**Week 8 – GIT**

**Task 1 : Git Configuration, Notepad++ Setup, and Initial Commit**

**Objective :**To configure Git on the local machine, set up Notepad++ as the default editor, create a new Git repository, add a file, and make the first commit.

**Step 1 – Check if Git is Installed**

**Command :**

git –version

**Explanation :**  
Displays the installed Git version to confirm Git is available on the system.

**Step 2 – Configure Git Username and Email**

**Commands:**

git config --global user.name "Hariharane-Kumar"

git config --global user.email hariharanek2201[@gmail.com](mailto:sivabasker006@gmail.com)

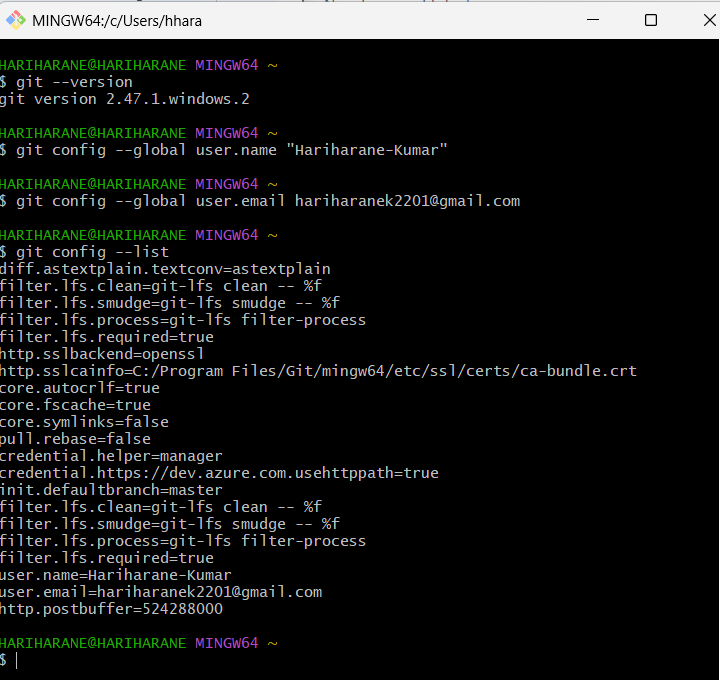
**Explanation:**  
Sets your Git identity for commits across all repositories.

**Step 3 – Verify Git Configuration**

**Command:**

git config –list

**Explanation:**  
Lists the current Git configuration, including the user name, email, and editor settings.



**Step 4 – Set Notepad++ as Default Editor**

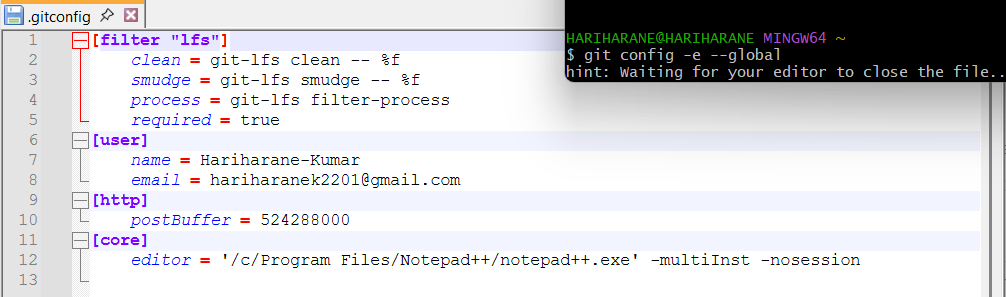
**Command :**

git config --global core.editor "'/c/Program Files/Notepad++/notepad++.exe' -multiInst -nosession"

