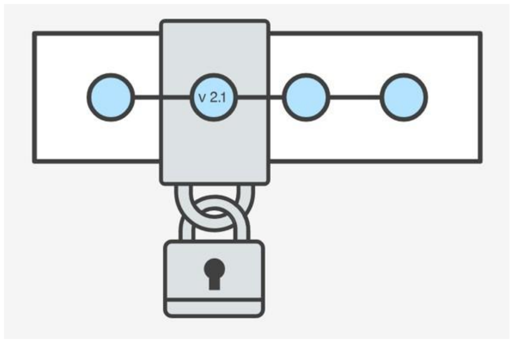
Version Control System

A version control system is software that tracks changes to a file or set of files over time so that you can recall specific versions later. It also allows you to work together with other programmers.

The version control system is a collection of software tools that help a team to manage changes in a source code. It uses a special kind of database to keep track of every modification to the code.

Developers can compare earlier versions of the code with an older version to fix the mistakes.



Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Purpose of the Version Control System

Following are the purpose of Version Control System (VCS)

1) Multiple people can work simultaneously on a single project. Everyone works on and edits their own copy of the files and it is up to them when they wish to share the changes made by them with the rest of the team.

2) It also enables one person to use multiple computers to work on a project, so it is valuable even if you are working by yourself.

3) It integrates the work that is done simultaneously by different members of the team. In some rare cases, when conflicting edits are made by two people to the same line of a file, then human assistance is requested by the version control system in deciding what should be done.

4) Version control provides access to the historical versions of a project. This is insurance against computer crashes or data loss. If any mistake is made, you can easily roll back to a previous version. It is also possible to undo specific edits that too without losing the work done in the meanwhile. It can be easily known when, why, and by whom any part of a file was edited.

Use of Version Control System:

1) A repository: It can be thought of as a database of changes. It contains all the edits and historical versions (snapshots) of the project.

2) Copy of Work (sometimes called as checkout): It is the personal copy of all the files in a project. You can edit to this copy, without affecting the work of others and you can finally commit your changes to a repository when you are done making your changes.

Benefits of the Version Control System

The Version Control System is very helpful and beneficial in software development; developing software without using version control is unsafe. It provides backups for uncertainty. Version control systems offer a speedy interface to developers. It also allows software teams to preserve efficiency and agility according to the team scales to include more developers.

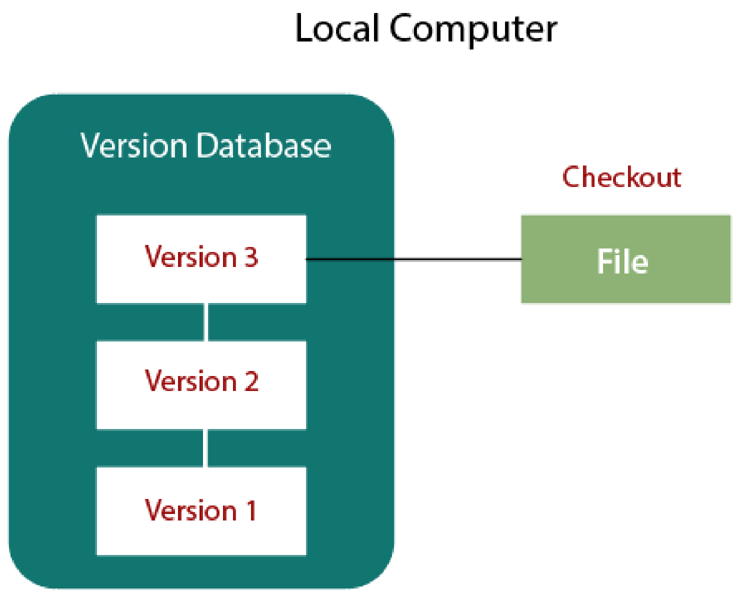
Some key benefits of having a version control system are as follows.

- Complete change history of the file
- Simultaneously working
- Branching and merging
- Traceability

Types of Version Control System

- Localized version Control System
- Centralized version control systems
- Distributed version control systems

Localized Version Control Systems



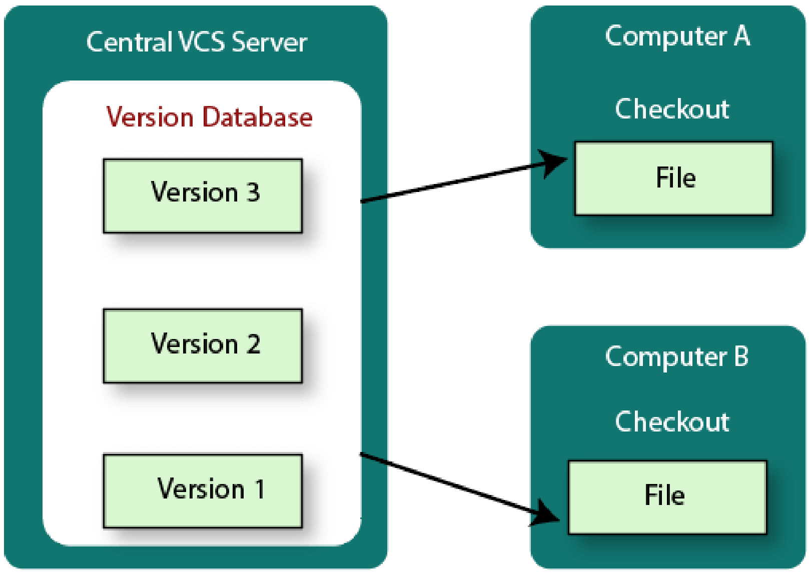
The localized version control method is a common approach because of its simplicity. But this approach leads to a higher chance of error. In this approach, you may forget which directory you're in and accidentally write to the wrong file or copy over files you don't want to.

To deal with this issue, programmers developed local VCSs that had a simple database. Such databases kept all the changes to files under revision control. A local version control system keeps local copies of the files.

The major drawback of Local VCS is that it has a single point of failure.

Centralized Version Control System

The developers needed to collaborate with other developers on other systems. The localized version control system failed in this case. To deal with this problem, Centralized Version Control Systems were developed.



These systems have a single server that contains the versioned files, and some clients to check out files from a central place.

Centralized version control systems have many benefits, especially over local VCSs.

