Git 101

By Abdelkader Belhadj

November 18, 2025

Overview

In this document, our goal is to build a solid understanding of Git and to develop enough mastery to use GitHub confidently for managing projects. To keep the material focused, we will assume that the reader is already familiar with a few basic ideas about how computers handle files, folders, and simple terminal commands.

As you may already know, there are many different ways to use Git. The original and most complete way is through the command line, but over the years a number of graphical interfaces have appeared, each offering its own style and level of convenience. While these GUIs can be helpful, none of them expose every Git feature. The command line, on the other hand, gives you full access to all of Git's capabilities. For this reason, our primary approach in this course will be the command-line version of Git.

We will also assume that the reader has Visual Studio Code installed, though this is not a strict requirement. You are free to use any editor you prefer. The author personally favors a terminal-based, Vim-style workflow, but such environments can feel overwhelming for beginners. For clarity and accessibility, most examples in this document will be demonstrated using VS Code.

Overview

By the end, the reader will have learned how to:

1. Install Git on all well-maintained operating systems,
2. Initialize local repositories, and clone remotes sources,
3. Track and revert changes across files,
4. Understand the roles of the working directory, staging area, and commit history,
5. Move changes between these areas, undo accidental edits, and restore files when needed,
6. Create clear, meaningful commits and revise them when appropriate,
7. Navigate through logs and branches,
8. Merge, re-base, and reset a repository,
9. Collaborate effectively through GitHub, which include:
   - Pushing commits to remote repositories,
   - Fetching, pulling, and merging updates from multiple branches,
   - Resolving conflicts when necessary,
   - Creating or reviewing pull requests to contribute to shared projects.

Outline

What is Git?

To understand what Git is, we first need to consider version control systems, which are systems that records changes to files over time, enabling us to revisit, compare, or restore previous states of a project. They keep a complete history of modifications of all tracked files, including what changed, created, removed, when, and by whom.

Traditional version control systems, despite their usefulness, have a major limitation: they rely on a central server. If this central server fails, access to the project history can be lost. To overcome this, distributed version control systems were developed. In a distributed system, every user has a complete copy of the repository, including all files, commits, branches, and history.

Unlike centralized systems, distributed systems allow users to commit changes locally and later synchronize with others through remote repositories, ensuring the project's history remains safe even if the central server becomes unavailable.

What is Git?

During the early years of Linux kernel development (1991–2002), changes were exchanged as patches and archived files. In 2002, the project adopted a proprietary DVCS called BitKeeper. By 2005, the relationship between the Linux community and the company behind BitKeeper collapsed, revoking its free-of-charge status. This event prompted Linus Torvalds, creator of Linux, to develop a new tool that incorporated lessons learned from BitKeeper, and thus Git was born.

Today, Git is the version control system most commonly used by software developers. It is the most popular DVCS, with nearly 95% of developers reporting it as their primary version control system as of 2022. It is the most widely used source-code management tool among professional developers with no competition what so ever.

Even though GitHub we will be our primarily subject, there are multiple offerings of Git repository services, including SourceForge, Bitbucket and GitLab.

Why do we need it?

We need Git because projects change constantly, and without a structured way to track those changes, it becomes easy to lose work or break something without knowing how or why. Git, by recording every modification as a commit, provides a complete history that anyone can revisit at any time to review, restore, or simply to debug. Git makes it simple to fix mistakes and compare versions. This ability makes development far more reliable and manageable.

We also need Git for collaboration. When multiple people work on the same files, chaos can happen without proper coordination. Before long, you will be faced with overwritten code and conflicting changes, and confusion about whose work to implement will only be in the way of your progress. Git, however, solves this through its many fancy tools, allowing us to work independently and then combine the work cleanly. This structured workflow ensures smooth teamwork and an efficient project management, whether you're working alone or with a large team, and this is the main reason why we need it.

How to Install Git?

Git is available on all major operating systems, including Linux, macOS, and Windows. This means that no matter what machine you use, whether it's a personal laptop, a school computer, or a workstation, you can install Git and benefit from the same version-control tools.

Although this tutorial will be somewhat Unix-oriented, the commands and core functionalities remain consistent across platforms. It is straightforward to collaborate with others, even if they are using a different operating system.

If you already have it *properly* installed, you can skip to this section 2.

MacOs

To install Git on mac, you'll first need to make sure Homebrew is installed, since it's the most common and convenient package manager on the platform. Homebrew allows you to install and manage software packages directly from the terminal. If you don't already have it, you can install Homebrew by running the official installation command from their website. Once Homebrew is set up, installing Git becomes straightforward.

Simply open your terminal and type the following command to download and install Git through Homebrew:

```
$ brew install git
```

The author would urgently like to stop all of us here to announce that the "$" symbol is *not* part of the command. In layman terms, that simply means that you should run it as a normal user, not as root.

