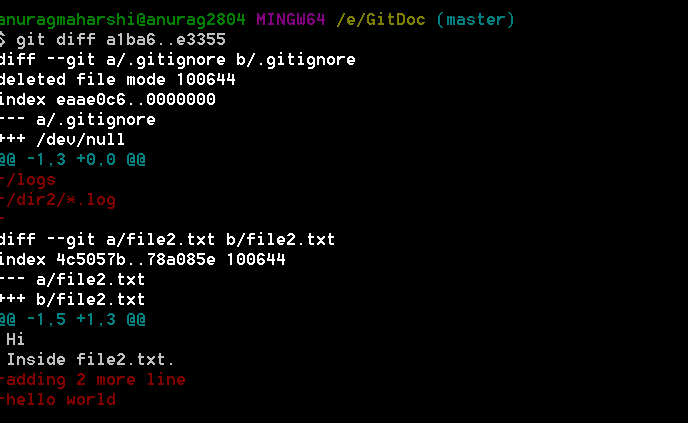
Git show HEAD displays all the information changed in last commit. For displaying any specific commit from head, we can use ~(tilde symbol). Git show HEAD~2. This will display second commit from HEAD.

We can use short sha no to also display it. Eg git show <ajs23>.



## Displaying difference between two commits. Command is git diff <sha 1>..<sha 2>

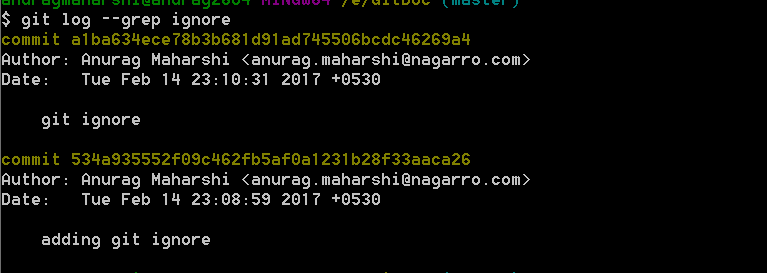
Git difference will display the difference between two commits. We need to provide 5 or 6 digit sha no which is generated with each commit.



We can also use HEAD instead of sha no. HEAD it the most recent commit. Previous commit from HEAD is HEAD~1(tilde). For e.g. git diff HEAD~2..HEAD will display the difference between HEAD which is current state of repository and two version previous.

## Filtering commit history on text in commit message. Command is: - git log –grep <text>

Git log –grep <text> will filter that commit history based on text in commit message



## Other switches for displaying commit history

* -(n)

Show only the last n commits

* --since, --after

Limit the commits to those made after the specified date.

git log --since=2.weeks

This command works with lots of formats – you can specify a specific date like "2008-01-15", or a relative date such as "2 years 1 day 3 minutes ago".

* --until, --before

Limit the commits to those made before the specified date.

* --author

Only show commits in which the author entry matches the specified string.

* --committer

Only show commits in which the committer entry matches the specified string.

* -S

Only show commits adding or removing code matching the string

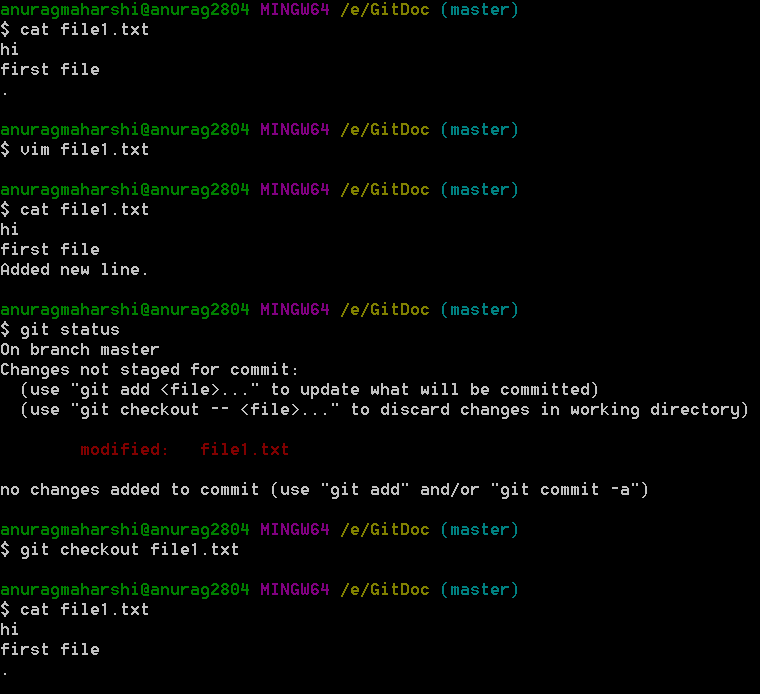
## Link for better understanding of logs

<https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History>

# Recovering Old files

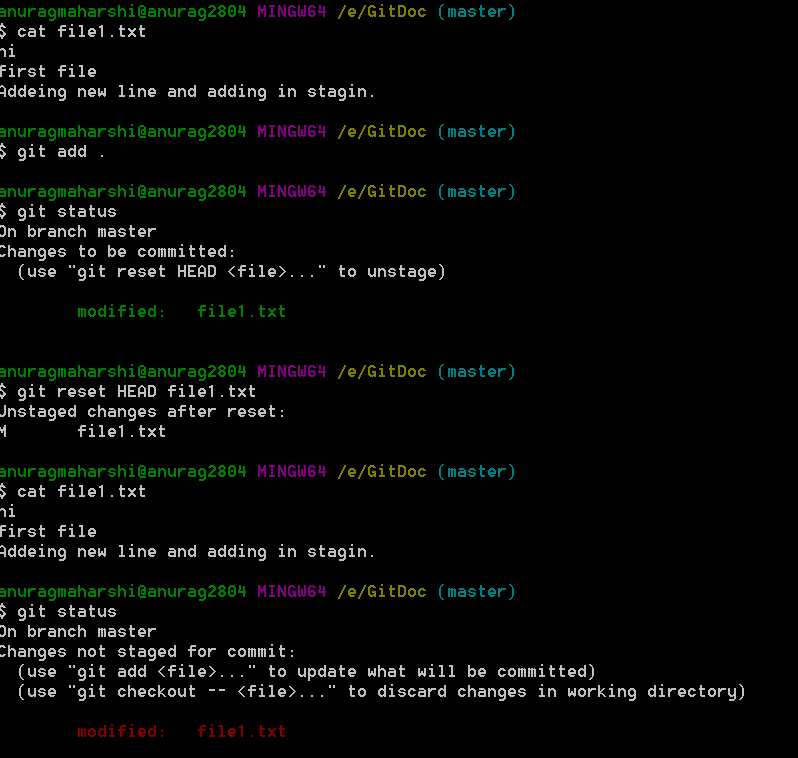
## File updated in working directory but not added in staging. Git checkout <filename>

If you have updated any file, have not yet added in staging area, then old verison can be recovered using git checkout <filename> command.