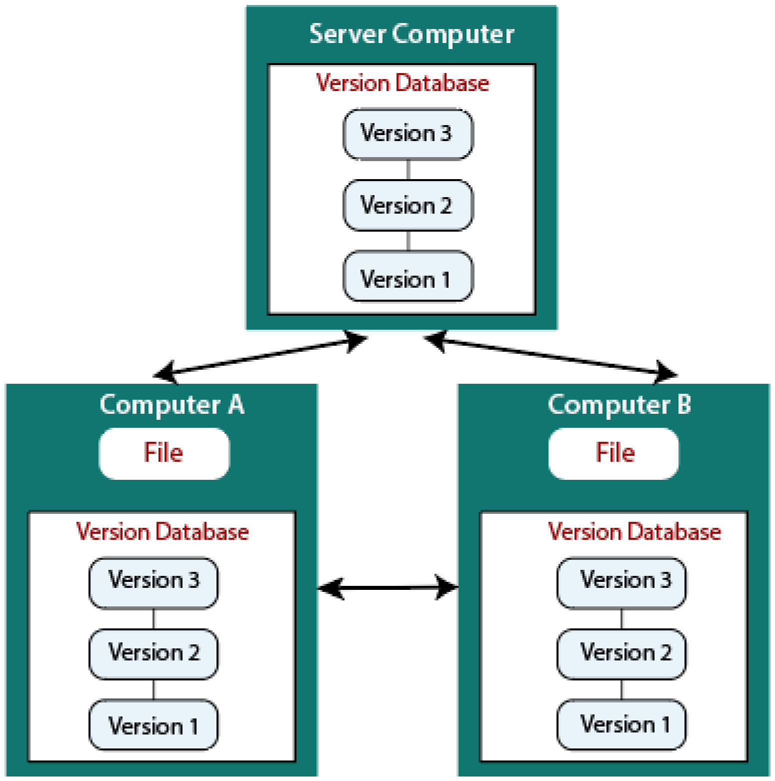
- Everyone on the system has information about the work what others are doing on the project.
- Administrators have control over other developers.
- It is easier to deal with a centralized version control system than a localized version control system.
- A local version control system facilitates with a server software component which stores and manages the different versions of the files.

It also has the same drawback as in local version control system that it also has a single point of failure.

Distributed Version Control System

Centralized Version Control System uses a central server to store all the database and team collaboration. But due to single point failure, which means the failure of the central server, developers do not prefer it. Next, the Distributed Version Control System is developed.

In a Distributed Version Control System (such as Git, Mercurial, Bazaar or Darcs), the user has a local copy of a repository. So, the clients don't just check out the latest snapshot of the files even they can fully mirror the repository. The local repository contains all the files and metadata present in the main repository.



DVCS allows automatic management branching and merging. It speeds up of most operations except pushing and pulling. DVCS enhances the ability to work offline and does not rely on a single location for backups. If any server stops and other systems were collaborating via it, then any of the client repositories could be restored by that server. Every checkout is a full backup of all the data.

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

Difference between Centralized Version Control System and Distributed Version Control System

Centralized Version Control Systems are systems that use client/server architecture. In a centralized Version Control System, one or more client systems are directly connected to a central server. Contrarily the Distributed Version Control Systems are systems that use peer-to-peer architecture.

There are many benefits and drawbacks of using both the version control systems. Let's have a look at some significant differences between Centralized and Distributed version control system.

Centralized Version Control System	**Distributed Version Control System**
In CVCS, The repository is placed at one place and delivers information to many clients.	In DVCS, Every user has a local copy of the repository in place of the central repository on the server-side.
It is based on the client-server approach.	It is based on the client-server approach.
It is the most straightforward system based on the concept of the central repository.	It is flexible and has emerged with the concept that everyone has their repository.
In CVCS, the server provides the latest code to all the clients across the globe.	In DVCS, every user can check out the snapshot of the code, and they can fully mirror the central repository.
CVCS is easy to administrate and has additional control over users and access by its server from one place.	DVCS is fast comparing to CVCS as you don't have to interact with the central server for every command.
The popular tools of CVCS are SVN (Subversion) and CVS.	The popular tools of DVCS are Git and Mercurial.
CVCS is easy to understand for beginners.	DVCS has some complex process for beginners.
If the server fails, No system can access data from another system.	if any server fails and other systems were collaborating via it, that server can restore any of the client repositories

What is 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.

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**.

Features of Git

Scalability
Security
Open Source
Features of Git
Speed

- **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.

- **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.

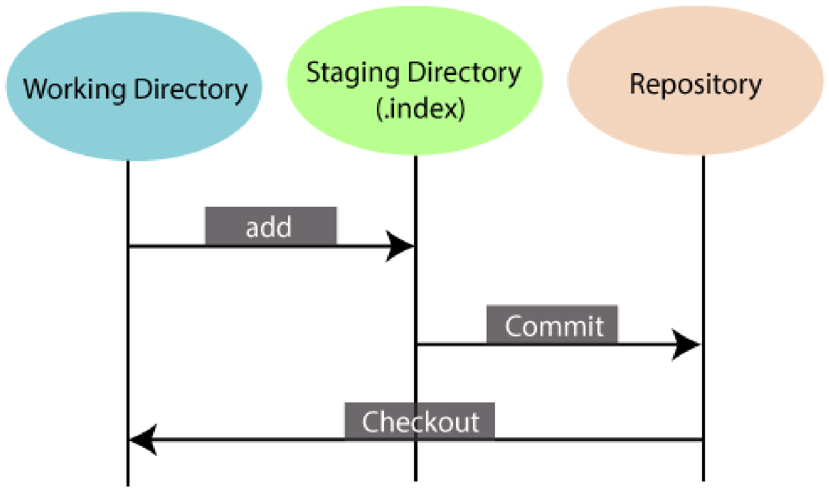
  - We can create a **demo branch** for the experiment and check if it is working. We can also remove it if needed.

  - 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.

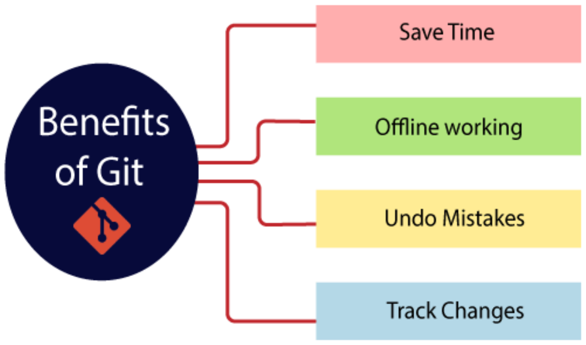
Some **significant benefits** of using Git are as follows:

- **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.

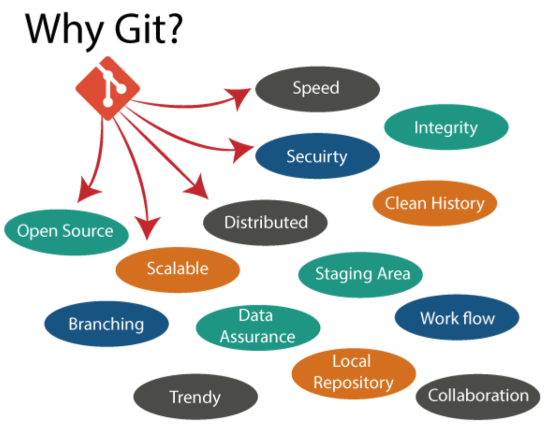
- **Undo Mistakes**
  One additional benefit of Git is we can **Undo** mistakes. Sometimes the undo can be a savior option for us. Git provides the undo option for almost everything.

- **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 should we 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.

Using Git with Command Line

To start using Git, we are first going to open up our Command shell.

For Windows, you can use Git bash, which comes included in Git for Windows. For Mac and Linux you can use the built-in terminal.

