Git, GitHub & Open Source

*Gunj Joshi*

# Git

# What Is Git?

Git is an open-source project that was started in 2005 and grew to be one of the most popular version control systems on the market. It is a distributed version control system. Meaning, any developer on the team with granted access can manage the source code and its history of changes using Git command line tools.

Git offers feature branches. This means every software engineer in the team can split off a feature branch that provides an isolated local repository for making changes to the code.

# Important Git terms to know

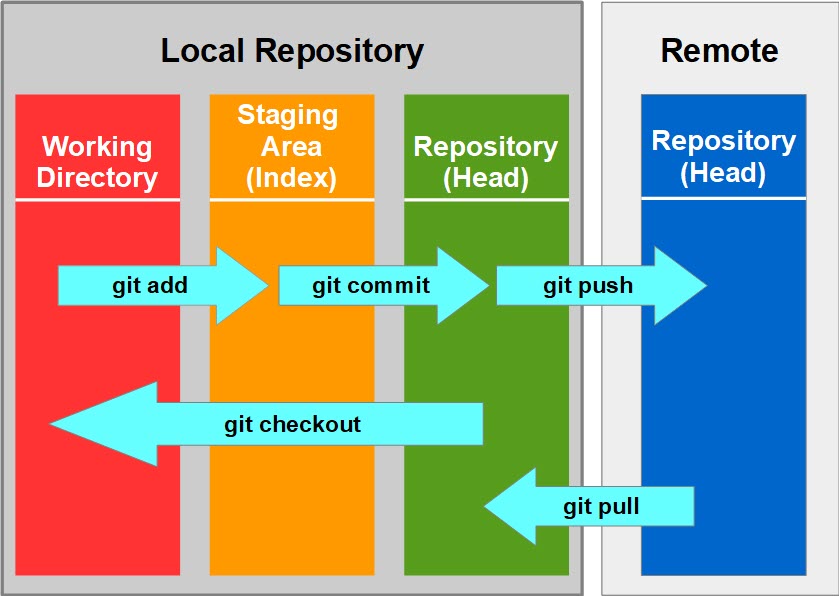
Here are some of the most-used Git terms you will have to know to understand the basics of Git workflow:

* **Branch** – Git has the **main** branch as its default branch. Creating a new Git branch means creating a new independent line of development from the main branch, such as developing a new feature.
* **Check out** – Select which line of development you want to apply your changes. Checking out a branch will make Git record new commits on that branch.
* **Clone** – Create a local copy of an existing repository.
* **Commit** – Save the changes you’ve made. The commit command also refers to the snapshot of your repo at the time of the commit.
* **Git repository** – Also known as **repo**, is a **.git** folder inside a project, where all the commits are located. It tracks and saves all changes made to the Git project files. Note that the files in the repository are the ones that Git recognizes.
* **HEAD** – A pointer that refers to the last commit command in the currently checked out branch. Think of it as the current branch.
* **Local repository** – A copy of a repository stored in your computer.
* **Pull** – Copy the changes from a remote repository to a local machine.
* **Push** – copy the changes from a local machine to a remote repository.
* **Staging area** – This is where users organize files for the next commit. It informs Git what will change between the current commit and the next one.
* **URL** – The location of the Git repository.
* **Working directory** – Also called a **working area** or a **working tree**, it refers to testing out your modifications before committing them to the staging area. It’s located in your local directory, and is the untracked area of Git.

# Workflow in Git

There are three stages in the basic workflow of Git:

1. Start by creating a feature branch from the main branch in the working directory and developing your features there. Move back to this stage whenever you make any changes to your files, such as adding, deleting, or renaming them.
2. Once you’re done, add the files to the staging area by merging the feature branch with the staging branch. Doing so helps review and organize the files you want to commit, making the project version easy to understand.
3. Once the development has reached a milestone, commit the local changes to the main Git repository permanently by merging the staging branch with the main branch. It stores all the snapshots of the project versions.



