My Own Git Book 📚🔧

MY GIT & GITHUB JOURNEY

Every expert was once a beginner. As I dive deeper into the world of Git, I embrace every merge, commit, and branch as opportunities to grow. Each command I learn builds my confidence and brings me closer to mastering this powerful version control system. This journey is not just about learning Git — it's about developing persistence, problem-solving skills, and a mindset for continuous growth. I am proud of every step I take on this path.

WHAT IS GIT

Git is a tool that helps you save different versions of your files while you work on a project. It keeps track of all the changes you make, so you can see what was changed, go back to older versions if needed, and work with other people on the same files without confusion. Git is very popular because it is fast, reliable, and works well for small and big projects.

WHAT IS GITHUB

GitHub is a website where you can save your Git projects online. It helps you share your work with other people, work together on the same project, and keep your code safe. GitHub makes it easy to work with friends or teams because it shows all changes clearly and helps you discuss ideas.

INSTALLING GIT ON YOUR SYSTEM

First, open your web browser and go to the official Git website. On the homepage, click the download button. The website will automatically start downloading the correct Git version for your Windows OS.

When the download finishes, open the installer file to start the installation. Follow the instructions on the screen carefully. For most users, it is best to use the default settings by clicking "Next" until the installation is complete.

To check if Git is installed, open PowerShell or CMD and type.

```
git --version
```

INSTALLING GIT ON LINUX

First, open your terminal on your Linux computer. Most Linux systems use package managers to install software easily, If you use Ubuntu or Debian, type the following command and press Enter to install Git:

```
sudo apt update && sudo apt install git -y
```

If you use CentOS or RHEL, type this command and press Enter:

```
sudo yum install git -y
```

If you use Fedora, type this command and press Enter:

```
sudo dnf install git -y
```

After the installation is complete, check if Git is installed correctly by typing this command and pressing Enter:

```
git --version
```

INSTALLING GIT ON MACOS

If Git is not already installed, you can install it using the Homebrew package manager. If you don't have Homebrew installed , go to and follow the instructions to install it.

Once Homebrew is installed, type the following command in Terminal and press Enter to install Git:

```
brew install git
```

After the installation is complete, check if Git is installed correctly by typing this command and pressing Enter:

```
git --version
```

Git is usually already installed on most Linux distro and macOS systems; if it is not, you can follow the previous pages to learn how to install it.

CREATING A GITHUB ACCOUNT

To create a GitHub account, go to **github.com** and click the "Sign up" button on the top right. Enter a username, your email, and a strong password. Choose the free plan and then check your email to verify the account. After verification, you can log in and start using GitHub to store and share your projects.

USING SSH KEYS FOR GITHUB AUTHENTICATION

SSH keys are a pair of cryptographic keys used for secure access and authentication between a client and a server over the Secure Shell (SSH) protocol. Instead of using passwords, SSH keys provide a more secure and convenient way to log into remote systems.

create a new SSH key pair (use your email for identification):

```
ssh-keygen -t ed25519 -C "ahmedsahal@gmail.com"
```

When you create the SSH key using ssh-keygen, it will ask you to enter a passphrase. You can Enter a secure passphrase (recommended for better safety), or Leave it empty (press Enter) if you don’t want a passphrase.

To connect your SSH key to GitHub, copy your public key, then go to your GitHub account settings, add a new SSH key by pasting it there, and finally test the connection using this command to make sure it works.

```
ssh -T git@github.com
```

CREATING A GITHUB REPOSITORY

To make a GitHub repository, log in to your GitHub account and click the "+" icon at the top right, then choose "New repository." Give your repo a name and set it to public or private. Then click "Create repository" to finish.

If your project has already started locally, you can initialize the GitHub repo with a README.md, .gitignore and LICENSE file, then pull from the remote repo for the first time to sync with your local repo and avoid conflicts.

INITIALIZING GIT IN A PROJECT

Initializing Git means starting Git to track changes in your project folder. When you initialize Git, your project folder becomes a Git repository. This lets you save versions of your files and work with Git commands.

Initializes your project folder to start tracking your files with Git.

```
git init
```

CONNECTING A LOCAL PROJECT TO GITHUB

After creating the GitHub repository, you usually want to connect your local project folder from your computer to this GitHub repository. To do this, you use the git remote add command like this.

```
git remote add origin git@github.com:cyberhappy/Dev.git
```

If you use SSH keys, it is best to connect to your GitHub repository with the SSH URL because it is more secure and does not require you to enter your password every time.

