GIT & GITHUB

COMPLETE HANDWRITTEN NOTES

Prepared by

SAFIYA FATIMA

LinkedIn : https://www.linkedin.com/in/safiya-fatima-534561328

TABLE OF CONTENTS

13. General Workflow when working with Local Git
14. Local vs Remote (in Git & GitHub)
15. Git Branches
16. Merging code
    - Merging methods (1 & 2)
    - What is Pull Request (PR)?
17. Pull Command
18. Resolving Merge Conflicts
    - Merge Conflicts
    - Resolving merge conflicts
19. Undoing Changes
    - Meaning
    - Why do we undo changes ?
    - Three types of Changes in Git
20. Git Log command & Commit Hash
    - Meanings
    - Why do we need commit hash ?
    - How to get the hash ?
21. Fork
    - Meaning
    - Why do we use Fork ?

WHAT IS GIT ?

→ GIT is a Version Control System.
[Version Control System is a tool that helps to track changes in code].

REASONS TO USE GIT ?

→ GIT is one of the most popular Version Control System (VCS)

→ free & Open Source

→ fast & scalable

GIT IS PRIMARILY USED FOR 2 MAIN REASONS (PURPOSES) :—

1) To track the history of our code.

2) To Collaborate (to work with a team)

In simple words:-

→ Git is a tool - basically a software that runs on our computer.

→ We use it to track changes in our files, manage versions, and work smoothly with others on projects.

WHAT IS GITHUB ?

→ GITHUB is a website that allows developers to store and manage their code using Git.

→ https://github.com

↳ make your github profile with this link

→ AFTER CREATING YOUR GITHUB ACCOUNT:-

- Create a new repository
- Make your first commit

TO SET UP GIT :-

(1) Install Visual Studio Code (VS Code)

(It is one of the most popular free & open source code editor by Microsoft. Here, we can code in any language)

VS Code for Windows / Mac / Linux

↓

https://code.visualstudio.com/download

↳ Download VS Code for windows / Mac / Linux with this link

(2) For Windows - (Git Bash) :-

git for windows

↓

git-scm.com/install/windows

↳ Download Git & Git Bash for windows with this link

↓

After download, you will get the git bash icon on the desktop.

↓

Click & Open the Git Bash icon

↓

3. Write the command "git --version" to check the version of your git installed.

③ For Mac — (Terminal) :-

- **Option 1** — If Mac already has Git pre-installed

go to finder
↓
search "terminal"
↓
write the command "git --version"
↓
this will give you the git version of Git if it is pre installed

- **Option 2** — If Mac doesn't have Git pre-installed

Git for Mac
↓
git-scm.com/install/mac
↳ Download Git for Mac with this link

After Setup & Installation of Git, the next step is

CONFIGURING GIT :—

Configuring Git means telling Git who you are.

You are telling Git:

- Your name
- Your email ID

So that Git knows which user is making the changes and who wrote each commit.

WHY DO WE CONFIGURE GIT ?

Because every time you commit, Git needs to attach :

- "Who made this change ?" → Your name
- "Which account/email ?" → your email ID

This helps in :

- Identifying the author of the commit
- Connecting commits to your GitHub account
- Working in teams without confusion

NOTE :—

- Configuring Git does NOT log you into GitHub. It only tells your identity.

- To connect Git to GitHub, you use username + token (for HTTPS) or SSH Keys

COMMANDS TO CONFIGURE GIT :—

→ To set your username so that Git knows who is making the commits

COMMAND :~

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

↳ Here, use the name which you are using in your GitHub account

→ To set your email so Git knows who is making the commits.

COMMAND :~

```
git config --global user.email "your-email@email.com"
```

↳ Here, use the email which you are using in your GitHub account

→ To check whether your Git is configured correctly i.e to verify if your username & email were set correctly.