**Explanation:**  
Configures Notepad++ as the default editor for commit messages.

**Test Command:**

git config -e –global



**Step 5 – Create a Project Folder**

**Command:**

mkdir GitDemo

**Explanation:**Creates a new folder named GitDemo to store the project.

**Step 6 – Navigate to the Project Folder**

**Command:**

cd GitDemo

**Explanation:**  
Moves into the GitDemo directory.

**Step 7 – Initialize an Empty Git Repository**

**Command:**

git init

**Explanation:**  
Creates a new empty Git repository in the gitd folder.

**Step 8 – Create a New File**

**Command:**

notepad welcome.txt

**Explanation:**  
Opens Notepad to create welcome.txt. Add text inside (e.g., Welcome to Git hands-on!) and save.

**Step 9 – Check Repository Status**

**Command:**

git status

**Explanation:**  
Shows the state of the working directory. welcome.txt will appear under "Untracked files."

**Step 10 – Stage the File**

**Command:**

git add welcome.txt

**Explanation:**  
Adds welcome.txt to the staging area.

**Step 11 – Commit the File**

**Command:**

git commit

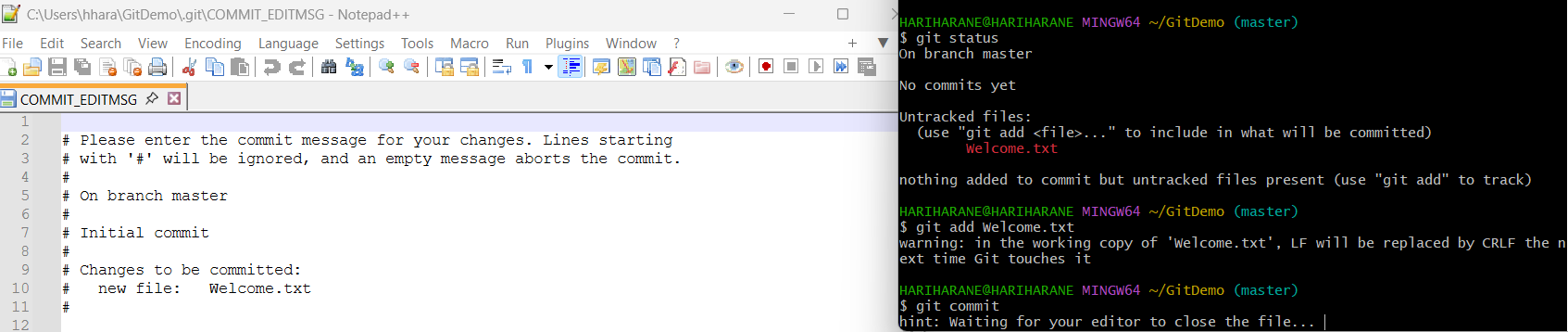
**Explanation:**  
Opens Notepad++ to enter a commit message. Example:

