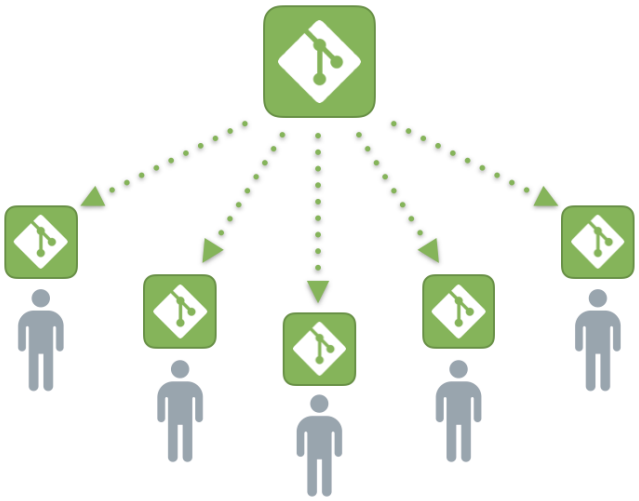
Version Control System

Version Control System (VCS) is a software that helps software developers to work together and maintain a complete history of their work.



Listed below are the functions of a VCS:

• Allows developers to work simultaneously.

• Does not allow overwriting each other's changes.

• Maintains a history of every version.

Following are the types of VCS:

• Centralized version control system (CVCS).

• Distributed/Decentralized version control system (DVCS).

we will concentrate only on distributed version control system

and especially on Git. Git falls under distributed version control system.

What is GitHub?

GitHub is a Git repository hosting service. GitHub also facilitates with many of its features, such as access control and collaboration. It provides a Web-based graphical interface.

GitHub is an American company. It hosts source code of your project in the form of different programming languages and keeps track of the various changes made by programmers.

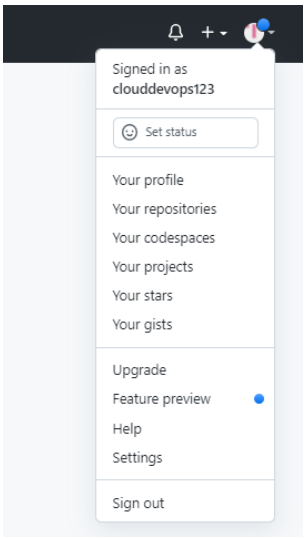
It offers both **distributed version control and source code management (SCM)** functionality of Git. It also facilitates with some collaboration features such as bug tracking, feature requests, task management for every project.

The key benefits of GitHub are as follows.

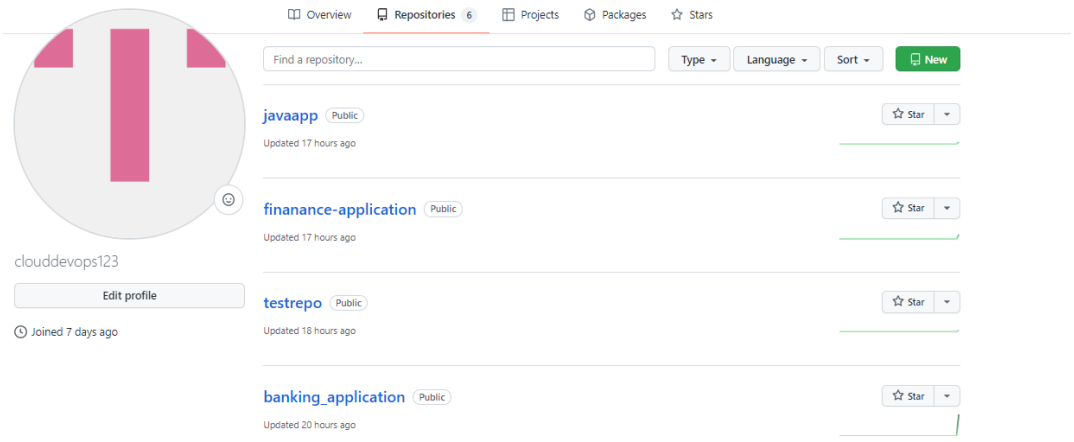
- It is easy to contribute to open source projects via GitHub.
- It helps to create an excellent document.
- You can attract recruiter by showing off your work. If you have a profile on GitHub, you will have a higher chance of being recruited.
- It allows your work to get out there in front of the public.
- You can track changes in your code across versions.

How to create Project repository in GITHub:

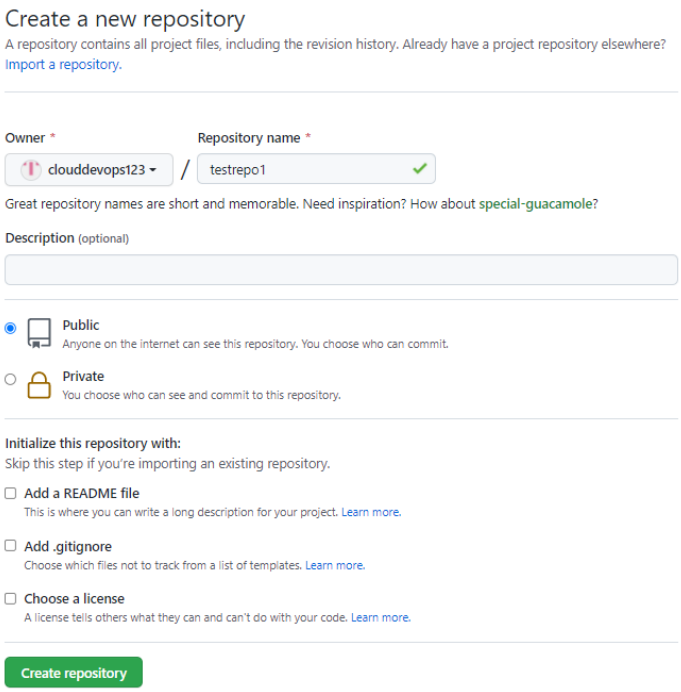
- Click on your profile and select Your repositories option



- Click on new option



- Name your repo and click on create



Difference between git and github:

Git	SVN
It's a distributed version control system.	It's a Centralized version control system
Git is an SCM (source code management).	SVN is revision control.
Git has a cloned repository.	SVN does not have a cloned repository.
The Git branches are familiar to work. The Git system helps in merging the files quickly and also assist in finding the unmerged ones.	The SVN branches are a folder which exists in the repository. Some special commands are required For merging the branches.
Git does not have a Global revision number.	SVN has a Global revision number.
Git has cryptographically hashed contents that protect the contents from repository corruption taking place due to network issues or disk failures.	SVN does not have any cryptographically hashed contents.
Git stored content as metadata.	SVN stores content as files.
Git has more content protection than SVN.	SVN's content is less secure than Git.
Linus Torvalds developed git for Linux kernel.	CollabNet, Inc developed SVN.
Git is distributed under GNU (General public license).	SVN is distributed under the open-source license.

GIT

Git is an **open-source distributed version control system**. It is designed to handle minor to major projects with high speed and efficiency. It is developed to co-ordinate the work among the developers. The version control allows us to track and work together with our team members at the same workspace.

it is foundation of many services like **GitHub** and **GitLab**, but we can use Git without using any other Git services. Git can be used **privately** and **publicly**.

Git was created by **Linus Torvalds** in **2005** to develop Linux Kernel. It is also used as an important distributed version-control tool for **the DevOps**.

Git is easy to learn, and has fast performance. It is superior to other SCM tools like Subversion, CVS, Perforce, and ClearCase.

