Week 8

Exercise-1 Hands-on: Git Basics

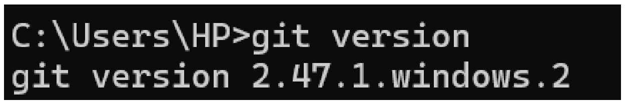
Objectives

- Be familiar with Git commands like `git init`, `git status`, `git add`, `git commit`, `git push`, and `git pull`.

Introduction to Git

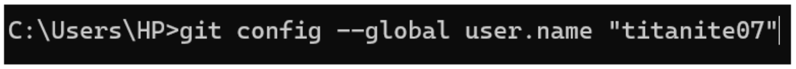
Git is a **version control system** that helps you track changes in your files. Think of it as a tool that takes "snapshots" of your project at different points in time, so you can always go back to a previous version if you need to. It's essential for collaboration and managing code projects.

1.git -–version



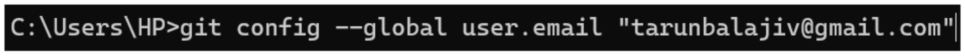
```
C:\Users\HP>git version
git version 2.47.1.windows.2
```

2.git config –global user.name “username”



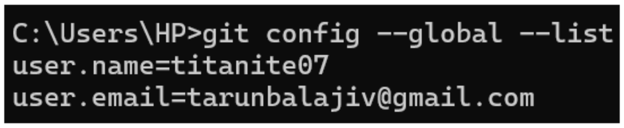
```
C:\Users\HP>git config --global user.name "titanite07"
```

3.git config –global user.email “email”



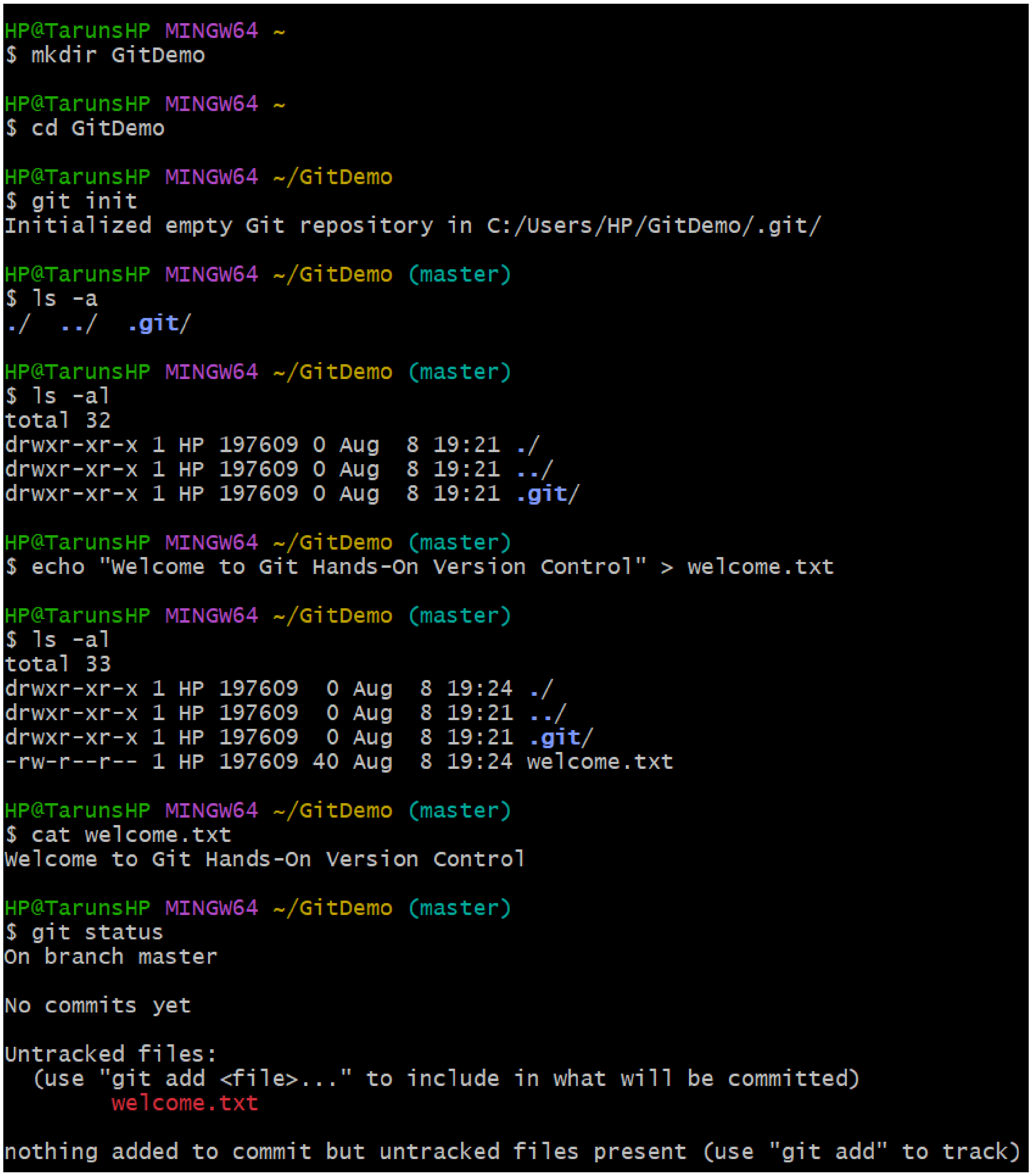
```
C:\Users\HP>git config --global user.email "tarunbalajiv@gmail.com"
```

