**GIT Learning**

YouTube Links:

<https://www.youtube.com/watch?v=iR5WIknxdkY&t=6s>

<https://www.youtube.com/watch?v=evknSAkUIvs&list=PLu0W_9lII9agwhy658ZPA0MTStKUJTWPi>

<https://www.youtube.com/watch?v=-U-eUHI6euM&list=PLhW3qG5bs-L8OlICbNX9u4MZ3rAt5c5GG>

<https://www.youtube.com/watch?v=HVsySz-h9r4&list=PL-osiE80TeTuRUfjRe54Eea17-YfnOOAx>

Content:

1) Overview of GIT

2) What is GIT

3) Why Git Version Control

4) History of Version Control System

a. Local VCS

b. Centralised VCS

c. Distributed VCS

5) How does Git work / GIT Life Cycle?

a. Initialization

b. Staging area

c. Commit

6) Installing Git

7) Environment Setup

8) GIT Features

9) Git – Three stage architecture

11) File life Cycle

12) Skipping The Staging Area

13) Moving and Renaming Files In Git

14) Git Log: Viewing & Changing Commits In Git

15) Unstaging & Unmodifying Files In Git

16) GitHub: Working with Remote Repositories

17) Setting Alias In Git

18) Creating & Switching Branches In Git

19) Branching & Merging a Production Grade Project

20) Directories Vs Repositories

21) ‘bare repository’ in GIT?

22) git fetch

23) git pull vs git fetch

24) Amending a commit

25) Command line fundamentals

26) Using Stash Command

27) git merge

28) git merge abort

29) Diff and Merge Tools

30) Difference between Add= “A”, “add – u”

31) git Commit Hash?

32) How to resolve Merge Conflicts

33) Git Revert

34) Git Reset

35) Git Rebase

36) 2 way commit and 3 way commit

37) Cherry Picking In GIT?

38) What does the commit object contain?

39) Git fork

40) Git Pull Request

41) git log

42) git reflog

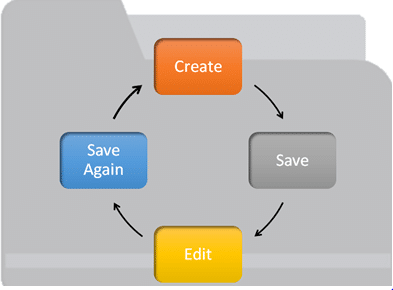
43) git squash

44) git hooks

**Getting Start With GIT**

**Overview of GIT:**

* Be it a designer, a creator, or a developer, all deal with projects and files every day. Our primary work cycle revolves around creating a file, saving it, editing or making required changes, and saving it again.
* Git is a small yet very efficient version control tool. It helps both programmers and non-programmers keep track of the history of their project files by storing different versions of them.



## What is Git?

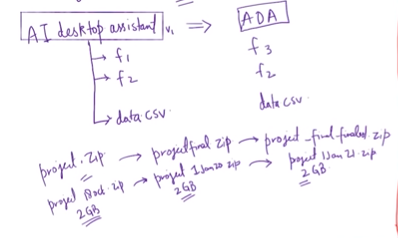
Git helps developers keep track of the history of their code files by storing them in different versions on its own server repository, i.e., GitHub. Git has all the functionality, performance, security, and flexibility that most of the development teams and individual developers need.

## Why Git Version Control

Below are some of the facts that make Git so popular:

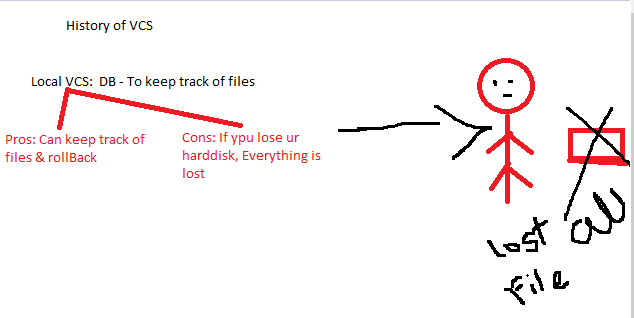
* **Works offline**: Git provides users very convenient options such as allowing them to work both online and offline. With other version control systems like SVN or CVS, users need to have access to the Internet to connect to the central repository.
* **Undoes mistakes**: Git allows us to undo our commands in almost every situation. We get to correct the last commit for a minor change, and also we can revert a whole commit for unnecessary changes.
* **Restores the deleted commits**: This feature is very helpful while dealing with large projects when we try out some experimental changes.
* **Provides security**: Git provides protection against secret alteration of any file and helps maintain an authentic content history of the source file.
* **Guarantees performance**: Being a distributed version control system, it has an optimized performance due to its features like committing new changes, branching, merging, comparing past versions of the source file, etc.
* **Offers flexibility**: Git supports different nonlinear development workflows, for both small and large projects.

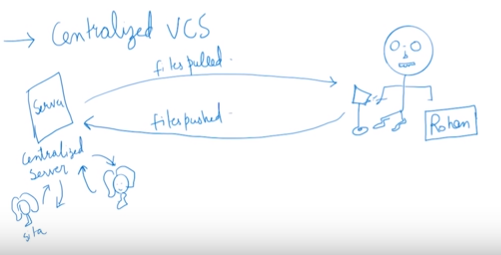
Eg:



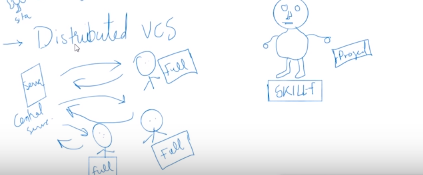
* Easily Recover File.
* Who Introduced an issue and When?
* Roll back to previously working state.

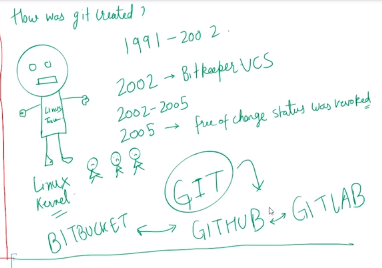
**History of Version Control System:**

Only one person has every files and collaboration was difficult



Can lose data from server

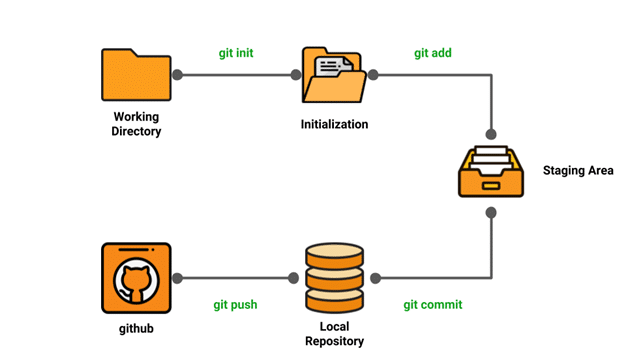




Pull -> get all project/ history at Pulling time->Everyone has his/her own files complete backup -> if lost-> pull from serve-> if server lost-> can setup server again and get all files 😊

## How does Git work/Git Life Cycle

* **Local working directory**: The first stage of a Git project life cycle is the local working directory where our project resides, which may or may not be tracked.



