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.



Features of GitHub

GitHub is a place where programmers and designers work together. They collaborate, contribute, and fix bugs together. It hosts plenty of open source projects and codes of various programming languages.

Some of its significant features are as follows.

* Collaboration
* Integrated issue and bug tracking
* Graphical representation of branches
* Git repositories hosting
* Project management
* Team management
* Code hosting
* Track and assign tasks
* Conversations
* Wikisc

Benefits of GitHub

GitHub can be separated as the Git and the Hub. GitHub service includes access controls as well as collaboration features like task management, repository hosting, and team management.

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.

Difference between git and gitHub

Programming language wordings are very intuitive these days. By hearing the name of a particular language, we start imagining what all it will be.

[Java](https://www.javatpoint.com/java-tutorial) and [Javascript](https://www.javatpoint.com/javascript-tutorial) are very similar to the names ham and hamster, the logo of [python](https://www.javatpoint.com/python-tutorial) is intertwined with the image of snakes.



So, someone looking at git and github would find any apparent connection between them. Let us see git and github in detail with the differences between them.

Git



There are many words to define [git](https://www.javatpoint.com/git), but it is an open-source distributed version control system in simpler words.

Let us break each component in the definition and understand it.

* **Open-source -** A type of computer software released under a specific license. The users are given permissions to use the code, modify the code, give suggestions, clone the code to add new functionality. In other words, if the software is open-source, it is developed collaboratively in a public manner. The open-source softwares is cheaper, more flexible, and lasts longer than an authority or a company. The products in the source code include code, documents, formats for the users to understand and contribute to it. Using open-source a project can be expanded to update or revise the current features. Unix and Linux are examples of open-source softwares.
* **Control system -** The work of a control system is to track the content. In other words, git is used to storing the content to provide the services and features to the user.
* **Version Control system -** Just like an app has different updates due to bugs and additional feature addition, version changes, git also supports this feature. Many developers can add their code in parallel. So the version control system easily manages all the updates that are done previously.  
  Git provides the feature of branching in which the updated code can be done, and then it can be merged with the main branch to make it available to the users. It not only makes everything organized but keeps synchronization among the developers to avoid any mishap. Other examples of version control systems are Helix core, Microsoft TFS, etc.
* **Distributed version control system -** Here distributed version control system means if a developer contributes to open source, the code will also be available in his remote repository. The developer changes his local repository and then creates a pull request to merge his changes in the central repository. Hence, the word distributed means the code is stored in the central server and stored in every developer's remote system.

**Why is git needed?**

When a team works on real-life projects, git helps ensure no code conflicts between the developers. Furthermore, the project requirements change often. So a git manages all the versions. If needed, we can also go back to the original code. The concept of branching allows several projects to run in the same codebase.

GitHub

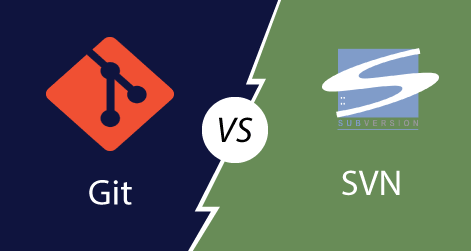


By the name, we can visualize that it is a Hub, projects, communities, etc. [GitHub](https://www.javatpoint.com/github) is a [Git repository](https://www.javatpoint.com/git-repository) hosting service that provides a web-based graphical interface. It is the largest community in the world. Whenever a project is open-source, that particular repository gains exposure to the public and invites several people to contribute.

The source code of several projects is available on github which developers can use in any means.

Using github, many developers can work on a single project remotely because it facilitates collaboration.

**Features of gitHub**

* Using github the project managers can collaborate, review and guide the developers regarding any changes. This makes project management easy.
* The github repositories can be made public or private. Thus allowing safety to an organization in case of a project.
* GitHub has a feature of pull requests and issues in which all the developers can stay on the same page and organize.
* All the codes and their documentation are in one place in the same repository. Hence it makes easy code hosting.
* There are some special tools that github uses to identify the vulnerabilities in the code which other softwares do not have. Hence there is safety among the developers from code start till launch.
* Github is available for mobile and desktops. The UI is so user-friendly that it becomes straightforward to get comfortable with and use it.
* Git vs SVN
* Apache Subversion or **SVN is one of the most popular centralized version control systems**. Now, SVN's popularity is on the decrease, but there are still millions of projects stored in it. It can continue to be actively maintained by an open-source community. In SVN, you can check out a single version of the repository. It stores data in a central server. The drawback of the SVN is, it has the entire history on a local repository which limits you. You can only do commits, diffs, logs, branches, merges, file annotations, etc.
* 
* While, **Git is a popular distributed version control system**, which means that you can clone your repository. Thus you can get a complete copy of your entire history of that project. This means you can access all your commits.
* **Git has more advantages than SVN**. It is much better for those developers who are not always connected to the master repository. Also, it is much faster than SVN.
* To better understand the differences between Git and Subversion. Let's have a look at following significance points.

|  |  |
| --- | --- |
| **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 vs Mercurial

Mercurial and Git both are two quite similar and most popular distributed version control systems. Their strengths and weaknesses make them ideal for different use cases. Both tools use a directed acyclic graph to store history.

**Mercurial is a distributed source control management tool.** It is free and open-source. It can handle projects of any size and offers an easy and intuitive interface.

Today, Git has more than 31 million users and is owned by Microsoft. Since the last decade, the Git has become the standard for most development projects.

Mercurial still has a handful tool of large development organizations. Some software development giants like Facebook, Mozilla, and World Wide Web Consortium are using it. But it only has approx 2 % of the VCS market share. Comparatively, Git has covered more than 80% market share.

Both version control systems, i.e., Mercurial and Git are distributed version control systems (DVCS).

To better understand the similarities and differences between Git and Mercurial, let's have a look at the following points.

|  |  |
| --- | --- |
| **Git** | **Mercurial** |
| Git is a little bit of complex than Mercurial. | Mercurial is simpler than Git. |
| No VCS are entirely secured, but Git offers many functions to enhance safety. | Mercurial may be safer for fresher. It has more security features. |
| Git has a powerful and effective branching model. Branching in Git is better than Branching in Mercurial. | Branching in Mercurial doesn't refer the same meaning as in Git. |
| Git supports the staging area, which is known as the index file. | There is no index or staging area before the commit in Mercurial. |
| The most significant benefit with Git is that it has become an industry-standard, which means more developers are familiar with it. | Mercurial's significant benefit is that it's easy to learn and use, which is useful for less-technical content contributors. |
| Git needs periodic maintenance for repositories. | It does not require any maintenance. |
| It holds Linux heritage. | It is python based. |
| Git is slightly slower than Mercurial. | It is faster than Git. |
| Git supports the unlimited number of parents. | Mercurial allows only two parents. |

# Git Version Control System

A version control system is a 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.

## 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.

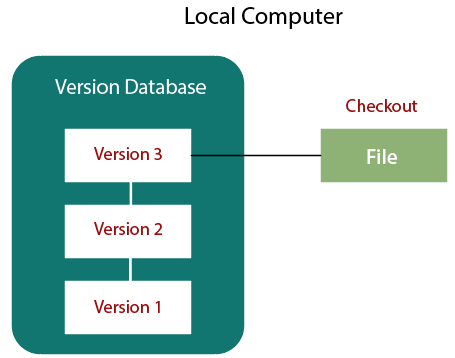
Some key benefits of having a version control system are as follows.

* 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**



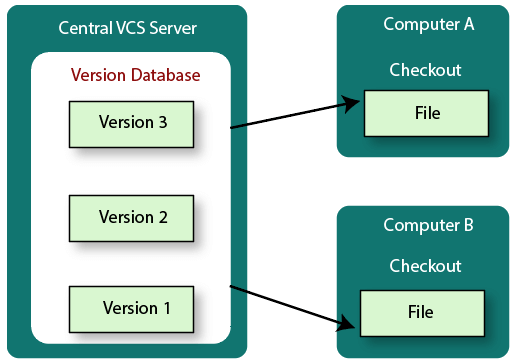
The localized version control method is a common approach because of its simplicity. But this approach leads to a higher chance of error. In this approach, you may forget which directory you're in and accidentally write to the wrong file or copy over files you don't want to.

