How to Use Git – Stunning Guide with Awesome Tips

By Oluwatobi Sofela • Last modified: April 9, 2021

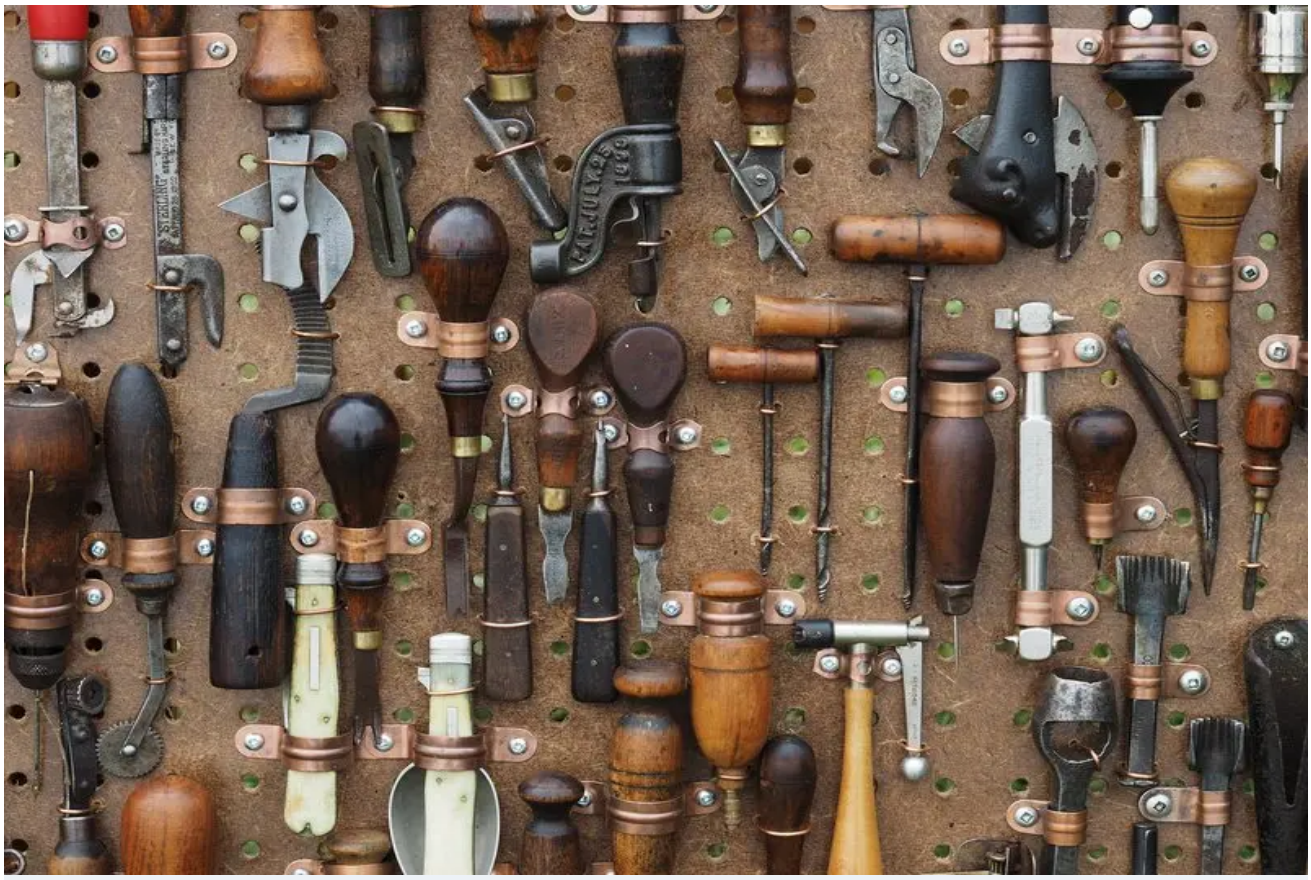

Table of Contents

This *“How to use Git”* post will take you through the essentials needed to get started with Git.

First, before you can use Git, you need to install it on your system — so let’s start with the installation process.

Git Installation

You can easily install Git from the Git download webpage.

A handy way to check the version installed on your system is to run:

```
git --version
```

After the installation, it is necessary to keep it up to date whenever there is a new version.

Update Git

If you are using a Windows system and your currently installed version is 2.16.1(2) or higher; freely get the latest version with this command:

```
git update-git-for-windows
```

Once you have the correct version installed on your system, you can then proceed with the setup process by *initializing Git* in your project's directory.