4.git config –global –list

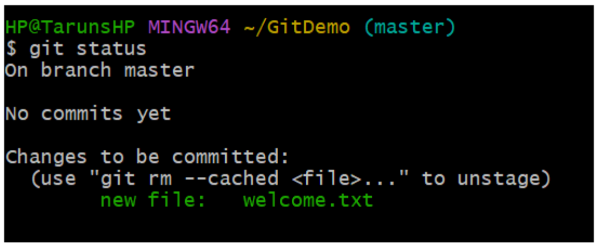


```
C:\Users\HP>git config --global --list
user.name=titanite07
user.email=tarunbalajiv@gmail.com
```

Add a File to Source Code Repository



```
HP@TarunsHP MINGW64 ~
$ mkdir GitDemo

HP@TarunsHP MINGW64 ~
$ cd GitDemo

HP@TarunsHP MINGW64 ~/GitDemo
$ git init
Initialized empty Git repository in C:/Users/HP/GitDemo/.git/

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ ls -a
./  ../  .git/

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ ls -al
total 32
drwxr-xr-x 1 HP 197609 0 Aug  8 19:21 ./
drwxr-xr-x 1 HP 197609 0 Aug  8 19:21 ../
drwxr-xr-x 1 HP 197609 0 Aug  8 19:21 .git/

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ echo "Welcome to Git Hands-On Version Control" > welcome.txt

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ ls -al
total 33
drwxr-xr-x 1 HP 197609  0 Aug  8 19:24 ./
drwxr-xr-x 1 HP 197609  0 Aug  8 19:21 ../
drwxr-xr-x 1 HP 197609  0 Aug  8 19:21 .git/
-rw-r--r-- 1 HP 197609 40 Aug  8 19:24 welcome.txt

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ cat welcome.txt
Welcome to Git Hands-On Version Control

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        welcome.txt

nothing added to commit but untracked files present (use "git add" to track)
```



```
HP@TarunsHP MINGW64 ~/GitDemo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   welcome.txt
```

Exercise -2 Hands-On: Implementing .gitignore

Introduction to `.gitignore`

When you're working on a project, there are often files you don't want to track with Git. These can include automatically generated files, temporary files, log files, or folders created by your operating system or IDE. The **`.gitignore`** file is a special text file that tells Git which files and folders to ignore. This keeps your repository clean and focused on the source code.

Code:

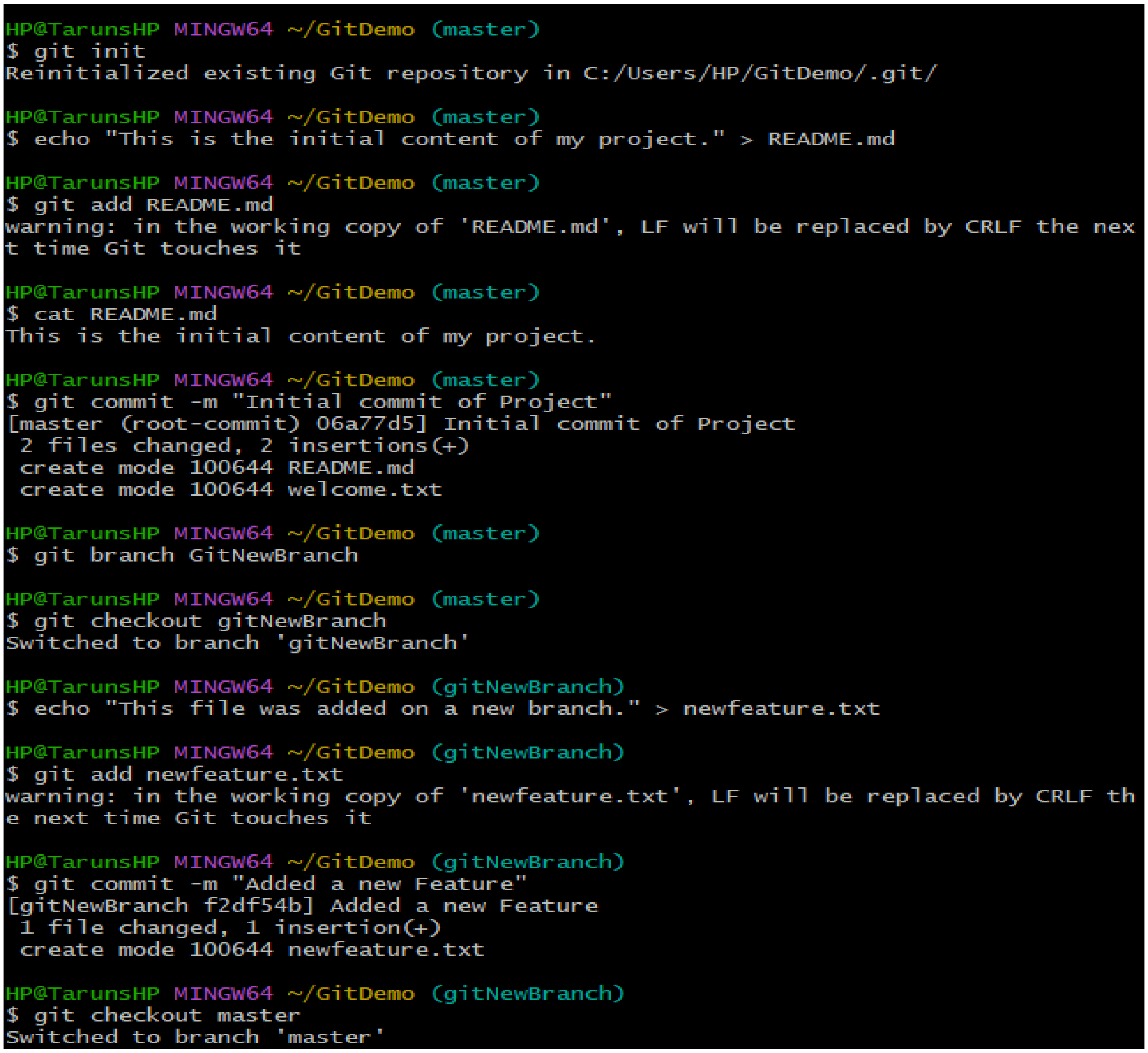
```
touch myapp.log
mkdir log
touch log/debug.log
git status
On Branch Master

No Commits Yet

Changes yet to be committed:
        (use “git rm  --cached <file>.. ” to unstage)

                new file: myapp.log
*.log
log/
git add .gitignore
git commit -m "Add .gitignore to ignore log files and folders"
git push origin master
```

Exercise -3 Hands-On: Introduction to Git Branching and Merging

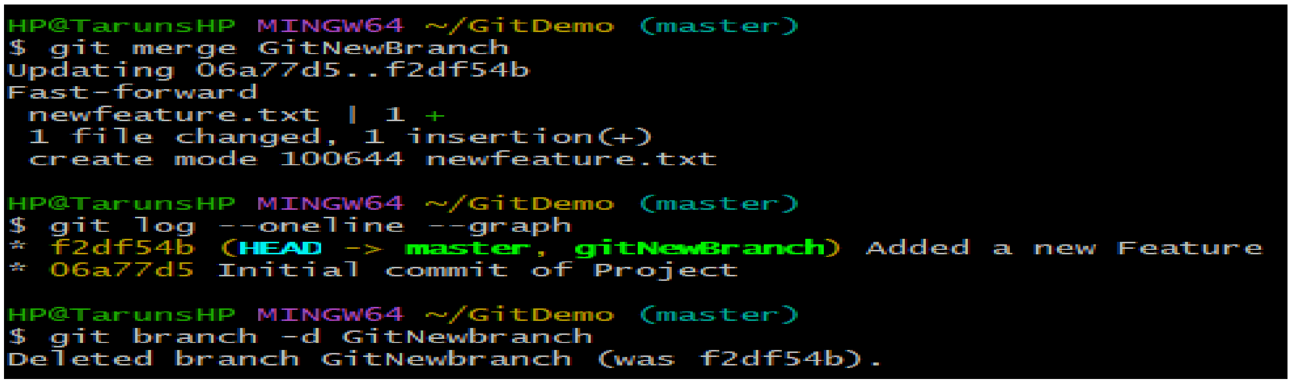
Introduction to Git Branching and Merging

