# **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 "AAA"

**Setting email id**

The Git uses this email id for each commit.

1. $ git config --global user.email  "aaa@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.

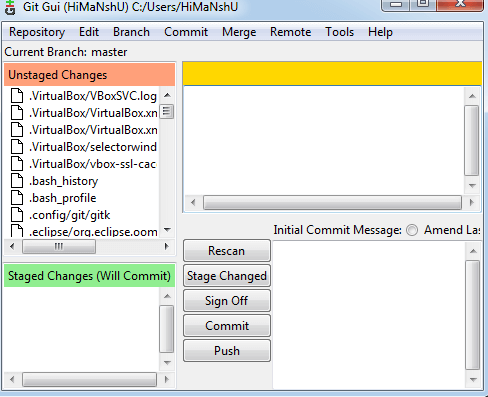
### **Git GUI**

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

Git Tools

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



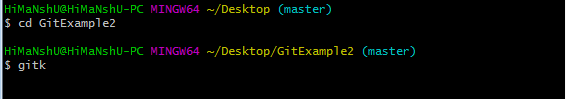
Git facilitates with some built-in GUI tools for committing (git-gui) and browsing (gitk), but there are many third-party tools for users looking for platform-specific experience.

### **Gitk**

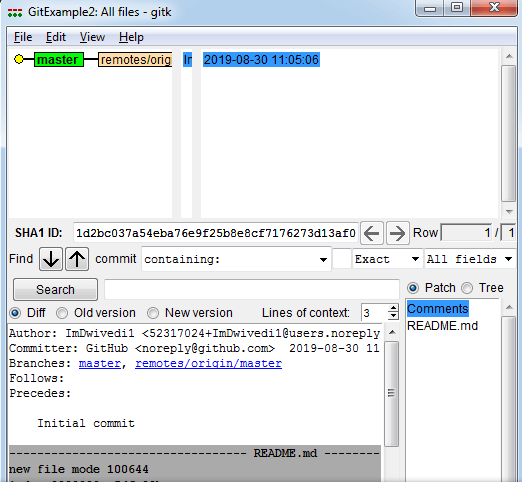
gitk is a graphical history viewer tool. It's a robust GUI shell over **git log** and **git grep**. This tool is used to find something that happened in the past or visualize your project's history.

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**

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**

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.

43.2M

848

How to find Nth Highest Salary in SQL

### **Cherry-Picking**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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.