The first thing we need to do, is to check if Git is properly installed:

Example

```
git --version

git version 2.30.2.windows.1
```

If Git is installed, it should show something like git version X.Y

Configure Git

Now let Git know who you are. This is important for version control systems, as each Git commit uses this information:

Example

```
git config --global user.name "Harry"

git config --global user.email "harry@gmail.com"
```

Note: Use global to set the username and e-mail for **every repository** on your computer.

If you want to set the username/e-mail for just the current repo, you can remove global

Creating Git Folder

Now, let's create a new folder for a project:

Example

```
mkdir myproject
cd myproject
```

mkdir **make**s a **new directory**.

cd **changes** the **current working directory**.

Now that we are in the correct directory. We can start by initializing Git!

Note: If you already have a folder/directory you would like to use for Git:

Navigate to it in command line, or open it in your file explorer, right-click and select "Git Bash here"

Initialize Git

Once you have navigated to the correct folder, you can initialize Git on that folder:

Example

```
git init

Initialized empty Git repository in /Users/user/myproject/.git/
```

You just created your first Git Repository!

Git Adding New Files

We just created our first local Git repo. But it is empty.

Lets add a file named index.html in current folder.

Go back to the terminal and list the files in our current working directory:

```
ls
index.html
```

`ls` will **list** the files in the directory. We can see that `index.html` is there.

Then we check the Git `status` and see if it is a part of our repo:

Example

```
git status
```

Output:

```
On branch master

No commits yet

Untracked files:

  (use "git add ..." to include in what will be committed)

    index.html

nothing added to commit but untracked files present (use "git add" to track)
```

Now Git is aware of the file, but has not added it to our repository!

Files in your Git repository folder can be in one of 2 states:

- Tracked - files that Git knows about and are added to the repository
- Untracked - files that are in your working directory, but not added to the repository

When you first add files to an empty repository, they are all untracked. To get Git to track them, you need to stage them, or add them to the staging environment.

Git Staging Environment

One of the core functions of Git is the concepts of the Staging Environment, and the Commit.

As you are working, you may be adding, editing and removing files. But whenever you hit a milestone or finish a part of the work, you should add the files to a Staging Environment.

Staged files are files that are ready to be **committed** to the repository you are working on. You will learn more about `commit` shortly.

For now, we are done working with `index.html`. So we can add it to the Staging Environment:

```
git add index.html
```

The file should be **Staged**. Let's check the status:

```
git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached ..." to unstage)     new file: index.html
```

Now the file has been added to the Staging Environment.

Adding all files to staging environment

Now add all files in the current directory to the Staging Environment:

```
git add –all

or
```

Using `--all` instead of individual filenames will `stage` all changes (new, modified, and deleted) files.

```
git add .

git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached ..." to unstage) new file: README.md new file: bluestyle.css new
file: index.html
```

Git Commit

Since we have finished our work, we are ready move from `stage` to `commit` for our repo.

Adding commits keep track of our progress and changes as we work. Git considers each `commit` change point or "save point". It is a point in the project you can go back to if you find a bug, or want to make a change.

When we `commit`, we should **always** include a **message**.

By adding clear messages to each `commit`, it is easy for yourself (and others) to see what has changed and when.

```
git commit -m "First release of Hello World!"
[master (root-commit) 221ec6e] First release of Hello World!
 3 files changed, 26 insertions(+)
 create mode 100644 README.md
 create mode 100644 bluestyle.css
 create mode 100644 index.html
```

The `commit` command performs a commit, and the `-m "message"` adds a message.

The Staging Environment has been committed to our repo, with the message:
"First release of Hello World!"

Git Commit without Stage

Sometimes, when you make small changes, using the staging environment seems like a waste of time. It is possible to commit changes directly, skipping the staging environment. The `-a` option will automatically stage every changed, already tracked file.

Let's add a small update to index.html and then try to commit:

It will give warning message:

Skipping the Staging Environment is not generally recommended.

Skipping the stage step can sometimes make you include unwanted changes.

Short Message in status:

Example

```
git status --short

 M index.html
```

Note: Short status flags are:

- ?? - Untracked files

- A - Files added to stage
- M - Modified files
- D - Deleted files

We see the file we expected is modified. So let's commit it directly:

Example

```
git commit -a -m "Updated index.html with a new line"
[master 09f4acd] Updated index.html with a new line
 1 file changed, 1 insertion(+)
```

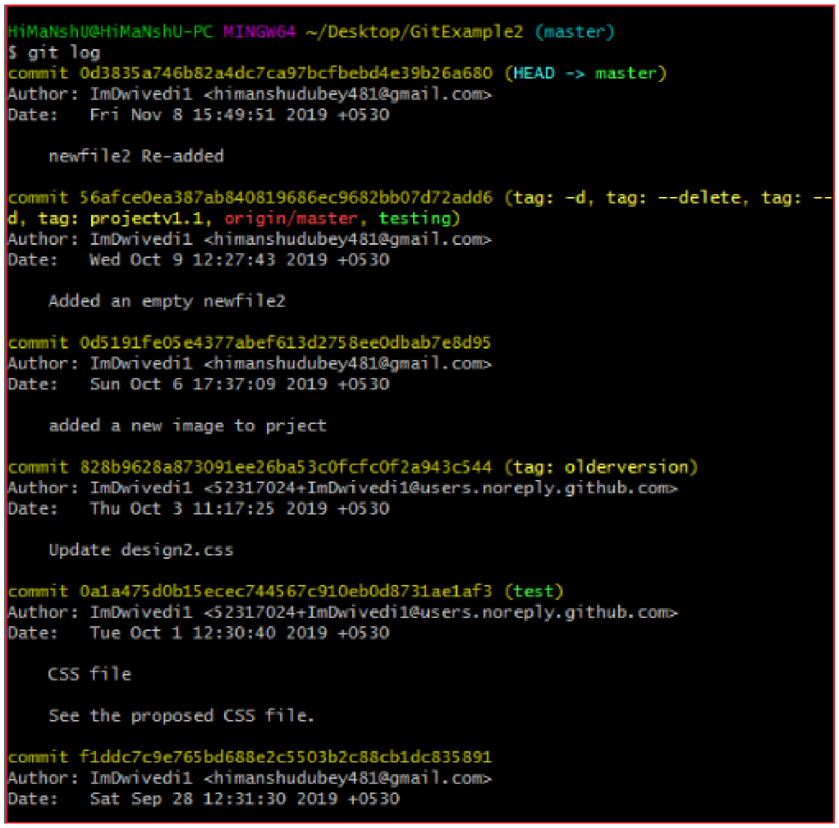
Git Commit Log

To view the history of commits for a repository, you can use the `log` command:

Example

```
git log
```

The above command will display the last commits.



```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: --
d, tag: projectv1.1, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530

    added a new image to prject

commit 828b9628a873091ee26ba53c0fcfc0f2a943c544 (tag: olderversion)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Thu Oct 3 11:17:25 2019 +0530

    Update design2.css

commit 0a1a475d0b15ecec744567c910eb0d8731ae1af3 (test)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Tue Oct 1 12:30:40 2019 +0530

    CSS file

    See the proposed CSS file.

commit f1ddc7c9e765bd688e2c5503b2c88cb1dc835891
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sat Sep 28 12:31:30 2019 +0530
```

The above command is listing all the recent commits. Each commit contains some unique sha-id, which is generated by the SHA algorithm. It also includes the date, time, author, and some additional details.