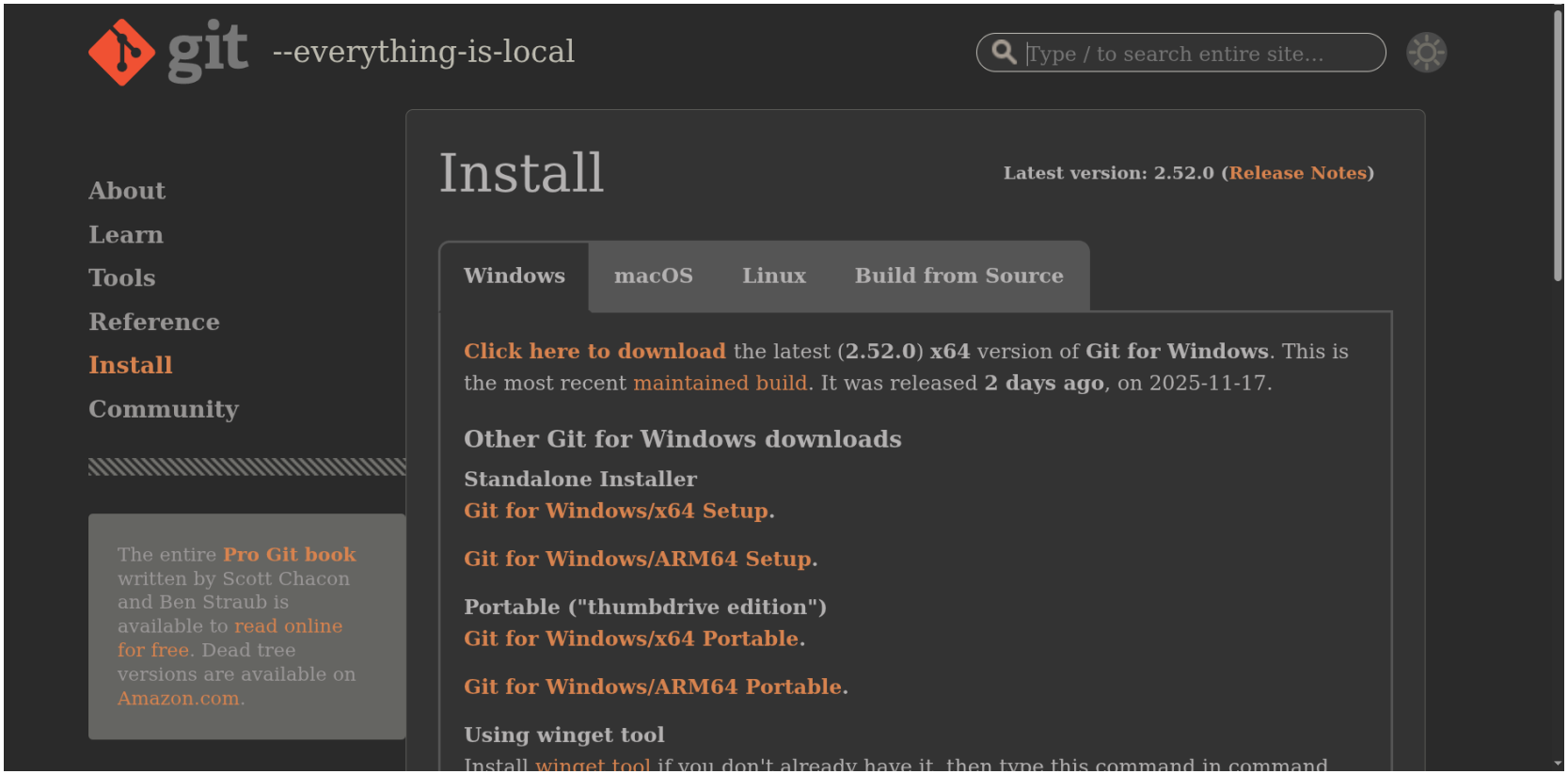
Linux

As for Linux, a package manager is almost always installed by default, since it's an essential part of how Linux systems handle software. However, the exact command you use to install Git will depend on your distribution, because each distro has its own package manager and its own repository structure. The full list can be found on the installation guide web page, but here are some of the most widely used package managers across different distributions include:

```
$ sudo pacman -Syu git              <------ For Arch-based systems
$ sudo apt install git              <------ For Debian-based systems
$ sudo emerge --ask dev-vcs/git     <------ For Gentoo
$ sudo dnf install git              <------ For Fedora
$ sudo zypper install git           <------ For openSUSE
$ sudo xbps-install -S git          <------ For Void Linux
$ sudo apk add git                  <------ For Alpine Linux
$ nix-env -iA nixpkgs.git           <------ For NixOS
```

Windows

For Windows users, the readers must visit the official Git website to download and install the latest stable version. Press “Click here to download,” and install the `.exe` file as you would for any other application.



It’s essential to remind the reader to leave the default settings as they are, unless you know what you are doing.