Features of Git

Some remarkable features of Git are as follows:

- **Open-Source**

  Git is an **open-source tool**. It is released under the **GPL** (General Public License) license.

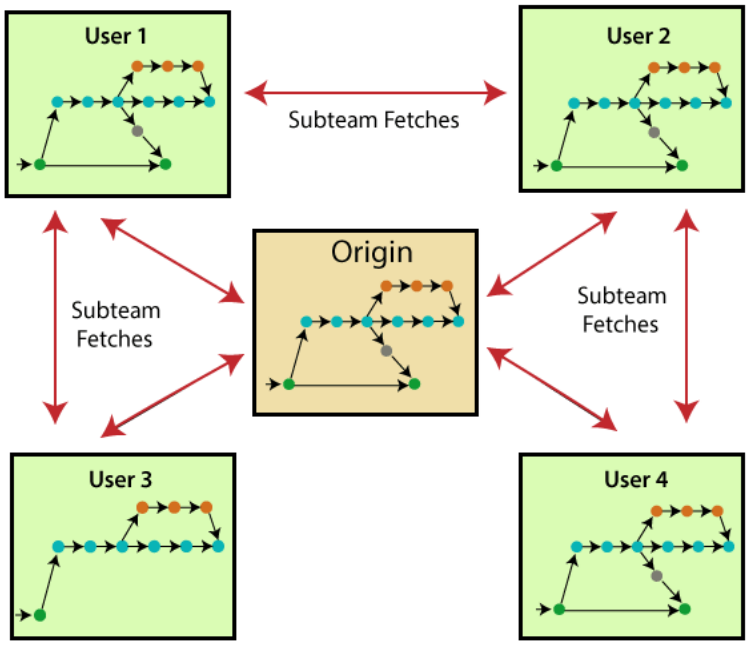
- **Scalable**

  Git is **scalable**, which means when the number of users increases, the Git can easily handle such situations.

- **Distributed**

  One of Git's great features is that it is **distributed**. Distributed means that instead of switching the project to another machine, we can create a "clone" of the entire repository. Also, instead of just having one central repository that you send changes to, every user

has their own repository that contains the entire commit history of the project. We do not need to connect to the remote repository; the change is just stored on our local repository. If necessary, we can push these changes to a remote repository.



- **Security**

  Git is secure. It uses the **SHA1 (Secure Hash Function)** to name and identify objects within its repository. Files and commits are checked and retrieved by its checksum at the time of checkout. It stores its history in such a way that the ID of particular commits depends upon the complete development history leading up to that commit. Once it is published, one cannot make changes to its old version.

- **Speed**

  Git is very **fast**, so it can complete all the tasks in a while. Most of the git operations are done on the local repository, so it provides a **huge speed**. Also, a centralized version control system continually communicates with a server somewhere. Performance tests conducted by Mozilla showed that it was **extremely fast compared to other VCSs**. Fetching version history from a locally stored repository is much faster than fetching it from the remote server. The **core part of Git** is **written in C**, which **ignores** runtime overheads associated with other high-level languages. Git was developed to work on the Linux kernel; therefore, it is **capable** enough to **handle large repositories** effectively. From the beginning, **speed** and **performance** have been Git's primary goals.

- **Supports-non-linear-development**

  Git supports **seamless branching and merging**, which helps in visualizing and navigating a non-linear development. A branch in Git represents a single commit. We can construct the full branch structure with the help of its parental commit.

- **Branching-and-Merging**

  Branching and merging are the **great feature**s of Git, which makes it different from the other SCM tools. Git allows the **creation of multiple branches** without affecting each other. We can perform tasks like **creation**, **deletion**, and **merging** on branches, and these tasks take a few seconds only. Below are some features that can be achieved by branching:

  - We can **create a separate branch** for a new module of the project, commit and delete it whenever we want.
  - We can have a **production branch**, which always has what goes into production and can be merged for testing in the test branch.
  - We can create a **demo branch** for the experiment and check if it is working. We can also remove it if needed.
  - The core benefit of branching is if we want to push something to a remote repository, we do not have to push all of our branches. We can select a few of our branches, or all of them together.

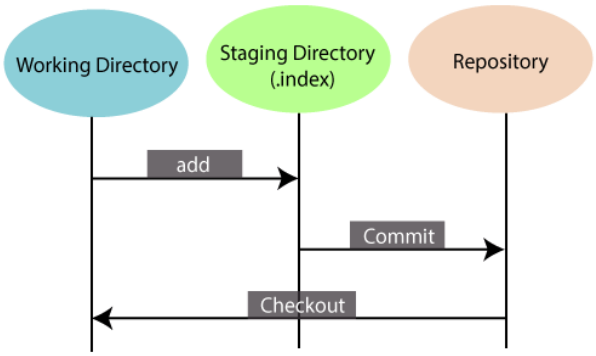
- **Data-Assurance**

  The Git data model ensures the **cryptographic integrity** of every unit of our project. It provides a **unique commit ID** to every commit through a **SHA algorithm**. We can **retrieve** and **update** the commit by commit ID. Most of the centralized version control systems do not provide such integrity by default.

- **Staging-Area**

  The **Staging area** is also a **unique functionality** of Git. It can be considered as a **preview of our next commit**, moreover, an **intermediate area** where commits can be formatted and reviewed before completion. When you make a commit, Git takes changes that are in the staging area and make them as a new commit. We are allowed to add and remove changes from the staging area. The staging area can be considered as a place where Git stores the changes.

  Although, Git doesn't have a dedicated staging directory where it can store some objects representing file changes (blobs). Instead of this, it uses a file called index.



Another feature of Git that makes it apart from other SCM tools is that **it is possible to quickly stage some of our files and commit them without committing other modified files in our working directory.**

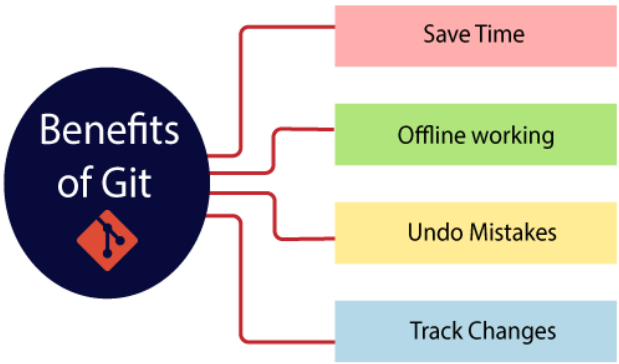
- **Maintain-the-clean-history**

  Git facilitates with Git Rebase; It is one of the most helpful features of Git. It fetches the latest commits from the master branch and puts our code on top of that. Thus, it maintains a clean history of the project.

Benefits of Git

A version control application allows us to **keep track** of all the changes that we make in the files of our project. Every time we make changes in files of an existing project, we can push those changes to a repository. Other developers are allowed to pull your changes from the repository and continue to work with the updates that you added to the project files.

Some **significant benefits** of using Git are as follows:



- **Saves-Time**

  Git is lightning fast technology. Each command takes only a few seconds to execute so we can save a lot of time as compared to login to a GitHub account and find out its features.

- **Offline-Working**

  One of the most important benefits of Git is that it supports **offline working**. If we are facing internet connectivity issues, it will not affect our work. In Git, we can do almost everything locally. Comparatively, other CVS like SVN is limited and prefer the connection with the central repository.