Git Log Oneline

The oneline option is used to display the output as one commit per line. It also shows the output in brief like the first seven characters of the commit SHA and the commit message.

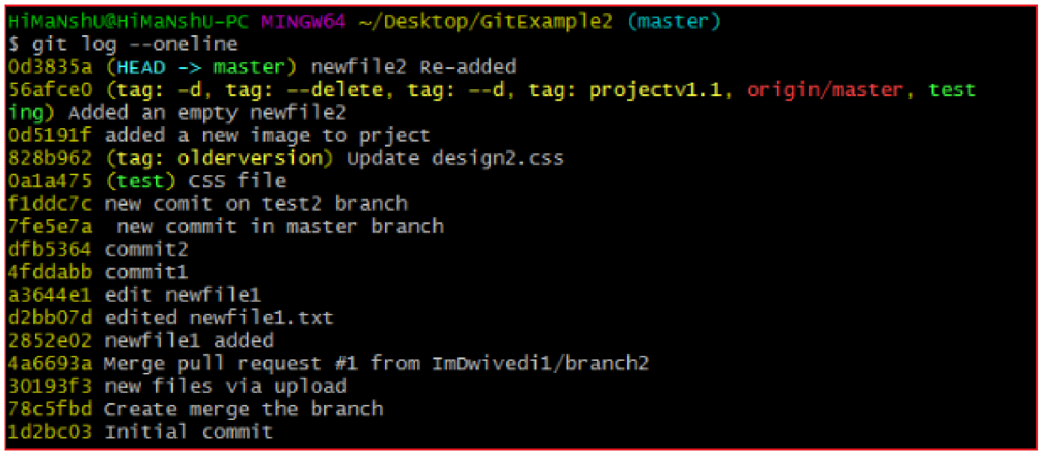
It will be used as follows:

$ git log --oneline

So, usually we can say that the --oneline flag causes git log to display:

- one commit per line
- the first seven characters of the SHA
- the commit message

Consider the below output:



```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log --oneline
0d3835a (HEAD -> master) newfile2 Re-added
56afce0 (tag: -d, tag: --delete, tag: --d, tag: projectv1.1, origin/master, test
ing) Added an empty newfile2
0d5191f added a new image to prject
828b962 (tag: olderversion) Update design2.css
0a1a475 (test) CSS file
f1ddc7c new comit on test2 branch
7fe5e7a  new commit in master branch
dfb5364 commit2
4fddabb commit1
a3644e1 edit newfile1
d2bb07d edited newfile1.txt
2852e02 newfile1 added
4a6693a Merge pull request #1 from ImDwivedi1/branch2
30193f3 new files via upload
78c5fbd Create merge the branch
1d2bc03 Initial commit
```

As we can see more precisely from the above output, every commit is given only in one line with a seven-digit sha number and commit message.

Git Log Stat

The log command displays the files that have been modified. It also shows the number of lines and a summary line of the total records that have been updated.

Generally, we can say that the stat option is used to display

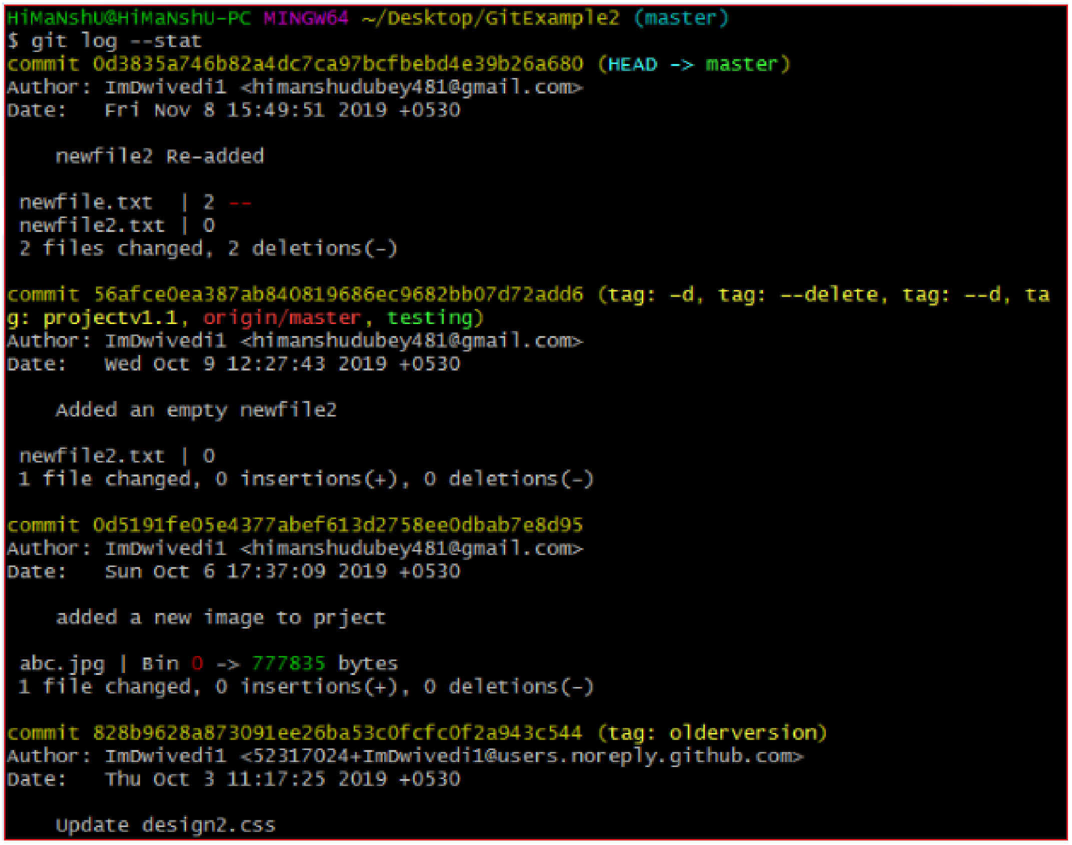
- the modified files,
- The number of lines that have been added or removed

- A summary line of the total number of records changed
- The lines that have been added or removed.

It will be used as follows:

1. $ git log --stat

The above command will display the files that have been modified. Consider the below output:



```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log --stat
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added

 newfile.txt  | 2 --
 newfile2.txt | 0
 2 files changed, 2 deletions(-)

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: --d, ta
g: projectv1.1, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

 newfile2.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530

    added a new image to prject

 abc.jpg | Bin 0 -> 777835 bytes
 1 file changed, 0 insertions(+), 0 deletions(-)

commit 828b9628a873091ee26ba53c0fcfc0f2a943c544 (tag: olderversion)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Thu Oct 3 11:17:25 2019 +0530

    Update design2.css
```

From the above output, we can see that all listed commits are modifications in the repository.