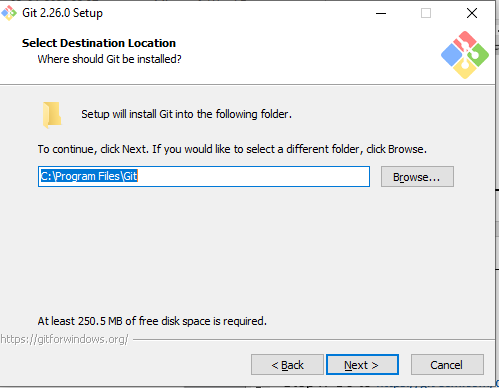
* **Initialization:** To initialize a repository, we give the **command git init**. With this command, we will make Git aware of the project file in our repository.
* **Staging area:** Now that our source code files, data files, and configuration files are being tracked by Git, we will add the files that we want to commit to the staging area by the git add command. This process can also be called indexing. The index consists of files added to the staging area.
* **Commit:** Now, we will commit our files using the **git commit -m ‘our message’ command.**

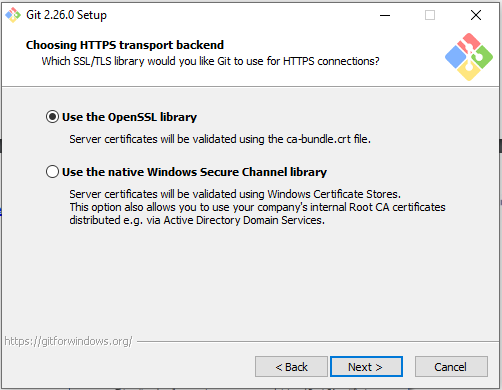
*We have successfully committed our files to the local repository. But how does it help in our projects? The answer is, when we need to collaborative projects, files may have to be shared with our team members.*

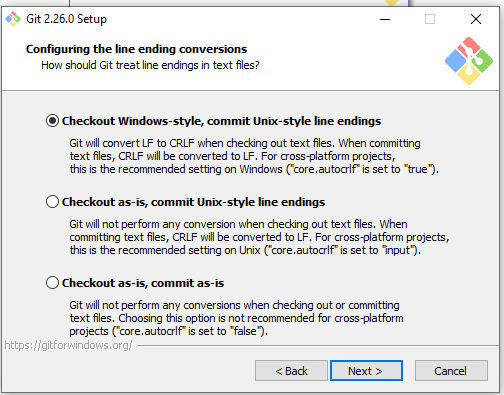
*This is when the next stage of the Git life cycle occurs, i.e., in GitHub, we publish our files from the local repository to the remote repository. And how do we do that? We do that by using the git push command.*

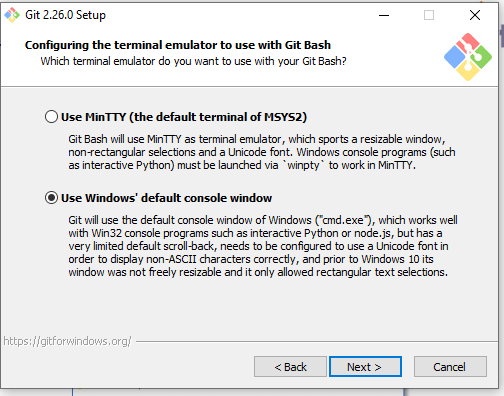
## Installing Git

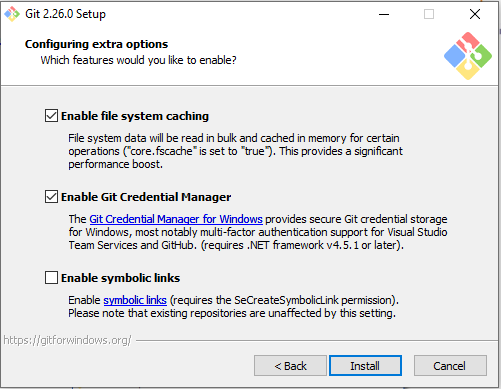
Step1: Go to <https://git-scm.com/download/win>

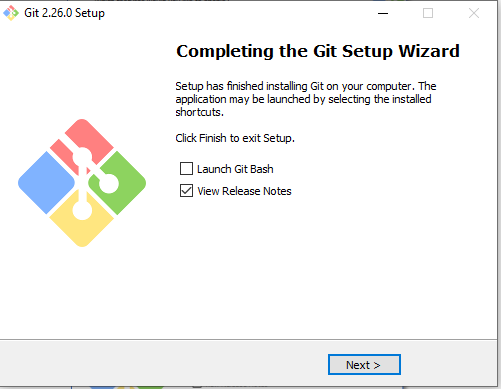








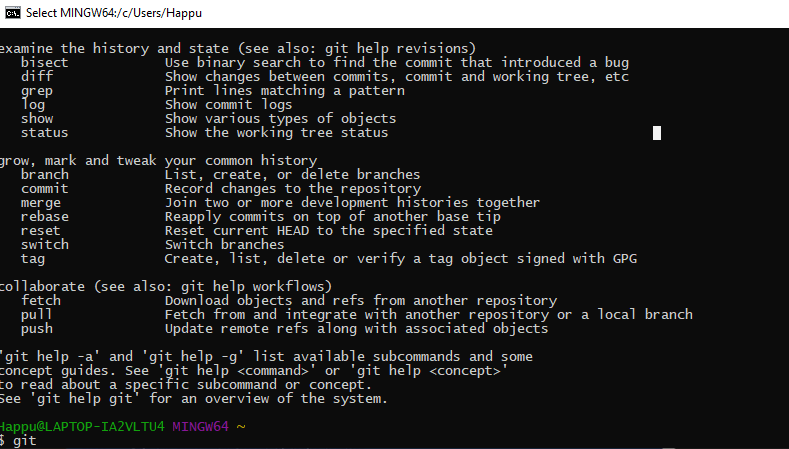


] 

After Insatallation we got Two thigns:

1: Git Command Line Tool (Interface)

2: Git Bash (TEminal Program)