- **Undo-Mistakes**

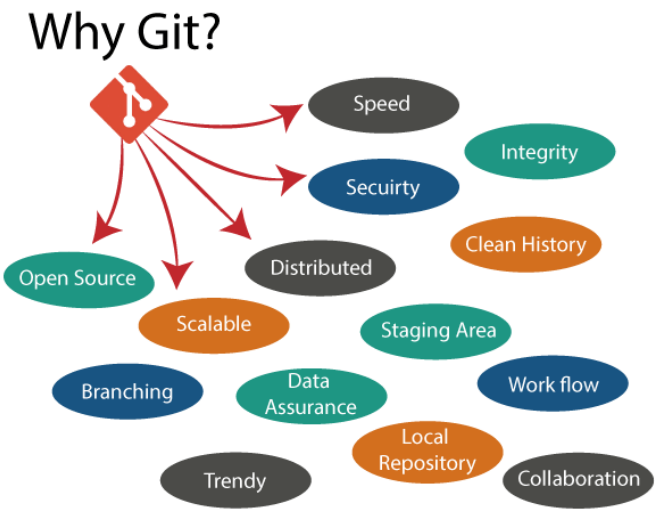
  One additional benefit of Git is we can **Undo** mistakes. Sometimes the undo can be a savior option for us. Git provides the undo option for almost everything.

- **Track-the-Changes**

  Git facilitates with some exciting features such as **Diff, Log,** and **Status**, which allows us to track changes so we can **check the status, compare** our files or branches.

Why Git?

We have discussed many **features** and **benefits** of Git that demonstrate the undoubtedly Git as the **leading version control system**. Now, we will discuss some other points about why we should choose Git.



- **Git-Integrity**

  Git is **developed to ensure** the **security** and **integrity** of content being version

controlled. It uses checksum during transit or tampering with the file system to confirm that information is not lost. Internally it creates a checksum value from the contents of the file and then verifies it when transmitting or storing data.

- **Trendy-Version-Control-System**
  Git is the **most widely used version control system**. It has **maximum projects** among all the version control systems. Due to its **amazing workflow** and features, it is a preferred choice of developers.
- **Everything-is-Local**
  Almost All operations of Git can be performed locally; this is a significant reason for the use of Git. We will not have to ensure internet connectivity.
- **Collaborate-to-Public-Projects**
  There are many public projects available on the GitHub. We can collaborate on those projects and show our creativity to the world. Many developers are collaborating on public projects. The collaboration allows us to stand with experienced developers and learn a lot from them; thus, it takes our programming skills to the next level.
- **Impress-Recruiters**
  We can impress recruiters by mentioning the Git and GitHub on our resume. Send your GitHub profile link to the HR of the organization you want to join. Show your skills and influence them through your work. It increases the chances of getting hired.

Prerequisites

Git is not a programming language, so you should have the basic understanding of Windows commands only.

Git - Environment Setup

Before you can use Git, you have to install and do some basic configuration changes. Below are the steps to install Git client on Ubuntu and Centos Linux.

Installation of Git Client

If you are using Debian base GNU/Linux distribution, then apt-get command will do the needful.

[ubuntu ~]$ sudo apt-get install git-core

[sudo] password for ubuntu:

[ubuntu ~]$ git --version

git version 1.8.1.2

And if you are using RPM based GNU/Linux distribution, then use yum command as given.

[CentOS ~]# yum -y install git

[CentOS ~]# git --version

git version 1.7.1

Git Environment Setup

The Git config command

Git supports a command called **git config** that lets you get and set configuration variables that control all facets of how Git looks and operates. It is used to set Git configuration values on a global or local project level.

Setting **user.name** and **user.email** are the necessary configuration options as your name and email will show up in your commit messages.

Setting username

The username is used by the Git for each commit.

$ git config --global user.name "Himanshu Dubey"

Setting email id

The Git uses this email id for each commit.

$ git config --global user.email himanshudubey481@gmail.com

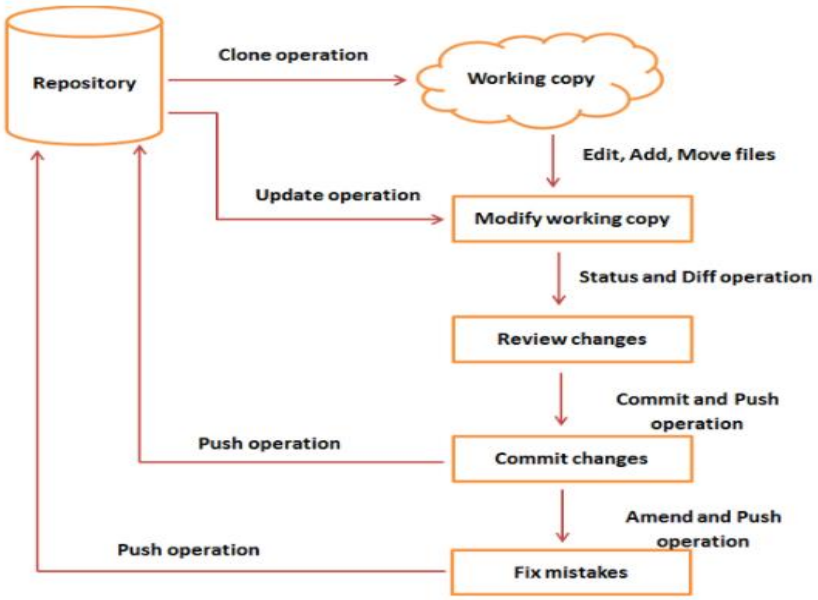
Checking Your Settings

You can check your configuration settings; you can use the **git config --list** command to list all the settings that Git can find at that point.

$ git config -list

What is a Git repository?

A Git repository is a virtual storage of your project. It allows you to save versions of your code, which you can access when needed.



Getting a Git Repository

There are two ways to obtain a repository. They are as follows:

1. Clone a remote repository (already exists on a server).
2. Create a local repository and make it as Git repository.

1. Cloning an Existing Repository

We can clone an existing repository. Suppose we have a repository on a version control system like subversion, GitHub, or any other remote server, and we want to share it with someone to contribute. The git clone command will make a copy for any user to contribute.

We can get nearly all data from server with git clone command. It can be done as:

Syntax:

$ git clone **<Repository** URL**>**

2. Create a local repository and make it as Git repository.

- Create a project folder:
- mkdir test
- Change to test project folder
- cd mkdir
- Create few files
- touch abc.txt
- create a repo with same in github as well & Add remote url of github repo using

  $ git remote add <githubrepourl>

  Add files to using

  $ git add .

  Commit files using

  $ git commit -m “adding”

  $ git push origin main or $ git push origin master

Important Terminologies:

Git Init

The git init command is the first command that you will run on Git. The git init command is used to create a new blank repository. It is used to make an existing project as a Git project. Several Git commands run inside the repository, but init command can be run outside of the repository.

The git init command creates a .git subdirectory in the current working directory. This newly created subdirectory contains all of the necessary metadata. These metadata can be categorized into objects, refs, and temp files. It also initializes a HEAD pointer for the master branch of the repository.

Creating the first repository

Git version control system allows you to share projects among developers. For learning Git, it is essential to understand that how can we create a project on Git. A repository is a directory that contains all the project-related data. There can also be more than one project on a single repository.

We can create a repository for blank and existing projects. Let's understand how to create a repository.

Create a Repository for a Blank (New) Project:

To create a blank repository, open command line on your desired directory and run the init command as follows:

$ git init

Git Add

The git add command is used to add file contents to the Index (Staging Area)

.This command updates the current content of the working tree to the staging area. It also prepares the staged content for the next commit. Every time we add or update any file in our project, it is required to forward updates to the staging area.

