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.

# **Git Branch**

[❮ Previous](https://www.w3schools.com/git/git_help.asp?remote=github)[Next ❯](https://www.w3schools.com/git/git_branch_merge.asp?remote=github)

## Change Platform:

[[](https://www.w3schools.com/git/git_branch.asp?remote=github)GitHub](https://www.w3schools.com/git/git_branch.asp?remote=github)[[](https://www.w3schools.com/git/git_branch.asp?remote=bitbucket)Bitbucket](https://www.w3schools.com/git/git_branch.asp?remote=bitbucket)[[](https://www.w3schools.com/git/git_branch.asp?remote=gitlab)GitLab](https://www.w3schools.com/git/git_branch.asp?remote=gitlab)

## Working with Git Branches

In Git, a branch is a new/separate version of the main repository.

Let's say you have a large project, and you need to update the design on it.

How would that work without and with Git:

Without Git:

* Make copies of all the relevant files to avoid impacting the live version
* Start working with the design and find that code depend on code in other files, that also need to be changed!
* Make copies of the dependant files as well. Making sure that every file dependency references the correct file name
* EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
* Save all your files, making a note of the names of the copies you were working on
* Work on the unrelated error and update the code to fix it
* Go back to the design, and finish the work there
* Copy the code or rename the files, so the updated design is on the live version
* (2 weeks later, you realize that the unrelated error was not fixed in the new design version because you copied the files before the fix)

With Git:

* With a new branch called new-design, edit the code directly without impacting the main branch
* EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
* Create a new branch from the main project called small-error-fix
* Fix the unrelated error and merge the small-error-fix branch with the main branch
* You go back to the new-design branch, and finish the work there
* Merge the new-design branch with main (getting alerted to the small error fix that you were missing)

Branches allow you to work on different parts of a project without impacting the main branch.

When the work is complete, a branch can be merged with the main project.

You can even switch between branches and work on different projects without them interfering with each other.

Branching in Git is very lightweight and fast!

## New Git Branch

Let add some new features to our index.html page.

We are working in our local repository, and we do not want to disturb or possibly wreck the main project.

So we create a new branch:

### Example

git branch hello-world-images

Now we created a new branch called "hello-world-images"

Let's confirm that we have created a new branch:

### Example

git branch

hello-world-images

\* master

We can see the new branch with the name "hello-world-images", but the \* beside master specifies that we are currently on that branch.

checkout is the command used to check out a branch. Moving us **from** the current branch, **to** the one specified at the end of the command:

### Example

git checkout hello-world-images

Switched to branch 'hello-world-images'

Now we have moved our current workspace from the master branch, to the new branch

Open your favourite editor and make some changes.

For this example, we added an image (img\_hello\_world.jpg) to the working folder and a line of code in the index.html file:

Example

<!DOCTYPE html>  
<html>  
<head>  
<title>Hello World!</title>  
<link rel="stylesheet" href="bluestyle.css">  
</head>  
<body>  
  
<h1>Hello world!</h1>  
<div><img src="img\_hello\_world.jpg" alt="Hello World from Space"  
style="width:100%;max-width:960px"></div>  
<p>This is the first file in my new Git Repo.</p>  
<p>A new line in our file!</p>  
  
</body>  
</html>

We have made changes to a file and added a new file in the working directory (same directory as the main branch).

Now check the status of the current branch:

Example

git status

On branch hello-world-images

Changes not staged for commit:

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

(use "git restore ..." to discard changes in working directory)

modified: index.html

Untracked files:

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

img\_hello\_world.jpg

no changes added to commit (use "git add" and/or "git commit -a")

So let's go through what happens here:

* There are changes to our index.html, but the file is not staged for commit
* img\_hello\_world.jpg is not tracked

So we need to add both files to the Staging Environment for this branch:

Example

git add --all

Using --all instead of individual filenames will **Stage** all changed (new, modified, and deleted) files.

Check the status of the branch:

Example

git status

On branch hello-world-images

Changes to be committed:

  (use "git restore --staged ..." to unstage)

    new file: img\_hello\_world.jpg

    modified: index.html

We are happy with our changes. So we will commit them to the branch:

Example

git commit -m "Added image to Hello World"

[hello-world-images 0312c55] Added image to Hello World

2 files changed, 1 insertion(+)

create mode 100644 img\_hello\_world.jpg

Now we have a new branch, that is different from the master branch.

**Note:** Using the -b option on checkout will create a new branch, and move to it, if it does not exist

Switching Between Branches

Now let's see just how quick and easy it is to work with different branches, and how well it works.

We are currently on the branch hello-world-images. We added an image to this branch, so let's list the files in the current directory:

Example

ls

README.md bluestyle.css img\_hello\_world.jpg index.html

We can see the new file img\_hello\_world.jpg, and if we open the html file, we can see the code has been altered. All is as it should be.

Now, let's see what happens when we change branch to master

Example

git checkout master

Switched to branch 'master'

The new image is not a part of this branch. List the files in the current directory again:

Example

ls

README.md bluestyle.css index.html

img\_hello\_world.jpg is no longer there! And if we open the html file, we can see the code reverted to what it was before the alteration.

See how easy it is to work with branches? And how this allows you to work on different things?

Emergency Branch

Now imagine that we are not yet done with hello-world-images, but we need to fix an error on master.

I don't want to mess with master directly, and I do not want to mess with hello-world-images, since it is not done yet.

So we create a new branch to deal with the emergency:

Example

git checkout -b emergency-fix

Switched to a new branch 'emergency-fix'

Now we have created a new branch from master, and changed to it. We can safely fix the error without disturbing the other branches.

Let's fix our imaginary error:

Example

<!DOCTYPE html>  
<html>  
<head>  
<title>Hello World!</title>  
<link rel="stylesheet" href="bluestyle.css">  
</head>  
<body>  
  
<h1>Hello world!</h1>  
<p>This is the first file in my new Git Repo.</p>  
<p>This line is here to show how merging works.</p>  
  
</body>  
</html>

We have made changes in this file, and we need to get those changes to the master branch.

Check the status:

Example

git status

On branch emergency-fix

Changes not staged for commit:

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

(use "git restore ..." to discard changes in working directory)

modified: index.html

no changes added to commit (use "git add" and/or "git commit -a")

stage the file, and commit:

Example

git add index.html

git commit -m "updated index.html with emergency fix"

[emergency-fix dfa79db] updated index.html with emergency fix

1 file changed, 1 insertion(+), 1 deletion(-)

Now we have a fix ready for master, and we need to merge the two branches.

* When I was working on this file with branches in Git, I faced a serious problem that prevented me from tracking it in two branches.
* So, I had taken the error that appeared and asked ChatGPT about it.
* It suggested to me a solution for the problem as follows:
  + Go to: D:/courses/Git/.git
  + Then delete the file called: index.lock
* After I had completed these steps, the problem was solved.
* So, I documented these steps to learn from similar problems in the future.