Note 1:-

**Do you need to configure Git before starting a new project?**

* **Yes, at least once on your computer**, you should set your user name and email. This info is used in every commit you make to identify the author.

**How to configure Git (one-time setup):**

Run these commands in your terminal **before you start committing**:

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

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

* The --global flag means this applies to **all your Git projects** on this computer.
* You only need to do this once unless you want to use different info for a specific repo.

**Goal:- Create project-make repo-add file –connect local repo to github –push your changes**

**Step 1: Create a Project Folder**

Open your terminal (Command Prompt, PowerShell, or Terminal) and create a new folder:

mkdir my-first-git-project

cd my-first-git-project

**note:+**

* You **don’t have to open Git Bash inside the folder first**.
* You can open Git Bash (or any terminal) **anywhere**, then run:

mkdir my-first-git-project

cd my-first-git-project

This will create the folder **and then move you into it**.

**Step 2: Initialize Git**

Initialize Git in this folder:

git init

You’ll see a message like:  
Initialized empty Git repository in ...

**Step 3: Create a File**

Create a file to track. For example, a README:

echo "# My First Git Project" > README.md

**Step 4: Check Status**

Check what Git sees as changed/untracked:

git status

You’ll see README.md as an untracked file.

**Step 5: Stage the File**

Add README.md to the staging area:

git add README.md

**Step 6: Commit Your Changes**

Commit the staged file with a message:

git commit -m "Add README with project title"

**Step 7: Create a GitHub Repository**

* Go to [GitHub](https://github.com)
* Click **New repository**
* Name it my-first-git-project
* **Do NOT** initialize with README (since you already have one locally)
* Click **Create repository**

**Step 8: Connect Local Repo to GitHub**

Back in your terminal, connect your local repo to GitHub’s remote repo:

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

Replace your-username with your actual GitHub username.

**Step 9: Push Your Commit**

Push your commit to GitHub:

git push -u origin main

*(If your branch is called master, replace main with master.)*

***Note:-***

### How to check your branch name?

Just run:

git branch

It will list branches and highlight the current one, like:

\* main

or

\* master

### So when pushing your code for the first time, use:

git push -u origin main

**if your branch is named main**, or

git push -u origin master

**if your branch is named master.**

**What’s going on?**

* You **edited README.md directly on GitHub** (online).
* You then went to your **local repo** and made commits, but your local README.md file doesn’t have the changes you made on GitHub yet.
* That’s because your local copy is **out of sync** with the remote repo.

**What you need to do: Pull changes from GitHub first**

Before making new commits locally, you want to **sync your local repo with the remote** (GitHub). To do that, run:

git pull origin main

(or replace main with master if that’s your branch)

This fetches and merges the changes from GitHub into your local repo — including your edits to README.md.

**Workflow tip**

1. **Pull** changes from remote:

git pull origin main

1. **Make your changes locally** (edit files).
2. **Add + Commit + Push** your changes:

git add README.md

git commit -m "Your commit message"

git push origin main

### Why?

Git tracks changes separately on local and remote. If remote changes happen (like your GitHub edit), you have to pull those updates so your local repo knows about them.

**FOR THOSE WHO HAVE JUST INSTALL A GIT**

1.INSTALL GITBASH

**2.Configure Git (one-time setup)**

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

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

1. **Initialize a Repository**

* In an existing project folder:

git init

1. **Check Status**

* See which files are changed, staged, or untracked:

git status

1. **Add Files**

* Add files to the staging area (ready to commit):

git add filename.txt # Add a specific file

git add . # Add all changed files

1. **Commit Changes**

* Save your changes with a message:

git commit -m "Your descriptive commit message"

1. **Connect to a Remote Repository**

* Usually GitHub, GitLab, etc.

git remote add origin https://github.com/username/repository.git

1. **Push Changes**

* Send your commits to the remote repository:

git push origin main

*(Replace main with master or the branch you use.)*

1. **Pull Changes**

* Update your local repo with remote changes:

git pull origin main

**Quick Commands Summary:**

| **Command** | **Purpose** |
| --- | --- |
| git init | Start a new git repository |
| git status | Show current changes |
| git add <file> | Stage changes |
| git commit -m "msg" | Commit staged changes |
| git push | Push commits to remote repo |
| git pull | Pull latest changes from remote |
| git clone <url> | Clone remote repository locally |
|  |  |

** Cloning a repo (grab a copy of an existing project)**

### How to clone a repo

#### Step 1: Get the repo URL

* Go to the GitHub (or GitLab, Bitbucket, etc.) page of the project you want to clone.
* Click the green **Code** button.
* Copy the HTTPS URL (it looks like https://github.com/username/repository.git).