Git **branching** allows you to work on different features or bug fixes in isolation from the main project. Imagine the main project as a tree trunk (`master` or `main` branch). When you want to work on something new, you create a new branch, like a new tree limb, so your work doesn't affect the stable "trunk" of the project.

Once your work on the new branch is complete, **merging** is the process of combining those isolated changes back into the main branch. This keeps the project history clean and allows multiple people to work on different things at the same time without interfering with each other.



```
HP@TarunsHP MINGW64 ~/GitDemo (master)
$ git init
Reinitialized existing Git repository in C:/Users/HP/GitDemo/.git/

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ echo "This is the initial content of my project." > README.md

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ git add README.md
warning: in the working copy of 'README.md', LF will be replaced by CRLF the nex
t time Git touches it

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ cat README.md
This is the initial content of my project.

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ git commit -m "Initial commit of Project"
[master (root-commit) 06a77d5] Initial commit of Project
 2 files changed, 2 insertions(+)
 create mode 100644 README.md
 create mode 100644 welcome.txt

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ git branch GitNewBranch

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ git checkout gitNewBranch
Switched to branch 'gitNewBranch'

HP@TarunsHP MINGW64 ~/GitDemo (gitNewBranch)
$ echo "This file was added on a new branch." > newfeature.txt

HP@TarunsHP MINGW64 ~/GitDemo (gitNewBranch)
$ git add newfeature.txt
warning: in the working copy of 'newfeature.txt', LF will be replaced by CRLF th
e next time Git touches it

HP@TarunsHP MINGW64 ~/GitDemo (gitNewBranch)
$ git commit -m "Added a new Feature"
[gitNewBranch f2df54b] Added a new Feature
 1 file changed, 1 insertion(+)
 create mode 100644 newfeature.txt

HP@TarunsHP MINGW64 ~/GitDemo (gitNewBranch)
$ git checkout master
Switched to branch 'master'
```



```
HP@TarunsHP MINGW64 ~/GitDemo (master)
$ git merge GitNewBranch
Updating 06a77d5..f2df54b
Fast-forward
 newfeature.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 newfeature.txt

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ git log --oneline --graph
* f2df54b (HEAD -> master, gitNewBranch) Added a new Feature
* 06a77d5 Initial commit of Project

HP@TarunsHP MINGW64 ~/GitDemo (master)
$ git branch -d GitNewbranch
Deleted branch GitNewbranch (was f2df54b).
```

Exercise -4 Hands-On: Understanding and Resolving Git Merge Conflicts

A **merge conflict** occurs when Git is unable to automatically integrate changes from one branch into another. This typically happens when two different branches have modified the same lines in the same file, and Git can't decide which change to keep. Resolving conflicts is a fundamental part of collaborative work in a version control system like Git.

1. The Scenario: Creating the Conflict

To create a conflict, we need two separate commit histories that diverge from a common point.

- **Initial State:** Your project starts with a file (e.g., `README.md`) on the **master** branch.
- **Branching:** You create a new branch, `GitWork`, and switch to it using `git checkout GitWork`.
- **Divergent Changes:**
  - On the `GitWork` branch, you create a new file, `hello.xml`, and add some content. You then commit this change.
  - You switch back to the `master` branch (`git checkout master`).
  - On the `master` branch, you create a different version of the *same file*, `hello.xml`, and commit it.

At this point, Git's history has diverged. The `master` and `GitWork` branches both contain a `hello.xml` file, but with different content.

2. The Failed Merge

When you attempt to merge the `GitWork` branch back into `master`, Git will stop and tell you there is a conflict.