Sometimes, you want to check which URL your local project uses to connect to GitHub—this helps you know if it's using SSH or HTTPS; to do this, use the following command to see the remote repository URL.

```
git remote -v
```

CONFIGURING GIT USER INFORMATION

Before using Git, you need to set your user name and email. This information is used to identify who makes changes in the project.

Git commands to configure your name and email:

```
git config --global user.name "Ahmed Sahal"
git config --global user.email "ahmedsahal@gmail.com"
```

SYNCING YOUR LOCAL PROJECT WITH THE REMOTE REPO

The remote repository might already have some important first files like **README.md**, **LICENSE**, and **.gitignore**. Now, we need to pull these initial commits to your local computer to make sure your local project and the remote repository are the same (in sync) This helps keep your work updated and can prevent conflicts when you make changes later.

Get the latest changes from the remote repository to your local repo.

```
git pull origin main
```

Sometimes, your remote repository on GitHub has a branch called **main**, but your local repo has a branch called **master**. If you try to push changes, this difference can cause conflicts. To avoid these problems, you should rename your local master branch to main so it matches the remote branch.

renames the existing branch named "master" to "main" in your Git repo.

```
git branch -m master main
```

CHECKING THE CURRENT STATE OF YOUR REPOSITORY

When working with Git, it's important to know what is happening in your repository at any moment. You can check which files have been changed, which are staged (ready to be committed), and which files are untracked (new files not yet added to Git).

This helps you understand what you have done and what you need to do next, like committing or adding files. to check the state use this command

```
git status
```

Understanding the output of git status is an important first step to managing your files correctly. This helps you prepare your changes carefully before saving or sharing them with others.

PREPARING FILES FOR COMMIT (STAGING AREA)

Before you can save your changes to Git, you need to tell Git which changes you want to include in the next commit. This is called adding changes to the staging area. The staging area is like a “waiting room” where you prepare your changes before committing them to the project history.

adds all changed files in the current folder to the staging area, preparing them for commit.

```
git add .
```

Add one specific file to the staging area by running:

```
git add filename.ext
```

SAVING CHANGES TO THE PROJECT HISTORY

After you have added your changes to the staging area using git add, you need to commit these changes. A commit saves a snapshot of your current staged files to the project’s history. This makes a permanent record of your changes with a message explaining what was done.

Save your staged changes to the project history with a clear message.

```
git commit -m "Your commit message here"
```

SHARING YOUR CHANGES WITH THE REMOTE REPOSITORY

After you commit your changes locally, you need to send these changes to the remote repository (like GitHub) so others can see and access them. This action is called pushing.

Send your committed changes to the remote repository's main branch

```
git push origin main
```

COPYING A REMOTE REPOSITORY TO YOUR COMPUTER

Cloning a repository means making a copy of a project from GitHub (or another remote server) to your own computer. This allows you to work on the files, make changes, and later send your updates back to the original project if you want.

When you clone a repository, you get all the project files and its full history, meaning you can see all the past changes made to the project.

Copy complete project files and history from a remote repo to your local

```
git clone git@github.com:cyberhappy/Dev-Example.git
```

VIEWING THE HISTORY OF PROJECT CHANGES

In Git, the commit history shows a list of all the changes made to a project over time. Each commit has information like the author, date, and message describing the change. Viewing the commit history helps you understand what changes were made and when.

displays the full commit history in the current branch.

```
git log
```

CHANGING THE DEFAULT TEXT EDITOR USED BY GIT

Git uses a text editor when it needs you to write messages, like commit messages. By default, Git usually uses **Vim** as the editor, but many people find Vim hard to use if they are not familiar with it.

use the git config command to set the editor for Git.

```
git config --global core.editor "code –wait"
```

After running git config, open VS Code, press Ctrl+Shift+P, type code, and select “Shell Command: Install 'code' command in PATH” to add the code command to your system path.

EXCLUDING FILES FROM VERSION CONTROL

When working on a project with Git, there are sometimes files or folders you do not want to track or upload to GitHub. These include temporary files created by your computer, compiled files (such as .deb or .exe), secret files containing passwords or API keys, and large files that do not need to be shared.

To tell Git to ignore these files, you create a special file called **.gitignore** in your project. Inside .gitignore, you write the names or patterns of files and folders to exclude.