First Commit

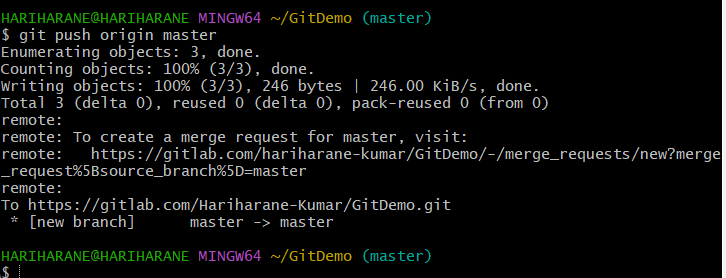
Added welcome.txt

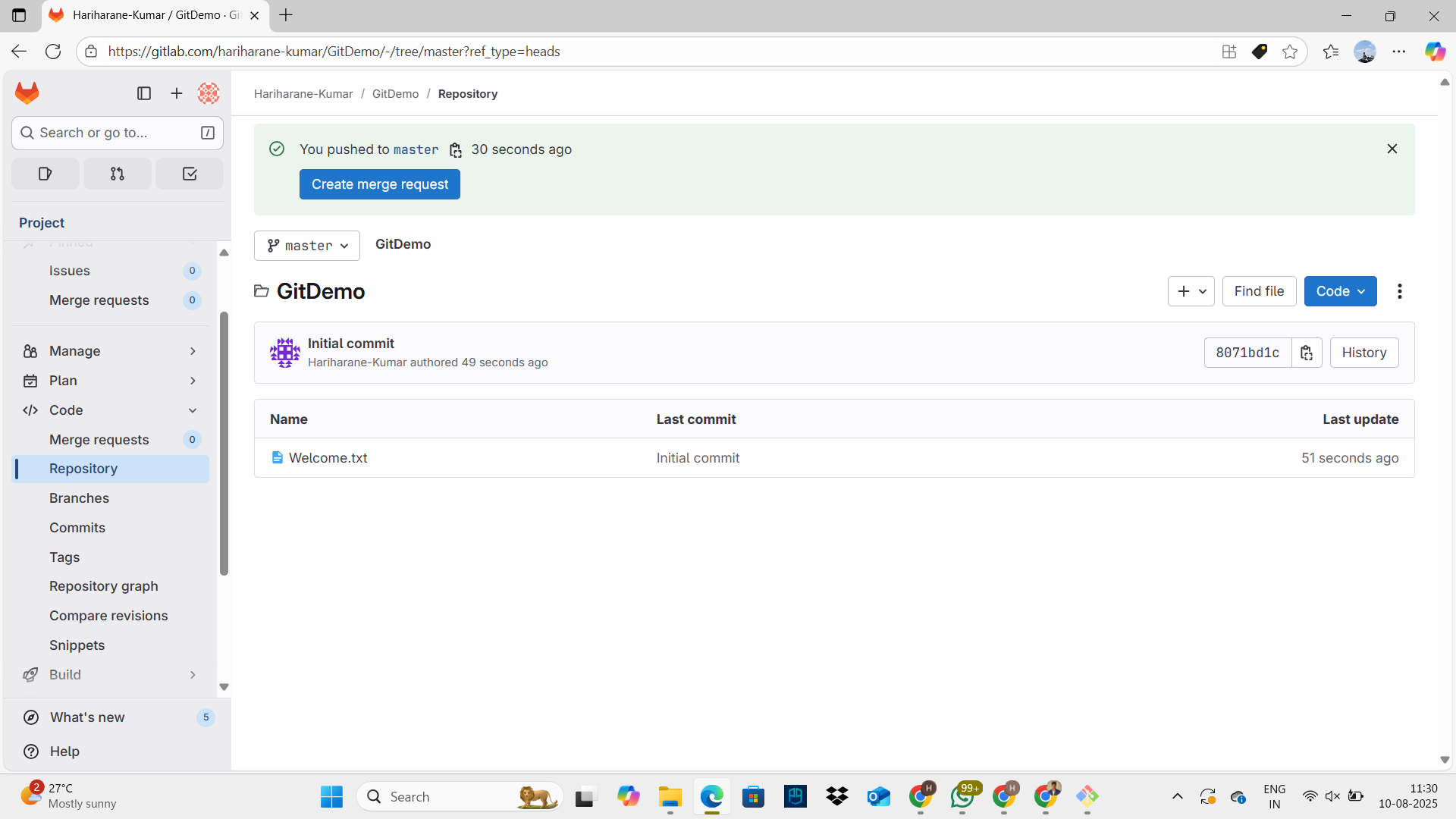
Git demo file





**Step 12 – Link with GitLab**





**Task 2 – Using .gitignore to Ignore Unwanted Files**

**Objective:**  
To learn how to use .gitignore in Git to prevent specific files and folders from being tracked.

**Step 1 – Navigate to the Repository**

**Command:**

cd GitDemo

**Explanation:**  
Moves into the local Git repository folder gitd where the .gitignore file will be created.

**Step 2 – Create a .log File and a log Folder**

**Commands:**

echo "This is a log file" > debug.log

mkdir log

echo "Folder log content" > log/example.txt

**Explanation:**

* debug.log is a sample log file.
* log/ is a folder containing a sample text file.

**Step 3 – Check Git Status Before Ignoring**

**Command:**

git status

**Explanation:**  
Lists all untracked files. At this point, debug.log and log/example.txt should appear in the output.

**Step 4 – Create a .gitignore File**

**Command:**

notepad++ .gitignore

**File Content:**

\*.log

log/

**Explanation:**

* \*.log → ignores all files ending in .log.
* log/ → ignores the log folder and everything inside it.