#### Step 2: Open your terminal (Git Bash, Terminal, etc.)

Navigate to the folder where you want the project to live. For example:

cd ~/projects

#### Step 3: Run the clone command

git clone https://github.com/username/repository.git

Replace the URL with the one you copied.

#### Step 4: Enter the project folder

cd repository

Now you have a full copy of the repo, including its entire history, and you can start working on it locally.

**how to make your first commit after cloning a repo:**

### Step 1: Clone the repo (if you haven’t already)

git clone https://github.com/username/repository.git

cd repository

### Step 2: Make changes to a file

For example, open or create a file and edit it. Let’s say you edit README.md:

echo "Adding a line to README" >> README.md

### Step 3: Check the status of your repo

See what files are changed/untracked:

git status

You should see README.md listed as modified.

### Step 4: Stage your changes

Add the changed files to the staging area:

git add README.md

### Step 5: Commit your changes

Commit the staged changes with a message:

git commit -m "Update README with new info"

### Step 6: Push your changes to the remote repo

git push origin main

(Replace *main* with *master* if your branch is called master.)

### How to create and switch branches in Git

#### Step 1: Check your current branch

git branch

This shows all branches and higitghlights the one you’re on.

#### Step 2: Create a new branch

git branch feature-branch

Replace feature-branch with a name describing your new feature or fix.

#### Step 3: Switch to your new branch

git checkout feature-branch

Now you’re working on the new branch!

**Shortcut:** You can create and switch at once with:

git checkout -b feature-branch

#### Step 4: Make changes and commit as usual

Edit files, then:

git add .

git commit -m "Describe your changes"

#### Step 5: Push your branch to the remote repo

git push -u origin feature-branch

The -u sets the remote branch to track your local one, so future git push commands work without extra arguments.

#### Step 6: Create a Pull Request (optional but common)

* Go to GitHub.
* You’ll see a prompt to create a Pull Request (PR) for your new branch.
* This is where you request to merge your changes into the main branch after review.

### Why use branches?

* Work on features independently.
* Avoid breaking main code.
* Collaborate safely with others.

Let’s talk about **merging branches** and then we’ll touch on **resolving conflicts** if they come up.

## Merging Branches

Merging means taking the changes from one branch (e.g., your feature branch) and combining them into another (usually main or master).

### Step 1: Switch to the branch you want to merge into

Usually, this is main:

git checkout main

### Step 2: Pull the latest changes (make sure your branch is up to date)

git pull origin main

### Step 3: Merge your feature branch into main

git merge feature-branch

### Step 4: Push the updated main branch to remote

git push origin main

## What if there are merge conflicts?

Conflicts happen when Git can’t automatically figure out how to combine changes because the same part of a file was changed differently in both branches.

### How to resolve merge conflicts:

1. Git will tell you which files have conflicts.
2. Open those files in a text editor.
3. Look for conflict markers like these:

<<<<<<< HEAD

Your changes in main branch

=======

Changes from feature-branch

>>>>>>> feature-branch

1. Edit the file to choose which changes to keep, or combine both as needed.
2. Remove the conflict markers.
3. Save the file.
4. Stage the resolved files:

git add filename

1. Complete the merge with a commit (if Git doesn’t do this automatically):

git commit -m "Resolved merge conflict in filename"

Let’s walk through a **step-by-step example of a Git merge conflict** and how to resolve it. We’ll simulate a common scenario:

## 🔧 Scenario: Merge Conflict in README.md

Let’s say:

* You’re working on a branch called main.
* You also created a branch called feature to edit README.md.

Both branches end up changing the **same line** in README.md.

### ✅ Step-by-step walkthrough

### 🔹 Step 1: Create a sample repo

mkdir merge-conflict-demo

cd merge-conflict-demo

git init

### 🔹 Step 2: Create and commit a README file

echo "Hello from main" > README.md

git add README.md

git commit -m "Initial commit on main"

### 🔹 Step 3: Create and switch to a new branch

git checkout -b feature

### 🔹 Step 4: Edit the README in the feature branch

echo "Hello from feature branch" > README.md

This replaces the previous content.

git add README.md

git commit -m "Edit README in feature branch"

### 🔹 Step 5: Switch back to main and edit the same file

git checkout main

echo "Hello from main branch (updated)" > README.md

git add README.md

git commit -m "Edit README in main branch"

### 🔹 Step 6: Now try to merge feature into main

git merge feature

⚠️ You will now get a **merge conflict** because both branches modified the same line.