COMMAND :~

```
git config --list
```

↳ After using this command, it will show all stored git settings, including the username & email we configured

2 TYPES OF GIT CONFIGURATIONS MOST COMMONLY USED :—

(1) GLOBAL CONFIGURATION :— (most commonly used)

- Global configuration means : settings for your entire computer.
- Whatever username & email you set in global, Git will use those details for all repositories on your computer.
- You can set global config like this :

```
git config --global user.name "Your Name"
git config -- global user.email "Your-email@gmail.com"
```

WHEN TO USE GLOBAL ?

- When your name is the same for all your Git Projects
- When your email is the same for all your GitHub repositories
- When you want one fixed identity for everything

Most students use only global configuration.

② LOCAL CONFIGURATION :—
(second most used)

- Local Configuration means : settings for one specific project / repository only.
- These settings apply only to that single folder (repo), not the whole computer
- You set local config like this (run this inside that repository folder):

```
git config user.name "Your Name"
git config user.email "your-email@gmail.com"
```

WHEN TO USE LOCAL ?

- When you work on different projects with different identities
- Example :
  - One project uses your college email
  - Another project uses your personal email
- When you are contributing to a company repo using a company email

Local config always overrides (replaces) global config for that specific project.

Additional Configuration :-

→ SYSTEM CONFIGURATION
(it is rarely used by normal users. It is mostly used by companies or administrators)

CREDENTIAL HELPER IN GIT :-

① WHAT IS CREDENTIAL HELPER?

Credential Helper is a Git feature that stores your GitHub login details (username, password / token) so that Git does not ask you again and again when you push or pull

② WHY DO WE NEED CREDENTIAL HELPER?

→ To avoid typing username and token every time

→ To make pushing & pulling faster

→ To store credentials safely (depending on the method)

③ HOW DO WE USE CREDENTIAL HELPER?

We use Credential Helper by running one command in Git Bash / Terminal

Common Commands :-

a) Save temporarily (few minutes)

```
git config --global credential.helper cache
```

b) Save permanently (less secure)

```
git config --global credential.helper store
```

c) Save securely (best)

```
git config --global credential.helper
manager-core
```

After running any one of these:

1.) Next time you do a git push
2.) Git asks for your GitHub username + token
3.) You enter them one time only
4.) Credential Helper saves it
5.) Git will not ask again

(4) WHY DOES credential.helper APPEAR IN git config --list ?

When you type:

```
git config --list
```

Git shows all saved settings.
Credential Helper is also a Git setting, so it appears in the list:

Example :-

```
credential.helper = store
user.name = yourname
user.email = youremail@gmail.com
```

(5) RELATION BETWEEN git config --list AND credential.helper :-

→ git config --list shows all Git configurations.

→ Credential Helper is one of those configurations.

→ That's why it appears when you list all Git settings

(6) DEFAULT CREDENTIAL HELPER BEHAVIOR :-

→ Git may show credential.helper even if we never configured it.

→ This is because Git automatically sets a default credential helper depending on the operating system (OS)

→ Default helpers in different systems :-

- Windows → manager-core
- Mac → osxKeychain
- Linux → sometimes shows no helper (null)

→ That is why credential.helper appears in git config --list even if we never ran any credential helper command.

(7) DO WE NEED TO MANUALLY SET CREDENTIAL HELPER ?

or

WHEN SHOULD WE SET CREDENTIAL HELPER MANUALLY ?

→ If credential.helper already appears when we run git config --list, then no need to set it manually, because Git already has a default helper

→ If credential.helper does NOT appear in git config --list, then we should manually set a helper using a command like:

```
git config --global credential.helper store
```

or

```
git config --global credential.helper manager-core
```

SOME IMPORTANT GIT COMMANDS THAT CAN BE WRITTEN IN TERMINAL / CODE EDITOR (vs code) TO USE GIT IN OUR SYSTEM :-

① CLONE :- Cloning a repository on our local machine [computer].