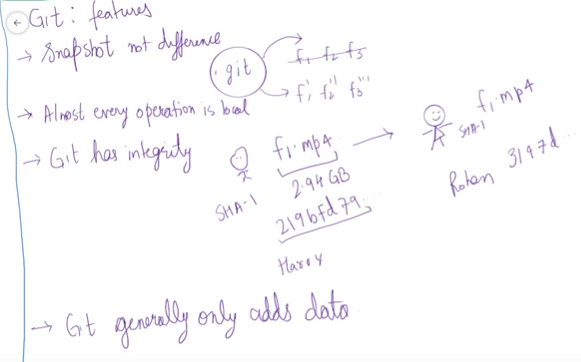
### Environment Setup

* Create a GitHub account
* Configure Git
* Create a local repository

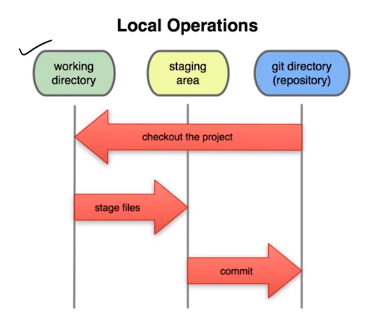
**Creating a GitHub account:**

* Go to https://Github.com
* Create a GitHub account
* Login

GIT Features:



**Git – Three stage architecture/ Workflow:**



### Repository

This is a local place/hub on your machine where the entire snapshot of your project is stored. Every minor change is stored and retrievable. Repository logs can easily be viewed and retrieved,

i.e, you can jump to any older state (time-travel in history) of your code.

### Working Directory

This is a local working copy of the project’s latest code.

### Index (Staging area, Cache)

An Index is the snapshot of your next commit.

It is where the code moves to once you stage or add (git add) the code in the Working directory. A Staging area is like a cache memory and acts as a middle layer between the working directory (where the code is developed) and the local repository (where the code resides).

This layer gives a quick preview of the project snapshot that you are about to **commit**. You can still edit (add, modify, or delete) your code in the Index.

You could also revert back an older version (state) of a project here.

### Commit

A commit is the latest snapshot (state) of a project.

A commit is a git object that stores the following attributes: commit ID, author name, authored date, and a commit message (header and body).

**Git Commands:**

#### 1) git config

**Utility** : To set your user name and email in the main configuration file.  
**How to** : To check your name and email type in **git config --global user.name** and **git config --global user.email**. And to set your new email or name **git config --global user.name = “Asha Thakur”** and **git config --global user.email =”ashathakur29@gmail.com”**

Syntax:

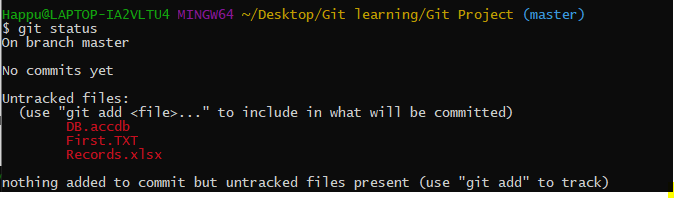
Usage: git config –global user.name “[name]”

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

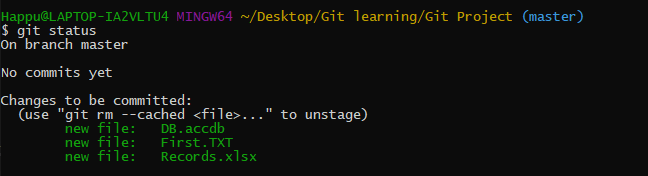
#### 4) git status

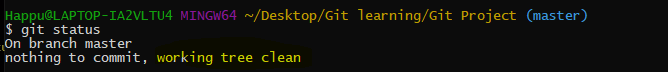
**git status**: After creating a new file, we can use the git status command and see the status of the files in the master branch.

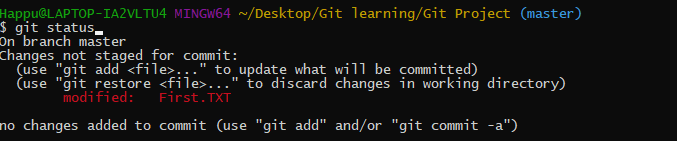




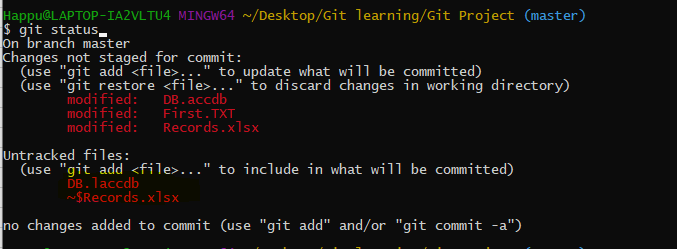
**git status**: Now, we will add the humble.txt file to the staging environment by using git add, before going ahead to the staging to see the change using git status.







Edited Two file



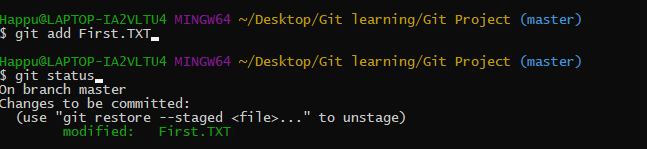
Now I want only one file to commit:

## Master

A master is the main default local branch when the project is first created as a git project.

#### git add:

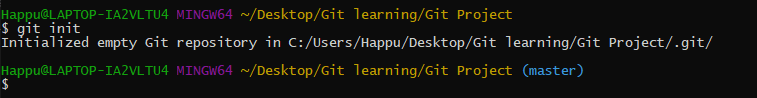
**Utility** : adds changes to stage/index in your working directory.  
**How to** : **git add .**



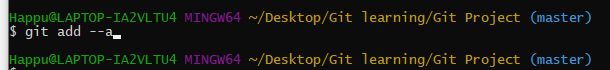
On Branch Master:

git init

**Utility** : To initialise a git repository for a new or existing project.  
**How to** : **git init** in the root of your project directory.