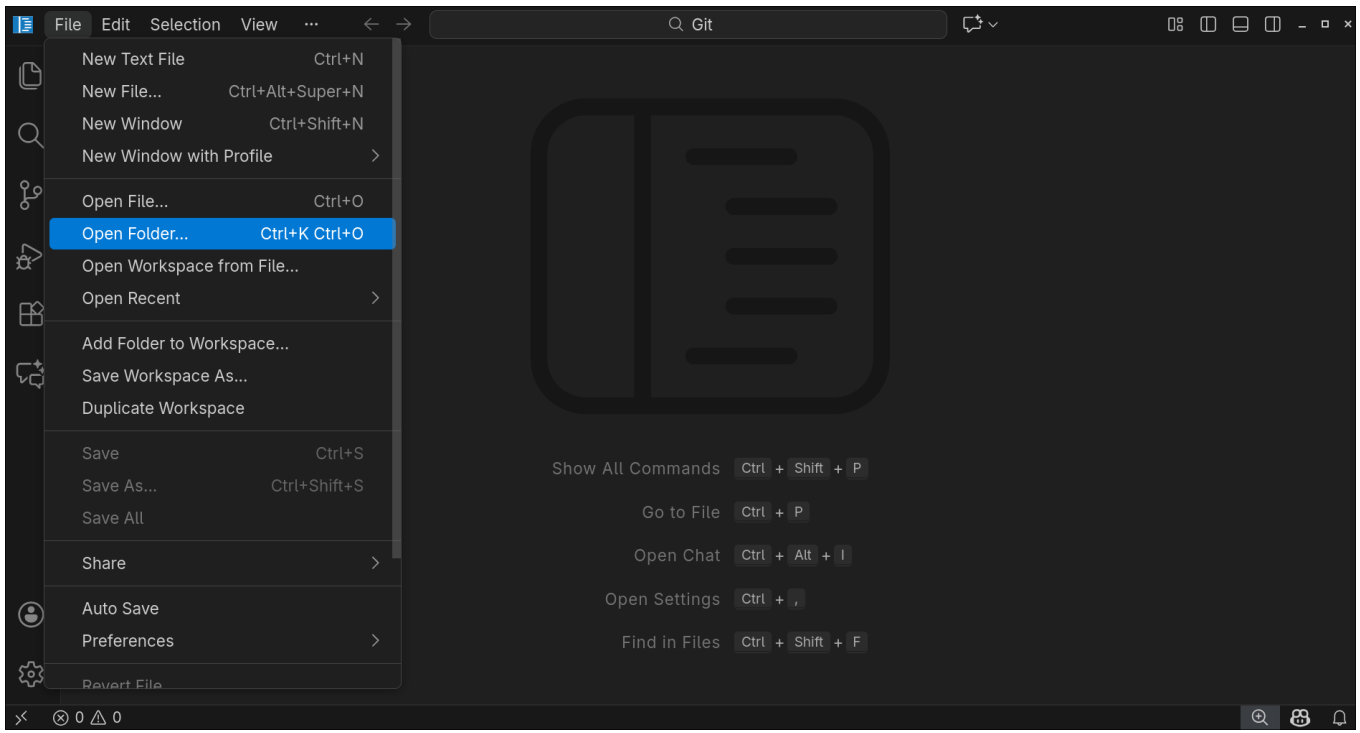
Initializing a repository

After installing Git on your system, the next step in this course is to initialize a repository. This process essentially tells Git that a particular folder (also called a directory) is something you want to track. From that point on, Git will monitor changes to files *within* that folder by record the history of those changes inside a hidden .git folder.

Initializing a repository also gives you the foundation for organizing your work. The developers can now experiment with new ideas safely, and keep a complete record of how the project evolves over time. Even if you are just starting with a small project, this step ensures that every change is accounted for and can be revisited or reversed if necessary.