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.

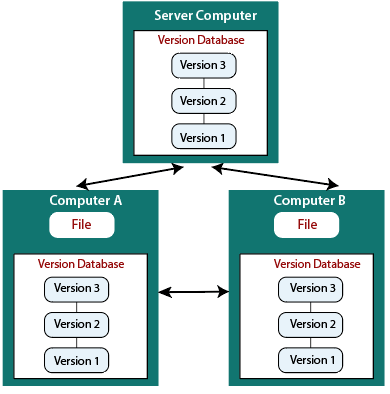
* Everyone on the system has information about the work what others are doing on the project.
* Administrators have control over other developers.
* It is easier to deal with a centralized version control system than a localized version control system.
* A local version control system facilitates with 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 |

How to Install Git on Windows

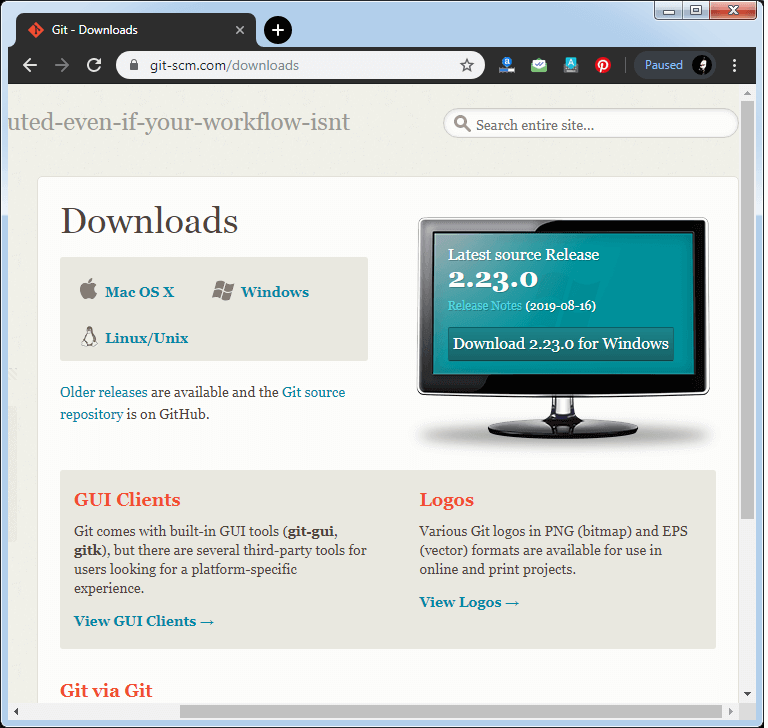
To use Git, you have to install it on your computer. Even if you have already installed Git, it's probably a good idea to upgrade it to the latest version. You can either install it as a package or via another installer or download it from its official site.

Now the question arises that how to download the Git installer package. Below is the stepwise installation process that helps you to download and install the Git.

How to download Git?

**Step1**

To download the Git installer, visit the Git's official site and go to download page. The link for the download page is <https://git-scm.com/downloads>. The page looks like as



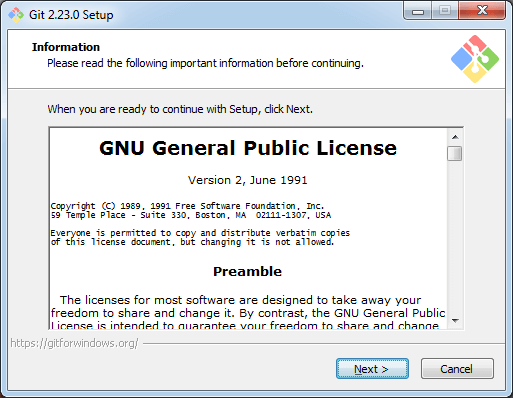
Click on the package given on the page as **download 2.23.0 for windows**. The download will start after selecting the package.

Now, the Git installer package has been downloaded.

Install Git

**Step2**

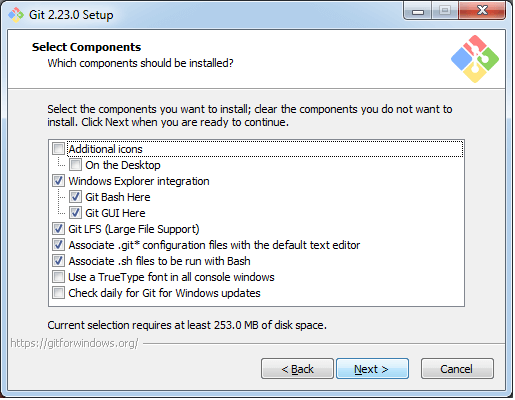
Click on the downloaded installer file and select **yes** to continue. After the selecting **yes** the installation begins, and the screen will look like as



Click on **next** to continue.

**Step3**

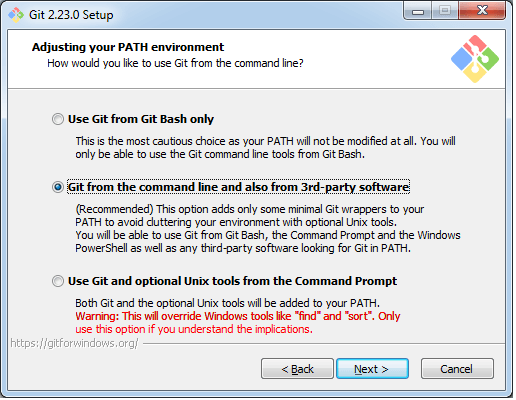
Default components are automatically selected in this step. You can also choose your required part.



Click next to continue.

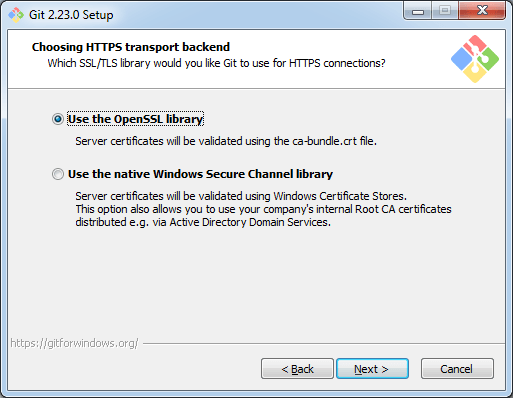
**Step4**

The default Git command-line options are selected automatically. You can choose your preferred choice. Click **next** to continue.



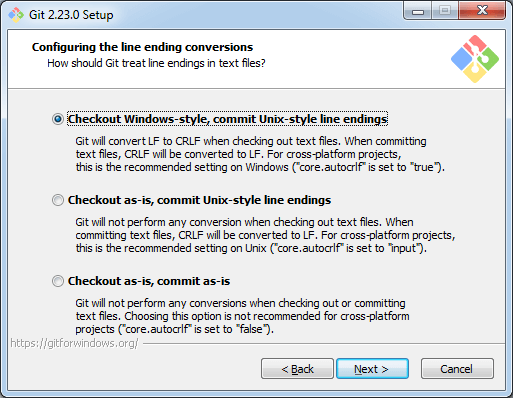
**Step5**

The default transport backend options are selected in this step. Click **next** to continue.



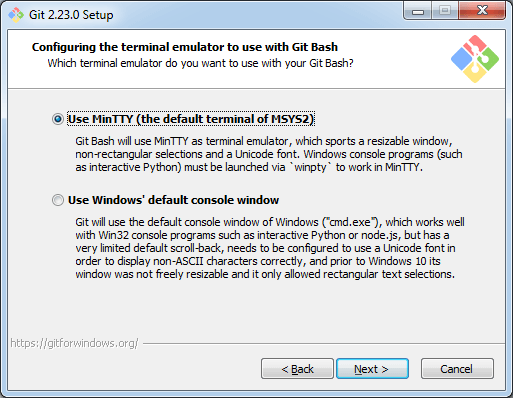
**Step6**

Select your required line ending option and click next to continue.