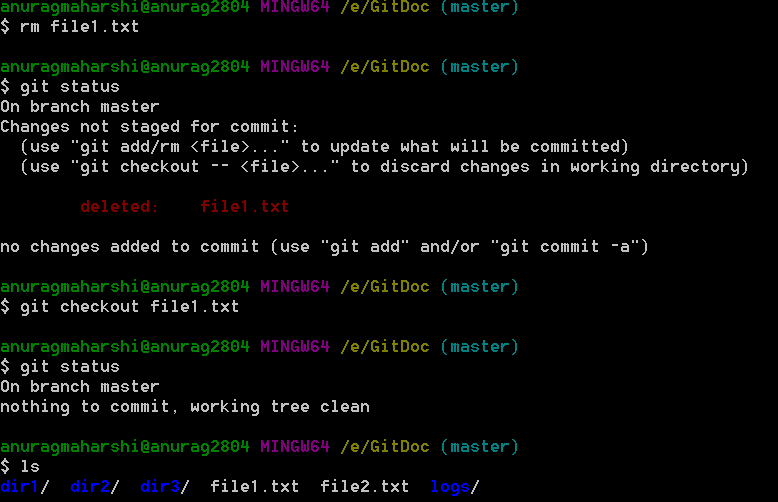
## Files added in staging area. Command is: - git reset HEAD <fileName>

Git reset HEAD <filename> can bring back the files in working area and will be removed from staging area. To discard the changes, this file can be git checkout.



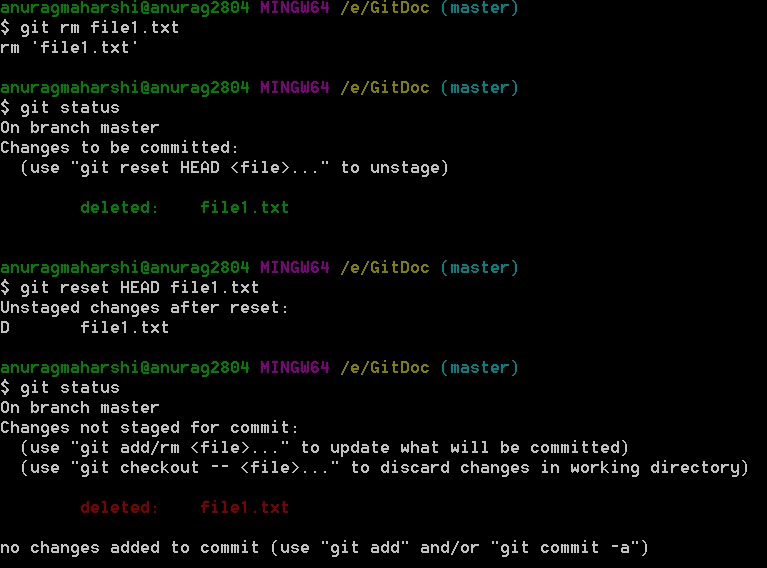
## Recovering deleted files not staged. Command is: - git checkout <fileName>

If file is deleted using simple rm command, then it can be recovered using git checkout



## Recovering deleted file from staging area. Git reset HEAD <fileName>

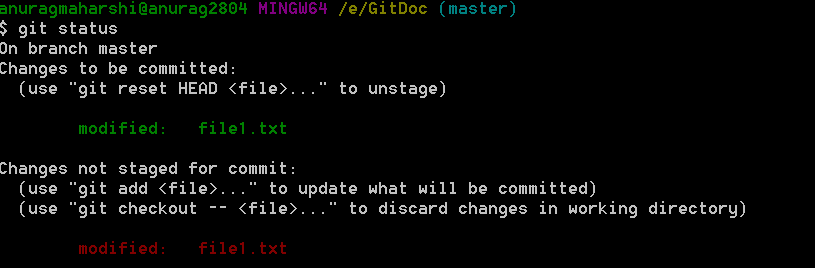
If file is deleted using git rm <filename> command, it is added directly in staging area. Or if user deletes the file and use git add command it is added in staging area. We can use git reset command to get back the file.

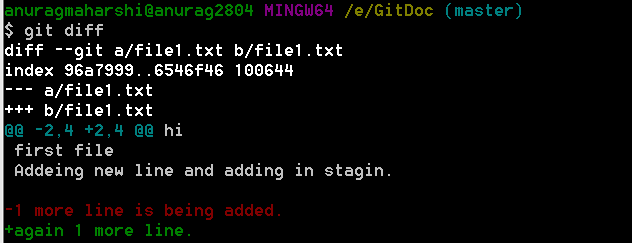


## Understanding git diff, git diff –cached and git diff HEAD

Suppose user have modified a file and added it in staging area. Now if user have modified it again, then this file will be both in staging area ready for commit and in modified file section.

* 1. Git diff is used to display the difference in file between staging and working area.





Line in red is present in staging area only and line in green is present only in working area.

* 1. Git diff --cached is used to display difference between Head ie last commit and staging area.



* 1. Git diff HEAD is used to display difference between HEAD and working directory.



## Git Reset. Command is: - git reset HEAD~<# from last commit> --soft/hard

For undoing changes that is committed we use git reset. This operation is mainly done when working privately and we need to discard the changes.

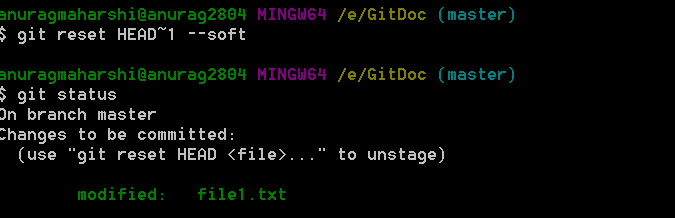
Git reset update the current branch with the specified commit where we need to role back.

In addition to moving the current branch, you can also get git reset to alter the staged snapshot and/or the working directory by passing it one of the following flags:

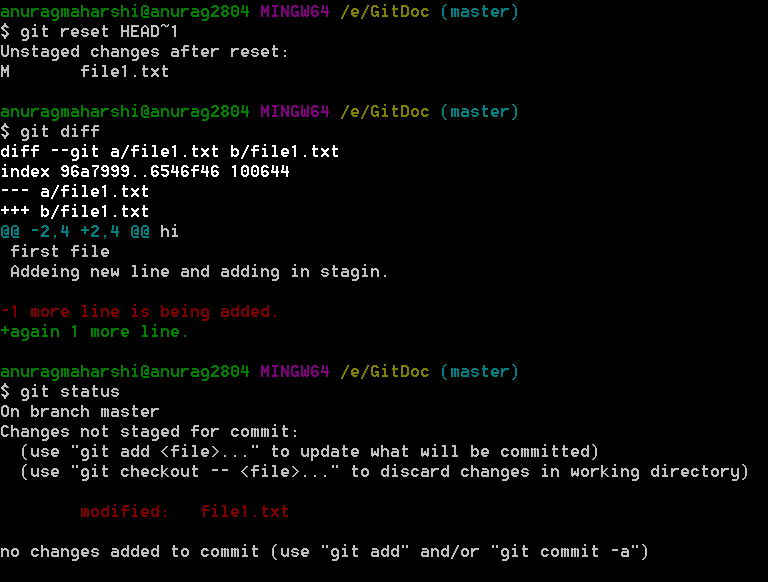
• --soft – The staged snapshot and working directory are not altered in any way.

• --mixed – The staged snapshot is updated to match the specified commit, but the working directory is not affected. This is the default option.

• --hard – The staged snapshot and the working directory are both updated to match the specified commit.



Use git diff command to know the changed content if we use soft reset.



Git reset will update all the files.

## Git Revert. Command is: - git revert HEAD/HEAD~<#>/sha no of commit

Git revert is also used for recovering changes made in previous commit. But this command reverts the changes as a new commit.