Initializing a repository

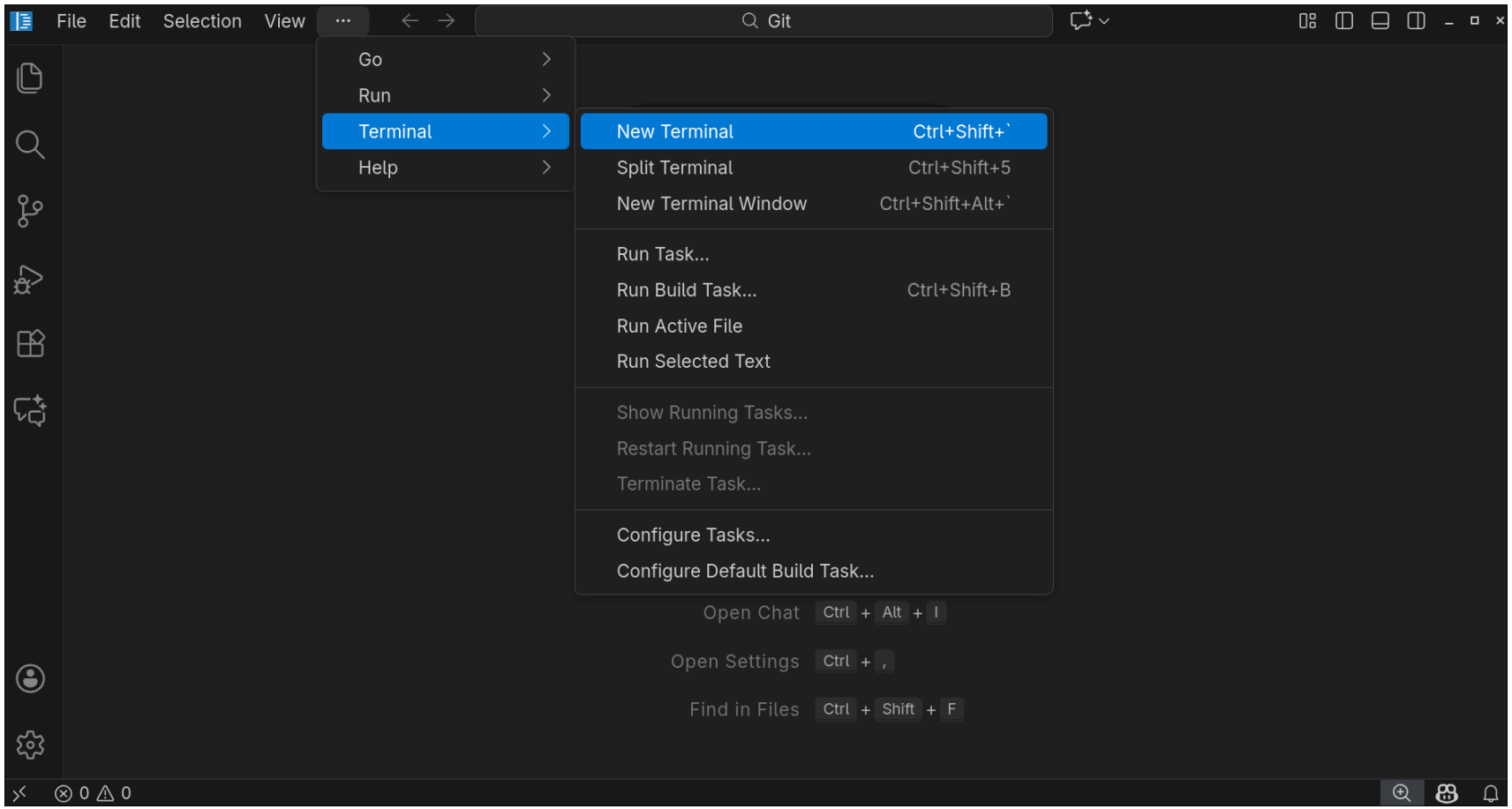
We begin fist by launching VS Code.



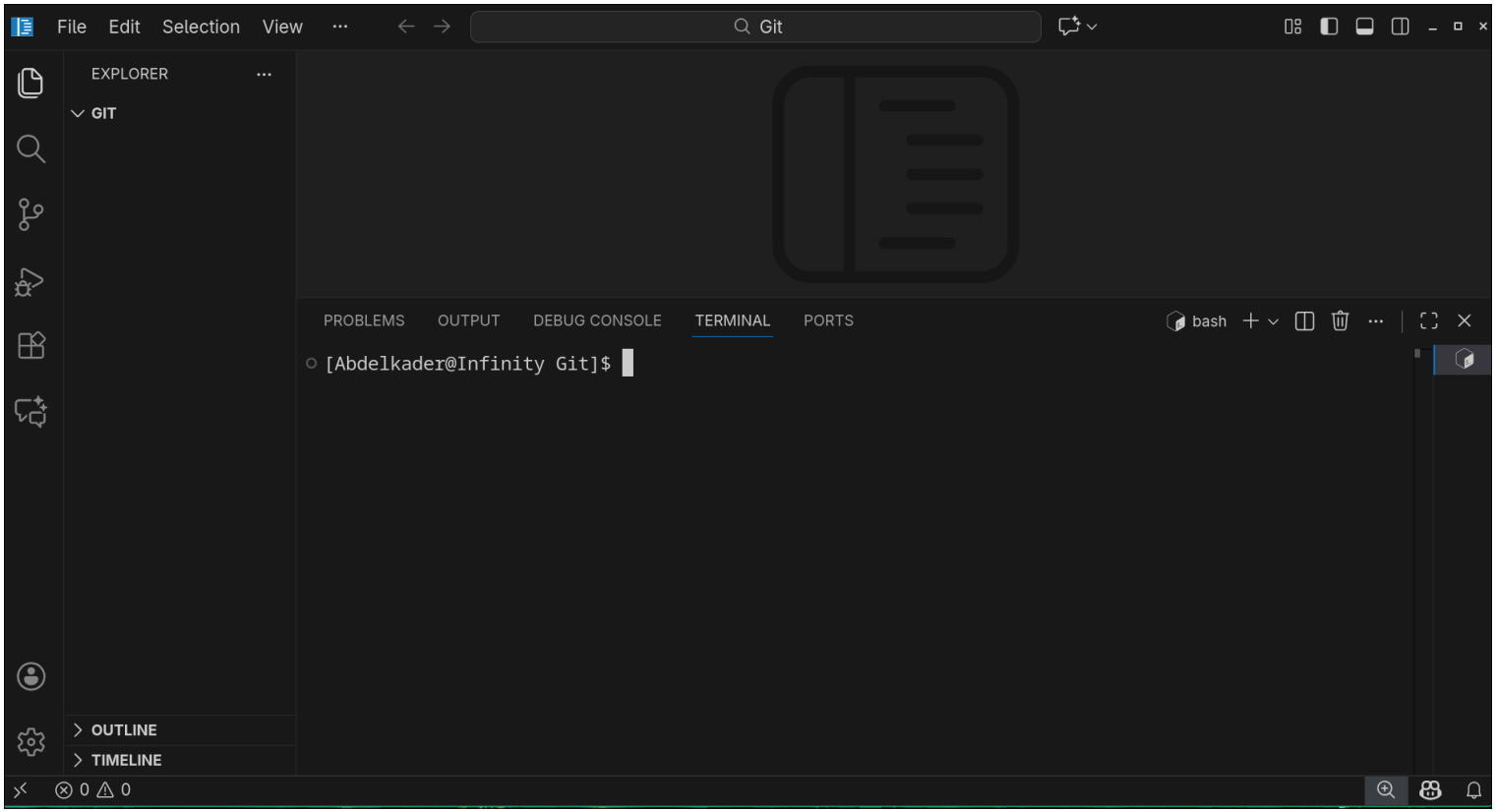
After that, click on “File”, then “Open Folder”, and select an empty folder to serve as your Git project. This folder, or directory, is what we will tell Git to track. From this point onward, we are able to ask Git to take a snapshot of our work at any point in time, to return only when there is a need to.