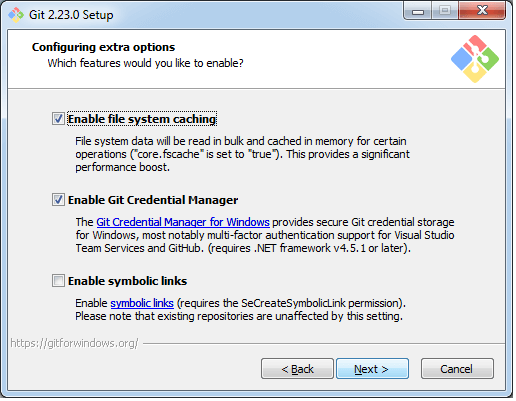
**Step7**

Select preferred terminal emulator clicks on the **next** to continue.



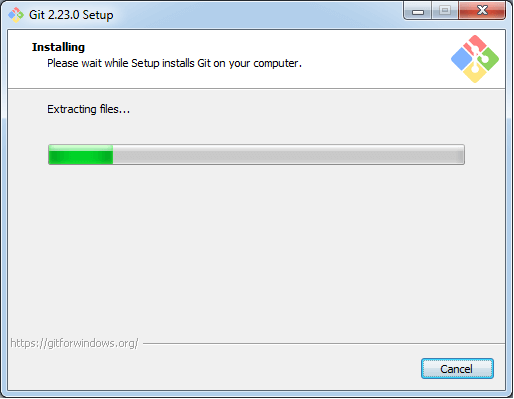
**Step8**

This is the last step that provides some extra features like system caching, credential management and symbolic link. Select the required features and click on the **next** option.



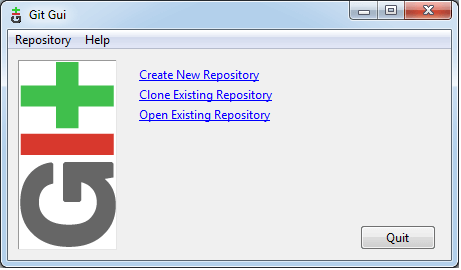
**Step9**

The files are being extracted in this step.



Therefore, The Git installation is completed. Now you can access the **Git Gui** and **Git Bash**.

The **Git Gui** looks like as



It facilitates with three features.

* Create New Repository
* Clone Existing Repository
* Open Existing Repository

The **Git Bash** looks like as

v

Install Git on Mac

There are multiple ways to install Git on mac. It comes inbuilt with Xcode or its other command-line tools. To start the Git, open terminal and enter the below command:

1. $ git --version

The above command will display the installed version of Git.

**Output:**

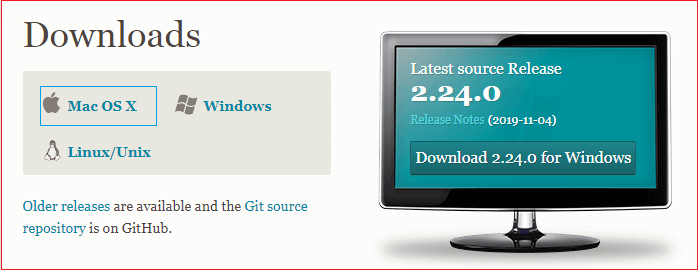
git version 2.24.0 (Apple Git-66)

If you do not have installed it already, then it will ask you to install it.

Apple provides support for Git, but it lags by several major versions. We may install a newer version of Git using one of the following methods:

Git Installer for Mac

This process is the simplest way to download the latest version of Git. Visit the [official page](https://git-scm.com/downloads) of git downloads. Choose the download option for **Mac OS X**.



The installer file will download to your system. Follow the prompts, choose the required installer option. After the installation process completed, verify the installation was successful by running the below command on the terminal:

1. $ git --version

The above command will display the installed version of Git. Consider the below output.

**Output:**

git version 2.24.0 (Apple Git-66)

Now, we have successfully installed the latest version on our mac OS. It's time to configure the version control system for the first use.

To register a username, run the below command:

1. $ git config --global user.name "javaTpoint"

To register an email address for the given author, run the below command:

1. $ git config --global user.email "javatpoint@xyz"

To go in-depth with the git config command, visit [Here](https://www.javatpoint.com/git-environment-setup).

Installation via MacPorts

Sometimes MacPorts also referred to DarwinPorts. It makes the straightforward installation of software on the Mac OS and Darwin operating systems. If we have installed MacPorts for managing packages on OS X, follow the below steps to install Git.

**Step1: Update MacPorts**

To update MacPorts, run the below command:

1. $ sudo port selfupdate

**Step2: Search for the latest Ports**

To search for the most recent available Git ports and variants, run the below command:

1. $ port search git
2. $ port variants git

The above command will search for the latest available port and options and will install it.

**Step3: Install Git**

To install Git, run the below command:

1. $ sudo port install git

We can also install some extra tools with Git. These tools may assist Git in different manners. To Install Git with bash-completion, svn, and the docs, run the below command:

1. $ sudo port install git +svn +doc +bash\_completion +gitweb

Now, we have successfully installed Git with the help of MacPorts on our system.

**Step4: Configure Git**

The next step for the first use is git configuration.

We will configure the Git username and email address as same as given above.

To register a username, run the below command:

1. $ git config --global user.name "javaTpoint"

To register an email address for the given author, run the below command:

1. $ git config --global user.email "javatpoint@xyz"

Install Git via Homebrew

Homebrew is used to make the software installation straight forward. If we have installed Homebrew for managing packages on OS X, follow the below steps to go with Git:

**Step1: install Git**

Open the terminal and run the below command to install Git using Homebrew:

1. $ brew install git

The above command will install the Git on our machine. The next step is to verify the installation.

**Step2: Verify the installation**

It is essential to ensure that whether the installation process has been succeeded or not.

To verify whether the installation has been successful or not, run the below command:

1. $ git --version

The above command will display the version that has been installed on your system. Consider the below output:

git version 2.24.0

**Step3: Configure Git**

We will configure the Git username and email address same as given above.

To register a username, run the below command:

1. $ git config --global user.name "javaTpoint"

To register an email address for the given author, run the below command:

1. $ git config --global user.email "javatpoint@xyz"

Git Environment Setup

The environment of any tool consists of elements that support execution with software, hardware, and network configured. It includes operating system settings, hardware configuration, software configuration, test terminals, and other support to perform the operations. It is an essential aspect of any software.

It will help you to understand how to set up Git for first use on various platforms so you can read and write code in no time.

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.

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

**Setting email id**

The Git uses this email id for each commit.

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

There are many other configuration options that the user can set.

**Setting editor**

You can set the default text editor when Git needs you to type in a message. If you have not selected any of the editors, Git will use your default system's editor.

To select a different text editor, such as Vim,

1. $ git config --global core.editor Vim

**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.

1. $ git config -list

This command will list all your settings. See the below command line output.

**Output**

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop

$ git config --list

core.symlinks=false

core.autocrlf=true

core.fscache=true

color.diff=auto

color.status=auto

color.branch=auto

color.interactive=true

help.format=html

rebase.autosquash=true

http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt

http.sslbackend=openssl

diff.astextplain.textconv=astextplain

filter.lfs.clean=git-lfs clean -- %f

filter.lfs.smudge=git-lfs smudge --skip -- %f

filter.lfs.process=git-lfs filter-process --skip

filter.lfs.required=true

credential.helper=manager

gui.recentrepo=C:/Git

user.email=dav.himanshudubey481@gmail.com

user.name=Himanshu Dubey

**Colored output**

You can customize your Git output to view a personalized color theme. The **git config** can be used to set these color themes.

**Color.ui**

1. $ Git config -global color.ui true

The default value of color.ui is set as auto, which will apply colors to the immediate terminal output stream. You can set the color value as true, false, auto, and always.

Git configuration levels

The git config command can accept arguments to specify the configuration level. The following configuration levels are available in the Git config.

* local
* global
* system

**--local**

It is the default level in Git. Git config will write to a local level if no configuration option is given. Local configuration values are stored in **.git/config** directory as a file.