- Basic Command to clone a Git repository is git clone followed by the repository's URL

```
git clone <repository-url>
```

② CD COMMAND :-

→ cd means Change Directory (folder)

→ The cd command is used to move into a folder from the terminal

→ We use it to go inside the project directory before running Git commands

→ Because Git Commands (like add, commit, push) only work inside the project folder. So, first we must enter the correct folder using cd.

- Go inside a folder :

```
cd foldername
```

- Go back one step (to previous folder):-

```
cd ..
```

- Go to Desktop (example):

```
cd Desktop
```

- Go to Downloads (example):

```
cd Downloads
```

- Go inside your project folder:

```
cd my-project
```

(3) HOW TO CHECK WHICH FOLDER WE ARE IN (CURRENT DIRECTORY)

In the terminal, the folder name shown on the left side of the % symbol (in Mac) or on the left side of $ (in Git Bash / Windows) tells us which folder we are currently inside.

EXAMPLES:—

(1) Mac Terminal :— `Desktop %`

This means → we are inside the Desktop folder

(2) Git Bash on Windows :— `~/Documents $`

This means → we are inside the Documents folder

④ TO CLEAR THE TERMINAL :—

- COMMAND :— `clear`
- SHORTCUT :— `ctrl + L`

Using this, the terminal becomes empty & gives a fresh black space.

⑤ ls COMMAND :—

The ls command is used to see all the files & folders present inside the current directory (folder).

It means "list files"

- COMMAND :— `ls`

Using this command we can see all the normal files but not the hidden files.

⑥ ls -a COMMAND :—

ls -a is used to see all files in the current folder, including hidden files.

- COMMAND :— `ls -a`

EXAMPLE OUTPUT :—

```
.   ..   .git   index.html   notes.txt
```

Here, in the above example output there is .git which means it is a GitHub file.

⑦ STATUS:- displays the state of the code.

- COMMAND:- `git status`

There are 4 TYPES of file status, when we use git status.

i, untracked (new file) → new files that git doesn't yet track.

ii, modified (changed file) → file in which you have made some changes

iii, staged (file is ready to commit) → file is ready to be committed

iv, unmodified (unchanged file) → file in which you have not made any changes. i.e everything is unchanged /unmodified in that file.

(8) ADD :— adds new or changed files in your working directory to the Git staging area.

```
git add <- file name ->
```

↳ use this to add a single file

```
git add .
```

↳ use this to add all the files

(9) COMMIT :— It is the record of change

```
git commit -m "some meaningful message"
```

(10) PUSH COMMAND :—

Push— upload local repo content to remote repo

```
git push origin main
```

(11) INIT COMMAND :—

init — used to create a new git repo

`git init` → used to create a new git repo in your current folder. It tells Git to start tracking the files in that folder.

`git remote add origin <-link->`

↳ This command connects your local project to a repo on GitHub

`git remote -v`

↳ (To verify remote)
This shows the URLs of the GitHub repo your project is connected to

`git branch`

↳ (To check branch)
It shows all the branches in your project

`git branch -M main`

↳ (To rename branch)
Changes the branch name to main

`git push origin main`

↳ Uploads your local main branch to the remote repository (GitHub)

`git push -u origin main`

↳ sets a default connection between your local branch & the remote branch, so next time you can push with just `git push` (Needs to be done once per project)

(12) mkdir Command :—

mkdir command is used to create a new folder (directory) in the current location

mkdir → make directory

```
mkdir folder_name
```

(13) GENERAL WORKFLOW WHEN WORKING WITH LOCAL GIT :—

1. **Create Repository**

Make a new repo on GitHub

↓

2. **Clone Repository**

Bring the repo to your local system using command git clone <repo-Link>

↓

3. **Make Changes**

Edit files, create new files, delete files, etc

↓

4. **Add Changes**

Add the updated files to the staging area using the command git add.

