# Git Tutorial

#### Git is a tool that helps you:

* **save and manage different versions of your files and code.**
* **work with others, keep track of changes, and undo mistakes.**

## Where to use Git?

Git works on your computer, but you also use it with online services like **GitHub**, **GitLab**, or **Bitbucket** to share your work with others. These are called **remote repositories**.

>git --version

## 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

## Key Git Concepts

* **Repository:** A folder where Git tracks your project and its history.
* **Clone:** Make a copy of a remote repository on your computer.
* **Stage:** Tell Git which changes you want to save next.
* **Commit:** Save a snapshot of your staged changes.
* **Branch:** Work on different versions or features at the same time.
* **Merge:** Combine changes from different branches.
* **Pull:** Get the latest changes from a remote repository.
* **Push:** Send your changes to a remote repository.

## 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.**

**Note:** Most Git actions (like staging, committing, and viewing history) happen on your own computer.

Only **Push** and **Pull** interact with remote servers like GitHub, GitLab, or Bitbucket to upload or download changes.

## 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.**

## Default Editor

>git config --global core.editor "code --wait" OR

>git config --global core.editor "notepad"

**Update:** Download and run the latest installer, or use your package manager  
(e.g., brew upgrade git or sudo apt-get upgrade git).

## Configure Git

## User Name

Your name will be attached to your commits.

>git config --global user.name "Your Name"

## Email Address

Your email is also attached to your commits

>git config --global user.email [you@example.com](mailto:you@example.com)

Note:

* Use --global to set the value for **every repository** on your computer.
* Use --local (the default) to set it only for the current repository.

## Why Configure Git?

Git uses your name and email to label your commits.

If you do not set these, Git will prompt you the first time you try to commit.

## Viewing Your Configuration

### List All Settings

>git config –list

### View a Specific Setting

>git config user.name

## Changing or Unsetting Config Values

To change a value, just run the git config command again with the new value.

To remove a setting, use --unset:

>git config --global --unset code.editor

## Default Branch Name

Set the default branch name for new repositories (for example, main instead of master):

>git config --global init.defaultBranch main

## Configuration Levels

There are three levels of configuration:

* **System** (all users): git config --system
* **Global** (current user): git config --global
* **Local** (current repo): git config –-local

### Set a Local Config

Local settings only apply to the current repository.

>git config user.name "Project Name"

### Set a Global Config

Global settings apply to all repositories for the current user.

>git config --global user.name "Global Name"

### Set a System Config

System settings apply to all repositories for all users.

>git config --system user.name "System Name"

## Get Started with Git

## Key Steps to Get Started

* Create a project folder
* Navigate to the folder
* Initialize a Git repository

## Creating Git Folder

>mkdir LearnGit

>cd LearnGit

* mkdir creates a new directory.
* cd changes our working directory.

## Initialize Git

> git init

## What Happens When You Run git init?

Git creates a hidden folder called .git inside your project.This is where Git stores all the information it needs to track your files and history.

### Show Hidden .git Folder

>ls –a

## What is a New File?

A **new file** is a file that you have created or copied into your project folder, but haven't told Git to watch.

Here are the key things to know:

* Create a new file (with a text editor)
* ls - List files in the folder
* git status - Check which files are tracked

## List Files in the Directory

>ls

## Check File Status with git status

## What is an Untracked File?

An **untracked file** is any file in your project folder that Git is not yet tracking.

These are files you've created or copied into the folder, but haven't told Git to watch.

## What is a Tracked File?

A **tracked file** is a file that Git is watching for changes.

To make a file tracked, you need to add it to the staging area

## What is the Staging Environment?

The **staging environment** (or **staging area**) is like a waiting room for your changes.

You use it to tell Git exactly which files you want to include in your next commit.

This gives you control over what goes into your project history.

Here are some key commands for staging:

* git add <file> - Stage a file
* git add --all or git add -A - Stage all changes
* git status - See what is staged
* git restore --staged <file> - Unstage a file

## Stage a File with git add

To add a file to the staging area, use git add <file>:

>git add index.html

## Stage Multiple Files (git add --all, git add -A)

You can stage all changes (new, modified, and deleted files) at once:

>git add -–all

## How to Unstage a File

If you staged a file by mistake, you can remove it from the staging area (unstage it) with:

>git restore --staged index.html

Now index.html is no longer staged. You can also use git reset HEAD index.html for the same effect.

## What is a Commit?

A **commit** is like a save point in your project.