**--global**

The global level configuration is user-specific configuration. User-specific means, it is applied to an individual operating system user. Global configuration values are stored in a user's home directory. **~ /.gitconfig** on UNIX systems and **C:\Users\\.gitconfig** on windows as a file format.

**--system**

The system-level configuration is applied across an entire system. The entire system means all users on an operating system and all repositories. The system-level configuration file stores in a **gitconfig** file off the system directory. **$(prefix)/etc/gitconfig** on UNIX systems and **C:\ProgramData\Git\config** on Windows.

The order of priority of the Git config is local, global, and system, respectively. It means when looking for a configuration value, Git will start at the local level and bubble up to the system level.

# Git Tools

To explore the robust functionality of Git, we need some tools. Git comes with some of its tools like Git Bash, Git GUI to provide the interface between machine and user. It supports inbuilt as well as third-party tools.

Git comes with built-in GUI tools like **git bash**, **git-gui**, and **gitk** for committing and browsing. It also supports several third-party tools for users looking for platform-specific experience.

## Git Package Tools

Git provides powerful functionality to explore it. We need many tools such as commands, command line, Git GUI. Let's understand some essential package tools.

### **GitBash**

Git Bash is an application for the Windows environment. It is used as Git command line for windows. Git Bash provides an emulation layer for a Git command-line experience. Bash is an abbreviation of **Bourne Again Shell**. Git package installer contains Bash, bash utilities, and Git on a Windows operating system.

Bash is a standard default shell on Linux and macOS. A shell is a terminal application which is used to create an interface with an operating system through commands.

By default, Git Windows package contains the Git Bash tool. We can access it by right-click on a folder in Windows Explorer.

#### **Git Bash Commands**

Git Bash comes with some additional commands that are stored in the **/usr/bin** directory of the Git Bash emulation. Git Bash can provide a robust shell experience on Windows. Git Bash comes with some essential shell commands like **Ssh**, **scp**, **cat**, **find**.

Git Bash also includes the full set of Git core commands like **git clone, git commit, git checkout, git push**, and more.

### **Git GUI**

Git GUI is a powerful alternative to Git BASH. It offers a graphical version of the Git command line function, as well as comprehensive visual diff tools. We can access it by simply right click on a folder or location in windows explorer. Also, we can access it through the command line by typing below command.

1. $ git gui

A pop-up window will open as Git gui tool. The Git GUI's interface looks like as:

Git facilitates with some built-in GUI tools for committing (git-gui) and browsing (gitk), but there are many third-party tools for users looking for platform-specific experience.

### **Gitk**

gitk is a graphical history viewer tool. It's a robust GUI shell over **git log** and **git grep**. This tool is used to find something that happened in the past or visualize your project's history.

Gitk can invoke from the command-line. Just change directory into a Git repository, and type:

1. $ gitk [git log options]

This command invokes the gitk graphical interface and displays the project history. The Gitk interface looks like this:

Gitk supports several command-line options, most of which are passed through to the underlying git log action.

## Git Third-Party Tools

Many third-party tools are available in the market to enhance the functionality of Git and provide an improved user interface. These tools are available for distinct platforms like Windows, Mac, Linux, Android, iOS.

A list of popular third party Git tools are as follows:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Tools** | **Platforms** | | | | | **Price** | **License Type** |
| **Windows** | **Mac** | **Linux** | **Android** | **iOS** |
| SourceTree | Yes | Yes | No | No | No | Free | Proprietary |
| GitHub Desktop | Yes | Yes | No | No | No | Free | MIT |
| TortoiseGit | Yes | No | No | No | No | Free | GNU GPL |
| Git Extensions | Yes | Yes | Yes | No | No | Free | GNU GPL |
| GitKraken | Yes | Yes | Yes | No | No | Free/$29/$49 | Proprietary |
| SmartGit | Yes | Yes | Yes | No | No | $79/user/free for non-commercial use | Proprietary |
| Tower | Yes | Yes | No | No | No | $79/user (30 days free trial) | Proprietary |
| Git Up | No | Yes | No | No | No | Free | GNU GPL |
| GitEye | Yes | Yes | Yes | No | No | Free | Proprietary |
| gitg | Yes | No | Yes | No | No | Free | GNUGPL |
| Git2Go | No | No | No | No | Yes | Free with in-app purchases | Proprietary |
| GitDrive | No | No | No | No | Yes | Free with in-app purchases | Proprietary |
| GitFinder | No | Yes | No | No | No | $24.95 | Proprietary |
| SnailGit | No | Yes | No | No | No | &9.99/Lite version | Proprietary |
| Pocket Git | No | No | No | Yes | No | 1.99€ | Proprietary |
| Sublime Merge | Yes | Yes | Yes | No | No | $99/user, $75 annual business sub, free eval | Proprietary |
|  |  |  |  |  |  |  |  |

# Git Terminology

Git is a tool that covered vast terminology and jargon, which can often be difficult for new users, or those who know Git basics but want to become Git masters. So, we need a little explanation of the terminology behind the tools. Let's have a look at the commonly used terms.

**Some commonly used terms are:**

### [**Branch**](https://www.javatpoint.com/git-branch)

A branch is a version of the repository that diverges from the main working project. It is an essential feature available in most modern version control systems. A Git project can have more than one branch. We can perform many operations on Git branch-like rename, list, delete, etc.

### [**Checkout**](https://www.javatpoint.com/git-checkout)

In Git, the term checkout is used for the act of switching between different versions of a target entity. The **git checkout** command is used to switch between branches in a repository.

### [**Cherry-Picking**](https://www.javatpoint.com/git-cherry-pick)

Cherry-picking in Git is meant to apply some commit from one branch into another branch. In case you made a mistake and committed a change into the wrong branch, but do not want to merge the whole branch. You can revert the commit and cherry-pick it on another branch.

### [**Clone**](https://www.javatpoint.com/git-clone)

The **git clone** is a Git command-line utility. It is used to make a copy of the target repository or clone it. If I want a local copy of my repository from GitHub, this tool allows creating a local copy of that repository on your local directory from the repository URL.

### [**Fetch**](https://www.javatpoint.com/git-fetch)

It is used to fetch branches and tags from one or more other repositories, along with the objects necessary to complete their histories. It updates the remote-tracking branches.

### [**HEAD**](https://www.javatpoint.com/git-head)

HEAD is the representation of the last commit in the current checkout branch. We can think of the head like a current branch. When you switch branches with git checkout, the HEAD revision changes, and points the new branch.

### [**Index**](https://www.javatpoint.com/git-index)

The Git index is a staging area between the working directory and repository. It is used as the index to build up a set of changes that you want to commit together.

### [**Master**](https://www.javatpoint.com/git-origin-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.

### [**Merge**](https://www.javatpoint.com/git-merge)

Merging is a process to put a forked history back together. The git merge command facilitates you to take the data created by git branch and integrate them into a single branch.

### [**Origin**](https://www.javatpoint.com/git-origin-master)

In Git, "origin" is a reference to the remote repository from a project was initially cloned. More precisely, it is used instead of that original repository URL to make referencing much easier.

### [**Pull/Pull Request**](https://www.javatpoint.com/git-pull)

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

Pull requests are a process for a developer to notify team members that they have completed a feature. Once their feature branch is ready, the developer files a pull request via their remote server account. Pull request announces all the team members that they need to review the code and merge it into the master branch.

### [**Push**](https://www.javatpoint.com/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.

### [**Rebase**](https://www.javatpoint.com/git-rebase)

In Git, the term rebase is referred to as the process of moving or combining a sequence of commits to a new base commit. Rebasing is very beneficial and visualized the process in the environment of a feature branching workflow.

From a content perception, rebasing is a technique of changing the base of your branch from one commit to another.

### [**Remote**](https://www.javatpoint.com/git-remote)

In Git, the term remote is concerned with the remote repository. It is a shared repository that all team members use to exchange their changes. A remote repository is stored on a code hosting service like an internal server, GitHub, Subversion and more.