↓

5. **Commit Changes**

Save your changes with a message using the command

git commit -m "your message"

↓

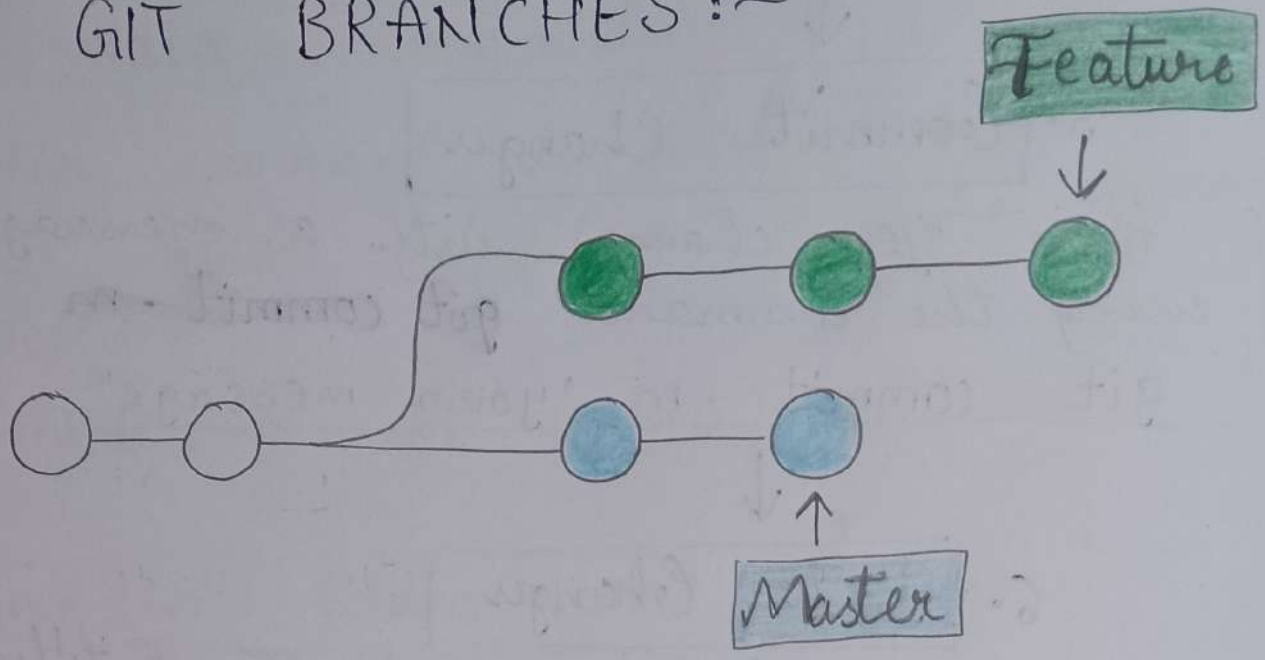
6. **Push Changes**

Send your commits to GitHub using the command git push

(14) LOCAL VS REMOTE (in Git & GitHub):-

LOCAL	REMOTE
→ Local means your computer	→ Remote means GitHub
→ All the project files you see & edit are in your local system	→ It is the online backup / storage of your project
→ Local changes are saved using Git commands like: • git add • git commit	→ A remote repository is connected using: • git remote add origin <link>
→ Local changes stay only on your device until you push	→ Your code reaches GitHub only when you push

(15) GIT BRANCHES :~



→ A Git Branch is like a separate line of work in your project.

→ Git Branches allow you to work on different versions of your project at the same time.

→ It lets you create a copy of your code so you can make changes or add new features without touching the main code.

- The main / master branch is the original project.
- A feature branch is created to work on something new
- After finishing the feature, the branch can be merged back into the main branch.