Git revert HEAD

This will bring back the changes made in last commit. Working directory will remain the same.

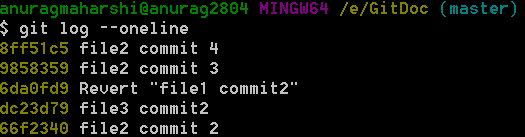
After reviewing the revert it can be committed again.



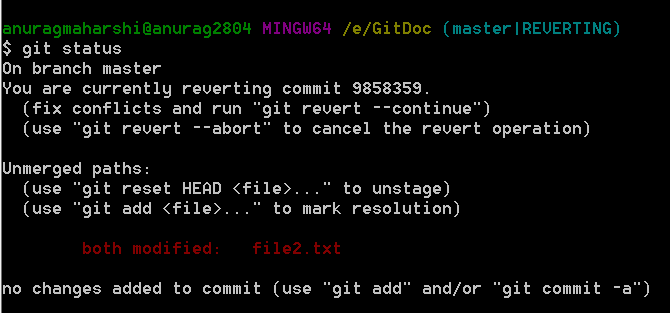
If we need to revert changes, we use git revert HEAD~2. This will revert 2nd commit from Head which is file1 commit 2.

After this revert, the content of file1 will be updated to that previous version of same file ie contains specified in commit with sha no 9ee4fde.

When there is conflict between then we need to modify the changes before committing. This happen when we have done 2 or 3 commits in the same file. And we want to revert changes from 2nd Commit. This saturation will create a conflict.







As specified we can use git revert –abort to cancel the revert.

Or we can modify the file and add it.

The steps for reverting in case of merge is then, just checkout the file that you want. Get the difference and update the file accordingly.

We can checkout specific file version and revert the changes. We need to resolve the conflict revert.

# Working with branches.

Branches are different working area in the repository. Each branch have entirely different working area. When we work on one branch, whatever changes we make is local to that branch. Our master copy is not affected by it.

This unique feature allow us to do research, modify and work on different module without interfering with other users.

## Displaying all branches in a repository. Command is: - git branch -r/--all/--list

* -r flag displays the remote branches
* --all or -a flag display all branches
* --list displays the list of branches. We can provide other parameter in this



## Creating new branch, Command is: - git branch <branchname> [<startpoint>]

Git branch <branchName> will create a new branch. This branch is by default not displayed.

When we create a new branch, all the files and directory included in the <startpoint>will be indexed and included in the working directory of new branch.

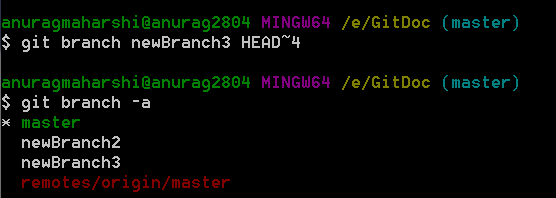
<Startpoint> is pointer to the commit history. If it is omitted new branch points to HEAD.

Git branch –all/a will display this new branch.



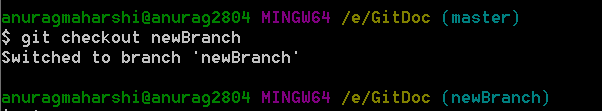


Other commit points can also be specified like HEAD~1, sha of commit.



## Checking out a new local branch. Git checkout <branchName> <startpoint>

Git checkout <branchName> will load the branch in our local working directory.



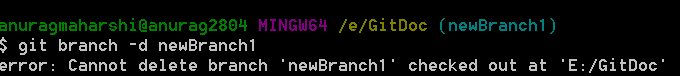
## Creating branch and checking out. Git checkout -b <branchname> [<startpoint>]

With -b flag, new branch will be created and checked out.

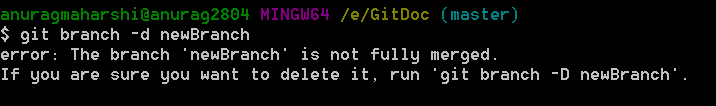
<Startpoint > is optional. It can point to anything in commit history.

## Deleting branch. Command is:- git branch -d/D <branchName>

For deleting a branch user must be checked out of current branch



* -d flag will delete the branch if it doesnot contain any commit.
* -D flag will force delete the branch.



## Renaming a branch. Command is :- git branch -m <oldBranchName> <newName>

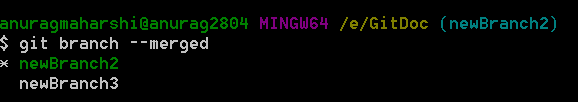


## Filtering branches. Git branch –merged/--no-merged [<branchName>]

* --merged flag. It displays branches which have same level of commit.
* --no-merged flag. It displays branches which are not merged and have some difference. <branchName> is optional. If omitted it tracks difference with current branch.

Then the filtered branches can be merged and/or deleted.





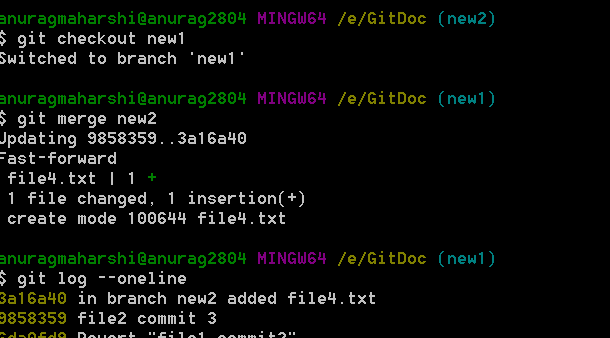
## Merging branches with no conflicts. Command is :- git merge <branchName>

Branches new1 and new2 are at same commit level we and add a new file in this. Then we merge branch new2 in new1.

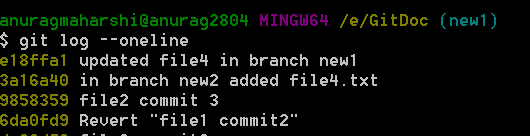
If the branches commit changes in different files then there will be no conflict.

Steps.

* Checkout branch in which we need to merge. Eg new1 branch we need to merge all changes made in branch new2.
* Merge using command :- git merge <branchName> eg git merge new2

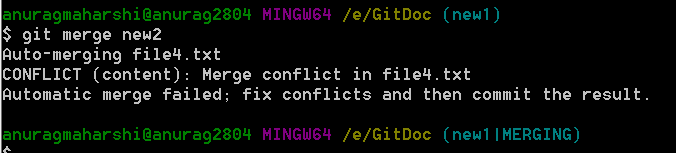


## Merging branches with conflicts with command line editors.





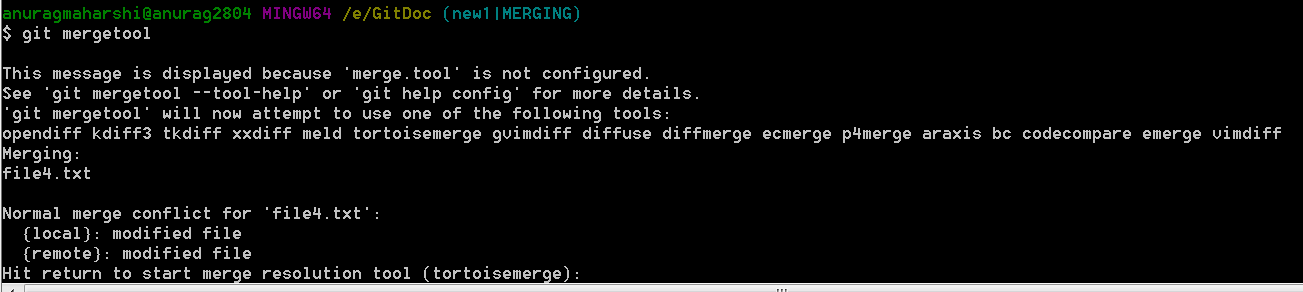
As we can see in the commit history, the two branches have updated the same file. This will create merge conflict.