Note that Git doesn’t track every modified file as it only includes the file in your commit if it is present in the staging area.

If you need to commit two different modified files, add the first file to the staging area, then perform the commit command. Once you’ve done with the first commit, repeat these steps to do the second commit.

# Importance of Using Git

* **Open-source** – It’s free to use. You can download its source code and modify it to suit your needs. Furthermore, many resources are available online to learn the best Git practices.
* **Fast and lightweight** – The core part of Git and most of its operations are written and performed locally, making it faster than the centralized version control system.
* **Moderate hardware power** – Git doesn’t require powerful hardware, as team members only need to interact with the server when they push or pull changes to the main repo. It’s also beneficial for growing teams, since Git doesn’t prompt further hardware requirements no matter how many people use it.
* **Secure environment** –It checks every file and commit during the checkout operation, so it’s not possible to alter data in the database without Git storing the changes.
* **Safer backup** – Git creates multiple copies of your data as it mirrors the repository to all clients, allowing more backups. Furthermore, Git can take snapshots, which are the representations of a file system, enabling you to roll back to the state when the snapshot was taken. This can be a helpful recovery solution in case of failure.
* **Easier branching** – Creating, deleting, and merging branches takes only a few seconds, making it more time-efficient and less complicated than a centralized version control system.

##### You can install Git from the URL : <https://git-scm.com/downloads>

# How to Use Git Commands

There are various ways to run commands in Git, from using command line tools like the terminal in macOS or the command prompt in Windows to graphical user interfaces (GUI).

This section will cover various Git commands you need to know to make the most of Git using the command line tool.

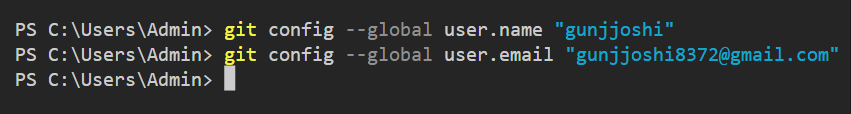
### *git config*

After you have installed Git, you need to configure it, i.e., letting Git know about yourself.

You can configure Git by using the following command

git config --global user.name “<your username>”

git config --global user.email “<your email>



### ***git init***

The first stage of working in Git is developing your project in a working directory. This working tree is located in your local Git repository – the main repo’s copy that exists in your computer.

The first step in creating a new repository is running the **git init** command. Doing so will create an empty Git repository. It sets up a new directory in a **.git** folder in your working tree, which contains the necessary Git metadata.

Navigate to your project folder, and enter the following command into your command line tool:

git init

You can also specify the directory you want to initialize a new repository. Use this syntax for doing so:

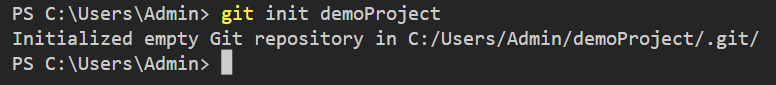
git init <folder>

For example, for a new repository in the **demo-project** folder, the syntax should look like this:

git init demo-project

This command will create a **.git** folder in the **demo-project** folder instead of your current working tree.

git init <folder>



### ***git add***

The **git add** command adds new or modified files from the working directory to the staging area.

Git doesn’t have a simple save feature. Instead, it uses a process called committing. The first step in committing a change to a Git repository is running the **git add** command to select the files you want to commit.

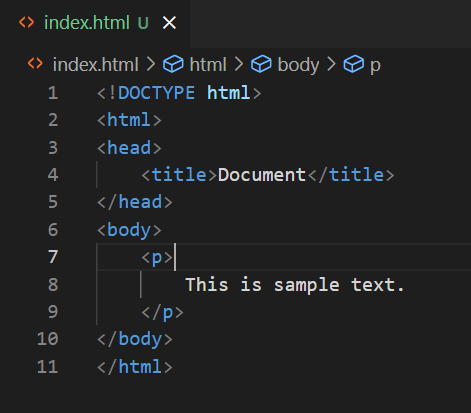
This command will move those selected files to the staging area. From there, you can use another Git command – **git commit** – to record the local changes made to those files into a Git repo.