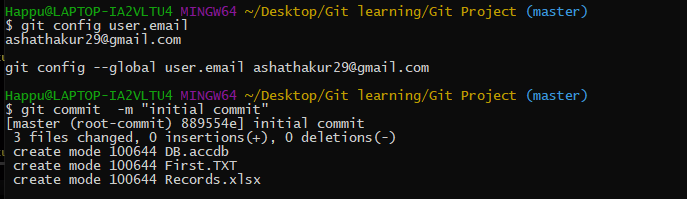
Git add –a:

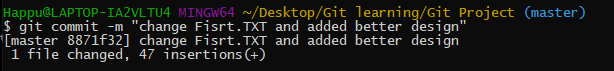


git commit

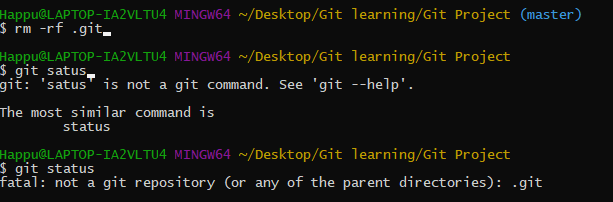
**Utility** : commits your changes and sets it to new commit object for your remote.  
**How to** : **git commit -m”sweet little commit message”**

Git commit – m “initial commit”

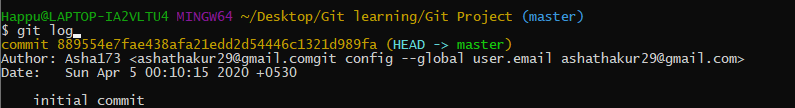




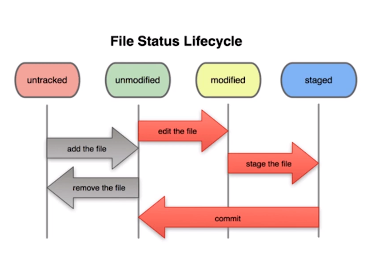
Command **rm -rf .git** : delete our content as .git will deleted will not able to track

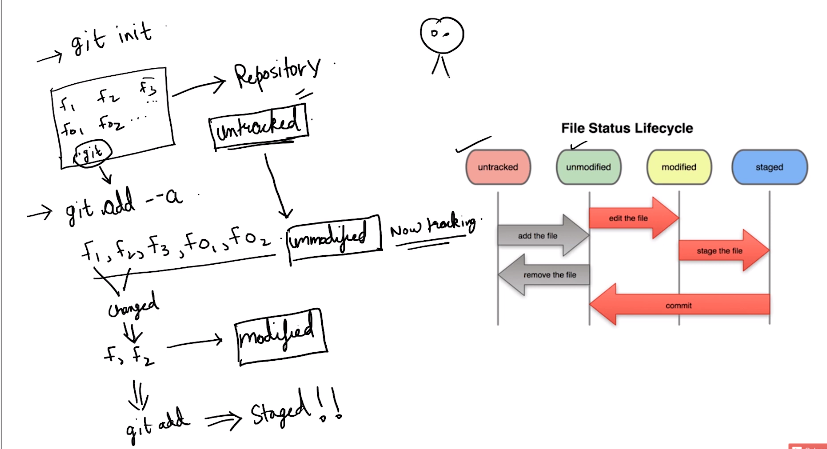


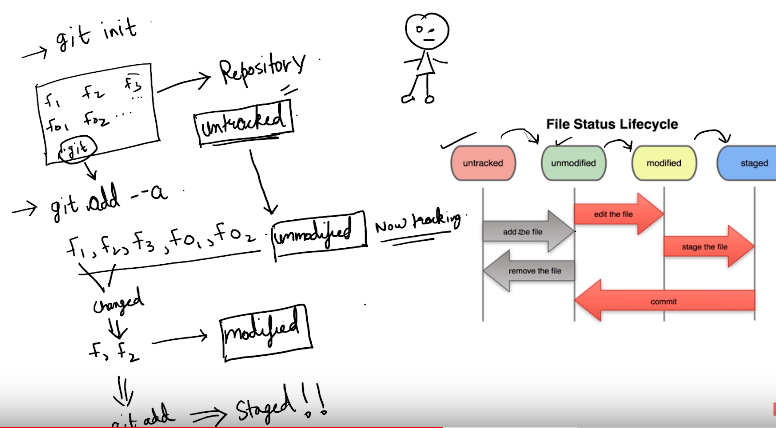
Git log:



**File life cycle:**







**git touch**: To add files to a project, we can use git touch. Let us see how we can add a file to the Git repository we have just created  
**Step 1**: Create a new file with the command touch  
**Step 2**: See the files present in our master branch

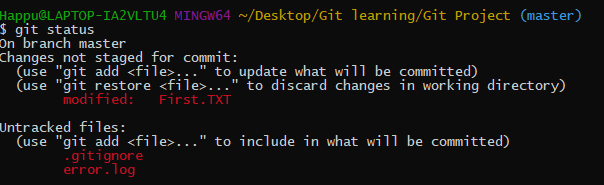
Touch Error.log will create automatic file.

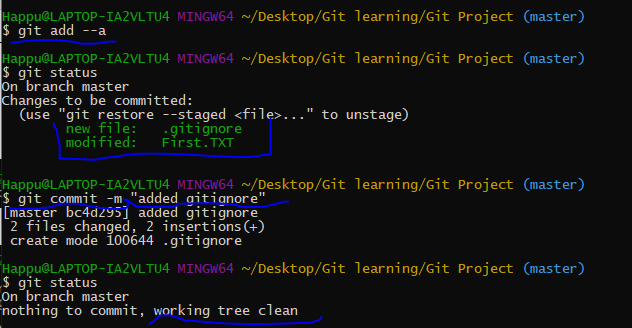


Git Ignore:

touch .gitignore





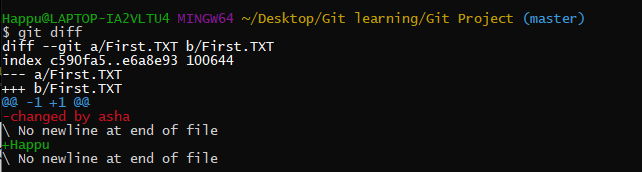


\*.Log

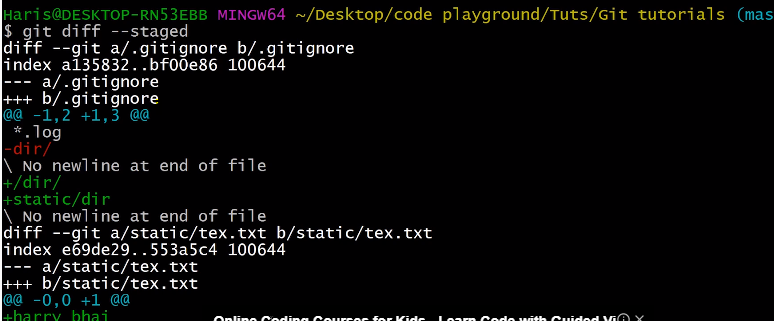
Dir/

/dir/

Git diff: Compare statging area and working directory



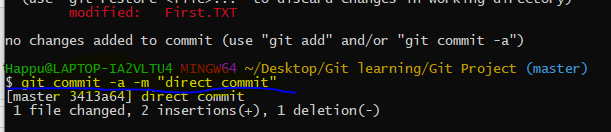
Git diff – stage: compare krta hai Pichele commit ko apki abhi ke staging area se