**Step 5 – Save .gitignore and Check Status Again**

**Command:**

git status

**Explanation:**  
Now debug.log and log/ should no longer appear in the untracked list, proving they are being ignored.

**Step 6 – Stage and Commit .gitignore**

**Commands:**

git add .gitignore

git commit -m "Added .gitignore to exclude log files and log folder"

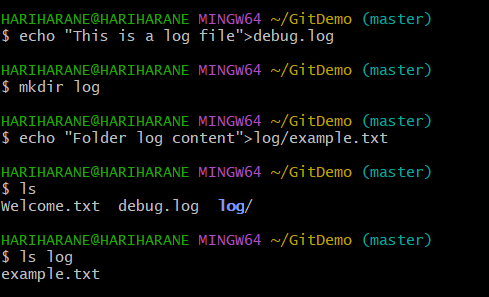
**Explanation:**  
Saves the .gitignore file to the repository so that the ignore rules are shared.

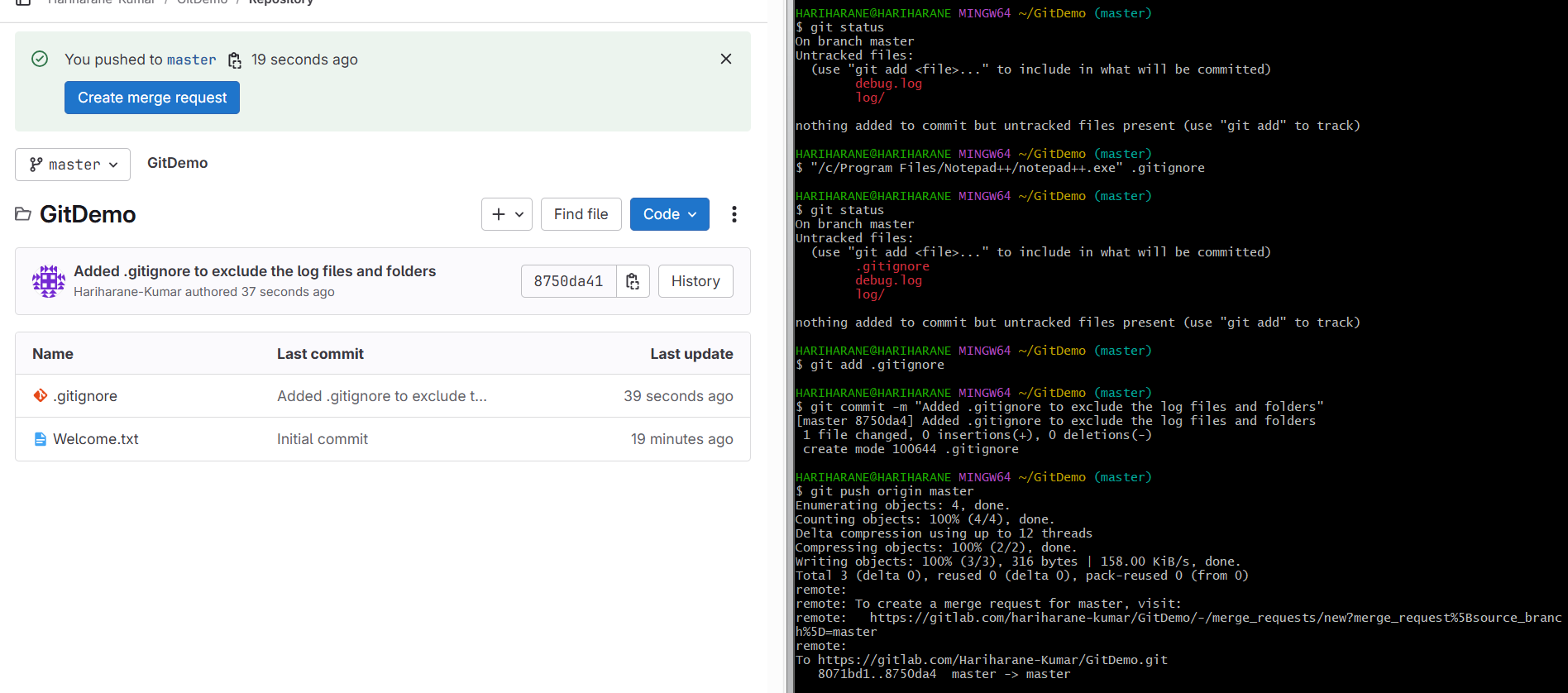
**Step 7 – Push Changes to Remote Repository**

**Command:**

git push origin master

**Explanation:**  
Uploads the .gitignore file to the GitLab repository. The ignored files (debug.log and log/) will not appear in the remote repo.