Git will say something like:

Auto-merging README.md

CONFLICT (content): Merge conflict in README.md

Automatic merge failed; fix conflicts and then commit the result.

### 🔹 Step 7: Open the conflicted file

Open README.md, and you’ll see something like:

<<<<<<< HEAD

Hello from main branch (updated)

=======

Hello from feature branch

>>>>>>> feature

This means:

* **HEAD** = the main branch version
* **feature** = the other branch you’re trying to merge

### 🔹 Step 8: Resolve the conflict

Edit the file manually to what you want. For example:

Hello from BOTH branches!

Or choose just one version. Once done, **remove the conflict markers** (<<<<<<<, =======, >>>>>>>).

### 🔹 Step 9: Stage the resolved file

git add README.md

### 🔹 Step 10: Complete the merge

git commit -m "Resolve merge conflict in README.md"

## 1. Creating Branches – Work on new features safely

Using branches allows you to work on features, experiments, or bug fixes **without affecting the main branch**.

### 🔹 Step-by-step: Creating and switching to a branch

You're on main, and you want to create a new branch for a feature:

git checkout -b feature/login-page

* This creates a new branch called feature/login-page and switches to it.
* The naming convention (feature/..., bugfix/...) is optional but helps keep things organized.

### 🔹 Make your changes, then commit them:

# Edit your files (e.g., login.html)

git add login.html

git commit -m "Add login page UI"

### 🔹 Push the branch to GitHub (optional but common if collaborating):

git push -u origin feature/login-page

Now others can also see or collaborate on your branch.

## ✅ 2. Merging Branches – Combine your changes back into *main*

Once your feature is ready and tested, you can merge it into main.

### 🔹 Step-by-step: Merge a feature branch into main

#### 1. ****Switch to main****

git checkout main

#### 2. ****Update main to latest version from GitHub (recommended)****

git pull origin main

#### 3. ****Merge the feature branch into main****

git merge feature/login-page

* If no conflicts: Git will automatically merge and create a merge commit.
* If conflicts: Git will notify you and you'll resolve them manually (we covered this earlier).

### 🔹 (Optional) Delete the feature branch (clean-up)

#### Locally:

git branch -d feature/login-page

#### On GitHub:

git push origin --delete feature/login-page

## 🚀 Summary Flow

# Create and switch

git checkout -b feature/something

# Make changes and commit

git add .

git commit -m "Work on something"

# Push (optional)

git push -u origin feature/something

# Merge back to main

git checkout main

git pull origin main

git merge feature/something

git push origin main

# Delete branch (optional cleanup)

git branch -d feature/something

git push origin --delete feature/something

use of merge:-

what **exactly happens when you merge two branches in Git** gives you a lot more confidence and control over your code.

## 🔍 What Exactly Happens During a Git Merge?

When you run:

git merge feature-branch

...from the main branch, Git is trying to **combine the commit history** and **file changes** from feature-branch into main.

### 🧠 Behind the scenes: Git’s merge process

#### 1. ****Git finds a common ancestor commit****

Git looks at the commit history of both branches to find the most recent shared commit (called the **merge base**). It uses this to understand what has changed in each branch since they diverged.

#### 2. ****Git compares the changes from both branches****

It performs a 3-way comparison:

* The base commit (common ancestor)
* The latest commit in main
* The latest commit in feature-branch

### 🤖 Git then tries to ****automatically apply**** all the changes:

* If files were changed in only one branch → Git **auto-merges** them.
* If the same part of a file was changed in both branches → Git **can’t decide** and creates a **merge conflict**.

### ✅ If there are no conflicts:

Git:

* Merges the file changes.
* Creates a new **merge commit** in main (unless using fast-forward).

Example:

A---B---C main

\

D---E feature

After merge:

A---B---C---M main (merged)

\ /

D---E

Where M is the merge commit combining changes from both main and feature.

### ⚠️ If there are conflicts:

Git:

* Pauses the merge.
* Marks the conflicting files.
* You manually resolve the conflicts.
* Then stage (git add) and commit the result.

### 🎯 Key takeaway

**Git merging is about combining two lines of development** by comparing what changed since they split, and applying those changes together in a new commit.

## 🧪 Merge vs Rebase (optional concept)

Just so you’re aware:

* **Merge** keeps both histories intact.
* **Rebase** rewrites commit history to make it linear.

Use merge when collaborating (especially with shared branches). Rebase is great for cleaning up your own local commits before pushing.