It records a snapshot of your files at a certain time, with a message describing what changed

Here are some key commands for commits:

* git commit -m "message" - Commit staged changes with a message
* git commit -a -m "message" - Commit all tracked changes (skip staging)
* git log - See commit history

## Other Useful Commit Options

* **Create an empty commit:**  
  git commit --allow-empty -m "Start project"
* **Use previous commit message (no editor):**  
  git commit --no-edit
* **Quickly add staged changes to last commit, keep message:**  
  git commit --amend --no-edit

## View Commit History (git log)

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

>git log

## How to Commit with a Message (-m)

To save your staged changes, use git commit -m "your message":

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

## Commit All Changes Without Staging (-a)

You can skip the staging step for **already tracked files** with git commit -a -m "message".

This commits all modified and deleted files, **but not new/untracked files**.

**>** git commit -a -m "Quick update to README"

## Other Useful Commit Options

* **Create an empty commit:**  
  git commit --allow-empty -m "Start project"
* **Use previous commit message (no editor):**  
  git commit --no-edit
* **Quickly add staged changes to last commit, keep message:**  
  git commit --amend --no-edit

## View Commit History (git log)

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

>git log

# Git Tagging

## Key Commands for Tagging

* git tag <tagname> - Create a lightweight tag
* git tag -a <tagname> -m "message" - Create an annotated tag
* git tag <tagname> <commit-hash> - Tag a specific commit
* git tag - List tags
* git show <tagname> - Show tag details

## What is a Tag?

A **tag** in Git is like a label or bookmark for a specific commit.

Tags are most often used to mark important points in your project history, like releases (v1.0 or v2.0).

Tags are a simple and reliable way to keep track of versions and share them with your team or users.

Some common tag types include:

* **Releases:** Tags let you mark when your project is ready for release, so you (and others) can always find that exact version later.
* **Milestones:** Use tags to highlight major milestones, like when a big feature is finished or a bug is fixed.
* **Deployment:** Many deployment tools use tags to know which version of your code to deploy.
* **Hotfixes:** If you need to fix an old version, tags make it easy to check out and patch the right code.

## Create a Lightweight Tag

A lightweight tag is just a name for a commit.

It's quick and simple, but does not store extra information.

### Annotated vs Lightweight Tags

* **Annotated Tag:** Stores author, date, and message.  
  Recommended for releases and sharing with others.
* **Lightweight Tag:** Just a simple name for a commit (no extra info, like a bookmark).

## Create a Lightweight Tag

A lightweight tag is just a name for a commit.

It's quick and simple, but does not store extra information.

### Annotated vs Lightweight Tags

* **Annotated Tag:** Stores author, date, and message.  
  Recommended for releases and sharing with others.
* **Lightweight Tag:** Just a simple name for a commit (no extra info, like a bookmark).

# >git tag v1.0

## Create an Annotated Tag (-a -m)

An annotated tag stores your name, the date, and a message.

This is recommended for most uses.

>git tag -a v1.0 -m "Version 1.0 release"

## List Tags

See all tags in your repository:

git tag

## Show Tag Details (git show)

See details about a tag and the commit it points to:

git show v1.0

## Push Tags to Remote

By default, tags exist only on your local computer.

If you want others to see your tags, you need to push them to your remote repository.

If you don't push your tags, only you will see them, and only locally.

* To push a single tag to your remote repository (for example, after creating a release tag):

### Example: Push a Single Tag

git push origin v1.0

* To push ****all**** your local tags to the remote at once (useful if you've created several tags):

### Example: Push All Tags

git push –-tags

## Delete Tags

Delete a tag locally:

> git tag -d v1.0

Delete a tag from the remote repository:

>git push origin --delete tag v1.0

## Update or Replace a Tag (Force Push)

If you need to move a tag to a different commit and update the remote, use --force:

git tag -f v1.0 <new-commit-hash>

git push --force origin v1.0

# **Git Stash**

## Key Commands for Stashing

* git stash - Stash your changes
* git stash push -m "message" - Stash with a message
* git stash list - List all stashes
* git stash branch <branchname> - Create a branch from a stash

## What is Git Stash? Why Use It?

Sometimes you need to quickly switch tasks or fix a bug, but you're not ready to commit your work.

git stash lets you save your uncommitted changes and return to a clean working directory.

You can come back and restore your changes later.

Here are some common use cases:

* **Switch branches safely:** Save your work before changing branches.
* **Handle emergencies:** Stash your work to fix something urgent, then restore it.
* **Keep your work-in-progress safe:** Avoid messy commits or losing changes.

## Stash Your Changes (git stash)

Save your current changes (both staged and unstaged tracked files) with:

**>** git stash

This command saves your changes and cleans your working directory so you can safely switch tasks or branches.

Your changes are now saved in a stack.