In case of a local repository, a remote typically does not provide a file tree of the project's current state, as an alternative it only consists of the .git versioning data.

### [**Repository**](https://www.javatpoint.com/git-repository)

In Git, Repository is like a data structure used by VCS to store metadata for a set of files and directories. It contains the collection of the file 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.

### [**Stashing**](https://www.javatpoint.com/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 branch without committing the current branch.

### [**Tag**](https://www.javatpoint.com/git-tag)

Tags make a point as a specific point in Git history. It is used to mark a commit stage as important. We can tag a commit for future reference. Primarily, it is used to mark a projects initial point like v1.1. There are two types of tags.

1. Light-weighted tag
2. Annotated tag

### [**Upstream And Downstream**](https://www.javatpoint.com/git-upstream-and-downstream)

The term upstream and downstream is a reference of the repository. Generally, upstream is where you cloned the repository from (the origin) and downstream is any project that integrates your work with other works. However, these terms are not restricted to Git repositories.

### [**Git Revert**](https://www.javatpoint.com/git-revert)

In Git, the term revert is used to revert some commit. To revert a commit, **git revert** command is used. It is an undo type command. However, it is not a traditional undo alternative.

### [**Git Reset**](https://www.javatpoint.com/git-reset)

In Git, the term reset stands for undoing changes. The **git reset** command is used to reset the changes. The git reset command has three core forms of invocation. These forms are as follows.

* Soft
* Mixed
* Hard

### [**Git Ignore**](https://www.javatpoint.com/git-ignore)

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

### [**Git Diff**](https://www.javatpoint.com/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.

### [**Git Cheat Sheet**](https://www.javatpoint.com/git-cheat-sheet)

A Git cheat sheet is a summary of Git quick references. It contains basic Git commands with quick installation. A cheat sheet or crib sheet is a brief set of notes used for quick reference. Cheat sheets are so named because the people may use it without no prior knowledge.

### [**Git Flow**](https://www.javatpoint.com/git-flow)

ADVERTISEMENT

GitFlow is a **branching model** for Git, developed by **Vincent Driessen**. It is very well organized to collaborate and scale the development team. Git flow is a collection of Git commands. It accomplishes many repository operations with just single commands.

### [**Git Squash**](https://www.javatpoint.com/git-squash)

In Git, the term squash is used to squash previous commits into one. Git squash is an excellent technique to group-specific changes before forwarding them to others. You can merge several commits into a single commit with the powerful interactive rebase command.

### [**Git Rm**](https://www.javatpoint.com/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.

### [**Git Fork**](https://www.javatpoint.com/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.

Great use of using forks to propose changes for bug fixes. To resolve an issue for a bug that you found, you can:

* Fork the repository.
* Make the fix.
* Forward a pull request to the project owner.

# 12 Git Commands

There are many different ways to use Git. Git supports many command-line tools and graphical user interfaces. The Git command line is the only place where you can run all the Git commands.

The following set of commands will help you understand how to use Git via the command line.

## Basic Git Commands

Here is a list of most essential Git commands that are used daily.

1. [Git Config command](https://www.javatpoint.com/git-commands#config-command)
2. [Git init command](https://www.javatpoint.com/git-commands#init-command)
3. [Git clone command](https://www.javatpoint.com/git-commands#clone-command)
4. [Git add command](https://www.javatpoint.com/git-commands#add-command)
5. [Git commit command](https://www.javatpoint.com/git-commands#commit-command)
6. [Git status command](https://www.javatpoint.com/git-commands#status-command)
7. [Git push Command](https://www.javatpoint.com/git-commands#push-command)
8. [Git pull command](https://www.javatpoint.com/git-commands#pull-command)
9. [Git Branch Command](https://www.javatpoint.com/git-commands#branch-command)
10. [Git Merge Command](https://www.javatpoint.com/git-commands#merge-command)
11. [Git log command](https://www.javatpoint.com/git-commands#log-command)
12. [Git remote command](https://www.javatpoint.com/git-commands#remote-command)

Let's understand each command in detail.

## 1) Git config command

This command configures the user. The Git config command is the first and necessary command used on the Git command line. This command sets the author name and email address to be used with your commits. Git config is also used in other scenarios.

**Syntax**

1. $ git config --global user.name "ImDwivedi1"
2. $ git config --global user.email "Himanshudubey481@gmail.com"

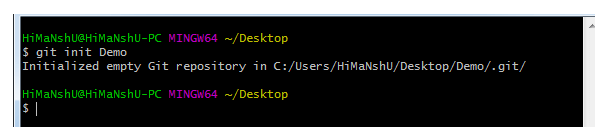
### **2) Git Init command**

This command is used to create a local repository.

**Syntax**

1. $ git init Demo

The init command will initialize an empty repository. See the below screenshot.

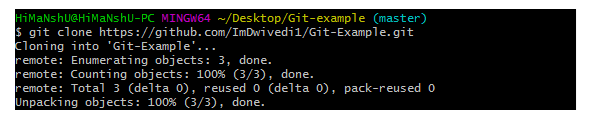


### **3) Git clone command**

This command is used to make a copy of a repository from an existing URL. If I want a local copy of my repository from GitHub, this command allows creating a local copy of that repository on your local directory from the repository URL.

**Syntax**

1. $ git clone URL



### **4) Git add command**

This command is used to add one or more files to staging (Index) area.

**Syntax**

To add one file

1. $ git add Filename

To add more than one file

1. $ git add\*

Git Commands

### **5) Git commit command**

Commit command is used in two scenarios. They are as follows.

**Git commit -m**

This command changes the head. It records or snapshots the file permanently in the version history with a message.

**Syntax**

1. $ git commit -m " Commit Message"

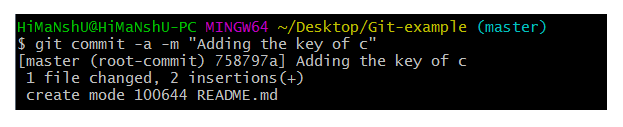
**Git commit -a**

ADVERTISEMENT

This command commits any files added in the repository with git add and also commits any files you've changed since then.

**Syntax**

1. $ git commit -a

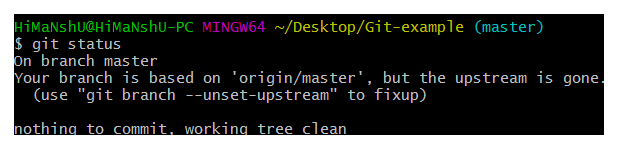


### **6) Git status command**

The status command is used to display the state of the working directory and the staging area. It allows you to see which changes have been staged, which haven't, and which files aren?t being tracked by Git. It does not show you any information about the committed project history. For this, you need to use the git log. It also lists the files that you've changed and those you still need to add or commit.

**Syntax**

1. $ git status



### **7) Git push Command**

It is used to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repo. It's the complement to git fetch, but whereas fetching imports commits to local branches on comparatively pushing exports commits to remote branches. Remote branches are configured by using the git remote command. Pushing is capable of overwriting changes, and caution should be taken when pushing.

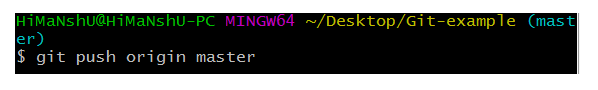
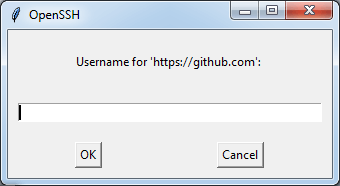
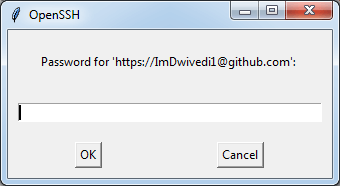
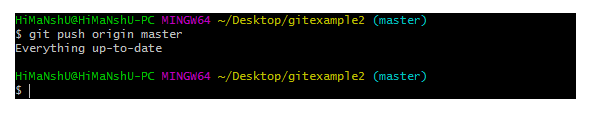
Git push command can be used as follows.

**Git push origin master**

This command sends the changes made on the master branch, to your remote repository.