Initialize Git

Initialization is to make Git *ready to start* monitoring files in a specified directory.

To initialize Git in a directory currently not under version control, you need first to **go inside that directory** from your terminal.

```
1.  cd path/to/the/directory
```

Afterward, initialize Git in that project's directory by running:

```
1.  git init
```

After running the command above, a Git repository, named `.git`, will be created in the project's folder.

Note:

- Initializing Git in a project does not mean Git is tracking anything in that project's directory. The `git init` command just creates the `.git` repository where almost everything that Git stores and manipulates lives.
- If you are curious about the contents in the `.git` directory, check Plumbing and Porcelain for more information.

Git Configuration

After the installation and initialization process, it is essential to configure your *username* and *email address* — as Git permanently bakes these details into your commits. Thus, helping team members identify each commit's creator.

To configure your identity, run the following commands — one by one — in your terminal:

```
1.  git config --global user.name "Your Name"
2.  git config --global user.email "your-email@address.com
```

Note:

The `--global` option enables you to run the above commands

only once. However, if you do not mind inputting your *name* and *email* for each project, you can omit the `--global` flag.

Change a project's configuration

To change the *configured name* or *email address* for a specific project, *go into that project's directory* and run the command(s) below — one after the other:

```
1.  git config user.name "The New Name"
2.  git config user.email "the-new-email@address.com"
```

Afterward, use the `git config --list` command to confirm your configurations.

Also, use the `git config <key>` command to check a specific key's value. For instance, to check the configured *email address's* value, you will run `git config user.email`.

Exit Git configuration space

To exit the Git configuration space, simply press the ***Q*** key on your computer's keyboard.

Track files

After initializing and configuring Git in a specific working directory, you can now use Git to begin tracking changes in your choice file(s). But the file(s) to be tracked must be in the working directory in which you initialized Git.

To begin tracking file(s) in the initialized project's folder, *go into that project's directory from the terminal* and run:

```
1.  git add <fileOrFolderName>
```

After running the command above, Git will add the specified file (or folder) to the staging area. Hence, Git will track the file (or folder): should incase any further modification occurs on it after its staging.

Note:

- Replace `<fileOrFolderName>` in the tracking command above with *the file's (or directory's) pathname*.
- Suppose `<fileOrFolderName>` is the pathname of a directory. In that case, the `git add` command will automatically add all the files in that directory into the staging area recursively.

Stage quickly

Attaching additional options to the `git add` command can help speed up the staging process — especially when there are multiple files to stage.

Below are *additional options* commonly used to add multiple files to the staging area simultaneously.

Option 1: "`-A`" Flag

To stage *all modified, and untracked,* files in a project's entire working directory — regardless of the project's directory you are currently in, run:

```
1.  git add -A
```

Note:

The `-A` flag used above is the shorthand notation for the `--all` flag. Hence, you can use any one of the two flags to add all modified files to the staging area.

Option 2: "`-u`" Flag

To stage *only modified* files you had committed previously, run:

```
1.  git add -u
```

Note:

- The `-u` flag will stage all modified files — currently being track — in the project's entire working directory — regardless of the project's directory you are in presently.
- `-u` flag *will not add any new file* (i.e., previously uncommitted files) to the staging area.
- The `-u` flag used above is the shorthand notation for the `--update` flag. Thus, you can use any one of the two flags to update all modified files currently under version control.

Option 3: "`.`" (dot) Symbol

To stage *all modified and untracked* files located ***only*** in the project's directory you are currently in, run:

```
1.  git add .
```

Note:

- *dot symbol* means *"current directory"*. Hence, the above command will stage all modified, and untracked files ***only of that project's directory*** wherein the `git add .`

command got invoked.
- The *dot* is essential in the `add` command above.
- There must be a *space* between the `add` command and the *dot mark*.

Option 4: " * " (asterisk) Symbol

Although the `git add *` command can also stage multiple files, however, to avoid unexpected results, **it is better not to use it**.

Note:

- Staging files does not mean you have saved the staged files into the Git directory. Instead, staging implies that you have added *details* about the staged file(s) into a file called *"index"* — located in the Git directory.
- Only the file versions in the staging area get committed to subsequent historic snapshots — not the working directory's versions. Therefore, committing at this point will only save the file versions presently in the staging area (when you ran `git add`) — not the versions in the working directory.
- After staging all necessary files, you can then commit (save) them to the `.git` directory.