BRANCH COMMANDS :-

`git branch`

↳ This command is used to see all branches in the project & to know which branch we are currently on. (In the terminal, the branch with *(star) is the current branch)

`git branch -M main`

↳ To rename branch

`git checkout <-branch name->`

↳ To go from one branch to another i.e to navigate

`git checkout -b <-new branch name->`

↳ To create new branch

`git branch -d <-branch name->`

↳ To delete branch

(Note:- You cannot delete the branch you are currently on. First switch to another branch, then delete it)

(16) MERGING CODE

There are 2 ways to merge the different branches you created with the main branch:

→ WAY 1 :-

USING GIT COMMANDS (Terminal) :-

- `git diff <-branch-name->`
  ↳ To see the differences
  [To compare commits, branches, files & more]
- `git merge <-branch name->`
  ↳ To merge a branch with another branch
  (You can merge one branch at a time)

→ WAY 2 :-

USING A PULL REQUEST (PR) ON GITHUB :-

This method is commonly used when working in teams

Simple flow: ↓

1.) Push your branch to GitHub
2.) Go to GitHub repository
3.) Click "Compare & Pull Request"
4.) Add comments (optional
5.) Create Pull Request (PR)

6.) Repo owner / maintainer reviews your changes

7.) After approval → Merge PR

WHAT IS PULL REQUEST (PR)?

A Pull Request (PR) is a request you send to the owner of the repository or the team members who manage the project on GitHub asking permission to add (merge) your branch changes into another branch (usually the main branch). It allows others to check your work before it becomes part of the main project.

* (17) PULL COMMAND :—

used to fetch & download content from a remote repo & immediately update the local repo to match that content

```
git pull origin main
```

* (18) RESOLVING MERGE CONFLICTS :—

? Merge Conflicts is an event that takes place when Git is unable to automatically resolve differences in code between two commits.

→ A merge conflict happens when two branches change the same part of the same file, & Git cannot decide which one is correct.

WHY MERGE CONFLICTS HAPPEN?

Merge conflicts occur when:

- Two people edited the same line in a file
- A file was deleted in one branch but edited in another
- You made different changes in two branches to the same section of code text.

Resolving merge conflicts simply means: You manually choose which changes should stay & which should be removed.

Because Git cannot guess what you want.

HOW TO RESOLVE MERGE CONFLICTS?

1.) Git shows you the file that has a conflict

2.) Open the file in VS Code or any editor.

3.) You will see conflict markers like this :

```
<<<<<<< HEAD
Your current branch changes
=======
The other branch's changes
>>>>>>> branch-name
```

4.) Decide which part you want to keep
- keep your changes
- keep the other branch's changes
- Or combine both (edit manually)

5.) Remove the conflict markers (<<<<<<< , ======= , >>>>>>>)

6.) Save the file.

7.) Run:

```
git add.
git commit
```

Conflict is resolved

(19) UNDOING CHANGES :—

→ Undoing Changes means going back to a previous state when you made a mistake or changed something that you don't want anymore

~) It is just like using Undo (Ctrl+Z) in any software, but in Git, you do it using commands.

WHY DO WE UNDO CHANGES?

- You made a mistake
- You added wrong code/text
- You committed something you didn't want
- You want to go back to a clean version

TYPES OF CHANGES IN GIT :~

When we talk about Undoing Changes, there are 3 types of situations. Each situation needs a different command to undo.

CASE 1 : STAGED CHANGES

These are the changes you added using:

`git add.`

But you have not committed them yet.

UNDO COMMAND:

* `git reset <-file name->`
  → use this to undo the changes of a single file

* `git reset`
  → use this to undo multiple changes in multiple files

CASE 2: COMMITTED CHANGES (Only one Commit)

You already made one commit, but now you want to undo or edit that commit

UNDO COMMAND:

* `git reset HEAD~1`
  → To undo the single latest commit you made

`git reset --soft HEAD~1`
→ To undo the last commit but keep the changes in working area

`git reset --hard HEAD~1`
→ To undo the last commit & also remove the changes

CASE 3: COMMITTED CHANGES (for many commits)

You want to undo multiple commits, not just the latest one.