The git add command is a core part of Git technology. It typically adds one file at a time, but there some options are available that can add more than one file at once.

The "index" contains a snapshot of the working tree data. This snapshot will be forwarded for the next commit.

The git add command can be run many times before making a commit. These all add operations can be put under one commit. The add command adds the files that are specified on command line.

Git add files

Git add command is a straight forward command. It adds files to the staging area. We can add single or multiple files at once in the staging area. It will be run as:

$ git add <File name>

The above command is added to the git staging area, but yet it cannot be shared on the version control system. A commit operation is needed to share it. Let's understand the below scenario.

We have created a file for our newly created repository in **NewDirectory**. To create a file, use the touch command as follows:

$ touch newfile.txt

And check the status whether it is untracked or not by git status command as follows:

$ git status

The above command will display the untracked files from the repository.. As we know we have created a newfile.txt, so to add this file, run the below command:

$ git add newfile.txt

Git Add All

We can add more than one files in Git, but we have to run the add command repeatedly. Git facilitates us with a unique option of the add command by which we can add all the available files at once. To add all the files from the repository, run the add command with **-A** option. We can use '.' Instead of **-A** option. This command will stage all the files at a time. It will run as follows:

```
$ git add -A
```

Or

```
$ git add .
```

Removing Files from the Staging Area

We can undo a git add operation. However, it is not a part of git add command, but we can do it through git reset command.

To undo an add operation, run the below command:

```
$ git reset <filename>
```

Git Commit

It is used to record the changes in the repository. It is the next command after the git add. Every commit contains the index data and the commit message. Every commit forms a parent-child relationship. When we add a file in Git, it will take place in the staging area. A commit command is used to fetch updates from the staging area to the repository.

The staging and committing are co-related to each other. Staging allows us to continue in making changes to the repository, and when we want to share these changes to the version control system, committing allows us to record these changes.

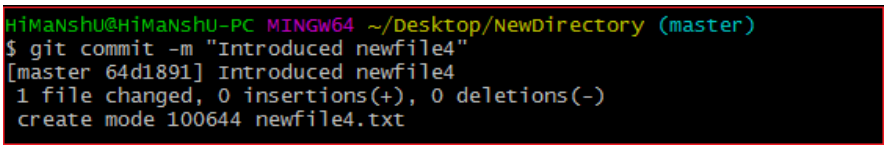
Commits are the snapshots of the project. Every commit is recorded in the master branch of the repository. We can recall the commits or revert it to the older version. Two different commits will never overwrite because each commit has its own commit-id. This commit-id is a cryptographic number created by **SHA (Secure Hash Algorithm)** algorithm.

Git commit -m

The -m option of commit command lets you to write the commit message on the command line. This command will not prompt the text editor. It will run as follows:

$ git commit -m "Commit message."

The above command will make a commit with the given commit message. Consider the below output:



Git Rm

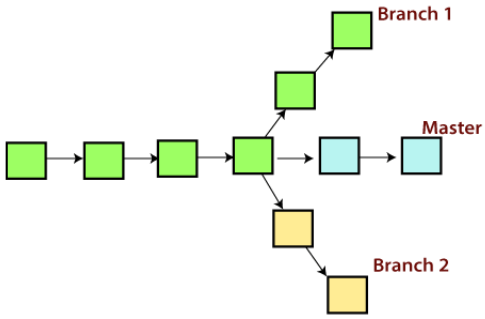
In Git, the term rm stands for remove. It is used to remove individual files or a collection of files. The key function of git rm is to remove tracked files from the Git index. Additionally, it can be used to remove files from both the working directory and staging index.

The git rm command is used to remove the files from the working tree and the index.

git rm <file Name>

Git Branch

A branch is a version of the repository that diverges from the main working project. It is a feature available in most modern version control systems. A Git project can have more than one branch. These branches are a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug, you spawn a new branch to summarize your changes. So, it is complex to merge the unstable code with the main code base and also facilitates you to clean up your future history before merging with the main branch.



Git Master Branch

The master branch is a default branch in Git. It is instantiated when first commit made on the project. When you make the first commit, you're given a master branch to the starting commit point. When you start making a commit, then master branch pointer automatically moves forward. A repository can have only one master branch.

Master branch is the branch in which all the changes eventually get merged back. It can be called as an official working version of your project.

Operations on Branches

We can perform various operations on Git branches. The **git branch command** allows you to **create**, **list**, **rename** and **delete** branches. Many operations on branches are applied by git checkout and git merge command. So, the git branch is tightly integrated with the **git checkout** and **git merge commands**.

Operations that can be performed on a branch:

Create Branch

You can create a new branch with the help of the **git branch** command. This command will be used as:

Syntax:

$ git branch **<branch** name**>**

Output:

This command will create the **branch B1** locally in Git directory.

List Branch

You can List all of the available branches in your repository by using the following command.

Either we can use **git branch - list** or **git branch** command to list the available branches in the repository.

$ git branch --list

or

$ git branch

Delete Branch

You can delete the specified branch. It is a safe operation. In this command, Git prevents you from deleting the branch if it has unmerged changes. Below is the command to do this.

Syntax: $ git branch -d**<branch** name**>**

Delete a Remote Branch

You can delete a remote branch from Git desktop application. Below command is used to delete a remote branch:

Syntax: $ git push origin -delete **<branch** name**>**

Switch Branch

Git allows you to switch between the branches without making a commit. You can switch between two branches with the **git checkout** command. To switch between the branches, below command is used:

$ git checkout**<branch** name**>**

Switch from master Branch

You can switch from master to any other branch available on your repository without making any commit.

Syntax: $ git checkout **<branch** name**>**

Rename Branch

We can rename the branch with the help of the **git branch** command. To rename a branch, use the below command:

Syntax:

$ git branch -m **<old** branch name**>** **<new** branch name**>**

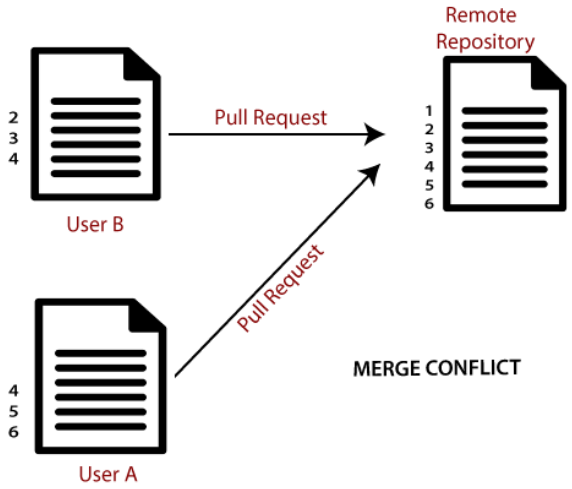
Merge Branch

Git allows you to merge the other branch with the currently active branch. You can merge two branches with the help of **git merge** command. Below command is used to merge the branches:

Syntax: $ git merge **<branch** name**>**

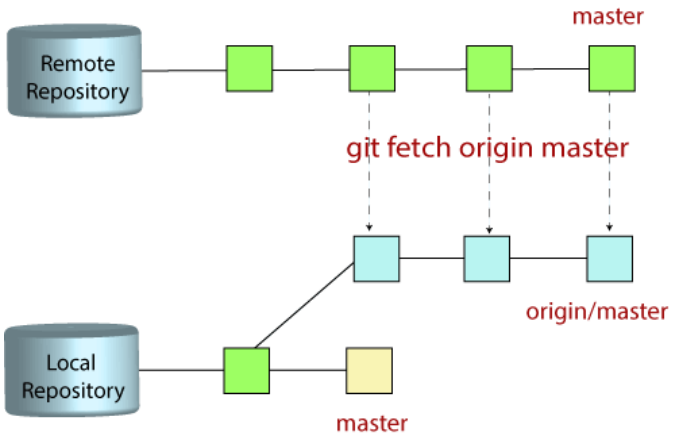
Git Merge Conflict

When two branches are trying to merge, and both are edited at the same time and in the same file, Git won't be able to identify which version is to take for changes. Such a situation is called merge conflict. If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually.