If you want to see what's ignored based on .gitignore, use:

```
git status --ignored
```

UNDERSTANDING THE GLOBAL GIT CONFIGURATION FILE

The Git global configuration file is usually located in your home directory and is named .gitconfig. This file stores your global Git settings, like your name, email and your default editor, which are used for all Git projects on your computer.

On Linux and macOS, it is usually at: /home/yourusername/.gitconfig.
On Windows, it is typically at: C:\Users\YourUsername\.gitconfig.

To check all global Git configuration settings, use this command

```
git config --global --list
```

To open or edit the Git global config file, use this command:

```
git config --global --edit
```

You can directly open and edit the **.gitconfig** file to add your email address, user name, default editor, signing key, and other settings if you want.

UNDERSTANDING THE INTERNAL STRUCTURE OF A GIT REPO

When you create a Git repository by running **git init**, Git creates a hidden folder called **.git** inside your project directory. This .git folder is very important because it stores all the information about your repository.

The .git directory contains all the data Git needs to track your project's history, branches, commits, and configuration. It includes:

- ❖ HEAD: Points to the current branch you are on.
- ❖ config: Stores repository-specific settings.
- ❖ objects/: Contains all the data Git stores, like your files and commits, saved as objects.
- ❖ refs/: Stores references to commits for branches and tags.
- ❖ logs/: Keeps a log of changes to references, like branch updates.

In summary, **.git directory** is the heart of your Git repository because it stores all Git data and history, so be careful not to delete or change .git files unless you know what you are doing.

REMOVING FILES FROM THE STAGING AREA SAFELY

When you modify a file and run git add, you stage the file, which prepares it to be included in the next commit. If you stage a file by mistake or want to keep your changes but remove it from staging,you can un-stage it—this clears the staging area while preserving your edits.

removes index.php from the staging area but keeps your edits.

```
git restore --staged index.php
```

Restoring a staged file back means removing the file from the staging area while keeping changes in the working directory, and you do this with the command git restore --staged <filename>.

UNDERSTANDING BRANCHES IN GIT

In Git, a branch is like a separate workspace or path where you can make changes to your project without affecting the main work. Imagine you are writing a book: the main story is your "main" branch, but you want to try a new idea. You create a branch and write your new idea there. If you like it, you can add (merge) it back into the main story. If not, you can ignore it, and the main story stays the same.

Branches help you work on new features, fix bugs, or experiment safely while keeping the main project clean.

list all branches in your Git project using this command:

```
git branch
```

create bugfix branch but stay on your current branch, use:

```
git branch bugfix
```

switch your current work to the existing branch named bugfix:

```
git checkout bugfix
```

COMBINING CHANGES USING GIT MERGE

When you work on a project, sometimes you create a separate branch to add new features or fix bugs without changing the main code. After finishing your work in that branch, you need to put (merge) those changes back into the main branch (usually called **main** or **master**). This process is called "**merging branches**."

HOW TO MERGE BRANCHES IN GIT?

First, switch to the branch where you want to add the changes (for example, main) then, merge the other branch (for example, feature) into it.

```
git checkout main && git merge feature
```

CLEANING UP UNUSED BRANCHES

Sometimes, after you finish working on a branch, that branch is no longer needed. To keep your project clean and organized, you can delete these unused branches.

There are two types of branches you might want to delete:
1. Local branches: These are branches on your computer.
2. Remote branches: These are branches stored on GitHub or gitlab.

To delete a branch on your computer | local branch, use this command:

```
git branch -d feature
```

This command is safe — it won't delete the branch if you have unmerged changes If you want to force delete the branch (even if it has unmerged changes), use:

```
git branch -D feature
```

To delete a branch from GitHub (or another remote), use this command:

```
git push origin --delete branch-name
```

RENAMING BRANCHES IN GIT

Sometimes, you may want to change the name of your branch to better describe what you are working on. You can rename your current branch or another branch easily.

To rename the current branch you are on, use:

```
git branch -m feature-update
```

To rename a branch you are not currently on, use:

```
git branch -m fixes bugfix bug-fix
```

RESOLVING MERGE CONFLICTS

When working with Git, a merge conflict happens when Git tries to combine changes from two different branches, but some changes overlap and Git does not know which one to keep.

For example, imagine two people edited the same line in the same file differently. When you try to merge their changes, Git cannot automatically decide which change is correct. This situation is called a merge conflict.