There are several ways to use **git add**, including:

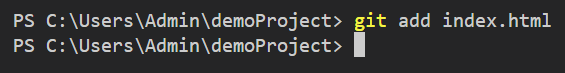
* **git add \*** – add all unstaged files in a repository.
* **git add \*.txt** – add all **.txt** files in the repository.
* **git add <file name>**– add a specific file.
* **git add <path>** – add a specific directory of files.

Here’s an example of applying **git add** command:

git add index.html



(index.html)

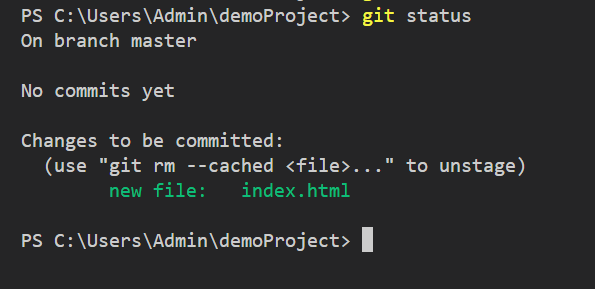


This command line will add an **index.html** file to the staging area.

### ***git status***

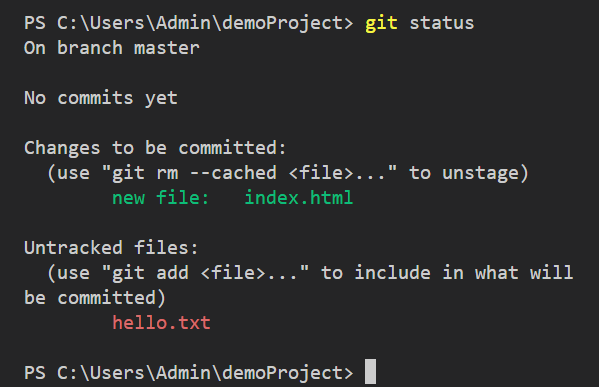
The **git status** command shows the current state of the working directory and the staging area.

This command will check which local changes have been staged or committed and which ones haven’t. If there are no new local changes, it will show a message that indicates there’s nothing to commit and that the working tree is clean.



It’s also helpful for new Git users, as the message provides hints on what to do under specific circumstances.

Suppose there are some untracked files that haven’t been staged or committed in your local repository. In response, this command will show the list of the untracked files, and you can follow that up by moving them to the staging area.



Furthermore, this command can also show if you have a merge conflict and which files are causing the problem.

### ***git commit***

The **git commit** command creates a snapshot of your repository’s content at a specific time.

It records the changes made to the files in your repository. On top of that, commits serve as the comprehensive project record, showing you how it has evolved.Each commit has a unique commit ID for easier reference. A commit ID is also helpful for referring to a specific commit when you need to undo a change.

This command is the second step in saving the changes made to a repository. The process starts with the **git add** command for staging any new changes you want to include in a commit. Then, **git commit** creates a commit with those changes to a repository. The **git add** command won’t affect anything until you execute **git commit**.

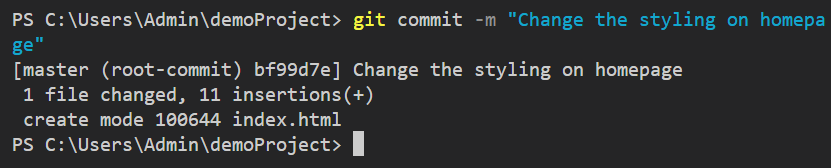
Once the code in your local branch is ready, use **git push** to commit to the remote repository. This enables various team members to commit their code in their local repo and push it to the main repo later when the code is fully ready.