Git Fetch

Git "fetch" Downloads commits, objects and refs from another repository. It fetches branches and tags from one or more repositories. It holds repositories along with the objects that are necessary to complete their histories to keep updated remote-tracking branches.



How to fetch Git Repository

We can use fetch command with many arguments for a particular data fetch. See the below scenarios to understand the uses of fetch command.

Scenario 1: To fetch the remote repository:

We can fetch the complete repository with the help of fetch command from a repository URL like a pull command does. See the below output:

Differences between git fetch and git pull

To understand the differences between fetch and pull, let's know the similarities between both of these commands. Both commands are used to download the data from a remote repository. But both of these commands work differently. Like when you do a git pull, it gets all the changes from the remote or central repository and makes it available to your corresponding branch in your local repository. When you do a git fetch, it fetches all the changes from the remote repository and stores it in a separate branch in your local repository. You can reflect those changes in your corresponding branches by merging.

So basically,

git pull = git fetch + git merge

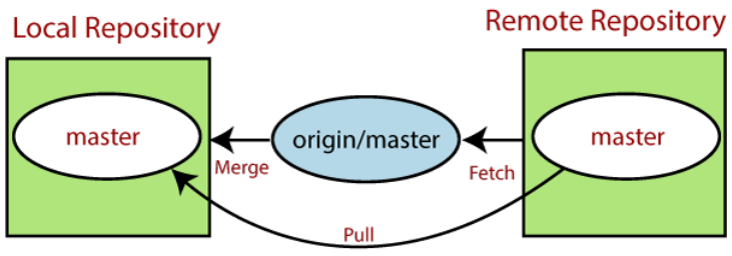
Git Fetch vs. Pull

Some of the key differences between both of these commands are as follows:

git fetch	git pull
Fetch downloads only new data from a remote repository.	Pull is used to update your current HEAD branch with the latest changes from the remote server.
Fetch is used to get a new view of all the things that happened in a remote repository.	Pull downloads new data and directly integrates it into your current working copy files.
Fetch never manipulates or spoils data.	Pull downloads the data and integrates it with the current working file.
It protects your code from merge conflict.	In git pull, there are more chances to create the **merge conflict**.
It is better to use git fetch command with git merge command on a pulled repository.	It is not an excellent choice to use git pull if you already pulled any repository.

Git Pull / Pull Request

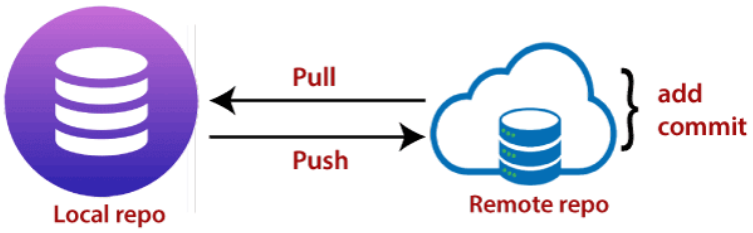
The term pull is used to receive data from GitHub. It fetches and merges changes from the remote server to your working directory. The **git pull command** is used to pull a repository.



$ git pull **<options><remote>/<branchname>**
$ git pull origin master

Git Push

The push term refers to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repository. Pushing is capable of overwriting changes; caution should be taken when pushing.

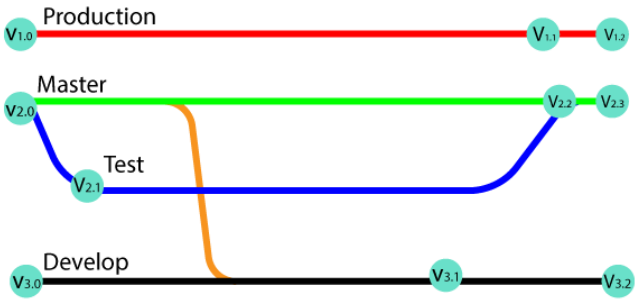



```
$ git push origin master
```

Git Tags

Tags make a point as a specific point in Git history. Tags are used to mark a commit stage as relevant. We can tag a commit for future reference. Primarily, it is used to mark a project's initial point like v1.1.

Tags are much like branches, and they do not change once initiated. We can have any number of tags on a branch or different branches. The below figure demonstrates the tags on various branches.



There are two types of tags.

- Annotated tag
- Light-weighted tag

Both of these tags are similar, but they are different in case of the amount of Metadata stores.

Git Create tag

To create a tag first, checkout to the branch where you want to create a tag. To check out the branch, run the below command:

```
$ git checkout <Branch name>
$ git tag <tag name>
```

Git List Tag

We can list the available tags in our repository. There are three options that are available to list the tags in the repository. They are as follows:

- git tag
- git show
- git tag -l ".*"

Git Push Tag

We can push tags to a remote server project. It will help other team members to know where to pick an update. It will show as **release point** on a remote server account. The git push command facilitates with some specific options to push tags. They are as follows:

- Git push origin <tagname>
- Git push origin -tags/ Git push --tags

The git push origin :

We can push any particular tag by using the git push command. It is used as follows:

Syntax:

```
$ git push origin <tagname>
```

Git Delete Tag

Git allows deleting a tag from the repository at any moment. To delete a tag, run the below command:

Syntax:

```
$git tag --d <tagname>
```

Or

```
$ git tag --delete <tagname>
```

Delete a Remote Tag:

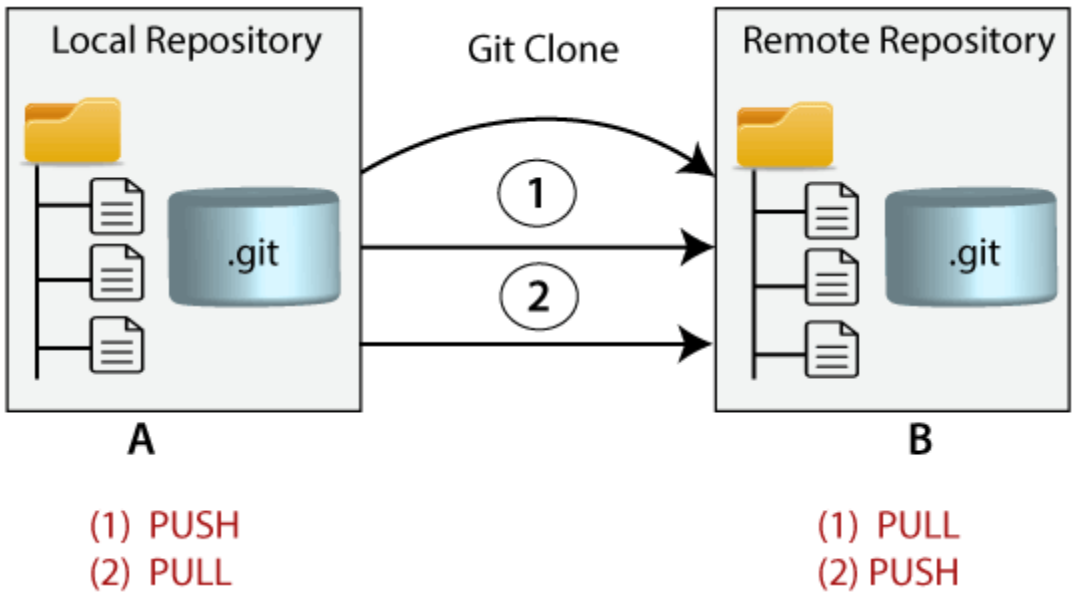
We can also delete a tag from the remote server. To delete a tag from the remote server, run the below command:

Syntax:

```
$ git push origin -d <tagname>
```

Git Clone

In Git, cloning is the act of making a copy of any target repository. The target repository can be remote or local. You can clone your repository from the remote repository to create a local copy on your system. Also, you can sync between the two locations.



