cd shop //To enter to C:\Users\capta\shop

or

$ cd d:/courses/Git // //To enter to d:/courses/Git

What is Git?

Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

It is used for:

* Tracking code changes
* Tracking who made changes
* Coding collaboration

What does Git do?

* Manage projects with **Repositories**
* **Clone** a project to work on a local copy
* Control and track changes with **Staging** and **Committing**
* **Branch** and **Merge** to allow for work on different parts and versions of a project
* **Pull** the latest version of the project to a local copy
* **Push** local updates to the main project

Working with Git

* Initialize Git on a folder, making it a **Repository**
* Git now creates a hidden folder to keep track of changes in that folder
* When a file is changed, added or deleted, it is considered **modified**
* You select the modified files you want to **Stage**
* The **Staged** files are **Committed**, which prompts Git to store a **permanent** snapshot of the files
* Git allows you to see the full history of every commit.
* You can revert back to any previous commit.
* Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

Why Git?

* Over 70% of developers use Git!
* Developers can work together from anywhere in the world.
* Developers can see the full history of the project.
* Developers can revert to earlier versions of a project.

What is GitHub?

* Git is not the same as GitHub.
* GitHub makes tools that use Git.
* GitHub is the largest host of source code in the world, and has been owned by Microsoft since 2018.
* In this tutorial, we will focus on using Git with GitHub.

Git Install

You can download Git for free from the following website: [https://www.git-scm.com/](https://git-scm.com/)

Using Git with Command Line

To start using Git, we are first going to open up our Command shell.

For Windows, you can use Git bash, which comes included in Git for Windows. For Mac and Linux you can use the built-in terminal.

The first thing we need to do, is to check if Git is properly installed:

Example

git --version

If Git is installed, it should show something like git version X.Y

## Configure Git

Now let Git know who you are. This is important for version control systems, as each Git commit uses this information:

### Example

git config --global user.name "w3schools-test"

git config --global user.email "test@w3schools.com"

Change the user name and e-mail address to your own. You will probably also want to use this when registering to GitHub later on.

**Note:** Use global to set the username and e-mail for **every repository** on your computer.

If you want to set the username/e-mail for just the current **repo**, you can remove global

## Creating Git Folder

Now, let's create a new folder for our project:

### Example

mkdir myproject

cd myproject

**mkdir** **make**s a**new directory.**

**cd** **changes** the **current working directory.**

Now that we are in the correct directory. We can start by initializing Git!

**Note:** If you already have a folder/directory you would like to use for Git:

Navigate to it in command line, or open it in your file explorer, right-click and select "Git Bash here"

## Initialize Git

Once you have navigated to the correct folder, you can initialize Git on that folder:

### Example

git init

Initialized empty Git repository in /Users/user/myproject/.git/

You just created your first Git Repository!

**Note:** Git now knows that it should watch the folder you initiated it on.

Git creates a hidden folder to keep track of changes.

## Git Adding New Files

You just created your first local Git repo. But it is empty.

So let's add some files, or create a new file using your favourite text editor. Then save or move it to the folder you just created.

And save it to our new folder as index.html.

Let's go back to the terminal and list the files in our current working directory:

### Example

ls

index.html

**ls** will **list** the files in the directory. We can see that index.html is there.

Then we check the Git status and see if it is a part of our repo:

Example

git status

On branch master

No commits yet

Untracked files:

  (use "git add ..." to include in what will be committed)

    index.html

nothing added to commit but untracked files present (use "git add" to track)

Now Git is **aware** of the file, but has not **added** it to our repository!

Files in your Git repository folder can be in one of 2 states:

* Tracked - files that Git knows about and are added to the repository
* Untracked - files that are in your working directory, but not added to the repository

 When you first add files to an empty repository, they are all untracked. To get Git to track them, you need to stage them, or add them to the staging environment.

## Git Staging Environment

One of the core functions of Git is the concepts of the Staging Environment, and the Commit.

As you are working, you may be adding, editing and removing files. But whenever you hit a milestone or finish a part of the work, you should add the files to a Staging Environment.

**Staged** files are files that are ready to be **committed** to the repository you are working on. You will learn more about commit shortly.

For now, we are done working with index.html. So we can add it to the Staging Environment:

### Example

git add index.html

The file should be **Staged**. Let's check the status::

### Example

git status

On branch master

No commits yet

Changes to be committed:

  (use "git rm --cached ..." to unstage)

    new file: index.html

Now the file has been added to the Staging Environment.

## Git Add More than One File

You can also stage more than one file at a time. Let's add 2 more files to our working folder. Use the text editor again.

Now add all files in the current directory to the Staging Environment:

### Example

git add --all

Using --all instead of individual filenames will stage all changes (new, modified, and deleted) files.

Now all 3 files are added to the Staging Environment, and we are ready to do our first commit.

**Note:** The shorthand command for git add --all is git add -A

## Git Commit

Since we have finished our work, we are ready move from stage to commit for our repo.

Adding commits keep track of our progress and changes as we work.  
Git considers each commit change point or "save point". It is a point in the project you can go back to if you find a bug, or want to make a change.

When we commit, we should **always** include a **message**.

By adding clear messages to each commit, it is easy for yourself (and others) to see what has changed and when.

### Example

git commit -m "First release of Hello World!"

[master (root-commit) 221ec6e] First release of Hello World!

3 files changed, 26 insertions(+)

create mode 100644 README.md

create mode 100644 bluestyle.css

create mode 100644 index.html

The commit command performs a commit, and the -m "message" adds a message.

The Staging Environment has been committed to our repo, with the message:  
"First release of Hello World!"

## Git Commit without Stage

Sometimes, when you make small changes, using the staging environment seems like a waste of time. It is possible to commit changes directly, skipping the staging environment. The **-a** option will **automatically stage** every changed, already tracked file.

And check the status of our repository. But this time, we will use the **--short** option to see the changes in a more compact way:

Example

git status --short

M index.html

**Note:** Short status flags are:

* ?? - Untracked files
* A - Files added to stage
* M - Modified files
* D - Deleted files

We see the file we expected is modified. So let's commit it directly:

Example

git commit -a -m "Updated index.html with a new line"

[master 09f4acd] Updated index.html with a new line

1 file changed, 1 insertion(+)

**Warning:** Skipping the Staging Environment is not generally recommended.

Skipping the stage step can sometimes make you include unwanted changes.

## Git Commit Log

To view the history of commits for a repository, you can use the log command

**Example**

git log

**Shift q** // To quit from search or log

Git Help

If you are having trouble remembering commands or options for commands, you can use Git help.

There are a couple of different ways you can use the help command in command line:

* git *command* -help -  See all the available options for the specific command
* git help --all -  See all possible commands

 Let's go over the different commands.

Git -help See Options for a Specific Command

Any time you need some help remembering the specific option for a command, you can use git *command* -help:

Example

git commit -help

**Note:** You can also use --help instead of -help to open the relevant Git manual page. (It means open the page in the browser)

## Git help --all See All Possible Commands

To list all possible commands, use the help --all command:

**Warning:** This will display a very long list of commands

**Note:** If you find yourself stuck in the list view, SHIFT + G to jump the end of the list, then q to exit the view.