Initializing a repository

After choosing our project folder, we click on the three dots next to “View” and select “New terminal.” The appearing windows is where all of our git commands will be typed.



Initializing a repository



After applying those instructions, the reader must have reached a similar state. It is not necessary to find an exact one-to-one replica, as these pictures provide only a general overview and are intended purely as a visual guide, but the general out-lay must be the same. To see if your VS Code has detected Git, run "`git --version`" in the terminal.

Initializing a repository

Now, to tell Git that this folder should be tracked, type the following command in your terminal:

```
$ git init
```

You should see a message in white text stating that an empty Git repository has been initialized. This means that Git has successfully created a new repository in the folder you selected. The hidden `.git` directory inside your folder contains all the necessary files for Git to operate properly. There is no need for the reader to manually intervene. Everything inside is handled automatically by the software.

Sometimes, depending on your shell, you may also see a yellow message. It is just a warning, not an error. It informs you that Git has named the initial branch "master" by default, but in the future (Git 3.0 and later), the default branch name will be "main."

Cloning a remote repository

While ‘`git init`’ is used to start a repository from scratch on your own machine, more often than not, you will want to work on a project that already exists somewhere else. For example, you could be in a team, working an open-source library, or taking an interest in a previously developed project. In such cases, Git allows you to create a full local copy of an existing repository using a process called *cloning*.

When you clone a repository, Git automatically remembers the location you cloned it from. This original source is called a remote. While you are working locally, the remote allows you to later fetch and synchronize changes without affecting your local copy immediately.

At this stage, it is enough to understand that remotes act as a bridge between your local repository and other copies of the project. Later in the course, when we discuss remote repositories and collaboration, you will learn how to connect your cloned repository back to its source to push your changes or pull updates from others. For now, just understand that cloning is the easiest way to start working with a project that already exists.

Clone command

To clone a repository, you need its Uniform Resource Locator.

```
$ git clone <URL>
```

Cloning a repository downloads all of its files, the full history of commits, and its branches to your local machine. This gives you a complete copy of the project, and sets the stage to make changes without affecting the original repository until you are ready to share your work.

Tracking changes in Git

Once a repository is initialized — or cloned from a remote — Git can start tracking changes to any file by typing the following command in your terminal:

```
$ git add ‘File name.extension’
```

Or, if you want to track all the files in your current working directory, you can simply do this instead:

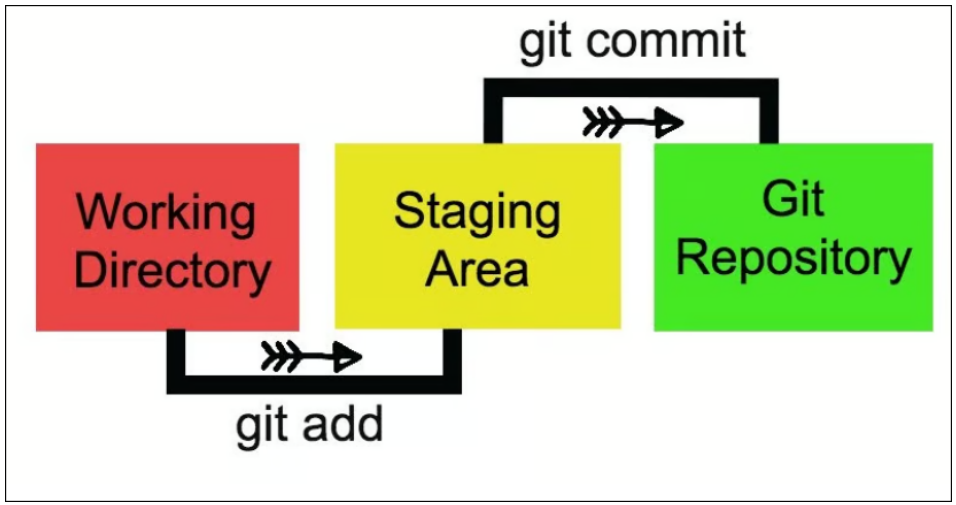
```
$ git add .
```

Now, to truly understand how this works, it’s mandatory to know the difference between two key words: the “Working Directory,” and “Staging Area.”

Tracking changes in Git

When you work on a project, the folder you see on your computer — where you open files, edit text, and save your work — is called the working directory. It always shows the current state of your project as it exists on your machine. Any time you type, delete, or modify something, those changes happen in the working directory.

Git, however, doesn't automatically save every change you make. Before a change becomes part of your project's history, it has to pass through a middle step called the staging area. When you run 'git add', you are not saving your work yet, you are simply telling Git that these are the changes we want to include in our next save.



Tracking changes in Git

Area	Description
Working Directory	• This is the folder on your computer where your project files actually live. Any edits, additions, or deletions you make happen here first. • Git notices changes in this directory but does **not** automatically save them as a snapshot yet.
Staging Area (Index)	• This is a "holding area" where you prepare changes before committing them. Running 'git add' copies the file's current state into the staging area. This ensures that the commit will contain that exact version, even if you continue editing the file afterward. • You can stage some changes but leave others out, giving you fine-grained control over what gets recorded.

Un-tracking and restoring changes in Git

Once you understand that key difference, the next step is learning how to move changes between them. Let's suppose that you've staged something by mistake. In that case, you can simply remove it from the staging area without altering the file itself in your working directory. To do that, simply type the code line below.

```
$ git restore --staged 'file name.extension'
```

This command tells Git to return the specified file to an unstaged state. Nothing inside the file is modified; you are simply removing its current changes from the list of what will be included in the next save.

There is also another command that behaves similarly, but only in a particular situation — when the file has not appeared in any previous commit.

Un-tracking and restoring changes in Git

```
$ git rm --cached ‘file name.extension’
```

This instructs Git to stop tracking the specified file altogether. The file remains in your working directory, but Git will treat it as if it does not exist. If a previous commit did contain this file, the next commit will record its removal.

There may also be situations where you modify a file in the working directory and later decide that you do not want those changes at all, neither in the staging area nor the working directory. In that case, instead of manually undoing the changes, run the following.

```
$ git restore ‘file name.extension’
```

Resetting everything to the last commit

Sometimes you may want to completely discard all changes in your project — whether they are staged, unstaged, or spread across multiple files. Instead of restoring each file one by one, you can reset the entire repository to the state of the last "save."

First, remove all files from the staging area. Then, discard all working–directory changes.

```
$ git restore --staged .
$ git restore .
```

Alternatively, both actions can be performed at once using a stronger command:

```
$ git reset --hard
```

This resets your entire project to the last committed state. Any uncommitted work will be lost, so use it with caution.

Importance of commits

Once you have learned how to move changes between the working directory and the staging area, the next major step is to actually save those changes into your project's history. In Git, this act is known as making a *commit*.

A commit in Git is a permanent, uniquely identified snapshot of the project's state at a specific point in time. It records the exact contents of every file that was staged, along with essential metadata such as the author, the date, the parent commit(s), and a message describing the purpose of the change. Each commit points to the one that came before it, forming a linked chain of versions that represents the full history of the repository.

A commit is not merely a save point — it is a permanent record that documents exactly what the project looked like at that point in time. Because commits cannot be modified without changing their identifiers, they provide a trustworthy and traceable sequence of changes.

How to commit?

To create a commit, you simply run the commit command and provide a short message describing what you changed. This message is important: it tells the story of your work. A good commit message is like a note you leave for your future self, or for anyone who might read your project later. It should be brief, but meaningful enough that someone can understand the purpose of that snapshot without opening every file in it.

```
$ git commit -m 'Commit Message'
```

Once the commit is created, all the changes that were in the staging area are sealed into the repository's history. From this point onward, you can always return to that version, compare it with others, or build new changes on top of it. Any edits still sitting in the working directory — those that were not staged — remain outside the commit and will wait until you decide whether to include them or discard them.

Amending changes

Sometimes, after creating a commit, you may realize that something about it is incomplete or incorrect. Perhaps the message was unclear, or you forgot to stage an important file before committing. Git provides a convenient way to fix such small mistakes without creating an entirely new entry in the project's history. This is done through the process of amending the most recent commit.

```
$ git commit --amend
```

Amending essentially replaces the last commit with a corrected version. The new commit contains whatever updated files you have staged, and it can also include an improved commit message. Although this operation preserves the intent of the original commit, it technically creates a new commit with a different identifier, since its contents and metadata have changed.