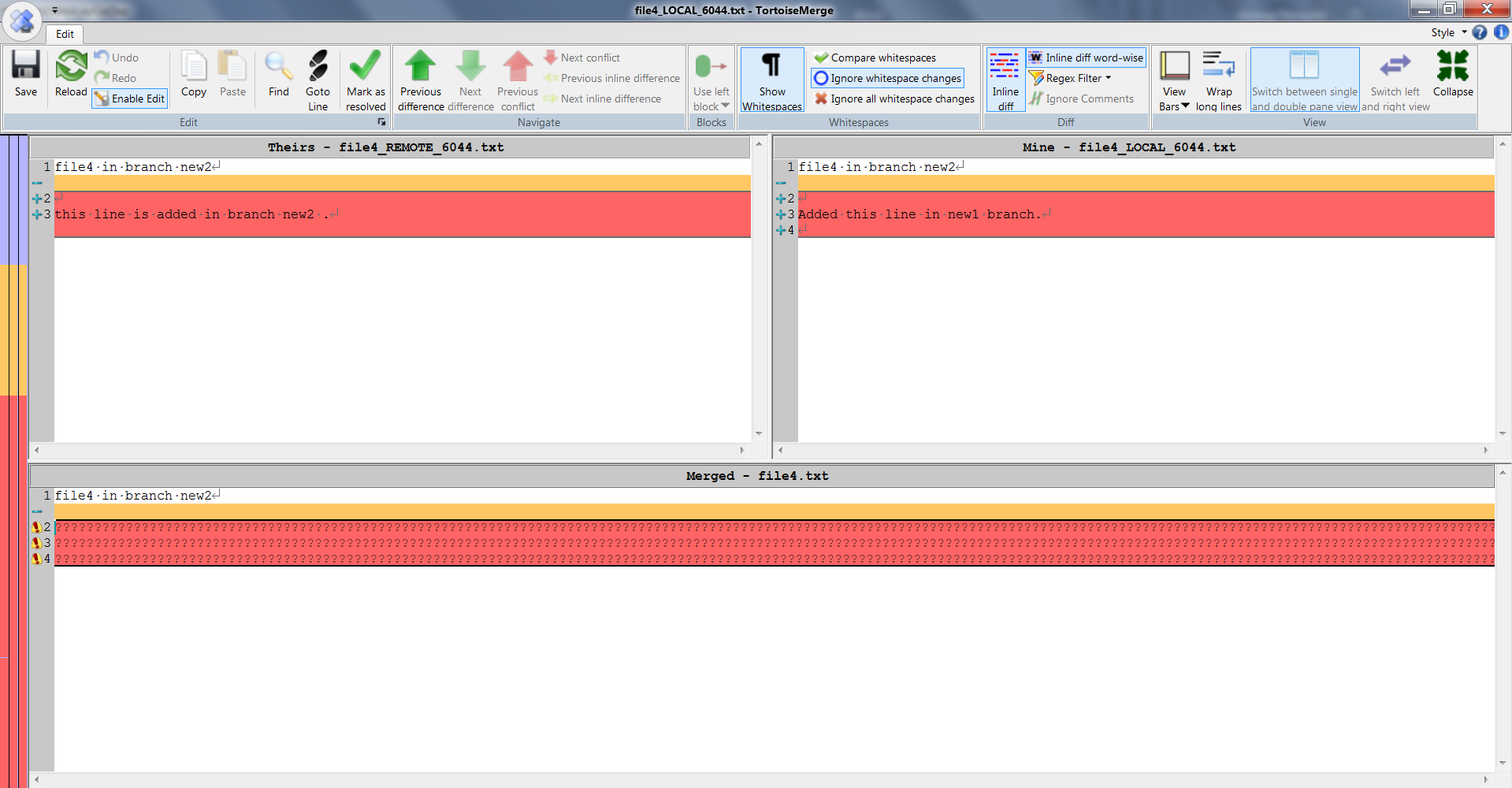
To resolve we can use the merge tool

### Merge Tool

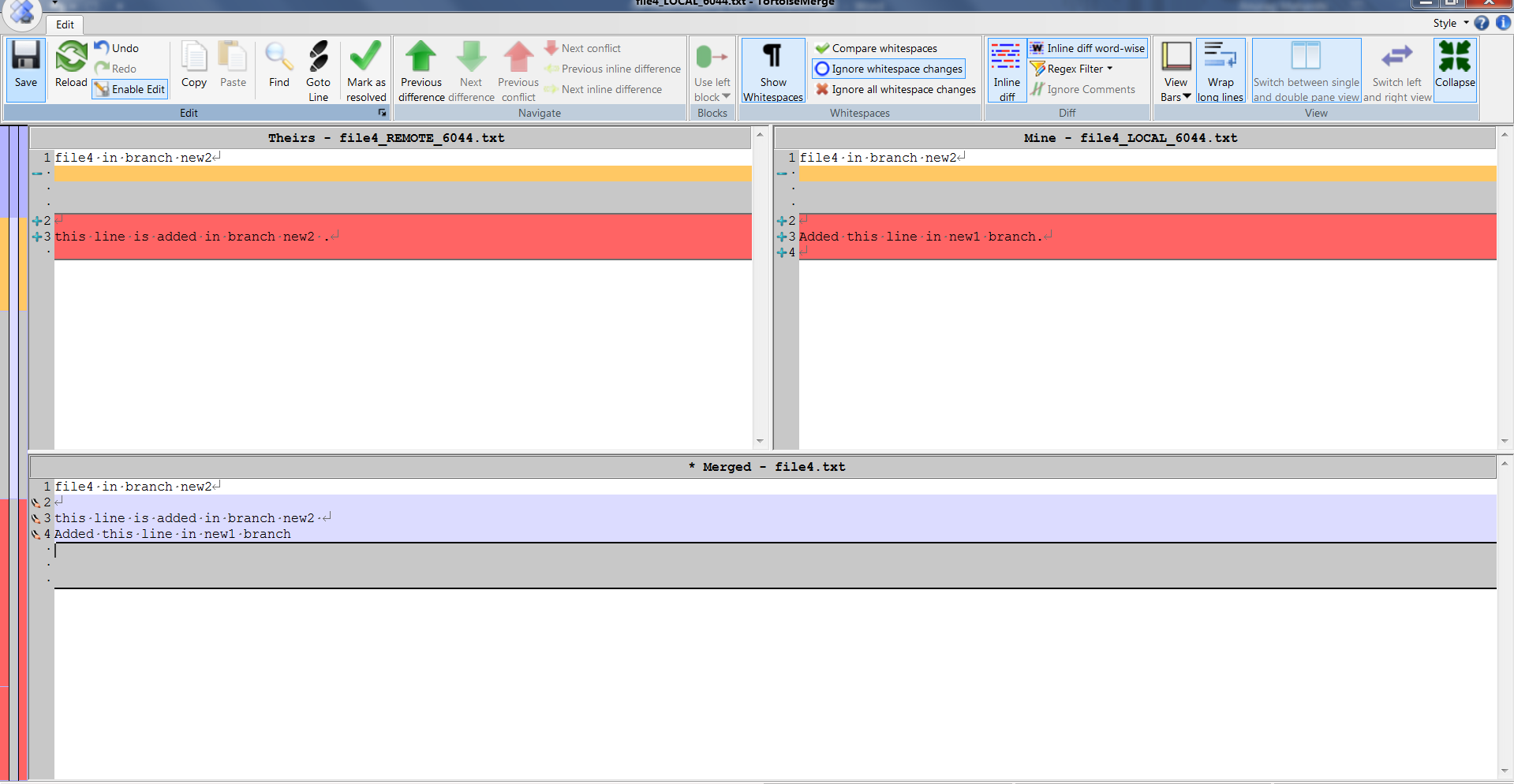
* Open merge tool. Command is git mergetool