Git will stop the merge and mark the file as conflicted. You need to open the file, find the conflicting parts (shown with <<<<<<<, =======, and >>>>>>>), and decide which code to keep:

- The code between <<<<<<< HEAD and ======= is your current branch's changes.
- The code between ======= and >>>>>>> branch-name is the other branch's changes.

start a merge (which may produce conflicts):

```
git merge feature
```

To resolve a merge conflict, first run to merge the branches,

```
git merge branch-name
```

then open the conflicted files and fix the conflicts manually by choosing the correct code and saving the files.

Afterward, add the fixed files using git add filename,

```
git add filename
```

finally, finish the merge with a commit using

```
git commit -m "Your commit message"
```

PUBLISHING LOCAL BRANCHES TO A REMOTE REPOSITORY

Pushing means sending your local changes from your computer to the remote repository (like GitHub). When you push a local branch, you upload your work so others can see it or so it is saved online. This is useful when you finish some work on a branch and want to share it or back it up.

Push local "feature" branch to the remote repository named "origin"

```
git push origin feature
```

UPDATING YOUR LOCAL BRANCH WITH REMOTE CHANGES

When you work with a team or store your project on GitHub, changes can be made by others on the remote (online) version of your project. To get the latest changes from the remote branch, you use the "git pull" command.

"Pull" means: download the changes from the remote branch and automatically merge them into your local copy of the branch you are working on. This helps to keep your work up-to-date with others' work.

Download remote branch bugfix to your local repo.

```
git pull origin bugfix
```

WORKING WITH TRACKING BRANCHES

A tracking branch is a local Git branch that has a connection (or link) to a branch in a remote repository (like GitHub). This connection helps Git know which remote branch your local branch should “push” changes to and “pull” changes from.

creates a local branch that tracks the remote branch named bugfix.)

```
git checkout --track origin/bugfix
```

If you want to see all your tracking branches, you can use:

```
git branch -vv
```

COMPARING CHANGES USING GIT DIFF

git diff shows you the differences between files you have changed but not yet saved (staged) for your next commit. It helps you see what changes you made in your working folder compared to the last saved version.

For example if you changed a file but haven't saved these changes to Git, git diff will show exactly what lines were added, changed, or removed.

shows the changes in your working directory that are staged for commit.

```
git diff --staged
```

Also you can see the differences between **two commits** by specifying their commit IDs (hashes). This helps you understand what changed from one commit to another.

To compare commit 9fd003d with commit 5dab5ae, use this command:

```
git diff 9fd003d 5dab5ae
```

git diff can also show you the differences between two branches This helps you understand what changes exist in one branch compared to another.

To see the differences between branch main and branch feature, use:

```
git diff main feature
```

We can also quickly see which files changed in a commit without showing the details by using git diff which lists only the names of the changed files and ignores the actual content changes.

Show only names of changed files (not full diffs):

```
git diff --name-only 5dab5ae
```

STASHING YOUR WORK IN GIT

Sometimes when you are working on a project, you have some changes in your files, but you need to switch to another task or branch without committing those changes. git stash lets you save your current changes temporarily so you can work on something else. Later, you can bring those changes back and continue working. It's like putting your changes in a short-term storage box and taking them out when you need them again.

Save your current changes without committing:

```
git stash
```

When you save changes using git stash, Git keeps those changes in a list. You can see all your saved (stashed) changes by using the command below. This helps you know what you saved and when.

```
git stash list
```

After you save your changes using git stash, you can bring them back anytime to continue working. This is called **applying** or **popping** the stash.

Bring back stashed changes but keep them in stash:

```
git stash apply
```

Bring back stashed changes and remove them from stash:

```
git stash pop
```

CHECKOUT THE COMMIT (DETACHED HEAD STATE)

When you use Git to move to a specific commit by its ID (hash), Git switches your working files to that point in history. This is called checking out a commit.

Normally, your HEAD points to a branch (main or feature). But if you checkout a specific commit (not a branch), Git puts you in a detached HEAD state. This means:

- ❖ You are no longer on a branch.
- ❖ You can look around or make changes, but if you make commits here, they are not connected to any branch and can be lost if you switch branches.
- ❖ If you want to keep new commits, you should create a new branch.

To checkout a commit and enter a detached HEAD state, use:

```
git checkout 161cb28
```

GO BACK TO YOUR PREVIOUS STATUS

When you are in a detached HEAD state after checking out a commit directly, you may want to go back to the state of your branch (like main or master), where your last commit is.