For this reason, amending should only be done on commits that have not yet been pushed to a shared remote repository.

Editing the commit message

When you amend a commit, Git often opens a text editor so that you can review or revise the commit message. On most systems, this editor is set to Vim by default. If you have never used Vim before, the interface may appear unfamiliar: the message is displayed in a plain text window, and Git expects you either to edit it or simply confirm it. If the message already suits your needs, you can save and close the editor to keep it unchanged.

The reader can explore a Vim guide to figure a way out without restarting the computer. The choice of editor, however, is not fixed. Git allows you to configure any editor you prefer, and will automatically use that tool whenever a commit message needs to be written or modified. On Windows, the user is prompted to change default editor during the installation process for Git, but if you changed your mind later, the user can still set VS Code, for example, to be the default editor by typing this command in the terminal.

```
$ git config --global core.editor ‘code --wait’
```

Navigating logs

Once you have made several commits, it becomes essential to be able to review the history of your project. Git maintains a complete record of all commits, including the author, time-stamp, commit message, and a unique identifier for each snapshot. This record is known as the commit history.

```
$ git log
```

Running this lists the commits in reverse chronological order. That is, the most recent commits appear first. Each entry shows the commit ID, the author's name and email, the date, and the commit message you provided. By examining this information, you can quickly identify what changes were made, when, and by whom.

Git also offers various ways to format and simplify the history for easier reading. For example, a condensed view shows only one line per commit, while a graphical representation can illustrate the relationship between branches and merges.

Useful arguments

In some cases, your project may have multiple branches, each representing different lines of development. Simply listing commits in a linear fashion may not give you the full picture of how these branches relate to one another. Fortunately, Git provides options to make the history easier to understand.

By using the "`--graph`" option, Git draws an ASCII-style diagram showing how commits branch and merge. Adding "`--all`" ensures that the history of all branches is displayed, not just the current one. Together, these options give you a clear, visual representation of the repository's structure, making it easier to see where different lines of development diverged and later merged.

```
$ git log --all --graph
```

This graphical view is especially useful when collaborating with others, as it helps you track the flow of changes across multiple contributors and branches. It transforms what could be a confusing list of commits into a structured, readable map of your project's evolution.

Comparing changes

Through the repository logs, the reader can see the history of all commits. Knowing that a commit is a snapshot of the project at a specific point in time, understanding the differences between these snapshots becomes essential to tracking how the project evolves. Fortunately, Git provides tools to compare these snapshots and highlight the changes made to files.

```
$ git diff
```

By default, running the command compares the current state of your working directory against the staging area. This shows you any edits that have been made but are not yet staged for the next commit.

Mastering the command is essential not only for tracking changes, but also for preparing merges, rebases, and other operations where knowing exactly how versions differ is critical.

Comparing changes

The comparison can be done at several levels. You can specify commit hashes, branch names, or tags, and Git will show the precise additions, deletions, and modifications that occurred between them.

```
$ git diff ‘Tag 1’ ‘Tag 2’
```

Comparing different points in your project’s history gives you a clearer understanding of how ideas evolve across time. However, not all changes occur in a single, continuous line of commits. In real projects, development often splits into parallel paths — some experimental, some stable, some temporary. To manage these different lines of work, Git provides one of its most powerful features: branches.

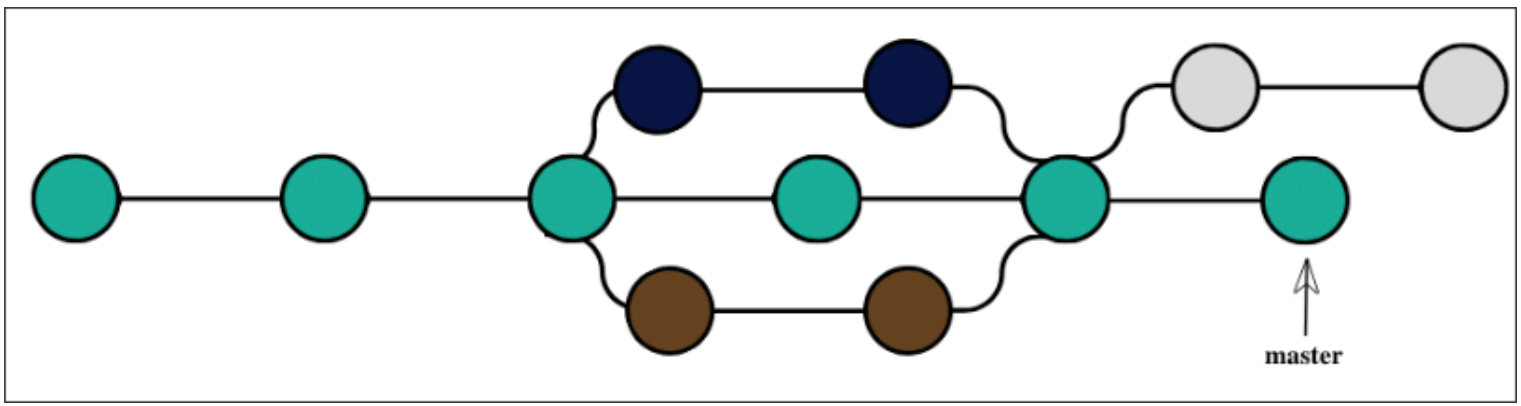
What are branches?