## Stash with a Message (git stash push -m)

Add a message to remember what you stashed:

### Example: Stash with a Message

git stash push -m "WIP: homepage redesign"

## List All Stashes (git stash list)

See all your saved stashes:

### Example: List Stashes

git stash list

## Show Stash Details (git stash show)

See what was changed in the latest stash:

### Example: Show Latest Stash

git stash show

This command gives a summary of what files and changes are in your most recent stash.

To see a full diff:

### Example: Show Full Diff

git stash show -p

This command shows the exact lines that were changed in your most recent stash.

## Apply the Latest Stash (git stash apply)

Restore your most recent stashed changes (keeps the stash in the stack):

### Example: Apply Latest Stash

git stash apply

## Apply a Specific Stash (git stash apply stash@{n})

Restore a specific stash from the list:

### Example: Apply a Specific Stash

git stash apply stash@{1}

This command lets you restore a specific stash from your list, not just the most recent one.

## Pop the Stash (git stash pop)

Apply the latest stash **and remove it from the stack**:

### Example: Pop the Stash

git stash pop

## Drop a Stash (git stash drop)

Delete a specific stash when you no longer need it:

### Example: Drop a Stash

git stash drop stash@{0}

This command deletes a specific stash from your list when you no longer need it.

## Clear All Stashes (git stash clear)

Delete all your stashes at once:

### Example: Clear All Stashes

git stash clear

## Branch from a Stash (git stash branch)

Create a new branch and apply a stash to it.

Useful if your stashed work should become its own feature branch:

### Example: Branch from a Stash

git stash branch new-feature stash@{0}

# **Git History**

## What is Git History? Why Use It?

Git keeps a detailed record of every change made to your project.

You can use history commands to see what changed, when, and who made the change.

This is useful for tracking progress, finding bugs, and understanding your project's evolution.

## Key Commands for Viewing History

* git log - Show full commit history
* git log --oneline - Show a summary of commits
* git show <commit> - Show details of a specific commit
* git diff - See unstaged changes
* git diff --staged - See staged changes

## See Commit History (git log)

Show a detailed list of all commits in your repository:

### Example: Full Commit History

git log

## Show Commit Details (git show <commit>)

See all the details and changes for a specific commit:

### Example: Show Commit Details

git show 09f4acd

## Compare Changes (git diff)

See what is different between your working directory and the last commit (unstaged changes):

### Example: See Unstaged Changes

git diff

## Compare Staged Changes (git diff --staged)

See what is different between your staged files and the last commit:

### Example: See Staged Changes

git diff --staged

## Compare Two Commits (git diff <commit1> <commit2>)

See what changed between any two commits:

### Example: Compare Two Commits

git diff 1234567 89abcde

## Show a Summary of Commits (git log --oneline)

Show a short summary of each commit (great for a quick overview):

### Example: Oneline Log

git log --oneline

## Show Commits by Author (git log --author="Alice")

See only the commits made by a specific author:

### Example: Commits by Author

git log --author="Alice"

## Show Recent Commits (git log --since="2 weeks ago")

See only commits made in the last two weeks:

### Example: Recent Commits

git log --since="2 weeks ago"

## Show Files Changed Per Commit (git log --stat)

See which files were changed in each commit and how many lines were added or removed:

### Example: Log with Stats

git log --stat

## Show a Branch Graph (git log --graph)

See a simple ASCII graph of your branch history (great for visualizing merges):

### Example: Log with Graph

git log --graph --oneline

# **Git Help**

## Why and When to Use Git Help?

Git has many commands and options.

If you forget how a command works or want to learn about its options, you can use Git's built-in help.

This is the fastest way to get answers without leaving your terminal.

## Key Commands for Getting Help

* git help <command> - See the manual page for a command
* git <command> --help - See help for a command (same as above)
* git <command> -h - See a quick summary of options
* git help --all - List all possible Git commands
* git help -g - List guides and concepts

## See Help for a Specific Command (git help <command>)

Shows the full manual page for a specific command, including all options and examples:

### Example: See Help for Commit

git help commit

## See Help with --help (git <command> --help)

This does the same as git help <command>. Most users prefer this form:

### Example: See Help for Status

git status --help

## See a Quick Summary with -h (git <command> -h)

Shows a short summary of the command's options, right in the terminal window (does not open the full manual):

### Example: Quick Help for Add

git add -h

## List All Git Commands (git help --all)

Lists every Git command available on your system, grouped by category:

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

### Example

$ git help --all

## List Guides and Concepts (git help -g)

Shows a list of guides and concept topics for deeper learning:

### Example: List Guides and Concepts

git help -g

# **Git Branch**

## What is a Git Branch?

In Git, a branch is like a separate workspace where you can make changes and try new ideas without affecting the main project. Think of it as a "parallel universe" for your code.

## Why Use Branches?

Branches let you work on different parts of a project, like new features or bug fixes, without interfering with the main branch.