There are various options of the **git commit** usage, with the most common options as follows:

* **git commit** – the primary command to commit the staged changes. It will launch a text editor that prompts you to submit a commit message that briefly describes the change you’ve made.
* **git commit -m “message”** – creates a commit with an inline message. Use any message, but make sure it is concise and understandable for other team members.
* **git commit -a** – creates a commit that includes all changes in a working directory. This command will only include modifications to tracked files, that have been added using the **git add** command.
* **git commit -am “message”** – combines the function of the **-a** and **-m** options, which will create a commit of all staged changes and includes an inline comment with it.
* **git commit –amend**– allows modification to the last commit, such as editing the last commit message without changing the snapshot.

For example, you have just changed the styling of a homepage and moved the **css/styles.css** file to the staging area. Now you can continue to perform **git commit** and put a commit message there to help anyone understand the change.

git commit -m "Change the styling on homepage"



### ***git log***

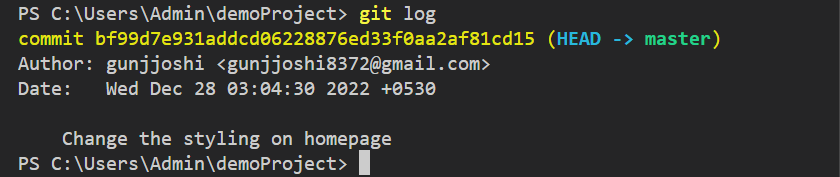
The **git log** command shows the commit history of a repository. By default, its output includes commit IDs, messages, and other metadata. It helps navigate your growing project history.

The difference between **git log** and **git status** is that **git status** applies to the current directory, while **git log** applies to the repository history.

Suppose you want to see a list of all or new commits that have been pushed to your team’s repository. Use the following command to do so:

git log

Git will return a list similar to the example below:



This output shows the latest log information in the repository. Here’s the breakdown of this example:

* **commit ID** – a random string of numbers and letters.
* **Author**– in this case, it’s “gunjjoshi”. It also shows the author’s email.
* **Date**– specifies the exact date and time.
* **Commit message** – in this example, it’s “Change the styling on homepage”.

# Working With Git Branches

One of the essential features in a Git version control system is branching. Git branches let you move away from the main codebase and make changes to it without affecting the original main code or other branches. In other words, it allows developers to work on independent development lines in parallel.

For example, in a website project, you can work on a new feature, while another developer is fixing a bug. Both of you will create a new branch and start working on those respective branches.

If you need to make an immediate change to the website, you can switch to the main branch, make the change, and push it live. After that, you can switch back to work on the new feature branch and merge it to the main branch once you’re done.

If another developer finishes his task at a different time, this will not affect someone else. All those changes will be safely kept in each branch and will only affect the main codebase when pushed to the main branch.



Branches also help reduce the possibility of unstable code being merged into the main branch since developers will only push their code into the main branch once they’re ready.

You can also ask developers to create a pull request first, which is a proposal for pushing a change to the main branch. Making a pull request lets other developers review that code first, ensuring it’s ready to be merged with the main branch.

### ***git branch***

The **git branch** command is used to create, rename, and delete branches. It can also retrieve a list of local and remote branches.

This command is commonly used together with **git checkout** and **git merge** to switch branches and merge different code versions.

Let’s start with creating a branch. In this example, we will create a new branch called **v2.0**. Use this syntax to do so:

git branch v2.0

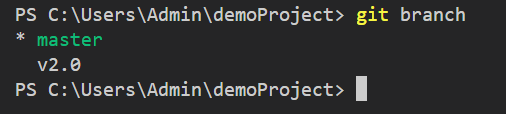
Screenshot 2022-12-28 031408.png

Once you run this command, your development environment will remain attached to the HEAD of the repository. You’ll need to use the **git checkout** command if you’d like to see this new branch.

Note that the repository history remains the same when you create a new branch, as Git only creates a pointer to that new branch.