- **The Command:** You run `git merge GitWork`.
- **The Output:** Git will display a message like `CONFLICT (add/add): Merge conflict in hello.xml`. This means a conflict occurred because both branches added the same file with different content.

Git will then modify the conflicting file (`hello.xml`) with special markers to show you where the problem is.

3. The Conflict Markers

When you open the conflicting file in a text editor, you'll see a section that looks like this:

```
<<<<<<< HEAD
<?xml version='1.0' encoding='UTF-8'?><greeting>Hello from master!</greeting>
=======
<?xml version='1.0' encoding='UTF-8'?><greeting>Hello from the
branch!</greeting>
>>>>>>> GitWork
```

- `<<<<<<< HEAD`: This marks the beginning of the conflicting change from your **current branch** (the `HEAD` of `master`).
- `=======`: This is the separator between the two conflicting versions.
- `>>>>>>> GitWork`: This marks the end of the conflicting change from the **branch you are merging** (the `GitWork` branch).

Your job is to manually edit this section to the final, correct version of the file, removing all of these markers.

4. Resolving the Conflict

To resolve the conflict, you must manually edit the file. There are a few options:

- **Accept your changes:** Delete everything between the ======= and >>>>>>> markers.
- **Accept their changes:** Delete everything between the <<<<<<< and ======= markers.
- **Combine both:** Edit the file to include elements from both versions. This is the most common approach.

In the example above, a good resolution would be to delete all the markers and combine the content:

```
<?xml version='1.0' encoding='UTF-8'?><greeting>Hello from the branch and the
master!</greeting>
```

5. Completing the Merge

After manually editing the file, you need to tell Git that you've finished resolving the conflict.

1. **Stage the resolved file:** You add the file to the staging area with `git add hello.xml`. This tells Git that the conflict for this file is resolved.
2. **Commit the merge:** You run `git commit -m "Merge branch 'GitWork' into master and resolve conflict"`. Git will automatically create a **merge commit** that records the new, final state of the file and completes the merge process.

6. Cleanup: Deleting the Branch

Once the `GitWork` branch has been successfully merged into `master`, you can delete it to keep your repository clean.

- **List branches:** Use `git branch` to see all local branches.

- **Delete the branch:** Use `git branch -d GitWork`. The `-d` flag is a safety measure; Git will only delete the branch if its changes have been fully merged into another branch. If you needed to force a deletion, you would use `-D`.

Exercise -5 Hands-On: Pushing and Cleaning Up Your Git Repository

1. Verifying a Clean State

Before you interact with the remote repository, it is always a good practice to check the status of your local working directory.

- **Command:** `git status`
- **Purpose:** This command shows you the state of your working directory and staging area.
- **Ideal Output:** You want to see `nothing to commit, working tree clean`. This means you have no uncommitted changes or unstaged files, so you can safely pull from or push to the remote repository without losing any work.

2. Pulling the Latest Remote Changes

Before pushing your changes, you should always pull any updates that others may have pushed to the remote repository since you last synchronized. This helps prevent merge conflicts.

- **Command:** `git pull origin master`
- **Purpose:** This is a shortcut for two commands:
    1. **`git fetch origin`**: This downloads the latest history from the remote repository (named `origin`). It updates your local copy of the remote branch but doesn't merge the changes into your current working branch.
    2. **`git merge origin/master`**: This merges the fetched changes from the remote `master` branch into your local `master` branch.
- **Result:** After a successful pull, your local `master` branch will be fully up-to-date with the remote version, and you're ready to push your own work.

3. Pushing Local Changes to Remote

Once your local branch is clean and has the latest changes from the remote, you can push your committed work to the shared repository.

- **Command:** `git push origin master`
- **Purpose:** This command uploads your local commits to the remote repository, making them available to your collaborators.
- **Understanding the terms:**
    - **`git push`**: The action of uploading your commits.
    - **`origin`**: The default name for the remote repository.
    - **`master`**: The name of the local branch you are pushing.

- **Result:** Your local `master` branch will now have the same commit history as the remote `master` branch.

4. Verifying Changes on the Remote Repository

The final step is to confirm that your changes have been successfully uploaded to the remote server.

- **How to verify:** You can't do this with a Git Bash command alone. You'll need to go to your Git hosting service's website (e.g., GitHub, GitLab, or Bitbucket) and navigate to your repository.
- **What to look for:** You should be able to see your new commit message and the updated files directly in the repository's file browser. This is your confirmation that the changes are live and visible to your team.