Commit files

Committing a file means to store the staged version of a working directory's file(s) into the `.git` repository.

To commit a file that is currently in the staging area, run:

```
1.  git commit -m "Write a message about the file(s) you a
```

Note:

- The `-m` flag in the command above allows you to write your message in-line with the `commit` command.
- If you omit the `-m` flag, your text editor will be launched with some comments prompting you to enter your commit message.
  Note that you can remove the editor's comments and type in your commit message, or you can leave the comments and add in your commit message.
- Once you've finished writing your commit message inside the text editor, exit it so that Git can create the commit.
- Writing commit messages with a text editor provides the opportunity to write long notes — even with multiple paragraphs. However, for short commit messages, the in-line method is sufficient.
- Suppose you prefer to use the text editor to input your commit message. In that case, you can optionally add the `-v` flag to the `git commit` command. In doing so, Git will

include the *diff* of your changes into the editor so you can see the exact changes you are committing.

- Crafting a good commit message will help your collaborators (and yourself — after some time) understand why you committed a specific version of your project's file(s). Also, it will help elucidate differences that exist between the file versions committed.
- Committed files reside in the local Git repository — named `.git`.
- A project's commit history is viewable by running `git log` on the command line. Note that the project's directory — whose commit history you want to check — *must be the current directory*.
- To skip the staging area (i.e., skip running the `git add` command), add an `-a` flag to the `git commit` command (i.e., `git commit -a`). Thus, Git will automatically stage all the files it is already tracking, before doing a commit.

Clone a Git repository

Git Cloning is mainly about getting (downloading) a copy of a `.git` repository.

For instance, you may need a copy of a project you intend to modify. In such a case, getting a clone of the project's `.git` directory puts at your possession all the file versions the project's contributor(s) have committed to the `.git` repository.

To clone a repository, run:

```
1.  git clone <theGitRepoURL> <state the place to put the
```

Thus, Git will download a copy of the specified `.git` repository into the place you've identified.

Note

- Replace `<theGitRepoURL>` in the `clone` command above with the URL of the `.git` directory you want to clone.
- Also, replace `<state the place to put the cloned git folder>` with your system's location, wherein you want the cloned `.git` repository to reside.
- Whenever you clone any remote repo, Git automatically names that repo's URL " `origin` ". However, you can specify a different name with the `git clone -o yourPreferredName` command.

Ignore files

To instruct Git to ignore specific file(s) or folder(s) in a particular project — including not showing them as untracked, create a `.gitignore` file.

Note:

- Most people create the `.gitignore` file in the project's *root directory*.
- *Root directory* refers to the folder containing all other files and sub-folders of a specific project. In other words, a root repository is a working directory that houses everything concerning a particular project. Hence, its content will also include the `.git` directory.

Create `.gitignore` file

To create a `.gitignore` file, *go into the root directory* wherein the file(s) you want to ignore are. Then, run the command below on your terminal:

```
1.  touch .gitignore
```

Specify files to ignore

After creating a `.gitignore` file in the *project's root directory*, open the `.gitignore` file and write in it, the names of the file(s), folder(s), or filetype(s) you want Git to ignore.

Note that you can use the hash symbol (`#`) to include comments inside the `.gitignore` file.