Git Clone Repository

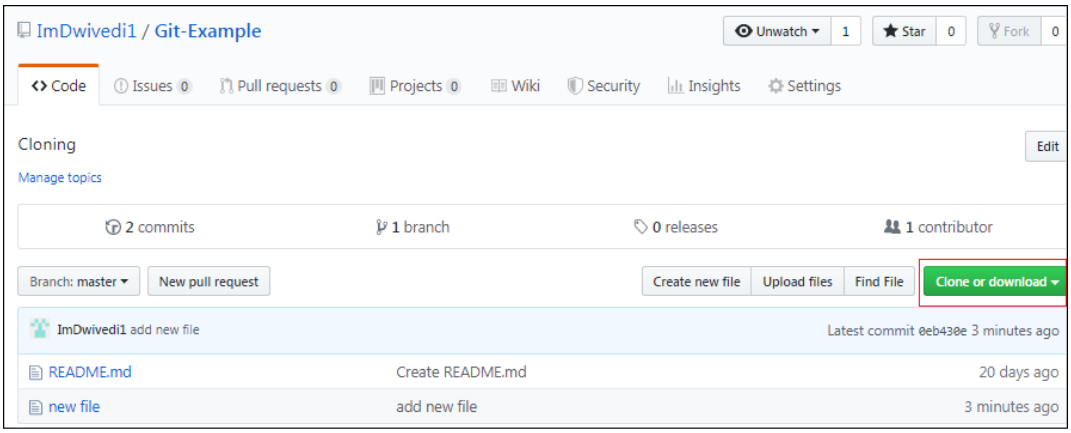
Suppose, you want to clone a repository from GitHub, or have an existing repository owned by any other user you would like to contribute. Steps to clone a repository are as follows:

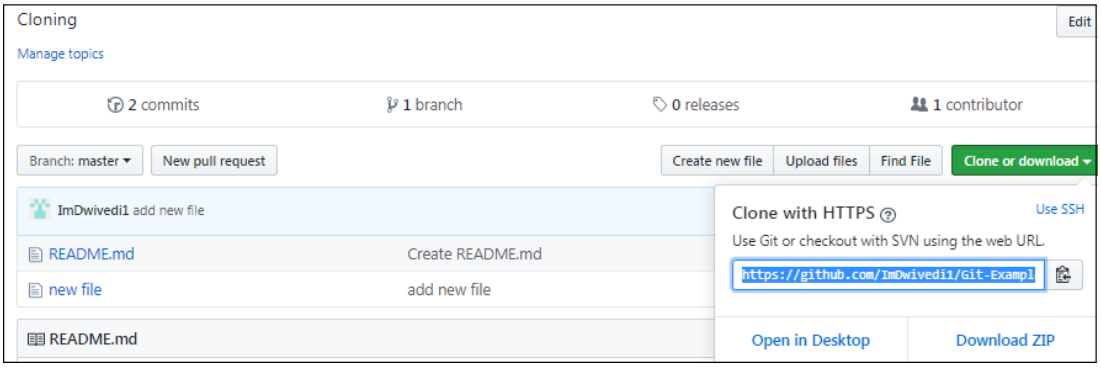
Step 1:

Open GitHub and navigate to the main page of the repository.

Step 2:

Under the repository name, click on **Clone or download**.





Cloning a Repository into a Specific Local Folder

Git allows cloning the repository into a specific directory without switching to that particular directory. You can specify that directory as the next command-line argument in git clone command. See the below command:

```
$ git clone https://github.com/ImDwivedi1/Git-Example.git "new folder(2)"
```

Git Clone Branch

Git allows making a copy of only a particular branch from a repository. You can make a directory for the individual branch by using the git clone command. To make a clone branch, you need to specify the branch name with -b command. Below is the syntax of the command to clone the specific git branch:

Syntax:

$ git clone -b **<Branch** name**><Repository** URL**>**

$ git clone -b master https://github.com/ImDwivedi1/Git-Example.git "new folder(2)"

Git Fork

A fork is a rough copy of a repository. Forking a repository allows you to freely test and debug with changes without affecting the original project. One of the excessive use of forking is to propose changes for bug fixing.

How to Fork a Repository?

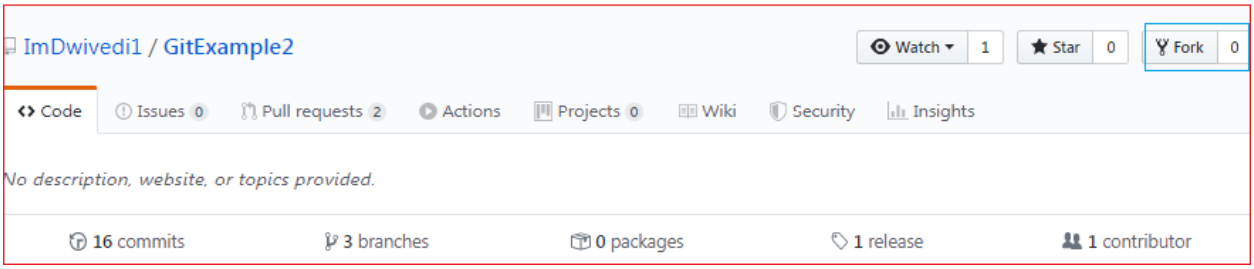
The forking and branching are excellent ways to contribute to an open-source project. These two features of Git allows the enhanced collaboration on the projects.

Forking is a safe way to contribute. It allows us to make a rough copy of the project. We can freely experiment on the project. After the final version of the project, we can create a pull request for merging.

It is a straight-forward process. Steps for forking the repository are as follows:

- Login to the GitHub account.
- Find the GitHub repository which you want to fork.
- Click the Fork button on the upper right side of the repository's page.

We can't fork our own repository. Only shared repositories can be fork. If someone wants to fork the repository, then he must log in with his account. Let's understand the below scenario in which a user pune2016 wants to contribute to our project **GitExample2**. When he searches or put the address of our repository, our repository will look like as follows:



Fork vs. Clone

Sometimes people considered the fork as clone command because of their property. Both commands are used to create another copy of the repository. But the significant difference is that the fork is used to create a server-side copy, and clone is used to create a local copy of the repository.

There is no particular command for forking the repository; instead, it is a service provided by third-party Git service like GitHub. Comparatively, git clone is a command-line utility that is used to create a local copy of the project.