UNDO COMMAND:

→ * `git reset <-commit hash->`

↳ use this to undo multiple commits (You can copy & paste the commit hash of the commit you want to go back to from the command git log)

→ * `git reset --hard <-commit hash->`

↳ use this to undo the changes we made after making the latest commit. Here, using --hard will help you undo the changes from both git & VS code

→ Example: Undo last 3 commits:

`git reset --hard HEAD~3`

→ `git reset HEAD ~ n`

↳ undo many commits,
Here, n = number of last commits to be undone

(20) git log command:-
&
Commit hash:-

- Git Log — git log is used to see the commit history of your project at any time

It shows: all previous commits, the author, the date, the commit message, & each commit's unique hash code.

- Commit Hash — Every commit in Git has a unique ID, called a "hash".
  → This hash looks like a long code (example: a3f6b92c1d....)
  → In the terminal, it usually appears in yellow color
- Why do we need commit hash?
  We use commit hash in many commands like:
  - git reset <commit-hash>
  - git checkout <commit-hash>
  - git revert <commit-hash>

Because Git needs to know exactly which commit you are talking about.

- How to get the hash?

just run: `git log`

Then copy the yellow commit hash & paste it into the command you want to use.

(21) FORK :-

→ We can fork repository's on GitHub

→ A Fork is a new repository that shares code & visibility settings with the original "upstream" repository.

→ So, basically Fork is a rough copy of any project.

→ Fork means creating your own copy of someone else's GitHub repository in your GitHub account.

→ You can freely study it, experiment on it, & make changes without affecting the original project.

Why do we use Fork?

→ To copy someone else's repository into your own account

→ To practice or experiment safely.

→ To make improvements & then send a Pull Request if needed

→ To contribute to open-source projects.

Steps to Fork a Repository on GitHub :—

1.) Go to the GitHub repository you want to copy

↓

2.) On the top-right side, you will see a Fork button

↓

3.) Click Fork

↓

4.) GitHub will ask:

- Repository name (you can keep the same or change it)
- Description (optional)
- Whether you want to copy only the main (master) branch or copy the whole project.
  - if you want the entire project → untick "copy the master branch only"
  - if you want only master branch → Tick that option

↓

5.) Click Create Fork.

↓

6.) GitHub will create a copy in your own account
(If the project is large, it may take a little time)

Now you will have the same project in your GitHub account.

Important Note :-

If you want to contribute to the original project:

→ Make useful changes in your forked repo

→ Then send a Pull Request (PR)

→ Only make PR if your changes is helpful

→ Never create unnecessary PRs

KEY TERMS IN GIT & GITHUB (Useful for 90% of real-life Git / GitHub tasks) :-

(1) Repository (Repo) :-

→ A repository is like a project folder on GitHub

→ It stores all your project files and their version history.

(2) Add :-

→ When you add, you are selecting the changed files that you want to save in Git

(3) Commit :-

→ A commit is like saving your changes permanently in Git.

→ It captures what you changed at that moment.

→ You also write a commit message (short note about what you changed).

(4) Push :-

→ Push means sending your commits from computer to GitHub.

→ It uploads your saved changes to the online repository.

(5) Pull :-

→ Pull means downloading the latest updates from GitHub to your computer.

→ It keeps your local project up-to-date.

(6) Clone :-

→ Clone means making a copy of an entire GitHub repository on your computer.

→ You get the full project with all files.

(7) Branch :-

→ A branch is like creating a separate workspace in your project.

→ You can try new features without affecting the main code.

(8) Merge :-

→ Merge means combining the changes from one branch into another (usually into main branch).

(9) Fork:—

→ A fork is a copy of someone else's repository into your GitHub account.

→ You use it when you want to modify or contribute to other projects.

(10) Pull Request (PR):—

→ A pull request is a request to merge your changes into someone else's repository or the main branch.

→ Used widely in teamwork.