**Task 3 – Branching and Merging**

**Objective:**  
To learn how to create a new branch, make changes, merge them into the main branch, and use both CLI and visual merge tools.

**Part A – Creating and Working on a New Branch**

**Step 1 – Create a New Branch**

**Command:**

git branch GitNewBranch

**Explanation:**  
Creates a new branch named GitNewBranch from the current branch (master).

**Step 2 – List All Local and Remote Branches**

**Command:**

git branch -a

**Explanation:**  
Lists all branches:

* \* marks the current branch.
* Remote branches are prefixed with remotes/.

**Step 3 – Switch to the New Branch**

**Command:**

git checkout GitNewBranch

**Explanation:**  
Moves your working directory to the GitNewBranch.

**Step 4 – Add a New File in the New Branch**

**Commands:**

echo "This is a new feature" > feature.txt

git add feature.txt

**Explanation:**  
Creates feature.txt with sample content and stages it for commit.

**Step 5 – Commit the Changes**

**Command:**

git commit -m "Added feature.txt in GitNewBranch"

**Explanation:**  
Commits the staged file to the GitNewBranch.

**Step 6 – Check Status**

**Command:**

git status

**Explanation:**  
Verifies that there are no uncommitted changes in the branch.

**Part B – Merging Changes**

**Step 1 – Switch Back to Master**

**Command:**

git checkout master

**Explanation:**  
Switches to the main branch to prepare for merging.

**Step 2 – Compare Changes Between Branches**

**Command:**

git diff master GitNewBranch

**Explanation:**  
Shows differences in files between master and GitNewBranch.

**Step 3 – View Visual Differences with P4Merge**

**Command:**

git mergetool

**Explanation:**  
Opens P4Merge to visually compare and resolve differences (only if configured).

**Step 4 – Merge the Feature Branch into Master**

**Command:**

git merge GitNewBranch

**Explanation:**  
Merges all commits from GitNewBranch into master.

**Step 5 – View Commit History After Merge**

**Command:**

git log --oneline --graph --decorate

**Explanation:**  
Shows a compact visual commit history with branch and merge points.

**Step 6 – Delete the Merged Branch**

**Command:**

git branch -d GitNewBranch

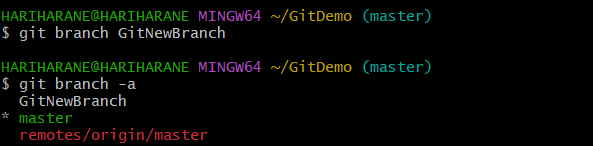
**Explanation:**  
Deletes the branch locally since it has been merged.