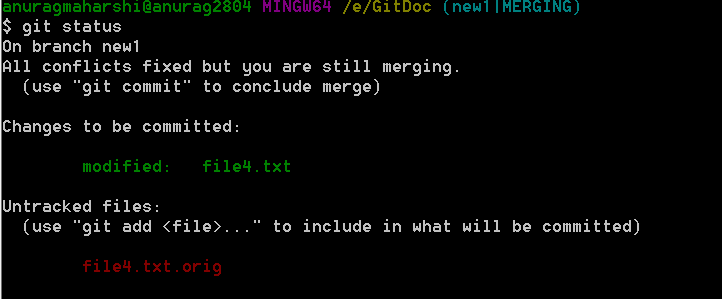
* Git merge window will open.



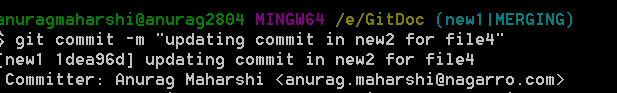
* Update file in merged section in. Keep the content from either file which you want.

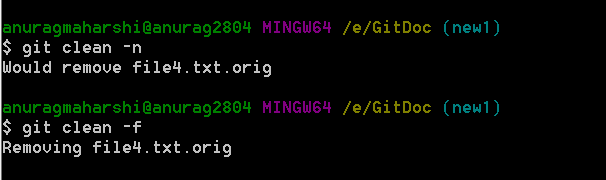


* Click on Mark as resolved button in above window
* Close the window
* Git bash window will be displayed with merging keyword



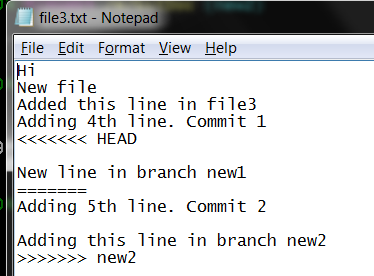
* Now commit the file in current branch. This will merge file4.txt in the branch. Note the file in other branch (new2)will remain same.
* There will be untracked file created. We need to delete it by using clean command
* Git clean -n (will display file which will be deleted)
* Git clean -f (will delete the untracked file)





### Default editor such as notepad or vim editor.

* After the merge conflict open the file in editor.
* Understand conflict markers:- To see the beginning of the merge conflict in your file, search the file for the conflict marker <<<<<<<. When you open the file in your text editor, you'll see the changes from the HEAD or base branch after the line <<<<<<< HEAD. Next, you'll see =======, which divides your changes from the changes in the other branch, followed by >>>>>>> BRANCH-NAME.



Content above <<<<<<< HEAD is present in HEAD or last commit.

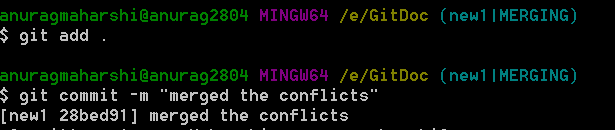
Line “New line in banch new1” is present in branch new1 only.

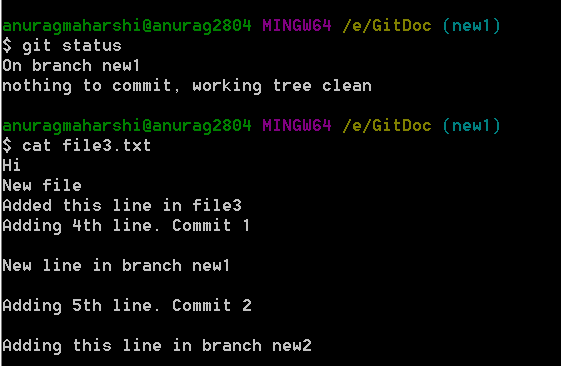
Content from “Adding 5th line… tile new2” is present in new2 only.

Updated the changes in editor and save it. Also remove the markers. 

After merging the file looks like this.

Now git add and commit the changes.