(11) Issues:—

→ Issues are like tasks or bug repoits in a project.

→ Used to track problems or upcoming work

(12) README :—

→ The README file explains the project.

→ It contains basic info like purpose, how to run, and features.

(13) Actions (GitHub Actions):—

→ Automated workflows to run tasks like testing or deployment.

→ Helps in CI/CD

(14) .gitignore :—

→ A file that tells Git which files to skip.
(e.g., temporary files, log files).

(15) Main / Master Branch :—

→ The main branch is the original & stable version of the project.

DIFFERENCES BETWEEN GIT, GIT BASH, & GITHUB :-

GIT	A version control system (VCS) used to track changes in files and manage project history. It works offline on your computer.
GIT BASH	A command-line tool for Windows that lets you run Git commands. It gives a linux-style terminal on Windows. (Mac already has Terminal, so no Git Bash needed.)
GITHUB	A website / platform where Git repositories are stored online. It helps you share, collabora-te, and work with others using Git.

STEPS TO CREATE A GITHUB REPOSITORY & UPLOAD A PROJECT :-

A.) Create a new repository on the GitHub website:

1.) Sign in to your GitHub account (open github.com and click sign in)

↓

2.) On the top-right, click the + icon (plus).

↓

3.) Click New repository

↓

4.) Repository name : type a short name (example : my-first-project).

↓

5.) Description (optional) : add one line explaining the project (optional).

↓

6.) Visibility : choose Public (anyone can see) or Private (only you & people you allow)

↓

7.) Initialize the repository (choose one or more):

- Add a README file – recommended. README explains the project
- Add .gitignore – optional ; pick a template (e.g., Python) to ignore unwanted files.

- Add a license - optional (like MIT) if you want others to use your code legally.

↓

8.) Click the green Create repository button

↓

9.) After creation, you land on the repo page.

Now you can:

- See the files (README, etc.)
- Click Code (green button) to get the clone URL (HTTPS or SSH)
- Click Add file → Upload files to upload files from your browser if you prefer.

B.) Put your local project on GitHub (using Git command)

[Use this Step B if you have files on your computer you want to send (push) to the new GitHub repo.]

PRE-STEPS (only once on your computer):

→ Install Git if not installed (visit git-scm.com and download)

→ Open Terminal (mac / linux) or Git Bash / Command Prompt (Windows).

→ Configure your identity (only first time):

```
git config --global user.name "Your name"
git config --global user.emal "Your-emal@example.com"
```

OPTION - 1 :— If you already have a project folder on your computer:

1.) Open terminal and go to your project folder:

```
cd path/to/your-project-folder
```

↓

2.) Initialize Git in that folder (makes it a Git repo):

```
git init
```

↓

3.) Connect your local repo to the GitHub repo (use the HTTPS URL you copied from GitHub):

```
git remote add origin https://github.com/your-username/my-first-project.git
```

↓

4) Add all files to staging (select files to save):

```
git add.
```

↓

5.) Commit your changes with a message:

```
git commit -m "Initial commit"
```

↓

6.) Rename your local branch to main (if needed):

```
git branch -M main
```

↓

7.) Push your local main branch to GitHub:

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

→ The first push may ask your GitHub username & password (or a personal access token) if using HTTPS.

OPTION – 2 :– If you want to start from the GitHub repo (clone it first):

1.) On the repo page on GitHub, click Code → HTTPS & copy URL

↓

2.) In terminal, run:

```
git clone https://github.com/your-username/my-first-project.git
```

3.) Go into the folder:

```
cd my-first-project
```

4.) Add files (copy or create files inside this folder), then:

```
git add.
git commit -m "Add project files"
git push
```

C.) If you prefer the web (no Git commands)

1.) Open your repo on GitHub

2.) Click Add file → Upload files

3.) Drag & drop files or click to choose files

4) Write a short commit message below (e.g., "Add project files")

5.) Click commit changes.

- This uploads files directly using the website.