To do this, you just checkout your branch again. This moves the HEAD pointer back to your branch tip, and you return to your latest checked-in work on that branch.

go back to your branch (example with branch main (Your Last Commit)

```
git checkout main
```

UNDOING CHANGES SAFELY GIT REVERT

Imagine you are working on a project and you made some changes (a commit) that caused problems. You want to go back to the previous version without losing all your history. Instead of deleting the bad changes, you create a new change that cancels out the bad ones. This way, your project moves forward, but the bad change is undone.

In Git, git revert creates a new commit that "goes back" to the previous version by undoing the changes from a specific commit, without removing any history.

Replace commit-hash with the ID of the commit you want to undo.

```
git revert 5dab5ae
```

SETTING UPSTREAM IN GIT

When you work with branches in Git, sometimes you want your local branch to "track" a branch on a remote (like GitHub). This means your local branch will know which remote branch to push changes to or pull updates from by default.

push your local main branch to the remote origin and set upstream tracking

```
git push -u origin main
```

This command does two things: it pushes your branch to the remote called origin, and it sets the remote branch as the upstream for your local branch.

PULL REQUEST PROCESS

A Pull Request (PR) is a way to ask a project owner or team to review and add your changes to their project. It is used mainly on platforms like GitHub. Here's how it works in easy steps:

- ❖ Fork or clone the repository: You copy the project's code to your own space or computer.
- ❖ Create a new branch: Make a separate branch for your new changes. This helps keep your work organized.
- ❖ Make your changes: Write code, fix bugs, or add new features on that branch.
- ❖ Push your branch to GitHub: Upload your changes to your copy on GitHub.
- ❖ Open a PR: Ask to merge your changes into the main project Done.

COMMON GIT MISTAKES & HOW TO FIX THEM

When using Git, beginners and even experienced users make mistakes; this guide helps you understand the most common errors and shows how to fix them easily.

Forgot to add files before commit: Sometimes you make changes to files but forget to use git add to include them in the commit, so the commit does not save those changes.

Commit message is not clear: Writing unclear or vague commit messages makes it hard to understand the history of the project later.

Committed to the wrong branch: At times, you accidentally commit your changes on the wrong branch instead of the one you intended.

Merge conflict: When two people edit the same part of a file, Git cannot automatically merge the changes, causing a conflict that needs to be fixed manually.

Deleted something by mistake: Sometimes you delete a file or commit by accident but don’t need to worry because Git can help you restore it.

Pushed wrong code to GitHub: Occasionally, you push code that is incomplete or wrong, but you can fix this by updating your code and pushing again or by undoing bad commits.

Forgot to pull before pushing: Sometimes you try to push your changes to GitHub without first pulling new changes from the remote repository, which causes errors. To fix this, always use git pull before pushing to update your local copy.

Accidentally staging unwanted files: Sometimes you add files to the staging area that you did not mean to include in the commit. To fix this, use git reset <file> to unstage those files before committing.

Complete Git Workflow (Start → Finish)

This chapter shows a **real-life Git workflow** from beginning to end.If you ever forget *how everything connects*, come back to this page.

First, get the project from GitHub This creates a local copy of the project.

```
git clone git@github.com:username/project.git
```

Before working, always sync with the remote repository This prevents conflicts later.

```
git pull
```

Create a separate branch for the new feature This keeps your work isolated and safe.

```
git checkout -b feature-new-function
```

Edit files using your editor and Check what has changed:

```
git status
```

Add the modified files to the staging area.

```
git add .
```

Create a commit with a clear message This records your work in Git history.

```
git commit -m "Add new function to improve feature"
```

Push your branch to GitHub Now others can see your work.

```
git push -u origin feature-new-function
```

After review, switch to the main branch: Merge your feature:

```
git merge feature-new-function
```

After merging, delete the branch:

```
git branch -d feature-new-function
```

(Optional) Delete it from the remote:

```
git push origin --delete feature-new-function
```

This Complete Git Workflow starts by cloning a project from GitHub to your computer. You create a new branch to work on changes separately.

After making and staging your changes, you commit them with a clear message. Before pushing, you pull the latest updates to avoid conflicts. Then, you push your branch to GitHub and open a Pull Request for review.

Once your code is approved, it is merged into the main branch. Finally, you delete your branch and update your local main branch with the latest changes.