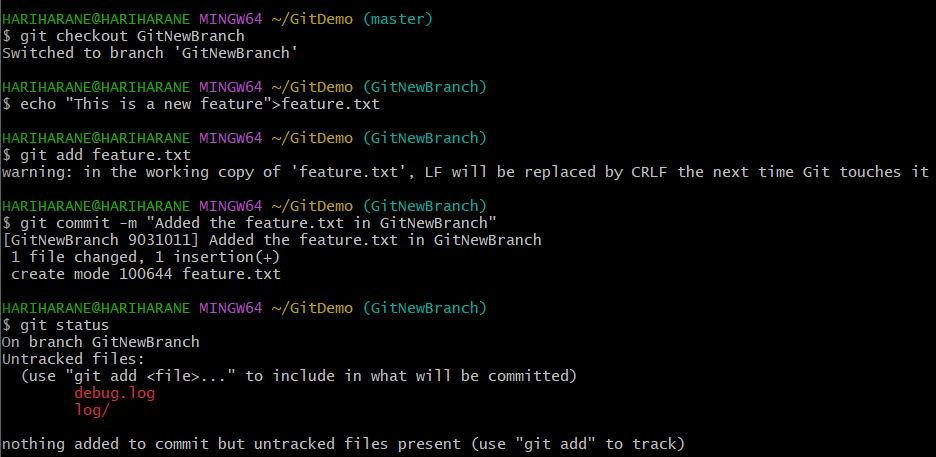
**Step 7 – Verify Status**

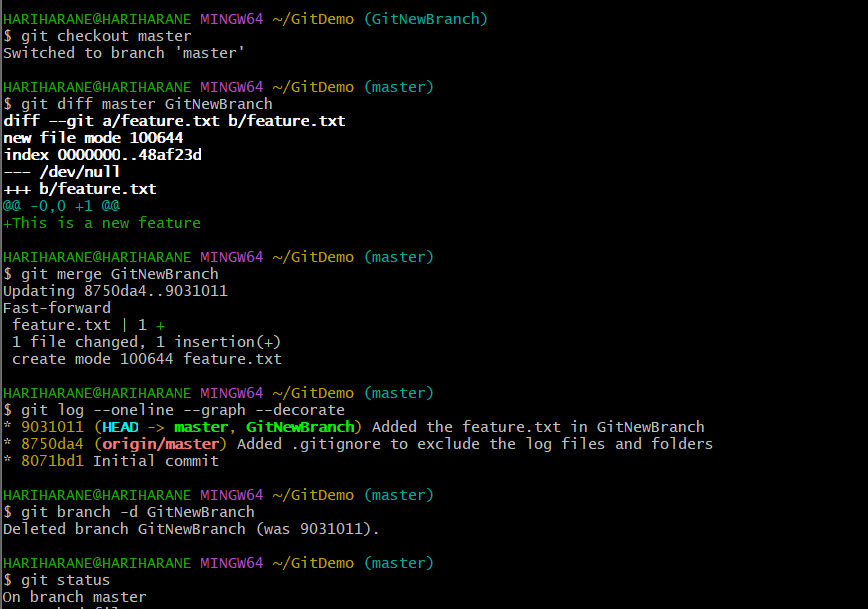
**Command:**

git status

**Explanation:**  
Ensures the working directory is clean after merging.







**Task 4 – Conflict Resolution during Merge**

**Objective:**  
To intentionally create a merge conflict between two branches, resolve it manually or with a merge tool, and complete the merge process.

**Part A – Creating the Conflict Scenario**

**Step 1 – Verify Clean Working Directory**

**Command:**

git status

**Explanation:**  
Checks if there are no pending changes before starting.

**Expected Output:**

On branch master

nothing to commit, working tree clean

**Step 2 – Create a New Branch and Add hello.xml**

**Commands:**

git checkout -b GitWork

notepad hello.xml

**Explanation:**  
Creates a new branch GitWork and opens a file hello.xml to add content.

Sample content:

<message>Hello from GitWork branch</message>

**Step 3 – Update hello.xml in GitWork**

**Commands:**

echo "<update>Some update in GitWork</update>" >> hello.xml

git status

**Explanation:**  
Appends additional content to the file and checks the modified status.

**Step 4 – Commit the GitWork Changes**

**Commands:**

git add hello.xml

git commit -m "Updated hello.xml in GitWork branch"

**Explanation:**  
Stages and commits changes to GitWork.

**Step 5 – Switch Back to Master**

**Command:**

git checkout master

**Explanation:**  
Moves back to the master branch to introduce conflicting changes.

**Step 6 – Create a Conflicting hello.xml in Master**

**Commands:**

notepad hello.xml

Add different content:

<message>Hello from master branch</message>

**Step 7 – Commit the Master Branch Changes**

**Commands:**

git add hello.xml

git commit -m "Added hello.xml in master"