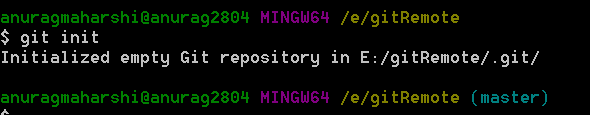




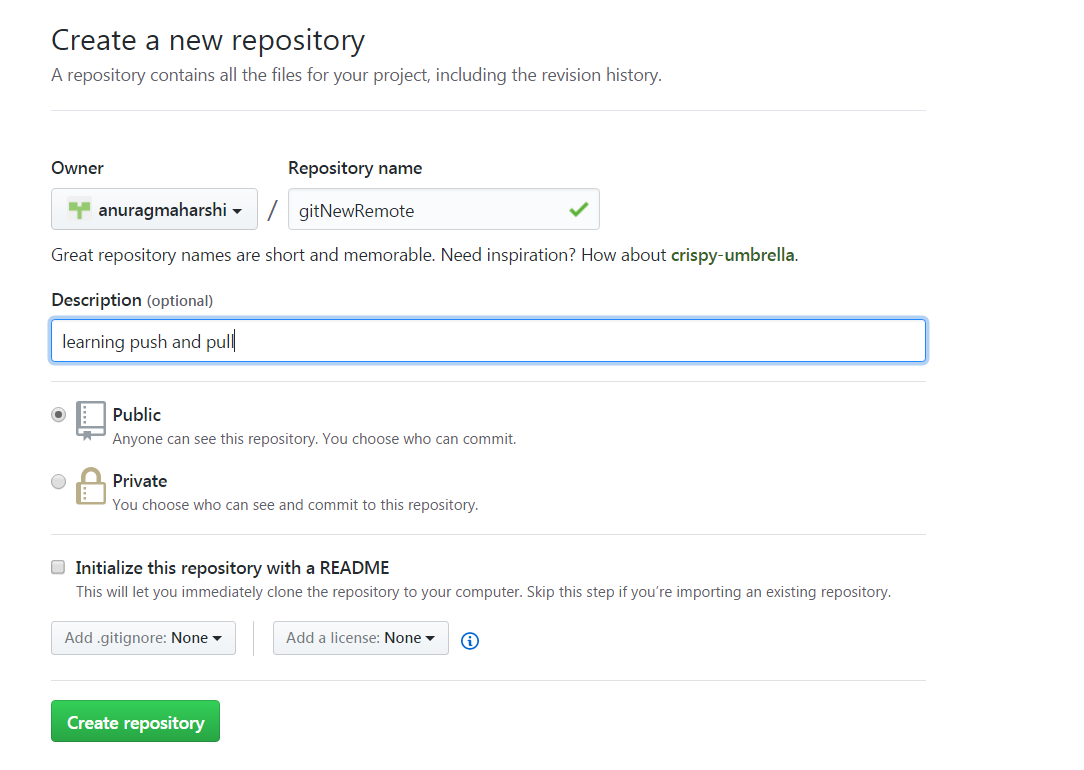
# Git Working with Remote

## Creating local repository and pushing

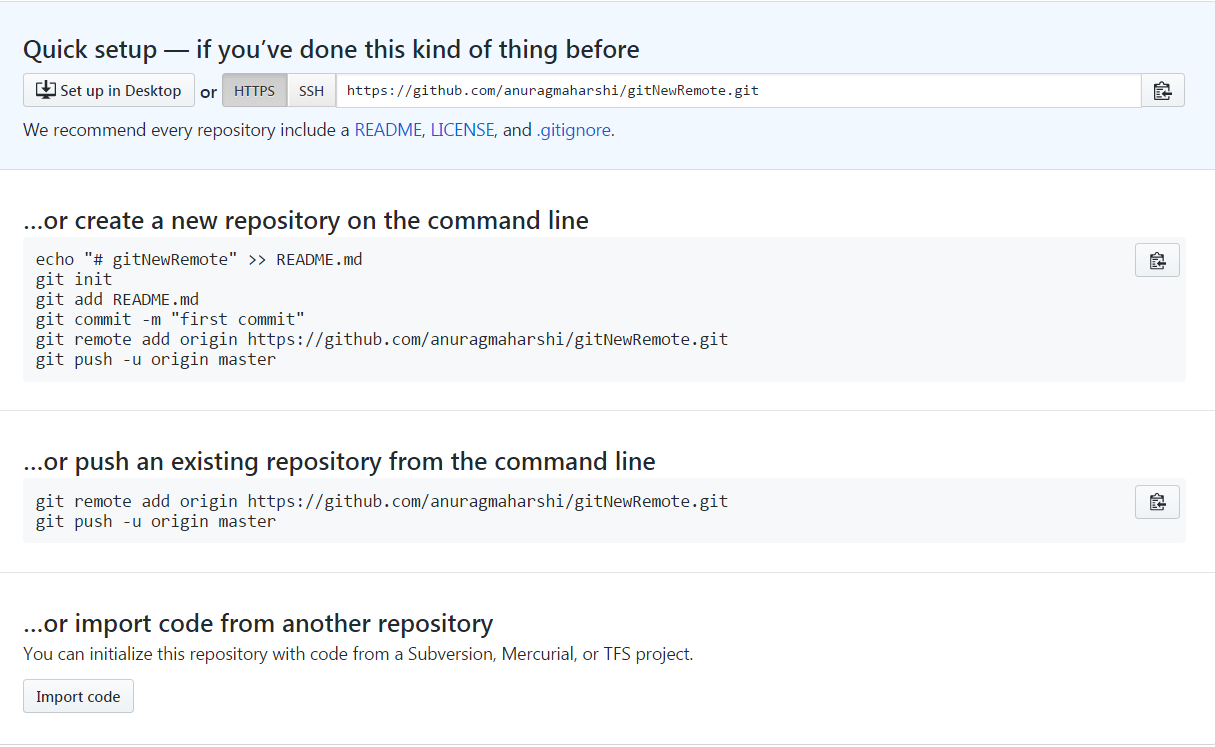
* 1. Local blank git repository



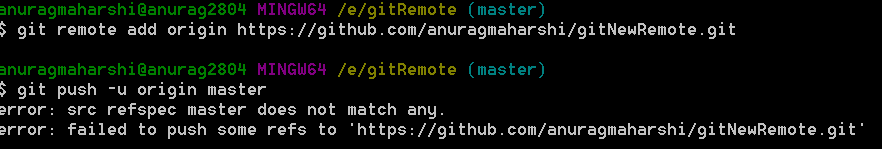
* 1. Creating remote server repository



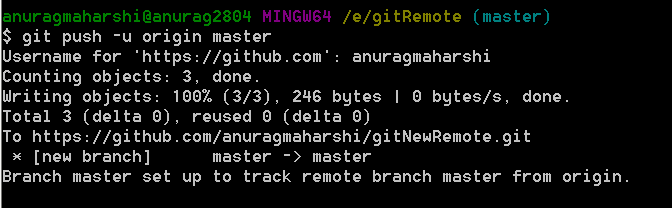
* 1. Snapshot of server after creating remote repository



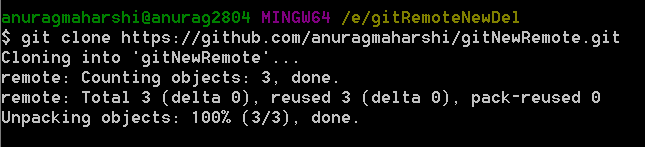
* 1. As my local repository is blank, we need to add initial commit for pushing.



* 1. For pushing the changes, username and password of remote server should be known



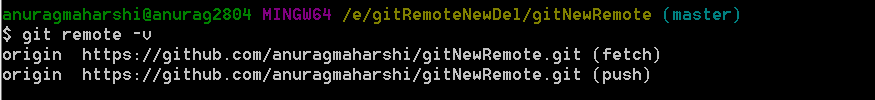
## Cloning from remote. Command is: - git clone <url>

For cloning from a remote use git clone.

This will download the git repository in new local directory.

## Displaying remote tracking branches. Command is: - Git remote -v

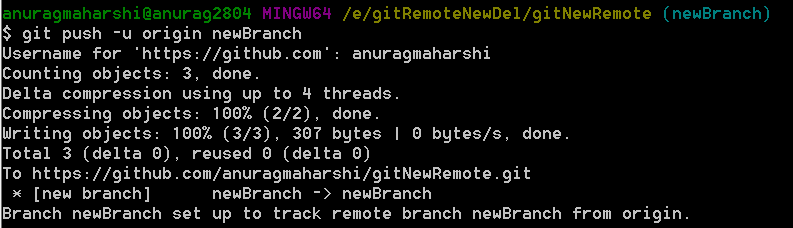
Git remote -v displays the remote tracking branches.



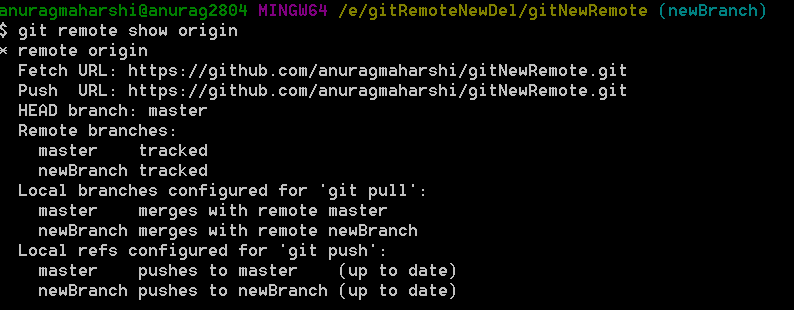
## Publishing local branch in remote. Command is: - Git push -u origin <local branch name>

To publish your local branch in remote so that it can be tracked and used by others use

Git push -u origin <local branch name>



## Inspecting a remote repository. Command is: - Git remote show origin



## Pushing changes in tracked branches. Command is: - Git push origin/origin HEAD/origin <name>

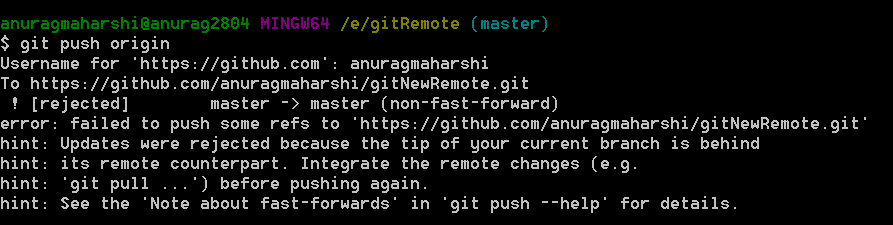
Git push origin / git push origin HEAD/git push origin <branchName>

This command will upload our changes from local repository to remote. Note it means that we must have origin configured with master.