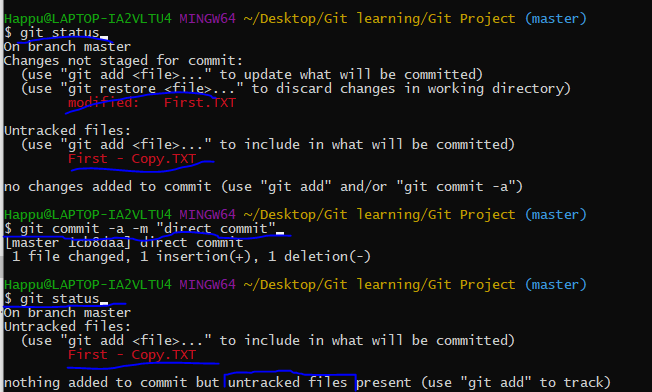


# **Git: Skipping The Staging Area**

Git commit -a -m “Direct Commit”

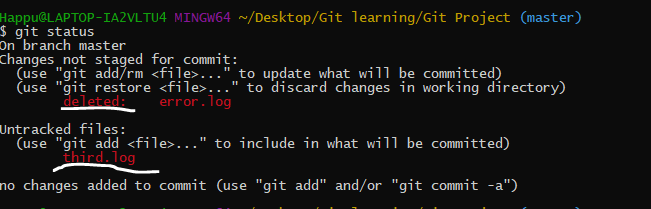
By this we will be able to commite tracked files directly without staging from unmodified to commit..

untracted file will not commit

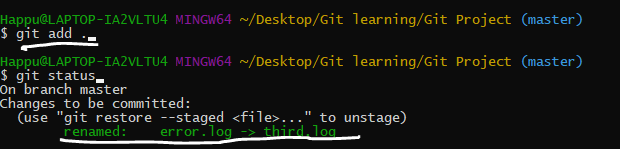


# **Moving and Renaming Files In Git**

Renamed error.log file to third.txt then checked status:



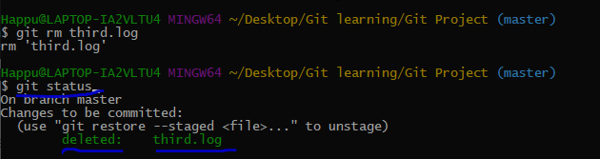
Git will say deleted error.log rather than rename, after doing git ADD. GIT WILL realized that ohhhh it was ranamed



Command to remove files:

Git rm file name:

Git rm third.log



=============================================================

\*.error.log

/dir/

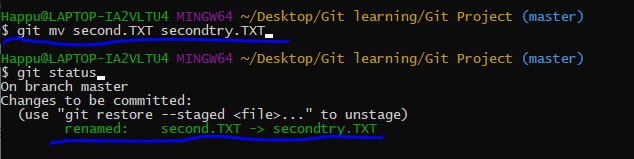
===========================================================

Rename file:

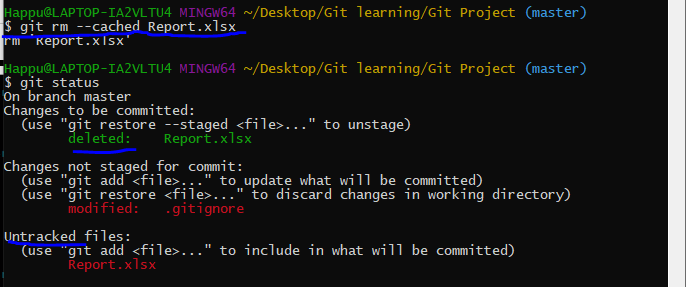
Git mv filename newfilename

Git mv second.TXT secondtry.TXT

This will rename file + staging also done at same time



Modified and Untracked Report.txt file



# **Git Log: Viewing & Changing Commits In Git**

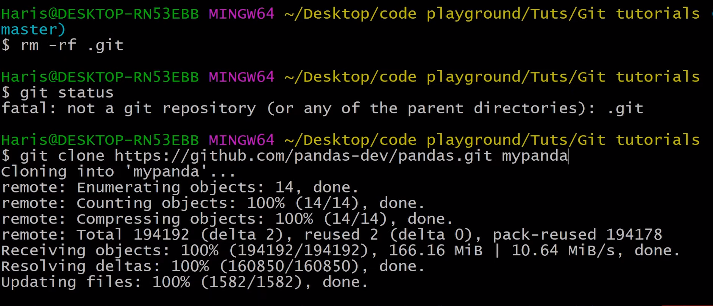
Rm -rf .git: to remove Repository

Clone:

A clone is a working copy of a remote repository.

The ‘git clone’ command downloads the remote repository and creates a working directory on your local machine.

In addition, this command also stores a remote handler or pointer reference from the local repository to the remote repository.

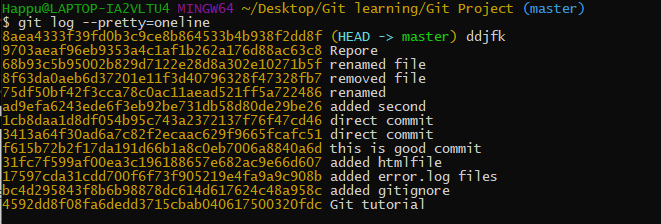


Gitlog – p

Gitlog -m

Gitlog – state

git log --pretty=oneline



git log --pretty=short

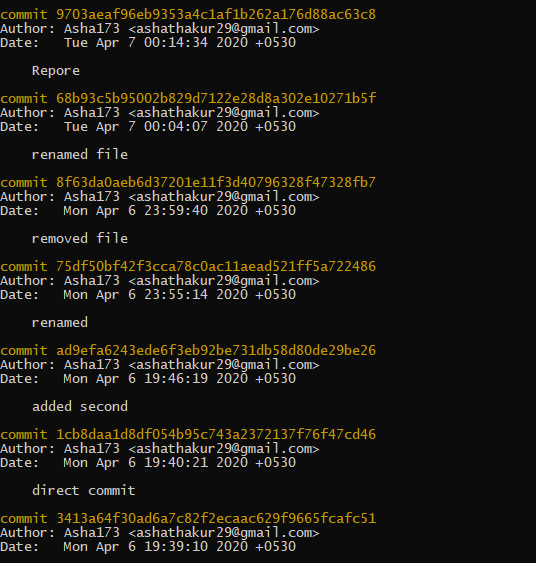
git log --pretty=full

Timer filter:

git log --since=2weeks

git log --since=2days

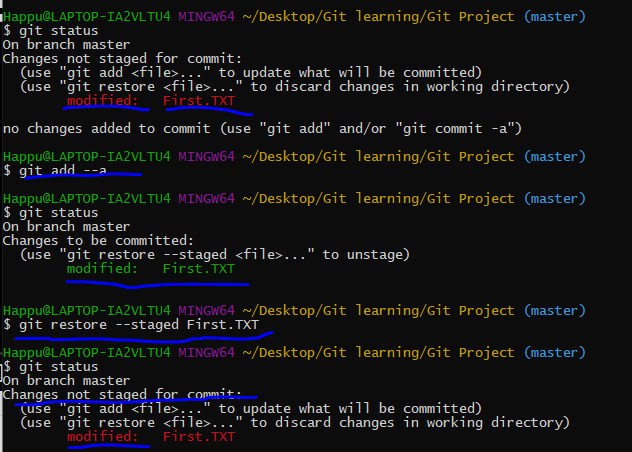
git log --since=2years



# **Unstaging & Unmodifying Files In Git**

Git restore –staged file name

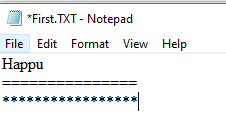
Git restore –staged first.txt

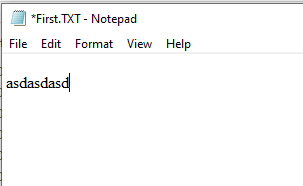


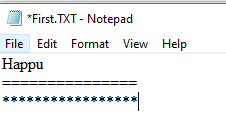
To Restore deleted/changed things we can restore/unmodify file:

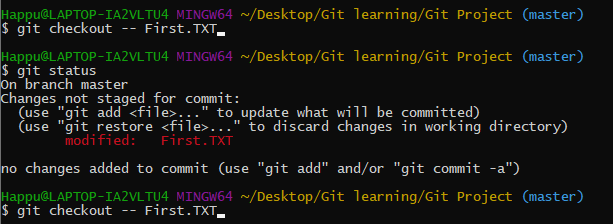
Git checkout – first.TXT



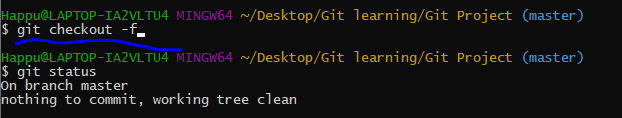








Git checkout -f : apki file last chenges hat jate hai aur hum clearn state me aa jate hai



# **GitHub: Working with Remote Repositories**

Git remote: git is distributed version system means apka ek server hota hai jisme aap apna code +history milti hai ager wo pull krta hai.

* **git pull**

When it comes to syncing a remote repository, the pull command comes in handy. Let us take a look at the commands that can leads to the pulling operation in Git.

* remote origin
* pull master

We will use these as shown below:  


Push krta hunt oh sirf incremental push hofa jo ki maine push ki hai

Pull: detail:code ko github se aone computer me lekar ana

Push github me dal dena

* **git push**

git push origin EnterBranchName

Don’t get confused by the word ‘origin’. What exactly is this command referring to?

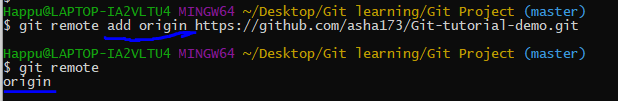
Instead of writing the whole Git URL, like **git push Git@Github.com:Git/Git.Git ourbranchname**, we can just use the following command, assuming that our brach name is ‘branch1’:

git push origin branch1

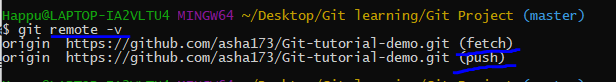
### push an existing repository from the command line

git remote add origin https://github.com/asha173/Git-tutorial-demo.git

git push -u origin master

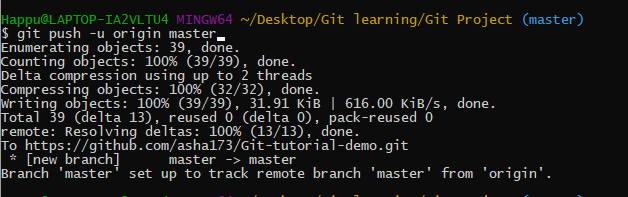


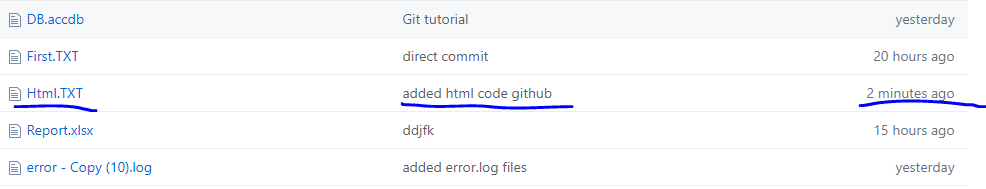
Git remote -v



To push our changes:

Check access by git push -u origin master

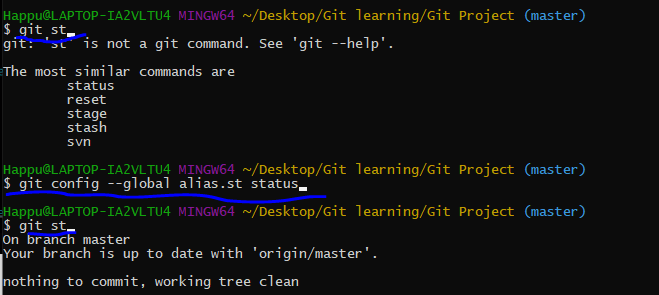




# **Setting Alias In Git**

Alis: means bada command likhne ki bajha koi chota command likhna

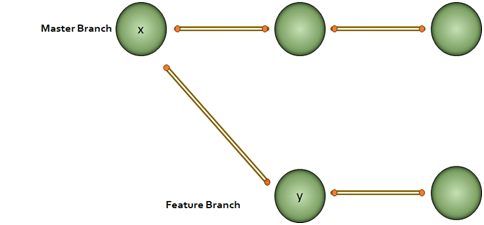
Eg: git config –global alias.st status



# **Git: Creating & Switching Branches In Git**

## Branching in Git

Branching, we mainly make use of the **git checkout** and **git branch** commands.

B  
Every node in the figure above represents commits. We have to note that for the ‘y’ node in the feature branch, ‘x’ is the base.