An example of a `.gitignore` file is:

```
# In a ".gitignore" file, an asterisk (*) is a wildcar
# means zero or more characters — except a slash — car
# asterisk.

# below command will ignore any document starting with
# anywhere in this project:
abc*

# below command will ignore all ".txt" filetypes locat
# this project:
*.txt

# In a ".gitignore" file, an exclamation mark (!) mear

# this command below will track "readme.txt" filetypes
# you've instructed Git — through the asterisk commanc
# ignore all ".txt" files:
!readme.txt

# the command below will ignore any "package-lock.jsor
# anywhere in this project:
package-lock.json

# below command will ignore only the "passwords.text"
# the specified directory:
directory/of/passwords.text

# this command will only ignore the "NOTE" file in the
# directory — not in any other directory — of this prc
/NOTE

# the following command will ignore all the contents i
# named "modules":
modules/
```

```
35.
36.  # ignore all ".txt" files directly inside the folder r
37.  mix/*.txt
38.  # Note that the above command will still track all ".t
39.  # are in the subdirectories of the "mix" folder. For i
40.  # will ignore "mix/test.txt" but will track "mix/real/
41.
42.  # In a ".gitignore" file, double asterisks (**) mark -
43.  # slash (e.g., **/) — is a wildcard symbol that means
44.  # directories' names can replace the double asterisks.
45.
46.  # below command will ignore all ".pdf" files inside bo
47.  # named "doc" and in any of its subdirectories:
48.  doc/**/*.pdf
49.
50.  # the question mark (?) in the command below means any
51.  # character — except a slash — can replace the questio
52.  sea?.txt
53.  # Hence, the above command will ignore files like "sea
54.  # "seat.txt". However, it will not ignore files like "
55.
56.  # In a ".gitignore" file, a pair of square brackets "[
57.  # the range of characters acceptable for a single char
58.  # Below are some examples.
59.
60.  # this command means the character after character "n"
61.  # character "s" or character "t":
62.  plan[st].js
63.  # Hence, the above command will match files like "plan
64.  # "plant.js". But it will not match "plane.js" nor "pl
65.
66.  # below command means the character after character "n
67.  # character between numbers "3 to 9" inclusive:
68.  plan[3-9].js
69.  # Hence, the above command will match files like "plan
70.  # "plan5.js". But it will not match "plan10.txt" nor "
71.
72.  # the following command means the character after char
```

```
# be any character between letters “f to z” inclusive:
plan[f-z].js
# Hence, the above command will match files like “plar
# “plany.js”. But it will not match “plan2.txt” nor “p

# this command means the character after character “n”
# character “s” nor character “t”:
plan[!st].js
# Hence, the above command will match files like “plar
# “plan2.js”. But it will not match “plans.js” nor “pl

# Note that an exclamation mark within a square bracke
# means “negate”.
```

Note:

A `.gitignore` file serves to ensure specific files that are currently untracked remain untracked. In other words, Git won’t ignore any file it is already tracking — even if specified in a `.gitignore` file — unless you delete such a file from the staging area.

Handy Git commands

Here are some other Git commands you may find handy:

Check the installation directory of Git by running:

```
1.  which git
```

View a project's commit history by running:

```
1.  git log
```

Note:

- After running the command above, if you get stuck on the Git console, just hit the ***Q*** key on your keyboard to exit.
- Adding `-3` to the log command above — i.e., `git log -3` — will limit the number of commit history displayed to the last three entries.
- The above `log` command only displays basic commit history — Author, Date, and Commit message. However, to include the *diff* introduced in each of the commits, add `-p` (short for `--patch`) to the `git log` command (i.e., `git log -2 -p`).
- To make the chunk less cumbersome, add the `--color-words` flag to the `--patch` command (i.e., `git log -p -2 --color-words`). Thus, each chunk will include just the modified *words* — not *lines* — and their context.
- A chunk refers to a modified section of a file.
- A chunk includes the modified line and its context (i.e., some unmodified lines before and after the altered line).

- To display a summarized statistic of the changes that occurred in each commit, add the `--stat` flag to the `git log` command (i.e., `git log --stat`).
- Check the "Viewing the Commit History" article by Git for more interesting options used with the `git log` command to customize the log output.

To check the status of your files, run:

```
1.  git status
```

Undo the staging of all files in the staging area:

```
1.  git reset
```

Undo the staging of a specific file in the staging area:

```
1.  git reset HEAD fileToUnstage
```

Rename a file:

```
1.  git mv currentFileName newFileName
```

See all the files currently in the staging area and the `.git` repository:

```
1.  git ls-files
```

Note:

Additional options for the `ls-files` command are listed the ls-files documentation by Git.

Confirm if a specific file or folder is in the `.git` directory:

```
1.  git ls-files | grep <fileOrFolderName>
```

Note:

- Replace `<fileOrFolderName>` in the command above with *the name* of the file (or directory) you want to check.
- If nothing shows after running the command above, it means the specified file or folder is not in the `.git`

directory.

- The `grep` command is a useful utility for searching for a specified pattern or characters and printing lines that match that pattern. For instance, `git ls-files | grep test.js` will look for and print lines that match `test.js`.
- Additional options for the `grep` command can be found in the grep documentation by Git.

Open gitignore manual page:

```
1.  git help gitignore
```

Note:

The gitignore manual will open in your default browser.