Git log P or Patch

The git log patch command displays the files that have been modified. It also shows the location of the added, removed, and updated lines.

It will be used as:

1. $ git log --patch

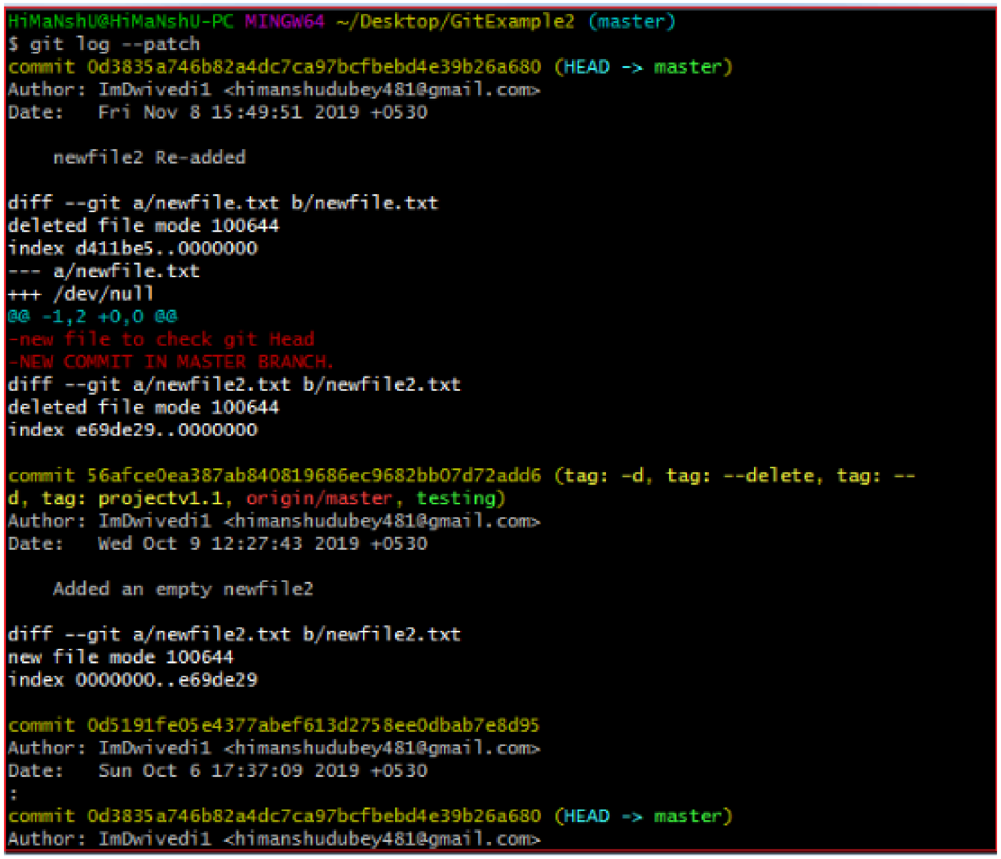
Or

1. $ git log -p

Generally, we can say that the --patch flag is used to display:

- Modified files
- The location of the lines that you added or removed
- Specific changes that have been made.

Consider the below output:



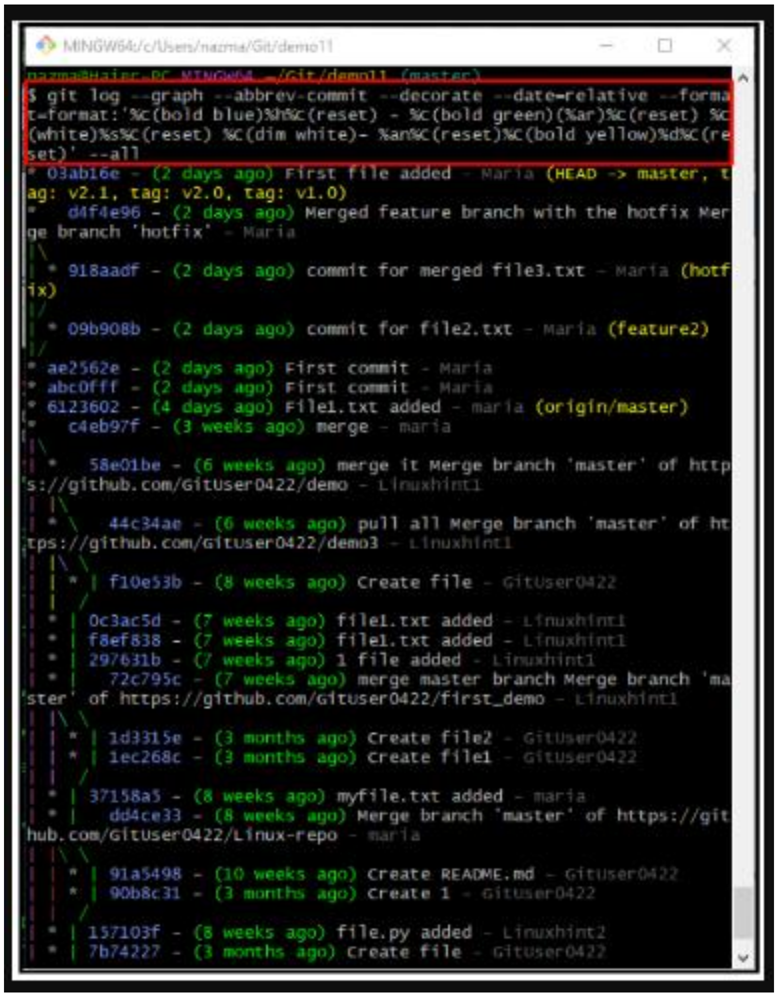
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log --patch
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added

diff --git a/newfile.txt b/newfile.txt
deleted file mode 100644
index d411be5..0000000
--- a/newfile.txt
+++ /dev/null
@@ -1,2 +0,0 @@
-new file to check git Head
-NEW COMMIT IN MASTER BRANCH.
diff --git a/newfile2.txt b/newfile2.txt
deleted file mode 100644
index e69de29..0000000

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: --
d, tag: projectv1.1, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

diff --git a/newfile2.txt b/newfile2.txt
new file mode 100644
index 0000000..e69de29