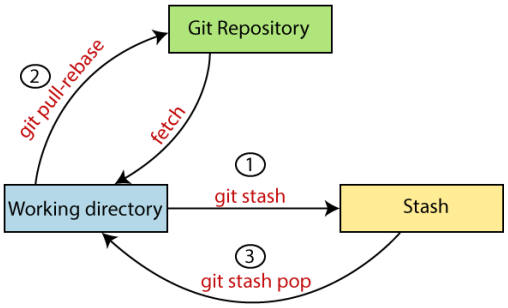
Generally, people working on the same project clone the repository and the external contributors fork the repository.

A pull request can merge the changes made on the fork repository. We can create a pull request to propose changes to the project. Comparatively, changes made on the cloned repository can be merged by pushing. We can push the changes to our remote repository.

Git Stash

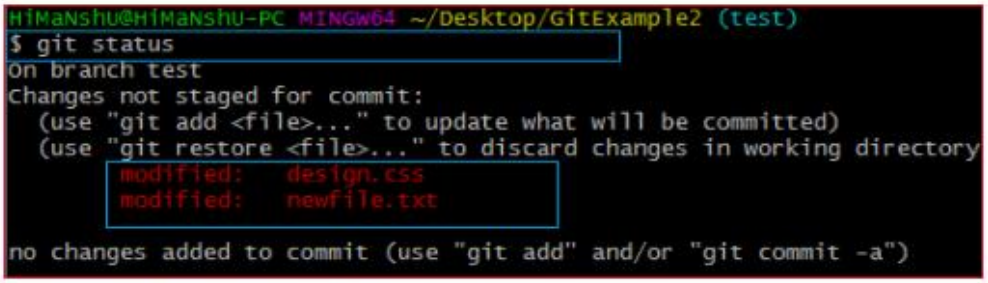
Sometimes you want to switch the branches, but you are working on an incomplete part of your current project. You don't want to make a commit of half-done work. Git stashing allows you to do so. The **git stash command** enables you to switch branches without committing the current branch.

The below figure demonstrates the properties and role of stashing concerning repository and working directory.



```
$ git status
```

Output:



From the above output, you can see the status that there are two untracked file **design.css** and **ne** use the git stash command. The git stash command is used as:

Syntax:

```
$ git stash
```

Output:

Git Stash Save (Saving Stashes with the message):

In Git, the changes can be stashed with a message. To stash a change with a message, run the below command:

Syntax:

```
$ git stash save "<Stashing Message>"
```

Git Stash List (Check the Stored Stashes)

To check the stored stashes, run the below command:

Syntax:

```
$ git stash list
```

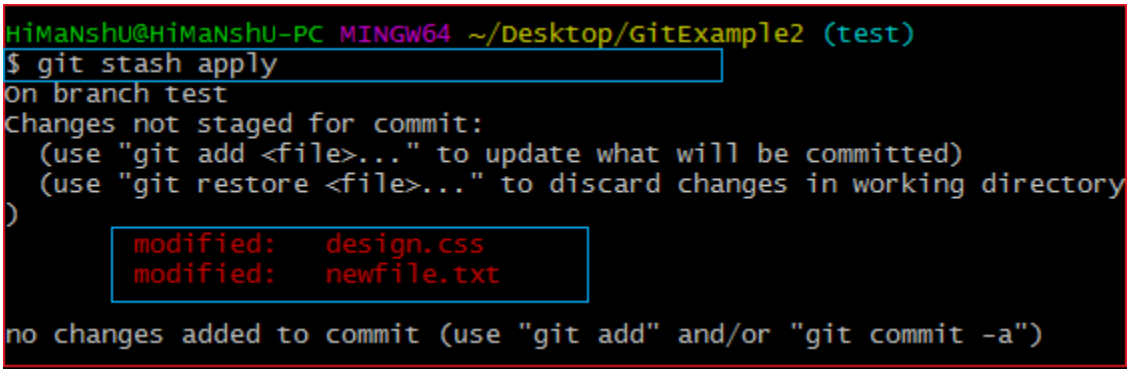
Git Stash Apply

You can re-apply the changes that you just stashed by using the git stash command. To apply the commit, use the git stash command, followed by the apply option. It is used as:

Syntax:

```
$ git stash apply
```

Output:



The above output restores the last stash. Now, if you will check the status of the repository, it will show the changes that are made on the file. Consider the below

Git Stash Pop (Reapplying Stashed Changes)

Git allows the user to re-apply the previous commits by using git stash pop command. The popping option removes the changes from stash and applies them to your working file.

The git stash pop command is quite similar to git stash apply. The main difference between both of these commands is stash pop command that deletes the stash from the stack after it is applied.

Syntax:

```
$ git stash pop
```

Git Stash Drop (Unstash)

The **git stash drop** command is used to delete a stash from the queue. Generally, it deletes the most recent stash. Caution should be taken before using stash drop command, as it is difficult to undo if once applied.

The only way to revert it is if you do not close the terminal after deleting the stash. The stash drop command will be used as:

Syntax:

```
$ git stash drop
```

Git Stash Clear

The **git stash clear** command allows deleting all the available stashes at once. To delete all the available stashes, operate below command:

Syntax:

```
$ git stash clear
```

Git Ignore

In Git, the term "ignore" is used to specify intentionally untracked files that Git should ignore. It doesn't affect the Files that already tracked by Git.

Sometimes you don't want to send the files to Git service like GitHub. We can specify files in Git to ignore.

The file system of Git is classified into three categories:

Tracked files are such files that are previously staged or committed.

Untracked:

Untracked files are such files that are not previously staged or committed.

Ignored:

Ignored files are such files that are explicitly ignored by git. We have to tell git to ignore such files.

How to Ignore Files Manually

There is no command in Git to ignore files; alternatively, there are several ways to specify the ignore files in git. One of the most common ways is the **.gitignore** file. Let's understand it with an example.

The .gitignore file:

Rules for ignoring file is defined in the .gitignore file. The .gitignore file is a file that contains all the formats and files of the ignored file. We can create multiple ignore files in a different directory. Let's understand how it works with an example:

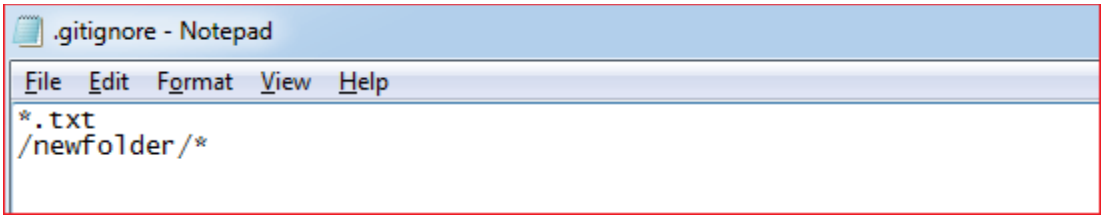
Step1: Create a file named .gitignore if you do not have it already in your directory. To create a file, use the command touch or cat. It will use as follows:

```
$ touch .gitignore
```