Write text into a file via the terminal:

```
1.  echo textToWrite >> fileToWriteInto
```

Note:

- `>>` will append your text to the end of the file's content.
- `>` will overwrite any existing content of the specified file.
- Some symbols — such as the exclamation mark (`!`) symbol

— needs the backslash (`\`) mark before each symbol for it to be rendered appropriately in the file into which you want to write.
For instance, to write, `Hello!!!` into a file via the command line, I would run:

```
1.  echo Hello\!\!\! >> fileToWriteInto
```

Compare the version of files in the working directory with the version you added to the staging area:

```
1.  git diff
```

Note:

- The `git diff` command shows the changes in a working directory's files that you've not yet staged.
- By default, `git diff` will compare a working directory's file version with the version in the staging area. However, for any file currently not in the staging area, `git diff` will compare the working directory's version with the last committed version.
- Suppose nothing shows after running the `diff` command

above. In that case, it implies that there is the same version of files in the working directory and the staging area.

- The `git diff` command does not show the entire content of a file. Instead, it displays only the chunks.
- A chunk refers to a modified section of a file.
  Note that the chunk includes both the modified line and its context (i.e., some unmodified lines before and after the altered line). Thus, making it easy to spot the displayed alteration's location.
- To make the chunk less cumbersome, add the `--color-words` flag to the `git diff` command (i.e., `git diff --color-words`). Thus, each chunk will include just the modified *words* — not *lines* — and their context.

Compare the file versions in the staging area with the last versions committed into the Git directory:

```
1.  git diff --staged
```

Note:

- The `git diff --staged` command shows the difference between file version(s) in the staging area and the last version committed to the Git directory.
- To make the chunk less cumbersome, add the `--color-`

`words` flag to the `git diff --staged` command (i.e., `git diff --staged --color-words`). Thus, each chunk will include just the modified *words* — not *lines* — and their context.

- The `git diff --staged` command is synonymous to the `git diff --cached` command. Hence, you may use any of them to compare the staging area's files with the last committed versions.

Change your last commit message:

```
1.  git commit --amend
```

Note:

- To modify the actual content of your last file version committed, simply make the changes on the file, stage the changes, and run the command above.
  Hence, Git will automatically replace your last commit with this updated version.
- Beware! Use this command cautiously! Amending a commit changes the commit's SHA-1. Therefore, to avoid confusing other collaborators, it's best not to amend any commit that you have pushed (shared)!
- Suppose you made only minor modifications to the file. In that case, you could skip writing a commit message by

adding the `--no-edit` option to the command above (i.e., `git commit --amend --no-edit`).

Remove file only from the staging area and not from your working directory:

```
1.  git rm --cached fileToDelete
```

Replace a working directory's file with its last committed version:

```
1.  git checkout -- localFileToDeleteAndReplace
```

Note:

This command is a dangerous command that will permanently delete all changes in the local version of the specified file. That is, Git will replace the local version with the last committed version.

Delete file from the working directory and the `.git` repository:

```
1.  git rm fileToDelete
```

Note:

- Suppose you only run `rm fileToDelete` — i.e., excluding the `git` command. In that case, the specified file will only get deleted from the working directory. Hence Git will still track it as an *unstaged* change.
  However, running `git rm fileToDelete` will delete the specified file from the working directory and automatically stage the removal. Hence, during the next commit, Git will delete the file from the `.git` directory.
- Use the `-f` flag — e.g., `git rm -f fileToDelete` — to forcefully remove modified files or files previously added to the staging area.
- Add the `-r` flag (e.g., `git rm -r fileToDelete`) to remove a folder.

Dangerous Git commands

Commands that permits you to *undo* changes are generally dangerous: as they can permanently delete your work — if you do it wrongly! Hence, be incredibly careful with these commands!

The Undoing Things article by Git is a good read on tools for undoing changes.

Share Git repository

Sharing a Git repository online makes it easy for collaborators to collaborate on a project from anywhere, at any time.

Moreover, GitHub — a popular online platform used for sharing `.git` repositories — takes Git collaboration to a whole new height.

To share your Git repository (i.e., your project's committed file versions) on GitHub, follow the "Host Git repository on GitHub" guide.

Useful resources

Below are links to other valuable content on how to use Git.

- About Version Control
- A collection of useful `.gitignore` templates
- Cheat Sheet

Credit

Featured Image: Tools awl pliers by Deborah Breen Whiting

#Client-side #Git and GitHub #Server-side

← PREVIOUS

Git vs. Working Directory – Quick Comparison of the Basics

NEXT →

Git vs. GitHub – Quick Look at the Key Differences

Similar Posts