## Creating a New Branch

Let's say you want to add a new feature. You can create a new branch for it.

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"

## Listing All Branches

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

To see all branches in your repository, use:

### Example

git branch

## Switching Between Branches

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

## 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

## Deleting a Branch

When you're done with a branch, you can delete it:

### Example

git branch -d hello-world-images

**Rename a branch:** git branch -m old-name new-name

# **Git Branch Merge**

## What is Merging in Git?

Merging in Git means combining the changes from one branch into another.

This is how you bring your work together after working separately on different features or bug fixes.

## Common git merge Options

* [git merge](https://www.w3schools.com/git/git_branch_merge.asp?remote=github#merge) - Merge a branch into your current branch
* [git merge --no-ff](https://www.w3schools.com/git/git_branch_merge.asp?remote=github#no-ff) - Always create a merge commit
* [git merge --squash](https://www.w3schools.com/git/git_branch_merge.asp?remote=github#squash) - Combine changes into a single commit
* [git merge --abort](https://www.w3schools.com/git/git_branch_merge.asp?remote=github#abort) - Abort a merge in progress

## Merging Branches (git merge)

To combine the changes from one branch into another, use git merge.

Usually, you first switch to the branch you want to merge into (often main or master), then run the merge command with the branch name you want to combine in.

First, we need to change to the master branch:

### Example

git checkout master

Switched to branch 'master'

Now we merge the current branch (master) with emergency-fix:

### Example

git merge emergency-fix

## Non-Fast-Forward Merge (git merge --no-ff)

By default, if your branch can be merged with a fast-forward (no new commits on the base), Git just moves the branch pointer forward.

If you want to always create a merge commit (to keep history clearer), use git merge --no-ff branchname.

### Example

git merge --no-ff feature-branch

## Squash Merge (git merge --squash)

If you want to combine all the changes from a branch into a single commit (instead of keeping every commit), use git merge --squash branchname.

This is useful for cleaning up commit history before merging.

### Example

git merge --squash feature-branch

## Aborting a Merge (git merge --abort)

If you run into trouble during a merge (like a conflict you don't want to resolve), you can cancel the merge and go back to how things were before with git merge --abort.

### Example

git merge --abort

## What is a Merge Conflict?

A merge conflict happens when changes in two branches touch the same part of a file and Git doesn't know which version to keep.

Think of it like two people editing the same sentence in a document in different ways—Git needs your help to decide which version to use.

## How to Resolve a Merge Conflict

Git will mark the conflict in your file.

You need to open the file, look for lines like <<<<<<< HEAD and =======, and decide what the final version should be.

Then, stage and commit your changes.

# Remove the hello-you branch from the local repository:

git branch –g hello-you

# **Git Workflow**

## Git Workflow Commands Overview

* [Working Directory](https://www.w3schools.com/git/git_workflow.asp?remote=github#working-directory) - Where you make changes
* [git add](https://www.w3schools.com/git/git_workflow.asp?remote=github#git-add) - Stage changes
* [git commit](https://www.w3schools.com/git/git_workflow.asp?remote=github#git-commit) - Save changes to your repository
* [git push](https://www.w3schools.com/git/git_workflow.asp?remote=github#git-push) - Share changes with others
* [git status](https://www.w3schools.com/git/git_workflow.asp?remote=github#git-status) - Check what's going on
* [Undo/Amend](https://www.w3schools.com/git/git_workflow.asp?remote=github#undo) - Fix mistakes (git restore, git reset, git commit --amend)

### The Three Areas of Git

* **Working Directory**: Where you make changes to your files.
* **Staging Area (Index)**: Where you prepare changes before committing.
* **Repository**: Where your committed history is stored.

### Workflow Diagram

[Working Directory] --git add--> [Staging Area] --git commit--> [Repository]

## Undoing and Amending Changes

Made a mistake? Git lets you fix things before you push!

* git restore <file> - Undo changes in your working directory (before staging).
* git restore --staged <file> - Unstage a file (move it out of the Staging Area).
* git reset HEAD~ - Undo your last commit (keeps changes in your working directory).
* git commit --amend - Change the last commit message or add files to your last commit.

### Example: Unstage a file

git restore --staged index.html

## Pull Before You Push

Always git pull before pushing.

This updates your local branch with changes from others, helps you avoid conflicts, and ensures your push will succeed.

**Why?** If someone else has pushed changes since your last pull, your push may be rejected or cause conflicts.

Pulling first lets you fix any issues locally.

### Example

git pull origin main

git push origin main

## Clone

Create a local copy of a remote repository.

[Learn more on our Git Clone page](https://www.w3schools.com/git/git_clone.asp)

### Example

git clone https://github.com/user/repo.git