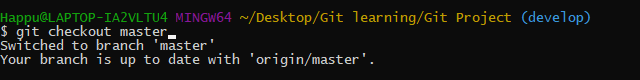
Why would we do branching in the first place? Say, we want to modify something but don’t want to make any change in the main project. This is when we make a branch out of the master branch, i.e., if we want to create a new branch to add a feature to the main project, we will make a branch out of it with the help of the following steps:

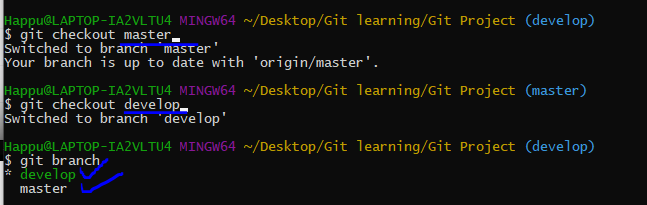
**Step 1**: We will run **git checkout -b <new branch name>**  
**Step 2**: Then, we will use the git branch command to confirm that our branch is created





Can swich to master again





Git commit -a -m “use for direct commit”

We can see from the above image that we have created a branch called ‘**develop**’ and we automatically got landed in the new branch. Again, just to see which branch we are currently in, we run another command, git branch.

**Note:** The **\*** mark before the branch name shows that it is the current branch.

Now, how to change a branch to the master branch? For that, we use the following command:

**git checkout master**

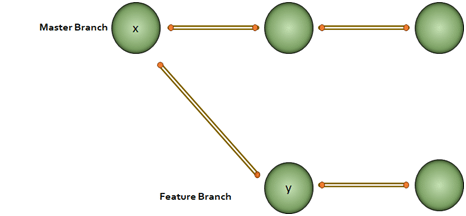
This command switches over to the specified branch and displays the current project state as it is in the branch.

It also restores the earlier working tree files.

## Merging in Git (git checkout, git add, git log, git merge, merging conflicts, and rebasing)

Now that we have learned how to create a branch and work on it, let us take a look at the merge feature in Git by merging the branch we created to the master branch.

Let’s take the above example. Say, we have a master branch and a feature branch.

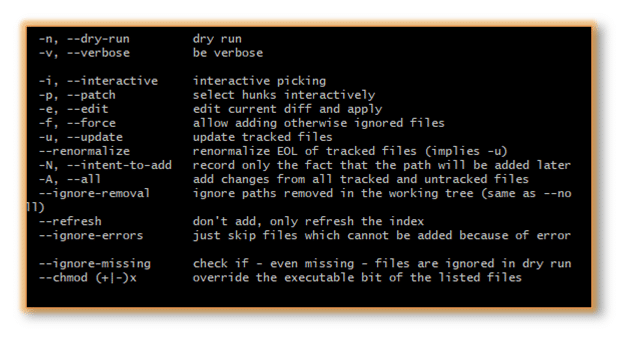
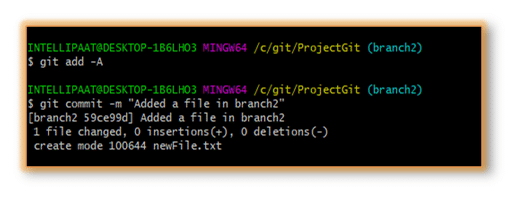
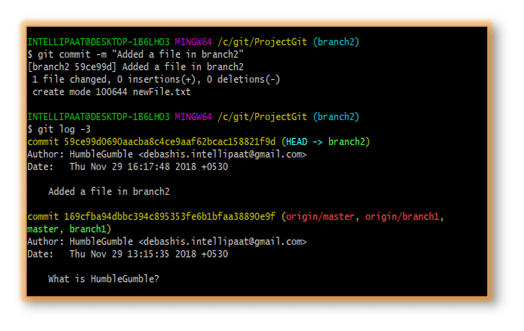
The merge commit represents every change that has occurred on the feature branch since it got branched out from the master.  
**Note**: Even after merging, we can go on with our work on both the master and the feature branches independently.

Let us see how to perform merging:

**Step 1**: Create a new branch called ‘branch2’



**Step 2**: Create a new file in the branch  
**Step 3**: Add changes from all tracked and untracked files

**Note**: Refer to the following **git add** attributes  
In our case, we have given the command as **git add -A** and after that, we will commit one sentence as shown below:  
**Step 4**: Check the last three logs by running the command: **git log -3**  
We have created another branch on our master branch. Now, we will see how to perform **merging**.

Let us get inside the master branch using the following command:

git checkout master

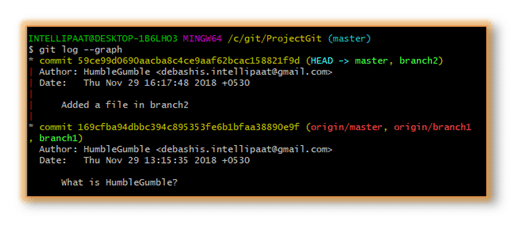
After that, we will perform merging with the help of the below command:

git merge branch2



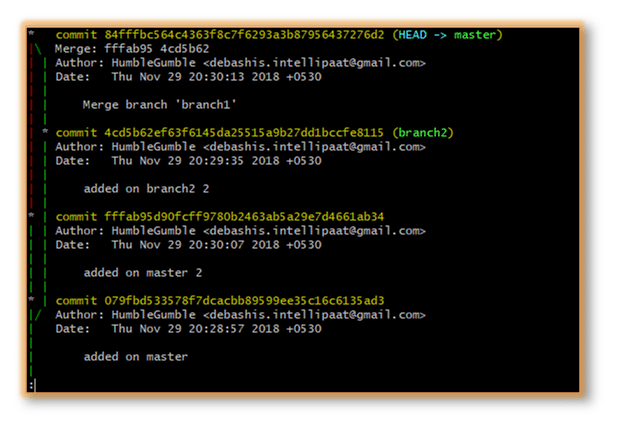
Now, we have successfully merged the feature branch into the master branch. Let us take a look at how it looks inside the master branch using the following command:

git log --graph



We can see the graph in the above image on the left. But **why the graph is linear?**

In our ProjectGit master branch, we did branching and committing. But after the branching, the master branch had not encountered any more commits. Hence, after merging, we have a linear graph.

Let us perform**git log –graph** with more than one commits in both master and feature branches that we have created.  
Here, we can see the graph with commits on both master and feature branches. The red part of the graph indicates the merging operation.

**Advantages of merging:**

* Merging allows parallel working in collaborative projects.
* It saves the time that is consumed by manual merging.

**The disadvantage of merging:**

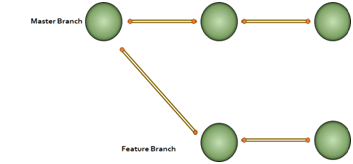
* Merging conflicts may occur while merging branches.

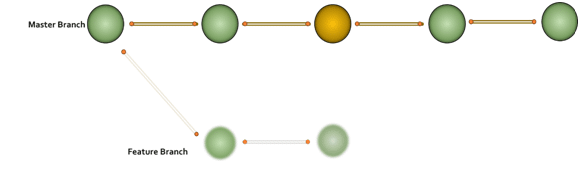
Now that we have successfully learned to branch and merge with Git and GitHub, further in this Git tutorial, let us look at yet another important Git operation, i.e., rebasing.