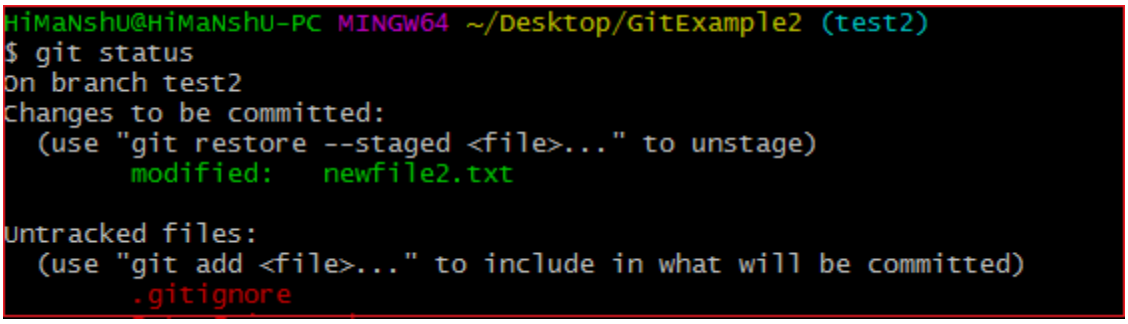
Or

$ cat .gitignore

Step2: Now, add the files and directories to the **.gitignore** file that you want to ignore. To add the files and directory to the .git ignore the file, open the file and type the file name, directory name, and pattern to ignore files and directories. Consider the below image:



Step3: Now, to share it on Git, we have to commit it. The .gitignore file is still now in staging area; we can track it by git status command. Consider the below output:

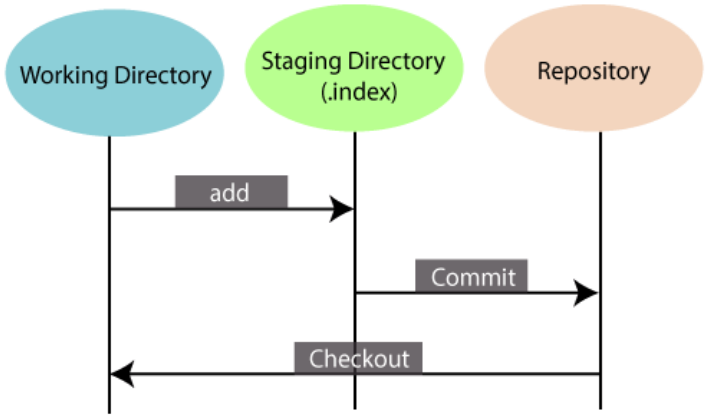


Now to stage it, we have to commit it. To commit it, run the below command:

1. $ git add .gitignore
2. $ git commit -m "ignored directory created."

Git Index

The Git index is a staging area between the working directory and repository. It is used to build up a set of changes that you want to commit together. To better understand the Git index, then first understand the working directory and repository.



Working directory:

When you worked on your project and made some changes, you are dealing with your project's working directory. This project directory is available on your computer's filesystem. All the changes you make will remain in the working directory until you add them to the staging area.

Staging area:

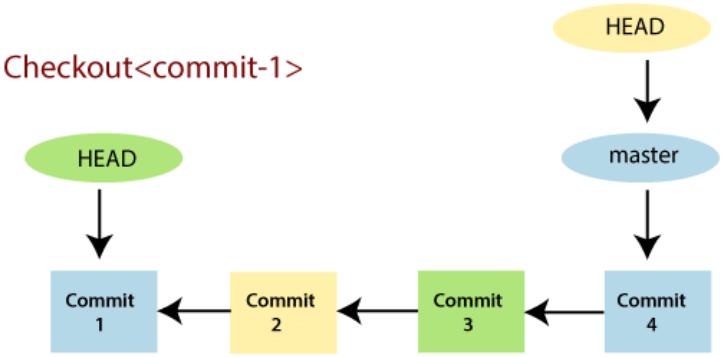
The staging area can be described as a preview of your next commit. When you create a git commit, Git takes changes that are in the staging area and make them as a new commit. You are allowed to add and remove changes from the staging area. The staging area can be considered as a real area where git stores the changes.

Repository:

In Git, Repository is like a data structure used by GIt to store metadata for a set of files and directories. It contains the collection of the files as well as the history of changes made to those files. Repositories in Git is considered as your project folder. A repository has all the project-related data. Distinct projects have distinct repositories.

Git Head

The **HEAD** points out the last commit in the current checkout branch. It is like a pointer to any reference. The HEAD can be understood as the "**current branch**." When you switch branches with 'checkout,' the HEAD is transferred to the new branch.



The above fig shows the HEAD referencing commit-1 because of a 'checkout' was done at commit-1. When you make a new commit, it shifts to the newer commit. The git head command is used to view the status of Head with different arguments. It stores the status of Head in **.git\refs\heads** directory. Let's see the below example:

Git Show Head

The **git show head** is used to check the status of the Head. This command will show the location of the Head.

1. $ git show HEAD

Git Revert

In Git, the term revert is used to revert some changes. The git revert command is used to apply revert operation. It is an undo type command. However, it is not a traditional undo alternative. It does not delete any data in this process; instead, it will create a new change with the opposite effect and thereby undo the specified commit. Generally, git revert is a commit.

It can be useful for tracking bugs in the project. If you want to remove something from history then git revert is a wrong choice.

Moreover, we can say that git revert records some new changes that are just opposite to previously made commits. To undo the changes, run the below command:

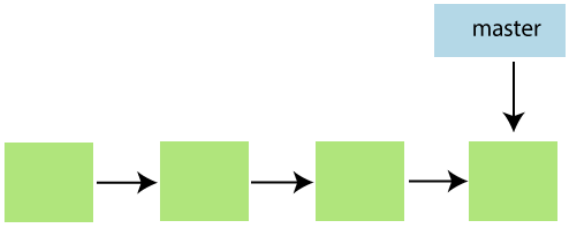
```
$ git revert commit-id
```

Git Origin Master

The term "git origin master" is used in the context of a remote repository. It is used to deal with the remote repository. The term origin comes from where repository original situated and master stands for the main branch. Let's understand both of these terms in detail.

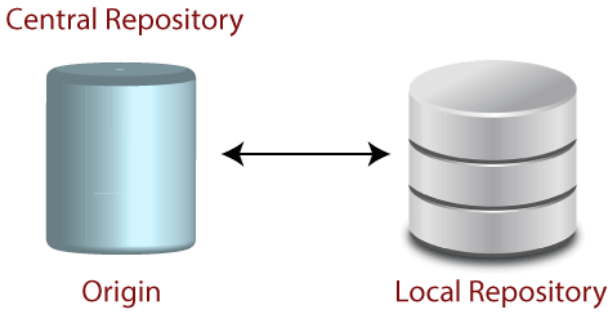
Git Master

Master is a naming convention for Git branch. It's a default branch of Git. After cloning a project from a remote server, the resulting local repository contains only a single local branch. This branch is called a "master" branch. It means that "master" is a repository's "default" branch.



Git Origin

In Git, The term origin is referred to the remote repository where you want to publish your commits. The default remote repository is called **origin**, although you can work with several remotes having a different name at the same time. It is said as an alias of the system.



The origin is a short name for the remote repository that a project was initially being cloned. It is used in place of the original repository URL. Thus, it makes referencing much easier.

Origin is just a standard convention. Although it is significant to leave this convention untouched, you could ideally rename it without losing any functionality.

In the following example, the URL parameter acts as an origin to the "clone" command for the cloned local repository:

```
$ git clone https://github.com/ImDwivedi1/Git-Example
```

Some commands in which the term origin and master are widely used are as follows:

- Git push origin master
- Git pull origin master

Git has two types of branches called local and remote. To use git pull and git push, you have to tell your local branch that on which branch is going to operate. So, the term origin master is used to deal with a remote repository and master branch. The term **push origin master** is used to push the changes to the remote repository. The term **pull origin master** is used to access the repository from remote to local.