**Syntax**

1. $ git push [variable name] master

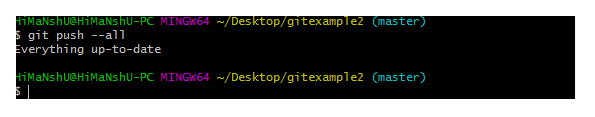
  
  
  


**Git push -all**

This command pushes all the branches to the server repository.

**Syntax**

1. $ git push --all

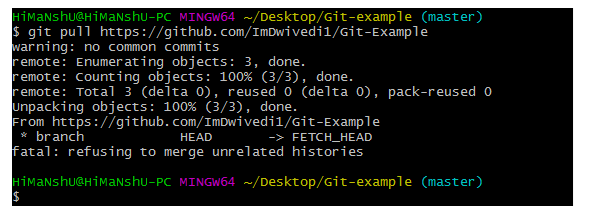


### **8) Git pull command**

Pull command is used to receive data from GitHub. It fetches and merges changes on the remote server to your working directory.

**Syntax**

1. $ git pull URL

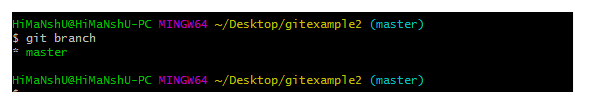


### **9) Git Branch Command**

This command lists all the branches available in the repository.

**Syntax**

1. $ git branch



### **10) Git Merge Command**

This command is used to merge the specified branch?s history into the current branch.

**Syntax**

1. $ git merge BranchName

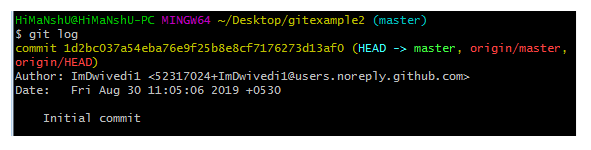


### **11) Git log Command**

This command is used to check the commit history.

**Syntax**

1. $ git log

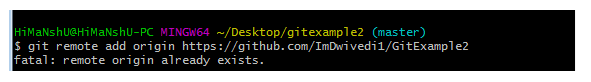


By default, if no argument passed, Git log shows the most recent commits first. We can limit the number of log entries displayed by passing a number as an option, such as -3 to show only the last three entries.

1. $ git log -3

### **12) Git remote Command**

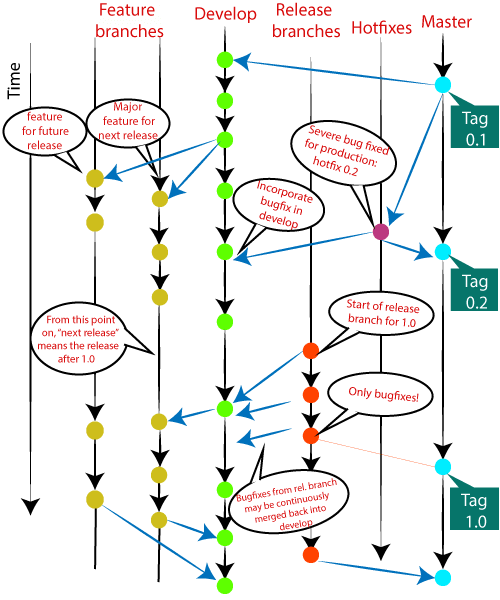
Git Remote command is used to connect your local repository to the remote server. This command allows you to create, view, and delete connections to other repositories. These connections are more like bookmarks rather than direct links into other repositories. This command doesn't provide real-time access to repositories.



# Git Flow / Git Branching Model

Git flow is the set of guidelines that developers can follow when using Git. We cannot say these guidelines as rules. These are not the rules; it is a standard for an ideal project. So that a developer would easily understand the things.

It is referred to as **Branching Model** by the developers and works as a central repository for a project. Developers work and push their work to different branches of the main repository.



There are different types of branches in a project. According to the standard branching strategy and release management, there can be following types of branches:

ADVERTISEMENT

ADVERTISEMENT

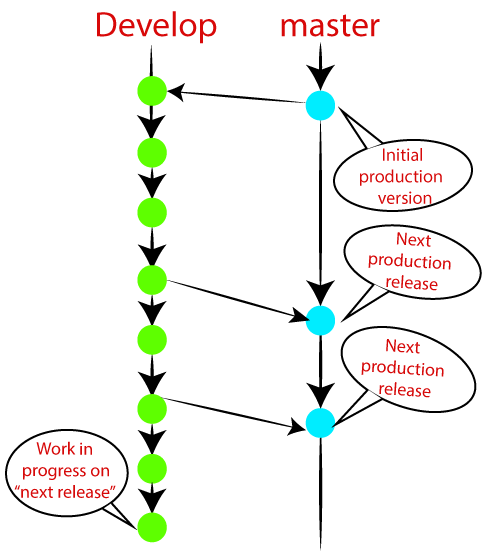
* **Master**
* **Develop**
* **Hotfixes**
* **Release branches**
* **Feature branches**

Every branch has its meaning and standard. Let's understand each branch and its usage.

## The Main Branches

Two of the branching model's branches are considered as main branches of the project. These branches are as follows:

* **master**
* **develop**



### **Master Branch**

The master branch is the main branch of the project that contains all the history of final changes. Every developer must be used to the master branch. The master branch contains the source code of HEAD that always reflects a final version of the project.

Your local repository has its master branch that always up to date with the master of a remote repository.

It is suggested not to mess with the master. If you edited the master branch of a group project, your changes would affect everyone else, and very quickly, there will be merge conflicts.

### **Develop Branch**

It is parallel to the master branch. It is also considered as the main branch of the project. This branch contains the latest delivered development changes for the next release. It has the final source code for the release. It is also called as a "**integration branch**."

When the develop branch reaches a stable point and is ready to release, it should be merged with master and tagged with a release version.

ADVERTISEMENT

## Supportive Branches

The development model needs a variety of supporting branches for the parallel development, tracking of features, assist in quick fixing and release, and other problems. These branches have a limited lifetime and are removed after the uses.

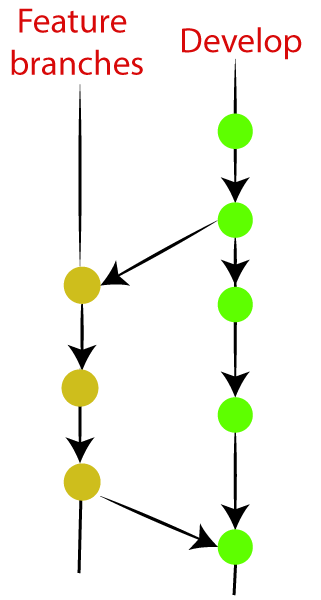
The different types of supportive branches, we may use are as follows:

* **Feature branches**
* **Release branches**
* **Hotfix branches**

Each of these branches is made for a specific purpose and have some merge targets. These branches are significant for a technical perspective.

### **Feature Branches**

Feature branches can be considered as topic branches. It is used to develop a new feature for the next version of the project. The existence of this branch is limited; it is deleted after its feature has been merged with develop branch.



To learn how to create a Feature Branch [**Visit Here**](https://www.javatpoint.com/git-branch).

### **Release Branches**

The release branch is created for the support of a new version release. Senior developers will create a release branch. The release branch will contain the predetermined amount of the feature branch. The release branch should be deployed to a staging server for testing.

Developers are allowed for minor bug fixing and preparing meta-data for a release on this branch. After all these tasks, it can be merged with the develop branch.

When all the targeted features are created, then it can be merged with the develop branch. Some usual standard of the release branch are as follows:

* Generally, senior developers will create a release branch.
* The release branch will contain the predetermined amount of the feature branch.
* The release branch should be deployed to a staging server for testing.
* Any bugs that need to be improved must be addressed at the release branch.
* The release branch must have to be merged back into developing as well as the master branch.
* After merging, the release branch with the develop branch must be tagged with a version number.