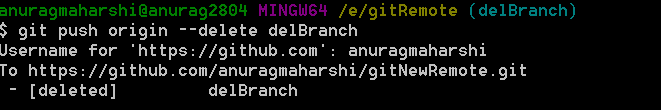


If your branch is behind remote an error message will be displayed.

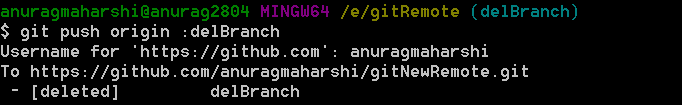


## Deleting remote branch. Command is: - Git push origin –delete <branchName>

To delete a branch from remote we need to use –delete switch.



Another way to delete is git push origin :<branchName>

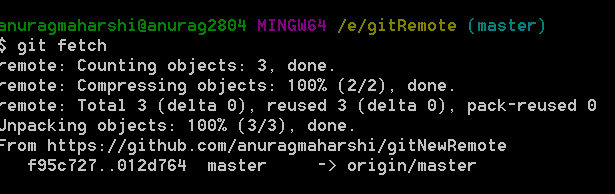


## Renaming remote branch name

To rename the remote branch name, simply edit the name locally, publish the changes to new branch and delete the old branch in remote.

## Fetching remote branch without updating local. Command is: -Git fetch/git fetch origin <branch name>

* Used to retrieve new work done by other people.
* Fetching from a repository grabs all the new remote-tracking branches and tags *without* merging those changes into your own branches.





We can use git log ..origin/branchname or git log [..@{u}](mailto:..@%7bu%7d) to view the difference.

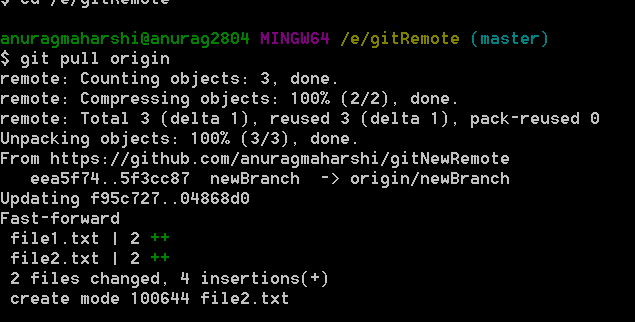




Git status will also display the no of commits by which our local repository is behind remote repository.

## Pulling all changes from remote in local. Command is: - Git pull origin /git pull origin <branchname>

* Git pull fetches the latest changes from remote into local and merge that. It is like git fetch plus git merge
* Git pull origin will pull the changes from the remote tracking branch of current branch.
* Git pull origin <branchName> will pull changes from the specified branch.
* It can be used to pull data from a remote to a new local branch.



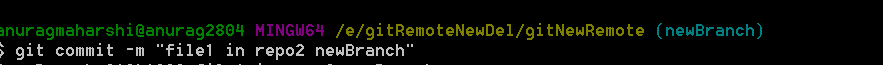




We need to remove the merge conflicts in this case.

## Git Pull with rebasing. Command is: - Git pull -rebase origin

* Rebasing allows local changes to be pushed on top of all the remote changes.
* If remote branches is updated by other users, and we want our commit to be most recent one we use git pull with rebase.

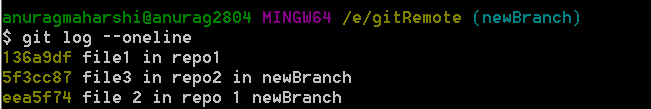


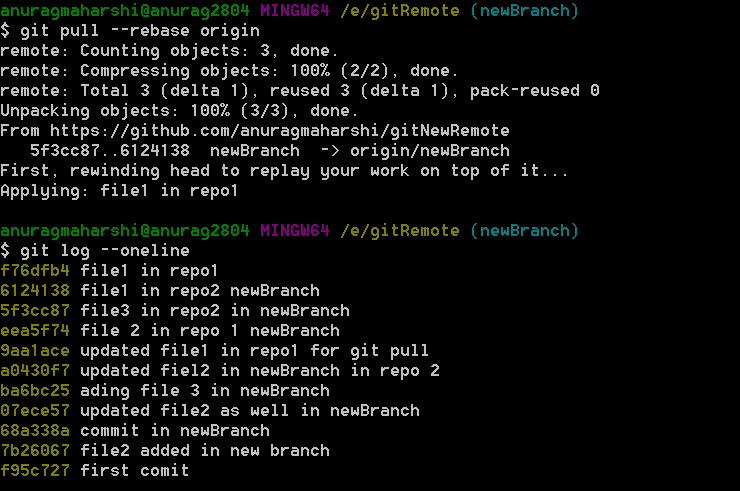


Switched to different repo









# Git Tagging

## Git Tag

* Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points (v1.0, and so on).

Git tag command displays all the tags in current branch



## Git annotated tags. Git tag -a <verison no> / git tag -a <verison no > -m <message>

* Annotated tags are stored as full objects in the Git database.
* They’re checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It’s generally recommended that you create annotated tags so you can have all this information



Without the -m switch and message, an editor will open to write the tag message.



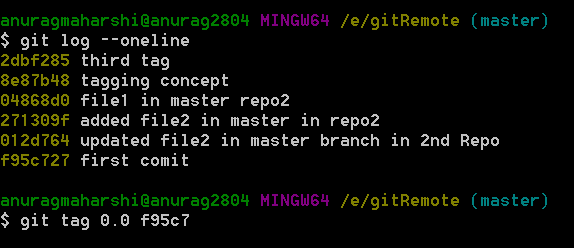
## Git lightweight tags. Git tag <versionNo>

* In lightweight tags we don’t provide any switch like -a,-m
* A lightweight tag is very much like a branch that doesn’t change – it’s just a pointer to a specific commit.