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530
:
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
```

The above output is displaying the modified files with the location of lines that have been added or removed.

Git Log Graph

Git log command allows viewing your git log as a graph. To list the commits in the form of a graph, run the git log command with --graph option. It will run as follows:

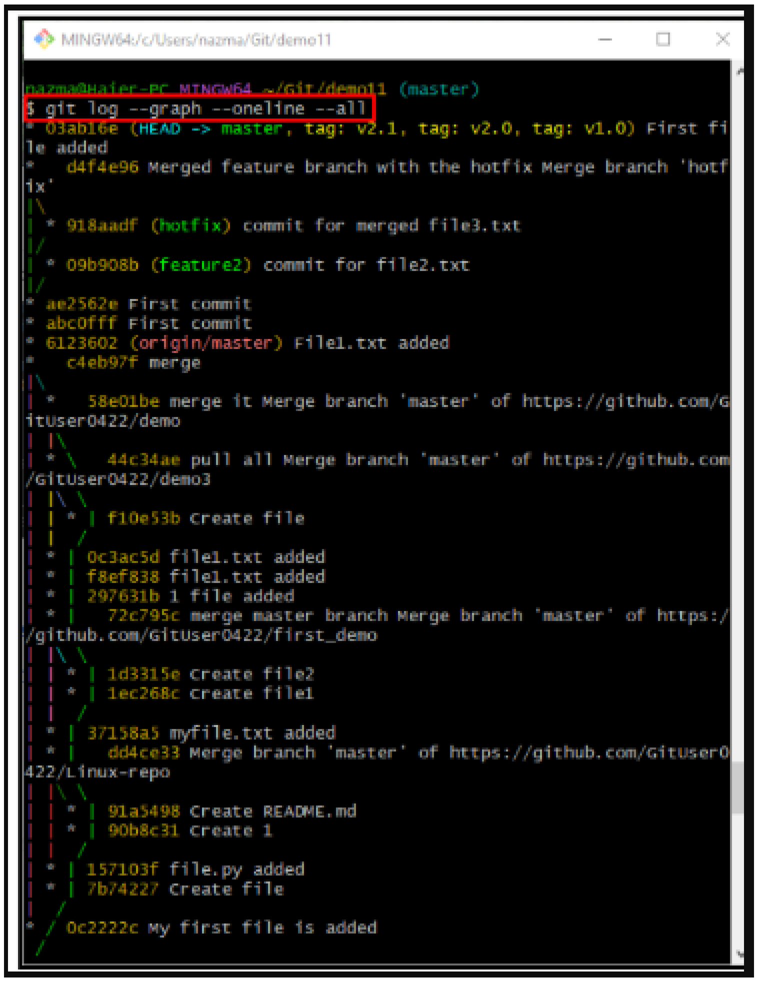
$ git log --graph



```
MINGW64:/c/Users/nazma/Git/demo11
nazma@Haier-PC MINGW64 ~/Git/demo11 (master)
$ git log --graph --abbrev-commit --decorate --date=relative --forma
t=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset) %C
(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(bold yellow)%d%C(re
set)' --all
* 03ab16e - (2 days ago) First file added - Maria (HEAD -> master, t
ag: v2.1, tag: v2.0, tag: v1.0)
*   d4f4e96 - (2 days ago) Merged feature branch with the hotfix Mer
ge branch 'hotfix' - Maria
|\
| * 918aadf - (2 days ago) commit for merged file3.txt - Maria (hotf
ix)
|/
| * 09b908b - (2 days ago) commit for file2.txt - Maria (feature2)
|/
* ae2562e - (2 days ago) First commit - Maria
* abc0fff - (2 days ago) First commit - Maria
* 6123602 - (4 days ago) File1.txt added - maria (origin/master)
*   c4eb97f - (3 weeks ago) merge - maria
|\
| *   58e01be - (6 weeks ago) merge it Merge branch 'master' of http
s://github.com/GitUser0422/demo - Linuxhint1
| |\
| * \   44c34ae - (6 weeks ago) pull all Merge branch 'master' of ht
tps://github.com/Gituser0422/demo3 - Linuxhint1
| |\ \
| | * | f10e53b - (8 weeks ago) Create file - GitUser0422
| | |/
| * | 0c3ac5d - (7 weeks ago) file1.txt added - Linuxhint1
| * | f8ef838 - (7 weeks ago) file1.txt added - Linuxhint1
| * | 297631b - (7 weeks ago) 1 file added - Linuxhint1
| * |   72c795c - (7 weeks ago) merge master branch Merge branch 'ma
ster' of https://github.com/Gituser0422/first_demo - Linuxhint1
| |\ \
| | * | 1d3315e - (3 months ago) Create file2 - GitUser0422
| | * | 1ec268c - (3 months ago) Create file1 - GitUser0422
| | |/
| * | 37158a5 - (8 weeks ago) myfile.txt added - maria
| * |   dd4ce33 - (8 weeks ago) Merge branch 'master' of https://git
hub.com/GitUser0422/Linux-repo - maria
| |\ \
| | * | 91a5498 - (10 weeks ago) Create README.md - GitUser0422
| | * | 90b8c31 - (3 months ago) Create 1 - GitUser0422
| | |/
| * | 157103f - (8 weeks ago) file.py added - Linuxhint2
| * | 7b74227 - (3 months ago) Create file - GitUser0422
```

To make the output more specific, you can combine this command with --oneline option. It will operate as follows:

$ git log --graph --oneline



MINGW64:/c/Users/nazma/Git/demo11

```
nazma@Haier-PC MINGW64 ~/Git/demo11 (master)
$ git log --graph --oneline --all
* 03ab16e (HEAD -> master, tag: v2.1, tag: v2.0, tag: v1.0) First fi
le added
*   d4f4e96 Merged feature branch with the hotfix Merge branch 'hotf
ix'
|\
| * 918aadf (hotfix) commit for merged file3.txt
|/
| * 09b908b (feature2) commit for file2.txt
|/
* ae2562e First commit
* abc0fff First commit
* 6123602 (origin/master) File1.txt added
*   c4eb97f merge
|\
| *   58e01be merge it Merge branch 'master' of https://github.com/G
itUser0422/demo
| |\
| * \   44c34ae pull all Merge branch 'master' of https://github.com
/GitUser0422/demo3
| |\ \
| | * | f10e53b Create file
| |  /
| * | 0c3ac5d file1.txt added
| * | f8ef838 file1.txt added
| * | 297631b 1 file added
| * |   72c795c merge master branch Merge branch 'master' of https:/
/github.com/GitUser0422/first_demo
| |\ \
| | * | 1d3315e Create file2
| | * | 1ec268c create file1
| |  /
| * | 37158a5 myfile.txt added
| * |   dd4ce33 Merge branch 'master' of https://github.com/GitUser0
422/Linux-repo
| |\ \
| | * | 91a5498 Create README.md
| | * | 90b8c31 Create 1
| |  /
| * | 157103f file.py added
| * | 7b74227 Create file
|  /
* / 0c2222c My first file is added
 /
```

Filtering the Commit History

We can filter the output according to our needs. It's a unique feature of Git. We can apply many filters like amount, date, author, and more on output. Each filter has its specifications. They can be used for implementing some navigation operations on output.

Let's understand each of these filters in detail.

By Amount:

We can limit the number of output commit by using git log command. It is the most specific command. This command will remove the complexity if you are interested in fewer commits.

To limit the git log's output, including the -<n> option. If we want only the last three commit, then we can pass the argument -3 in the git log command. Consider the below output:

```
$ git log -3
```

Git Diff

Git diff is a command-line utility. It's a multiuse Git command. When it is executed, it runs a diff function on Git data sources. These data sources can be files, branches, commits, and more. It is used to show changes between commits, commit, and working tree, etc.

It compares the different versions of data sources. The version control system stands for working with a modified version of files. So, the diff command is a useful tool for working with Git.

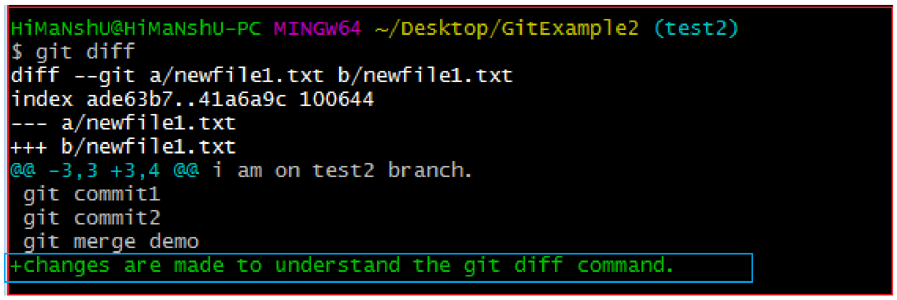
However, we can also track the changes with the help of git log command with option -p. The git log command will also work as a git diff command.

Let's understand different scenarios where we can utilize the git diff command.

Scenerio1: Track the changes that have not been staged.

The usual use of git diff command that we can track the changes that have not been staged.

Suppose we have edited the newfile1.txt file. Now, we want to track what changes are not staged yet. Then we can do so from the git diff command. Consider the below output:



```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git diff
diff --git a/newfile1.txt b/newfile1.txt
index ade63b7..41a6a9c 100644
--- a/newfile1.txt
+++ b/newfile1.txt
@@ -3,3 +3,4 @@ i am on test2 branch.
 git commit1
 git commit2
 git merge demo
+changes are made to understand the git diff command.
```

From the above output, we can see that the changes made on newfile1.txt are displayed by git diff command. As we have edited it as "changes are made to understand the git diff command." So, the output is displaying the changes with its content. The highlighted section of the above output is the changes in the updated file. Now, we can decide whether we want to stage this file like this or not by previewing the changes.

Scenerio2: Track the changes that have staged but not committed:

The git diff command allows us to track the changes that are staged but not committed. We can track the changes in the staging area. To check the already staged changes, use the --staged option along with git diff command.

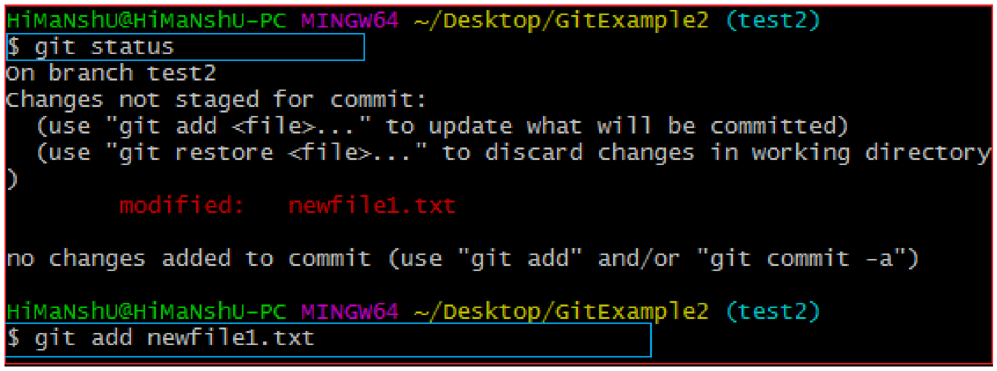
To check the untracked file, run the git status command as:

1. $ git status

The above command will display the untracked file from the repository. Now, we will add it to the staging area. To add the file in the staging area, run the git add command as:

1. $ git add < file name>

The above command will add the file in the staging area. Consider the below output:

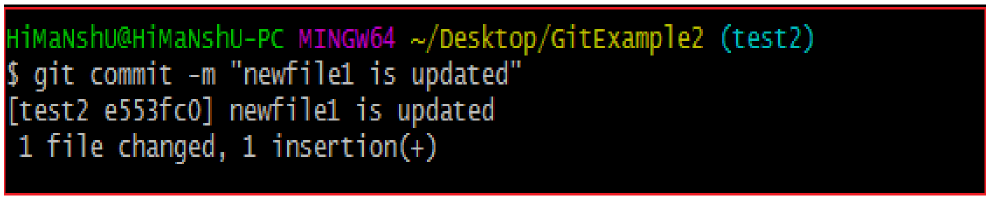


```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git status
On branch test2
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory
)
        modified:   newfile1.txt

no changes added to commit (use "git add" and/or "git commit -a")

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git add newfile1.txt
```

Scenerio3: Track the changes after committing a file:

Git, let us track the changes after committing a file. Suppose we have committed a file for the repository and made some additional changes after the commit. So we can track the file on this stage also.

In the below output, we have committed the changes that we made on our newfile1.txt. Consider the below output:



```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git commit -m "newfile1 is updated"
[test2 e553fc0] newfile1 is updated
 1 file changed, 1 insertion(+)
```

Git Status

The git status command is used to display the state of the repository and staging area. It allows us to see the tracked, untracked files and changes. This command will not show any commit records or information.

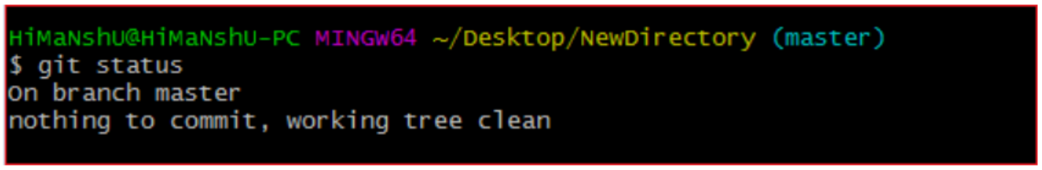
Mostly, it is used to display the state between **Git Add** and **Git commit** command. We can check whether the changes and files are tracked or not.

Let's understand the different states of status command.

Status when Working Tree is cleaned

Before starting with git status command, let's see how the git status looks like when there are no changes made. To check the status, open the git bash, and run the status command on your desired directory. It will run as follows:

1. $ git status
   Output:



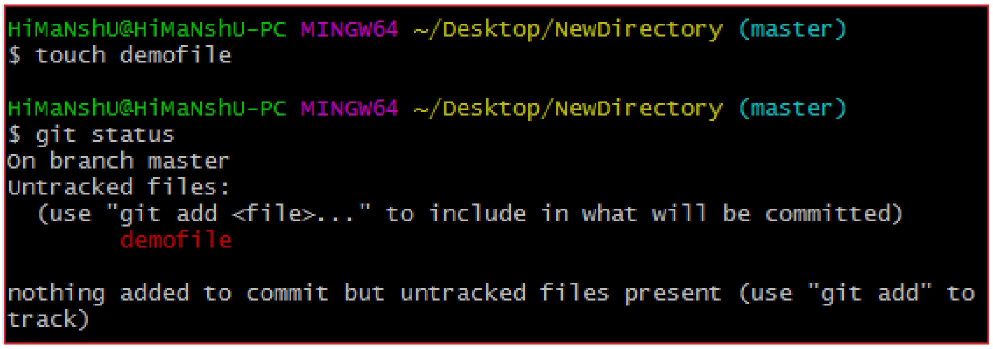
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Since there is nothing to track or untrack in the working tree, so the output is showing as the **working tree is clean**.

Status when a new file is created

When we create a file in the repository, the state of the repository changes. Let's create a file using the **touch** command. Now, check the status using the status command.

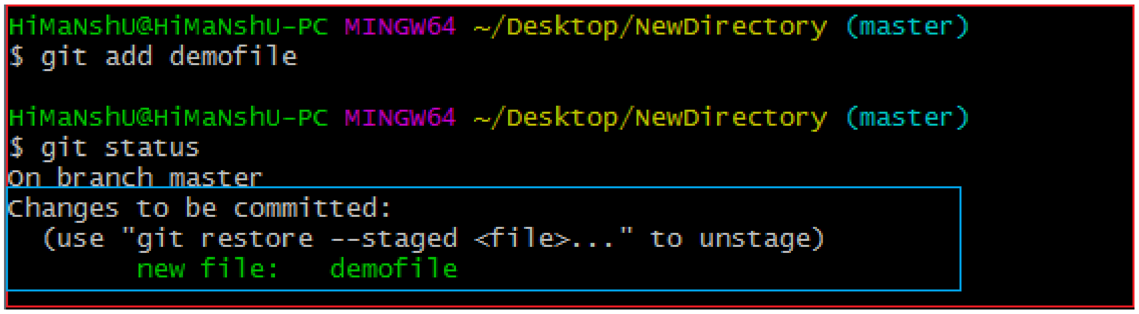
Consider the below output:



```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ touch demofile

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        demofile

nothing added to commit but untracked files present (use "git add" to
track)
```

As we can see from the above output, the status is showing as "**nothing added to commit but untracked files present (use "git add" to track)**". The status command also displays the suggestions. As in the above output, it is suggesting to use the add command to track the file.

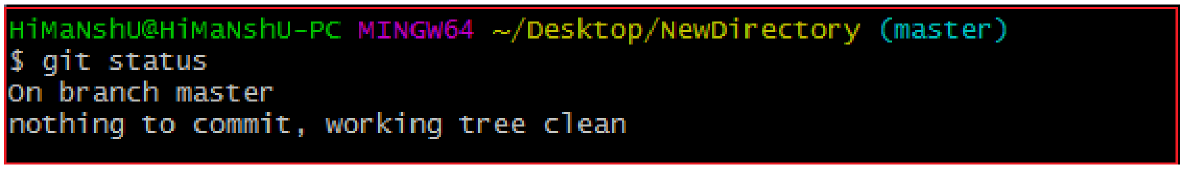
Let's track the file and will see the status after adding a file to the repository. To track the file, run the add command. Consider the below output:



```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git add demofile

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   demofile
```

From the above output, we can see that the status after staging the file is showing as "**changes to be committed**".

Before committing blindly, we can check the status. This command will help us to avoid the changes that we don't want to commit. Let's commit it and then check the status. Consider the below output:



```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
nothing to commit, working tree clean
```

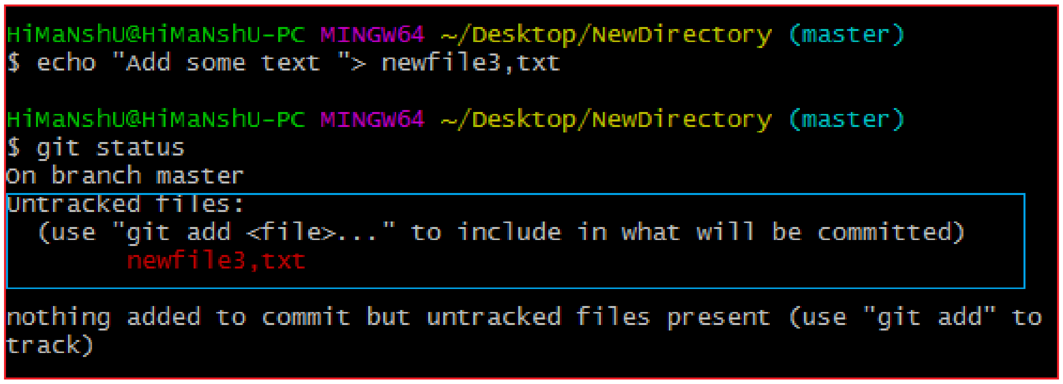
We can see that the current status after committing the file is clean as it was before.

Status when an existing file is modified

Let's check the status when an existing file is modified. To modify file, run the **echo** command as follows:

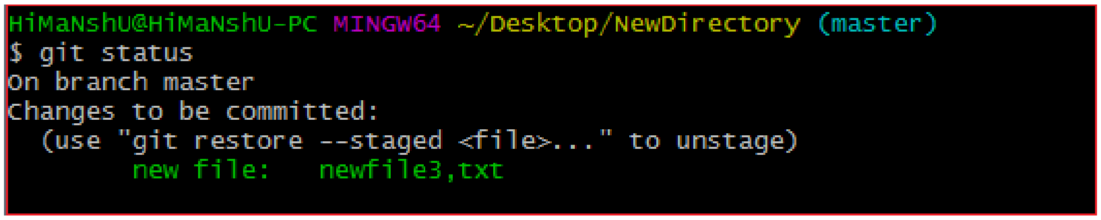
1. $ echo "Text"> Filename
    2. The above command will add the text to the specified file, now check the status of the repository. Consider the below output:

The above command will add the text to the specified file, now check the status of the repository. Consider the below output:



```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ echo "Add some text "> newfile3,txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        newfile3,txt

nothing added to commit but untracked files present (use "git add" to
track)
```

We can see that the updated file is displayed as untracked files. It is shown in red color because it is not staged yet. When it will stage, its color will change to Green. Consider the below output:



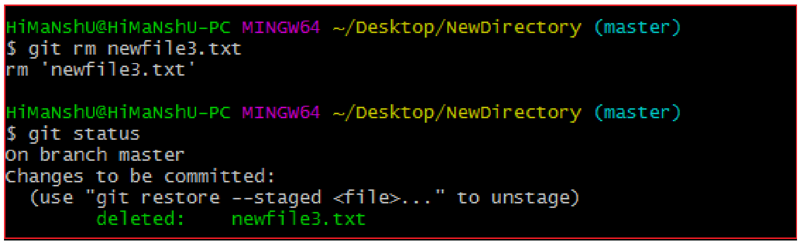
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   newfile3,txt
```

Status when a file is deleted

Let's check the status when a file is deleted from the repository. To delete a file from the repository, run the rm command as follows:

1. $ git rm < File Name>

The above command will delete the specified file from the repository. Now, check the status of the repository. Consider the below output:



```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git rm newfile3.txt
rm 'newfile3.txt'

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    newfile3.txt
```

The current status of the repository has been updated as deleted.