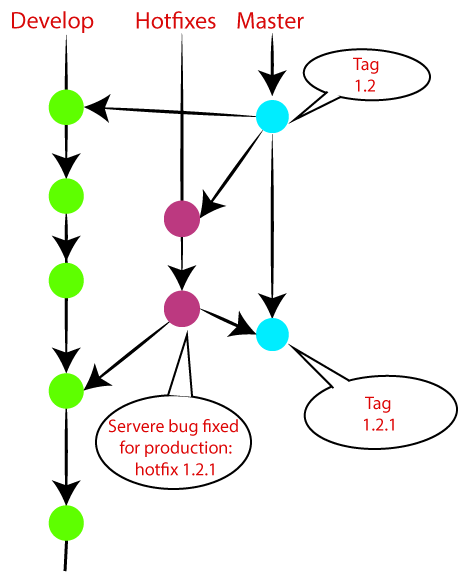
To create a release branch, visit [**Git Branching**](https://www.javatpoint.com/git-branch).

To tag branch after merging the release branch, Visit [**Git tag**](https://www.javatpoint.com/git-tags).

### **Hotfix Branches**

Hotfix branches are similar to Release branches; both are created for a new production release.

The hotfix branches arise due to immediate action on the project. In case of a critical bug in a production version, a hotfix branch may branch off in your project. After fixing the bug, this branch can be merged with the master branch with a tag.



# Git Cheat Sheet

### **1. Git configuration**

* **Git config**  
  Get and set configuration variables that control all facets of how Git looks and operates.  
  **Set the name:**  
  $ git config --global user.name "User name"  
  **Set the email:**  
  $ git config --global user.email "himanshudubey481@gmail.com"  
  **Set the default editor:**  
  $ git config --global core.editor Vim  
  **Check the setting:**  
  $ git config -list
* **Git alias**  
  **Set up an alias** for each command:  
  $ git config --global alias.co checkout  
  $ git config --global alias.br branch  
  $ git config --global alias.ci commit  
  $ git config --global alias.st status

### **2. Starting a project**

* **Git init**  
  **Create a local repository:**  
  $ git init
* **Git clone**  
  **Make a local copy** of the server repository.  
  $ git clone

### **3. Local changes**

* **Git add**  
  **Add a file** to staging (Index) area:  
  $ git add Filename  
  **Add all files** of a repo to staging (Index) area:  
  $ git add\*
* **Git commit**  
  **Record** or snapshots the file permanently in the version history **with a message**.  
  $ git commit -m " Commit Message"

### **4. Track changes**

* **Git diff**  
  Track the changes that have not been staged: $ git diff  
  Track the changes that have staged but not committed:  
  $ git diff --staged  
  Track the changes after committing a file:  
  $ git diff HEAD  
  Track the changes between two commits:  
  $ git diff Git Diff Branches:  
  $ git diff < branch 2>
* **Git status**  
  Display the state of the working directory and the staging area.  
  $ git status
* **Git show Shows objects:**  
  $ git show

### **5. Commit History**

* **Git log**  
  Display the most recent commits and the status of the head:  
  $ git log  
  Display the output as one commit per line:  
  $ git log -oneline  
  Displays the files that have been modified:  
  $ git log -stat  
  Display the modified files with location:  
  $ git log -p
* **Git blame**  
  Display the modification on each line of a file:  
  $ git blame <file name>

### **6. Ignoring files**

* **.gitignore**  
  Specify intentionally untracked files that Git should ignore. Create .gitignore:  
  $ touch .gitignore List the ignored files:  
  $ git ls-files -i --exclude-standard

### **7. Branching**

* **Git branch Create branch:**  
  $ git branch List Branch:  
  $ git branch --list Delete a Branch:  
  $ git branch -d Delete a remote Branch:  
  $ git push origin -delete Rename Branch:  
  $ git branch -m
* **Git checkout**  
  Switch between branches in a repository.  
  Switch to a particular branch:  
  $ git checkout  
  Create a new branch and switch to it:  
  $ git checkout -b Checkout a Remote branch:  
  $ git checkout
* **Git stash**  
  Switch branches without committing the current branch. Stash current work:  
  $ git stash  
  Saving stashes with a message:  
  $ git stash save ""  
  Check the stored stashes:  
  $ git stash list  
  Re-apply the changes that you just stashed:  
  $ git stash apply  
  Track the stashes and their changes:  
  $ git stash show  
  Re-apply the previous commits:  
  $ git stash pop  
  Delete a most recent stash from the queue:  
  $ git stash drop  
  Delete all the available stashes at once:  
  $ git stash clear  
  Stash work on a separate branch:  
  $ git stash branch
* **Git cherry pic**  
  Apply the changes introduced by some existing commit:  
  $ git cherry-pick

### **8. Merging**

* **Git merge**  
  Merge the branches:  
  $ git merge  
  Merge the specified commit to currently active branch:  
  $ git merge
* **Git rebase**  
  Apply a sequence of commits from distinct branches into a final commit.  
  $ git rebase  
  Continue the rebasing process:  
  $ git rebase -continue Abort the rebasing process:  
  $ git rebase --skip
* **Git interactive rebase**  
  Allow various operations like edit, rewrite, reorder, and more on existing commits.  
  $ git rebase -i

### **9. Remote**

* **Git remote**  
  Check the configuration of the remote server:  
  $ git remote -v  
  Add a remote for the repository:  
  $ git remote add Fetch the data from the remote server:  
  $ git fetch  
  Remove a remote connection from the repository:  
  $ git remote rm  
  Rename remote server:  
  $ git remote rename  
  Show additional information about a particular remote:  
  $ git remote show  
  Change remote:  
  $ git remote set-url
* **Git origin master**  
  Push data to the remote server:  
  $ git push origin master Pull data from remote server:  
  $ git pull origin master

### **10. Pushing Updates**

* **Git push**  
  Transfer the commits from your local repository to a remote server. Push data to the remote server:  
  $ git push origin master Force push data:  
  $ git push -f  
  Delete a remote branch by push command:  
  $ git push origin -delete edited

### **11. Pulling updates**

* **Git pull**  
  Pull the data from the server:  
  $ git pull origin master  
  Pull a remote branch:  
  $ git pull
* **Git fetch**  
  Download branches and tags from one or more repositories. Fetch the remote repository:  
  $ git fetch< repository Url> Fetch a specific branch:  
  $ git fetch  
  Fetch all the branches simultaneously:  
  $ git fetch -all  
  Synchronize the local repository:  
  $ git fetch origin

### **12. Undo changes**

* **Git revert**  
  Undo the changes:  
  $ git revert  
  Revert a particular commit:  
  $ git revert
* **Git reset**  
  Reset the changes:  
  $ git reset -hard  
  $ git reset -soft:  
  $ git reset --mixed

### **13. Removing files**

* **Git rm**  
  Remove the files from the working tree and from the index:  
  $ git rm <file Name>  
  Remove files from the Git But keep the files in your local repository:  
  $ git rm --cached

# 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.

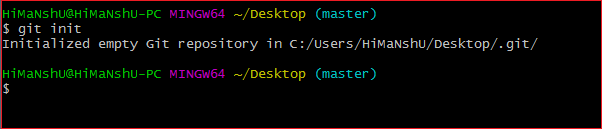
We can create a repository for blank and existing projects. Let's understand how to create a repository.

### **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:

1. $ git init

The above command will create an empty .git repository. Suppose we want to make a git repository on our desktop. To do so, open Git Bash on the desktop and run the above command. Consider the below output:



The above command will initialize a **.git** repository on the desktop. Now we can create and add files on this repository for version control.

To create a file, run the cat or touch command as follows:

1. $ touch <file Name>

To add files to the repository, run the git add command as follows:

1. $ git add <file name>

Learn more about git add command visit Git Add.