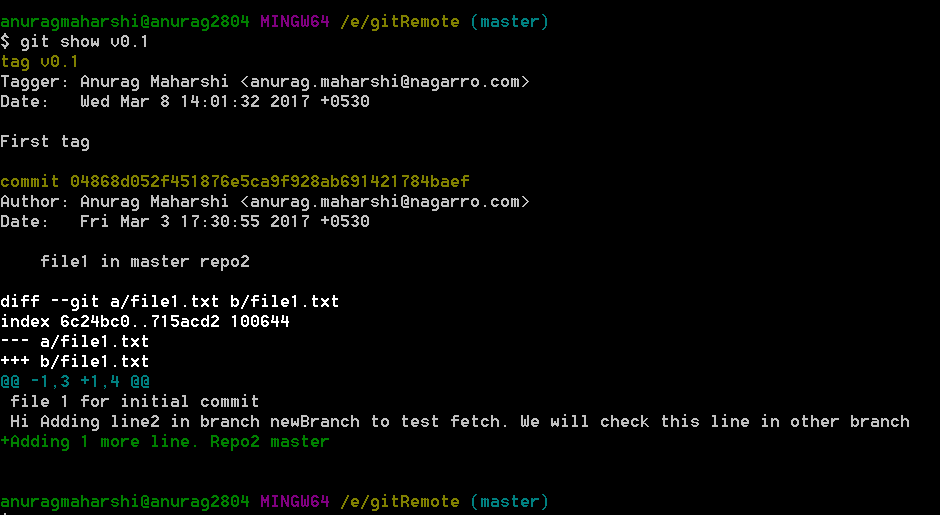
## Git tagging later. Git tag <tagName> <5,6 digit of sha # of commit>

* If we want to tag an specific version at later point we can use sha no of commit



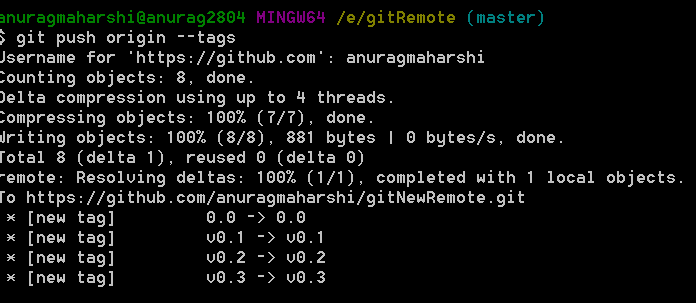
## Displaying information of tag. Git show <tagNo>

This displays the information about the tag. If version in annotated tag then extra information like email and name of tagged is also displayed.



## Pushing tags to remote. Git push origin –tags

By default tags are not pushed in remote. We need to specifically do that.





## Git checkout specific tag. Git checkout -b <newBranchName> <tagNo>

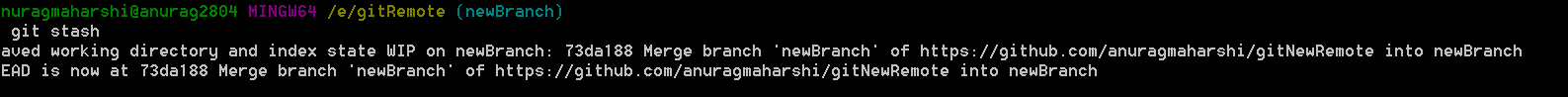


# Git Stash

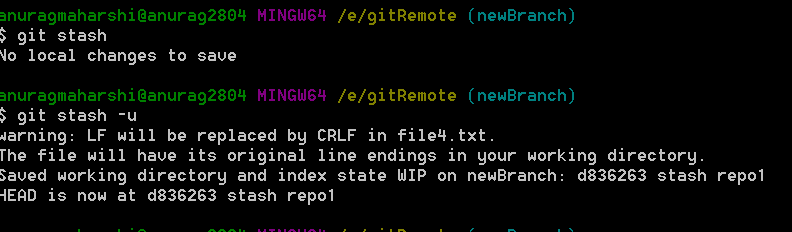
* Use git stash when you want to record the current state of the working directory and the index, but want to go back to a clean working directory.
* The command saves your local modifications away and reverts the working directory to match the HEAD commit.
* When different user in same branch modify different files and commit there changes in remote, there will no conflict in merging.
* When different user update same file in same branch, there will conflict in merging. Git pull will open the merge section where we need to resolve the conflict.

## Git stash / git stash -u

* This command will remove the changes in working directory to a temporary directory

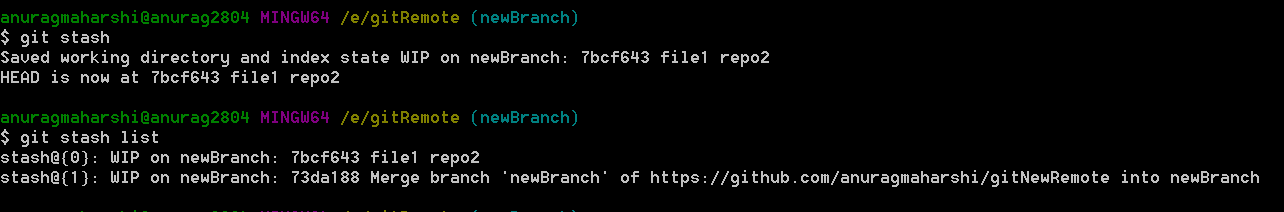


* -u switch will keep untracked files as well.



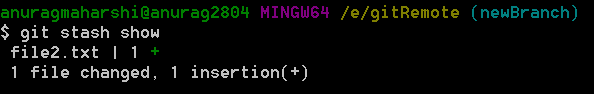
## Git stash list

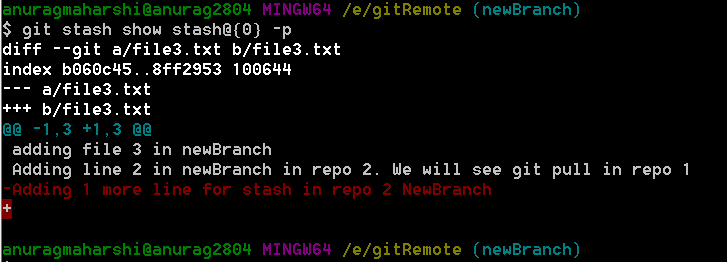
* This command shows the stashes.



## Git stash show [<stash>]/git stash show [<stash>] -p

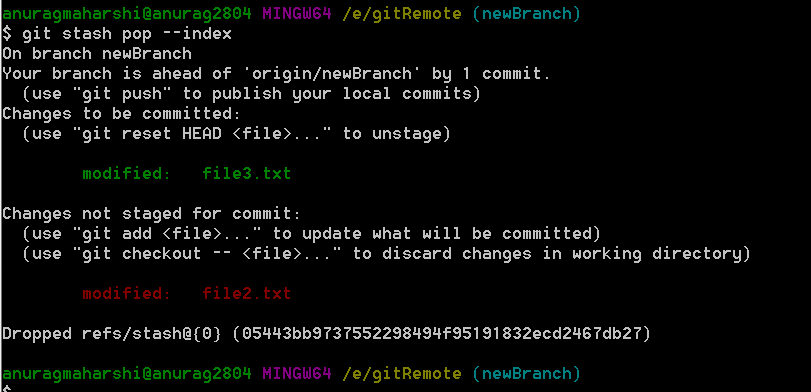
* Git stash show will show the file difference between Head and Stash
* <Stash> is the optional parameter for specifying a specific stash in case of multiple stash
* -p switch display the actual difference.





## Git stash pop [--index] [<stash>]

* Git pop will bring the changes back from temporary tree to working tree.
* --index will also update the index ie it will bring changes in committing area
* <stash> no can be given. If not specified stash0 is popped up.



## Removing single temporary stash. Git stash drop [<Stash>]

* Removes single stash from temporary area
* If <stash> is omitted 0th entry will be removed.

## Removing all temporary stash. Git stash clear

* It will remove all the stash entry

## Adding stash entry without removing it. Git stash apply [--indes] [<stash>]

* It will apply the stash entry but will not pop it.

## Creating branch from stashed changes. Git stash branch <branchname> <stashNo>

* We can create a new branch from stashed changes. This is helpful in merging