## Git Rebase

When our project becomes relatively large, the commit log and the history of the repository become messy. Here, we use rebasing. Rebasing will take a set of commits, copy them, and store them outside our repository. This helps us maintain a linear sequence of commits in our repository.

Let us take the same example. Here, we will rebase the master branch and see what happens.

**Note**: In rebasing, the base of the feature branch gets changed and the last commit of the master branch becomes the new base of the feature branch.

Now, let us perform rebasing in Git Bash.

**The advantage of rebasing:**

* Rebasing provides a cleaner project history.

**The disadvantage of rebasing:**

* In a collaborative workflow, re-writing the project history can be potentially catastrophic.

Now that we understood what branching, merging, and rebasing are, next in this Git tutorial, we will see where to use merging and rebasing.

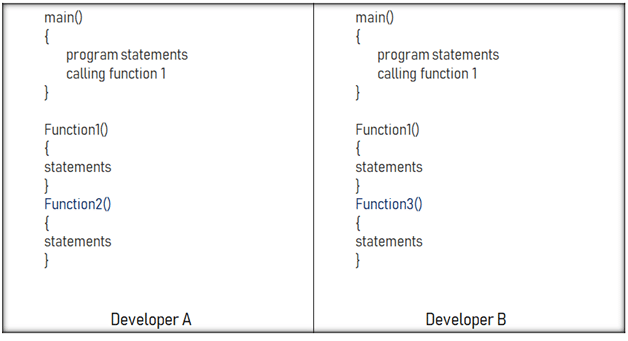
**The golden rule of Git merging and rebasing:**

* **When to use git merge**: We can use git merge while working on the public branches.
* **When to use git rebase**: We will use git rebase while working on the local branches.

As we have already mentioned earlier in this Git tutorial, one major disadvantage of merging is merge conflicts, which can be backbreaking for a team project. Let us understand how merge conflicts occur and how to resolve them.

## Git Merge Conflict and Rebase Conflict

This drawback of the merging operation in Git can be explained with a simple example shown below:

Say, we have two branches of the master branch, branch1 and branch2, and two developers are working on these two branches independently but on the same code file.

As we have seen in the above image, the developers have made the below-mentioned modifications:

* Developer A added a function called ‘function2’ to the main code.
* Developer B added a different function called ‘function3’ to the same code file.

How to merge these two modifications? That is where the merging conflict occurs.

Similarly, git rebase also exhibits conflicts

### git hash or the ‘SHA-1’

The unique commit ID is called a ‘git hash’ or ‘[SHA-1](https://en.wikipedia.org/wiki/SHA-1)’. Every filesystem change (add, delete, edit, move, copy, rename, file permissions, etc.) is treated as a file and its contents are converted into a unique SHA-1 code.

Here is a sample commit ID: 8db083e7df7c9241e640b66c89c6f02649ac885a

They are often referred by the first 7 unique digits, such as 8db083e

You never ever have to remember the entire hash ID. git has a beautiful way of handling these commit (hash) IDs using references such as branches and tags.

**Difference between Git Pull and Fetch:**

Both fetch and pull are used to download data from the remote repository. Which is appropriate, depends on your need.

command for fetch in git is

**$ git fetch origin**

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.

  Pull on the other hand downloads the data and integrates it with the current working files.

 command for pull is

**$ git pull origin master**

 So when you want to update your current HEAD branch with the latest changes from the remote you always use pull. Since git pull, merges the remote repository changes with the local ones it has a few things to be kept in mind

 ->Merge conflict: That is when you try to merge files where either two people make changes to the same line in the same file or when two people add files with same name unknowingly etc.

->Its recommended to start a git pull only after committing the files you currently working with or save them temporarily and have git pull with a clean working copy.

# What is a bare Git repository?

by Ryan Irelan

The standard way of initializing a new Git repository is to run git init. The directory in which you do this will be become the [Working Tree](https://mijingo.com/blog/what-is-the-working-tree-in-git) for the repository.

As part of the initialization process, Git creates a .git directory (which his hidden by default because of the . in the name) that contains the repository itself. This is brains of the repository; it's where Git tracks your changes, stores commit objects, refs, etc. You probably only rarely interact with that hidden directory.

Okay, so all of this is to lay the groundwork for understanding a bare Git repository. What the heck is it?

A bare Git repository is a repository that is created without a Working Tree. Go ahead and create one to see.

git init --bare .

Run ls on that directory and you won't see a Working Tree but just the contents of what is typically in the .git directory.

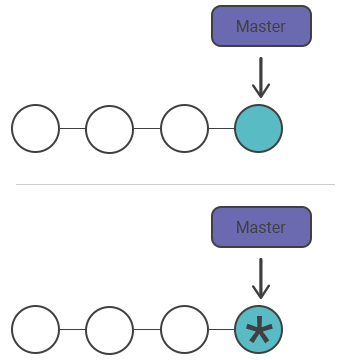
Why this setup?

A bare Git repository is typically used as a [Remote Repository](https://mijingo.com/blog/what-is-a-git-remote-repository) that is sharing a repository among several different people. You don't do work right inside the remote repository so there's no Working Tree (the files in your project that you edit), just bare repository data.

**Amending a commit:**

## Definition and usage

While working with Git, you can often forget to format the commit or to stage a file. The git commit -- amend command is the easiest way of correcting such mistakes. It is used to edit the latest commits. Instead of creating a completely new commit, you can run this command for combining staged changes with the previous commit. Besides, this command can modify the previous commit message without changing its snapshot. Take into account, that git commit -- amend replaces the current commit entirely. We are going to see some ways of usage of git commit -- amend.



## Changing the most recent Git commit message

If you have committed and made a mistake in the commit log message, you can execute git commit -- amend for modifying the log message of the previous commit without changing its snapshot. You can pass in a new message from the command line without receiving a prompt for opening an editor by using -m option.

git commit --amend -m "an updated commit message"

## Changing committed files

Let’s imagine that you have modified some files that you want to commit in a singular snapshot, but you have forgotten to add one of the files the first time around. You can solve this problem, if you stage the other file and commit by using the --amend flag. Add --no-edit flag to modify your commit without changing the commit message. As a result, the wrong commit will be replaced by the right one, and it will look like the following example:

# Modify project.py and text.py git add project.py git commit

# You forgot to add the changes from text.py git add text.py

git commit --amend --no-edit

### The inadmissibility of amending public commits

In fact, amended commits are new ones and after amending a commit, the previous commit will not be removed from the current branch. That’s why you should never amend a commit which is being developed by other team members because it will cause a lot of problems for collaboration. So you notice that the consequences are the same as in the case of resetting a public snapshot