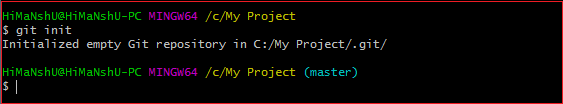
### **Create a Repository for an existing project**

If you want to share your project on a version control system and control it with Git, then, browse your project's directory and start the git command line (Git Bash for Windows) here. To initialize a new repository, run the below command:

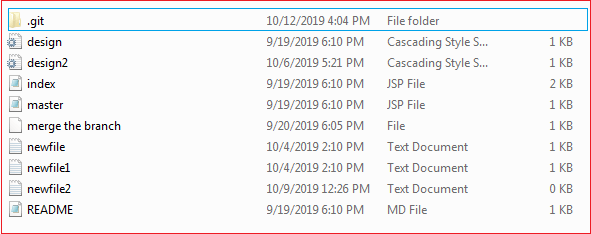
**Syntax:**

1. $ git init

**Output:**



The above command will create a new subdirectory named **.git** that holds all necessary repository files. The .git subdirectory can be understood as a Git repository skeleton. Consider the below image:

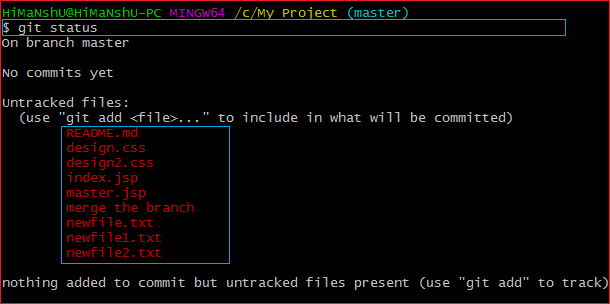


An empty repository .git is added to my existing project. If we want to start version-controlling for existing files, we have to track these files with git add command, followed by a commit.

We can list all the untracked files by git status command.

1. $ git status

Consider the below output:



In the above output, the list of all untracked files is displayed by the git status command. To learn more about status command, visit [**Git Status**](https://www.javatpoint.com/git-status).

We can track all the untracked files by Git Add command.

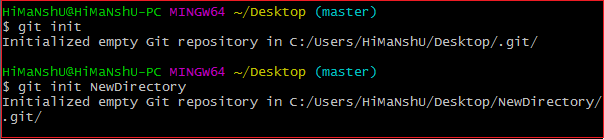
### **Create a Repository and Directory Together**

The git init command allows us to create a new blank repository and a directory together. The empty repository .git is created under the directory. Suppose I want to create a blank repository with a project name, then we can do so by the git init command. Consider the below command:

ADVERTISEMENT

1. $ git init NewDirectory

The above command will create an empty .git repository under a directory named **NewDirectory**. Consider the below output:



In the above output, the directory and the repository both are created.

Hence we can create a repository using git init command. Two other commands are handy to start with git. They are [**Git Add**](https://www.javatpoint.com/git-add), and [**Git commit**](https://www.javatpoint.com/git-commit).

Also, see various operations on the repository, see [**Git Repository**](https://www.javatpoint.com/git-repository).

Git Add

The git add command is used to add file contents to the [Index (Staging Area)](https://www.javatpoint.com/git-index).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.

The git add command does not add the [.gitignore](https://www.javatpoint.com/git-ignore) file by default. In fact, we can ignore the files by this command.

Let's understand how to add files on Git?

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:

1. $ 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:

1. $ touch newfile.txt

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

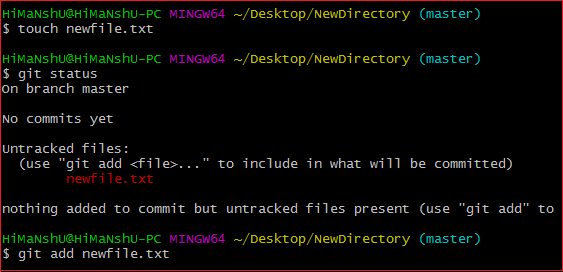
ADVERTISEMENT

1. $ git status

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

1. $ git add newfile.txt

Consider the below output:



From the above output, we can see **newfile.txt** has been added to our repository. Now, we have to commit it to share on Git.

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:

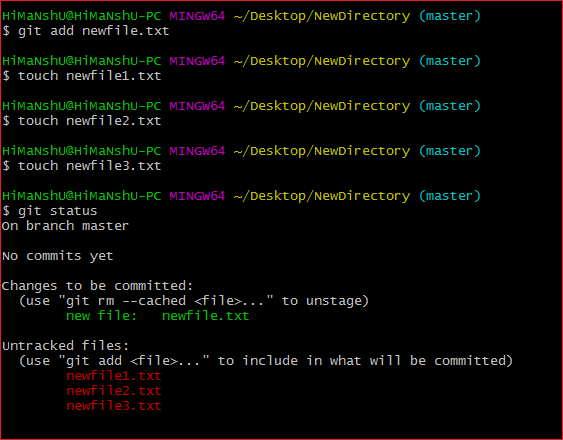
1. $ git add -A

Or

1. $ git add .

The above command will add all the files available in the repository. Consider the below scenario:

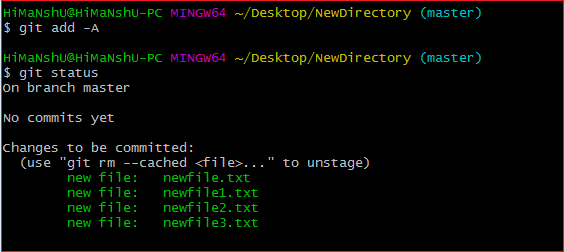
We can either create four new files, or we can copy it, and then we add all these files at once. Consider the below output:



In the above output, all the files are displaying as untracked files by Git. To track all of these files at once, run the below command:

1. $ git add -A

The above command will add all the files to the staging area. Remember, the **-A** option is case sensitive. Consider the below output:

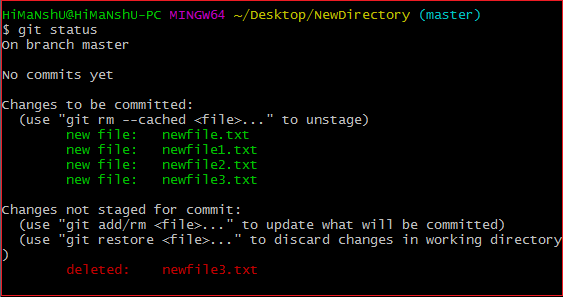


In the above output, all the files have been added. The status of all files is displaying as staged.

Removing Files from the Staging Area

The git add command is also used to remove files from the staging area. If we delete a file from the repository, then it is available to our repository as an untracked file. The add command is used to remove it from the staging area. It sounds strange, but Git can do it. Consider the below scenario:

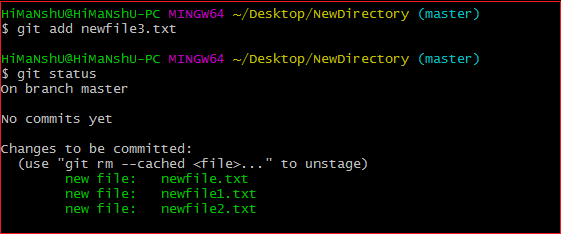
We have deleted the **newfile3.txt** from the repository. The status of the repository after deleting the file is as follows:



As we can see from the above output, the deleted file is still available in the staging area. To remove it from the index, run the below command as follows:

1. $ git add newfile3.txt

Consider the below output:



From the above output, we can see that the file is removed from the staging area.

Add all New and Updated Files Only:

Git allows us to stage only updated and newly created files at once. We will use the ignore removal option to do so. It will be used as follows:

1. $ git add --ignore-removal .

ADVERTISEMENT

Add all Modified and Deleted Files

Git add facilitates us with a variety of options. There is another option that is available in Git, which allows us to stage only the modified and deleted files. It will not stage the newly created file. To stage all modified and deleted files only, run the below command:

1. $ git add -u

Add Files by Wildcard

Git allows us to add all the same pattern files at once. It is another way to add multiple files together. Suppose I want to add all java files or text files, then we can use pattern .java or .txt. To do so, we will run the command as follows:

1. $ git add \*.java

The above command will stage all the Java files. The same pattern will be applied for the text files.

The next step after adding files is committing to share it on Git.

Git Undo Add

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:

1. $ git reset <filename>

To learn more about git reset command, visit [Git Reset](https://www.javatpoint.com/git-reset).