If you want to retrieve a list of branches in a Git repository, run this syntax:

git branch

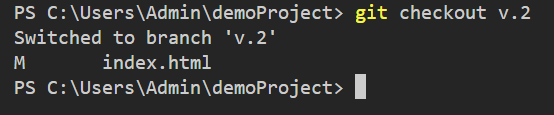


### ***git checkout***

To start modifying a branch, you need to switch to it. Here’s where you’ll use the **git checkout** command to switch between different branches.

Once you check out a branch, your working directory will be changed. For example, you’re on **branch v.1**, and you want to move to **branch v.2**. To do so, apply this command:

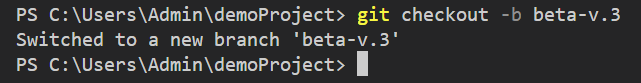
git checkout v.2



Now your current directory is branch v.2.

The **git checkout** command can also create a new branch. Usually, you’ll need to run the **git branch** command first to create a new branch, then run the **git checkout** command for checking it out. However, the command shown below provides a more convenient method:

git checkout -b beta-v.3



This command simultaneously creates a new branch named **beta-v.3** and checks it out, allowing you to switch to it immediately.

### ***git merge***

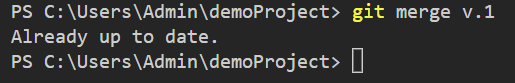
The **git merge** command combines an independent line of development with another, integrating them into a single branch.

For example, when a developer merges the changes made in a staging branch with the main branch, this command will put back together a forked repository history into a unified project history.

Note that when you’re merging a target branch to the main branch, only the main branch will be updated to reflect the merge, while the target branch will remain unchanged.

To execute this command, ensure you’re in the main branch. If you’re not there yet, move to your main branch by running the **git checkout** command. If the branch that you want to merge is **v.1**, the merge command should look like this:

git merge v.1



If the two branches you’re merging both changed the same section of the same file, Git won’t be able to determine which version to use, therefore triggering a merge conflict.

### ***git push***

The **git push** command transfers the commits in a local repository to a remote repository.

git push <remote name> <branch name>

For example, if the remote repository name is **origin** and the branch name is **main**, the syntax should look like this:

git push origin main

Screenshot 2022-12-28 033723.png

This command will send local commits to the remote repository’s main branch.

### ***git pull***

The **git pull** command retrieves and downloads the content of a remote repository to your local machine. Here’s the syntax to do so:

git pull <remote repository name>

For example, to retrieve the content of the **main** branch, use this command:

git pull main

Screenshot 2022-12-28 033958.png

### ***git clone***

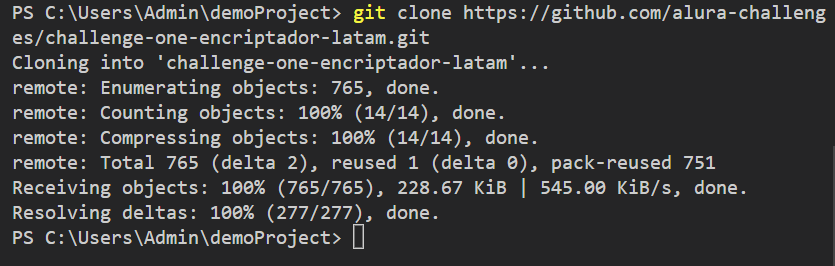
The **git clone** command creates a local copy of a remote repository. Developers use this command to obtain a copy of an existing repository to develop it in their local machine without affecting the main codebase.

It’s also one of the two ways in Git to set up a repository – by using the **git init** command to create a new Git repository or using the **git clone** command to clone an existing one.

Same as the **git init** command, the **git clone** command is typically only executed once. With cloning, you’ll have all the metadata, Git branches, and the complete repository history.

Use this command to clone a remote repository:

git clone <remote URL>



# Conclusion

Git helps you manage your projects and easily monitor project history. It enables you to create a new project, copy an existing repository, create a new feature, and work on it together with your team. On top of that, it provides a lightweight, secure working environment for your development project.