As projects grow in complexity, it often becomes necessary to work on different tasks simultaneously without interfering with the main line of development. In Git, this is accomplished through the concept of branches.

A branch in Git is a movable pointer to a specific commit, representing an independent line of development within a repository. It allows you to diverge from the main project history to work on new features, bug fixes, or experiments in isolation. Each branch maintains its own series of commits, enabling multiple lines of development to coexist simultaneously without interfering with one another. Branches can later be merged back into other branches, integrating their changes while preserving a clear history of how the project evolved.

What are branches?

You can imagine a branch as a side path in your project's history. The main branch represents the stable path, the version of your project that works and can be shared safely. When you want to try something new, you step off that main path and follow a branch. Everything you do on that branch stays there, leaving the main line untouched.



This separation gives you freedom to explore without fear of breaking what already works. You can make as many changes as you like, commit them, and see how they evolve. Later, when you're confident, you can bring the branch back into the main path, merging your work so it becomes part of the project's history. In a sense, they act like safe playgrounds within your repository.

Creating branches

Once you understand why branches are useful, the next step is learning how to create one. In Git, that is achieved via the following command.

```
$ git branch 'Branch Name'
```

Branches are extremely lightweight in Git. Creating one does not duplicate your files or take up extra space. It is simply a pointer to a specific commit. This makes it easy to create multiple branches for different experiments and bug fixes to merge them back into the main branch when ready.

You can rename an existing branch by adding '`-m`' before the branch name.

```
$ git branch -m 'New Name'
```

If the name you want already exists, Git won't overwrite it unless you use the capital '`-M`' option. The only difference is that '`-M`' forces the rename, while '`-m`' refuses if it would overwrite another branch. Other than that small detail, the process works the same.

Navigating branches in Git

Once you start working with branches, it's important to know how to move around, inspect your repository, and manage your different lines of development. Git provides several commands to help you navigate efficiently and make sure you always know which branch you're on and what changes are staged or committed.

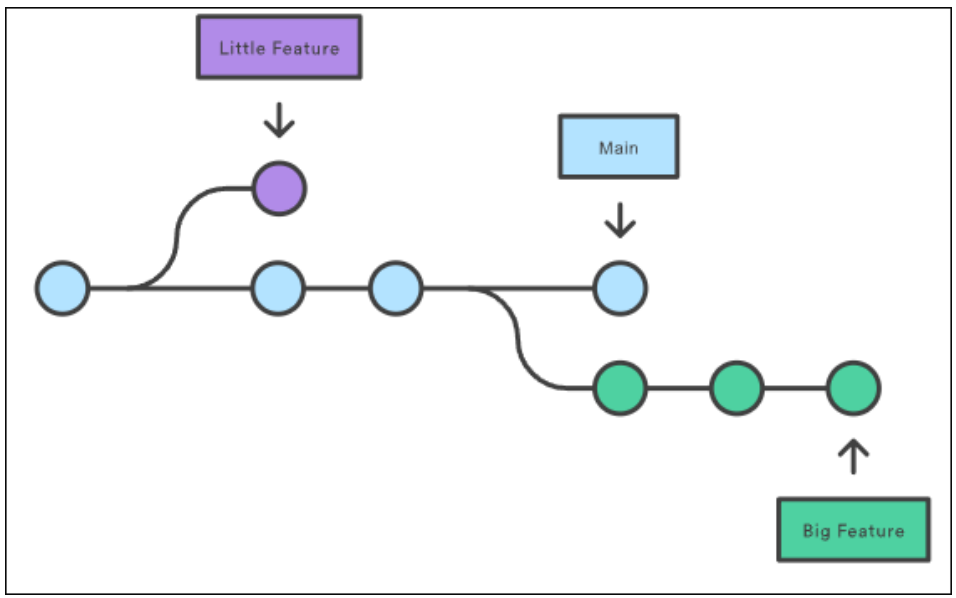
```
$ git branch --list
```

First, you can see a list of all the branches in your repository. This helps you understand what branches exist and which one is currently active. The current branch is usually highlighted with an asterisk or some other indicator. You can switch between branches at any time with one of the following.

```
$ git checkout 'Branch Name'
$ git switch 'Branch Name'
```

Navigating branches in Git

When you switch, Git updates your working directory to match the state of the branch you've chosen. This means files and commits from that branch become visible, while any work on other branches remains safely separate. If you ever want to create a new branch, Git will start it from your current location in the history, giving you a fresh pointer to begin committing new changes.



To delete a branch, simply run "`git branch -D 'Branch Name'`"