However, it can be daunting for beginners to learn Git, as its operation involves a lot of commands. It also takes some time to understand Git’s multiple development areas – from the local repository to the remote one – but once you get a better grasp of it, you’ll be able to navigate your development project way more effectively.

## GitHub

# What Is GitHub and Why Is It So Popular?

GitHub hosts a large number of repositories, the majority of which are open-source projects. GitHub is among the most popular Git GUI (Graphical User Interface) clients and is used by various professionals and large businesses.

All GitHub users can track and manage changes being made to the source code in real-time while having access to all of the other Git functions available to them at the same place.

What’s more, GitHub’s user interface is more user-friendly than Git, making it accessible to people who possess little-to-no technical knowledge. This means that more team members can be included in the progress and management of a project, making the development process smoother. GitHub’s interface is user-friendly enough so even novice coders can take advantage of Git. Without GitHub, using Git generally requires a bit more technical savvy and use of the command line.



|  |  |
| --- | --- |
| **Git** | **GitHub** |
| Git is a version control system. | GitHub is a Git repository hosting service. |
| Git is installed on our local machine and we don't need internet access to use Git. | GitHub is completely cloud-based and an internet connection is needed to use GitHub. |
| Git is maintained by the Linux Foundation. | GitHub is maintained by Microsoft. |
| Git is software and used as a command-line tool. | GitHub provides a service and provides a Graphical Interface to its users. |
| Git is completely free to use. | GitHub provides free as well as paid plans. |

## Open Source

# What do you mean by open-source contribution?

Open-Source Software is a type of software whose code is publicly available to use and modify. Open-Source Contribution involves contributing to the development or improvement of open-source software.

Some popular Open-Source Software are the Linux Operating System, Android, Mozilla Firefox, Chromium (which powers Google Chrome and Microsoft Edge), VSCode IDE, VLC Media Player, WordPress Content Manager System, etc.

# Why should I contribute to open-source?

Ever wonder if all these projects/software were single-handedly maintained by a single developer?  
What would have been its impact on them and pretty much on us?

The software would not have had so many features and upgradations. This is where open source contribution comes in. Contributors from around the world help develop and improve the software for every one of us who use it. Being a contributor will give you the super-power to be a part of something that is impacting so many lives.

Apart from the impact that you get to create, it also helps you become a better developer and with time a good mentor, leader, and passionate team player.

# Start contributing to Open-Source actively

* Find projects or organizations that you are interested in contributing to.
* Go to their GitHub repository, read the documentation, and search for first-timer issues as mentioned above.
* Try to work on as many issues as you can either across projects or for a single project.
* Join their IRC channel (Gitter/Discord/Slack, etc.). Introduce yourself and ask for help when stuck. You can find the link to the channels on their GitHub pages.
* You can also create issues after running the application locally.
* Once you are comfortable with contributing to open-source, start participating in open source programs.

# Open-Source Programs/Contests you can participate in

There are many open-source coding programs that you can participate in.

* Google Summer of Code (GSoC)

GSoC is the Olympics of Open Source. It is a global program focused on encouraging more student developers to do open source software development.  
Students work with one of the selected open-source organizations for 3 months and get a handsome stipend on completing the project. Students need to propose changes that they want to work on to get selected.  
It is a good idea to start contributing to your favorite orgs/project much before GSoC.

* HacktoberFest

HacktoberFest is a month-long celebration of open source software carried out in October. You can sign up anytime between October 1 and October 31.  
It is open to everyone in the global community!  
One needs to complete a certain amount of quality PRs to get swags in return. The swag motivates many people to get started with open-source contributions through this program.

* Outreachy

Outreachy (previously the Free and Open Source Software Outreach Program for Women) is a program that organizes three-month paid internships with free and open-source software projects.  
It is for people who are typically underrepresented in those projects.  
This is generally carried out biannual throughout the year.

* MLH Fellowship

The MLH Fellowship is an internship alternative for software engineers.  
Instead of working on a project for just one company, selected candidates contribute to Open Source projects that are used by companies around the world and are paid a competitive stipend during this tenure.