**Part B – Detecting the Conflict**

**Step 8 – View Both Branches’ Histories**

**Command:**

git log --oneline --graph --decorate --all

**Explanation:**  
Shows that the branches have diverged.

**Step 9 – Compare the Files**

**Command:**

git diff GitWork master -- hello.xml

**Explanation:**  
Displays textual differences between the file in both branches.

**Step 10 – Merge GitWork into Master**

**Command:**

git merge GitWork

**Explanation:**  
Attempts to merge, causing a **merge conflict** in hello.xml.

**Step 11 – View Conflict Markers**

**Command:**

cat hello.xml

**Part C – Resolving the Conflict**

**Step 12 – Edit File to Resolve Conflict**

**Command:**

notepad hello.xml

**Resolved Content Example:**

<message>Hello from master and GitWork branch</message>

<update>Some update in GitWork</update>

**Step 13 – Stage and Commit Resolved File**

**Commands:**

git add hello.xml

git commit -m "Resolved conflict and merged GitWork to master"

**Step 14 – Handle Backup Files from Merge Tool**

If hello.xml.orig or other backups exist:  
**Commands:**

echo "\*.orig" >> .gitignore

git add .gitignore

git commit -m "Added .orig files to .gitignore"

**Step 15 – Delete the Merged Branch**

**Command:**

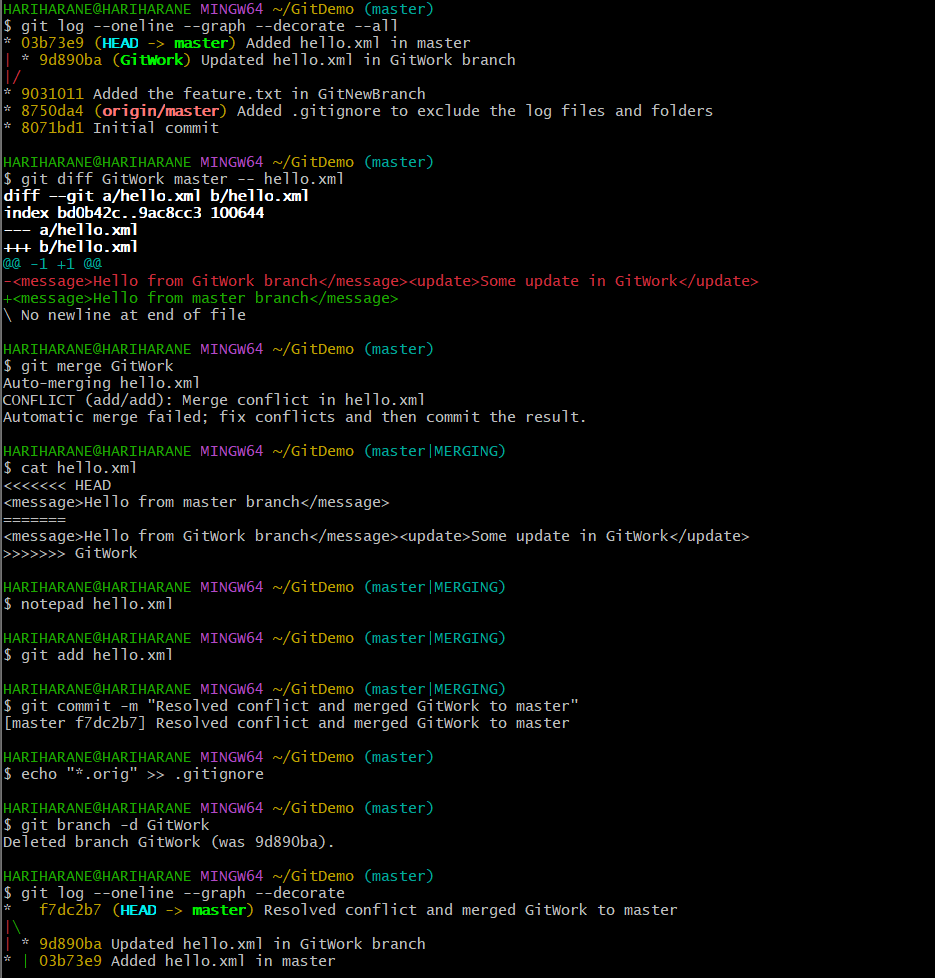
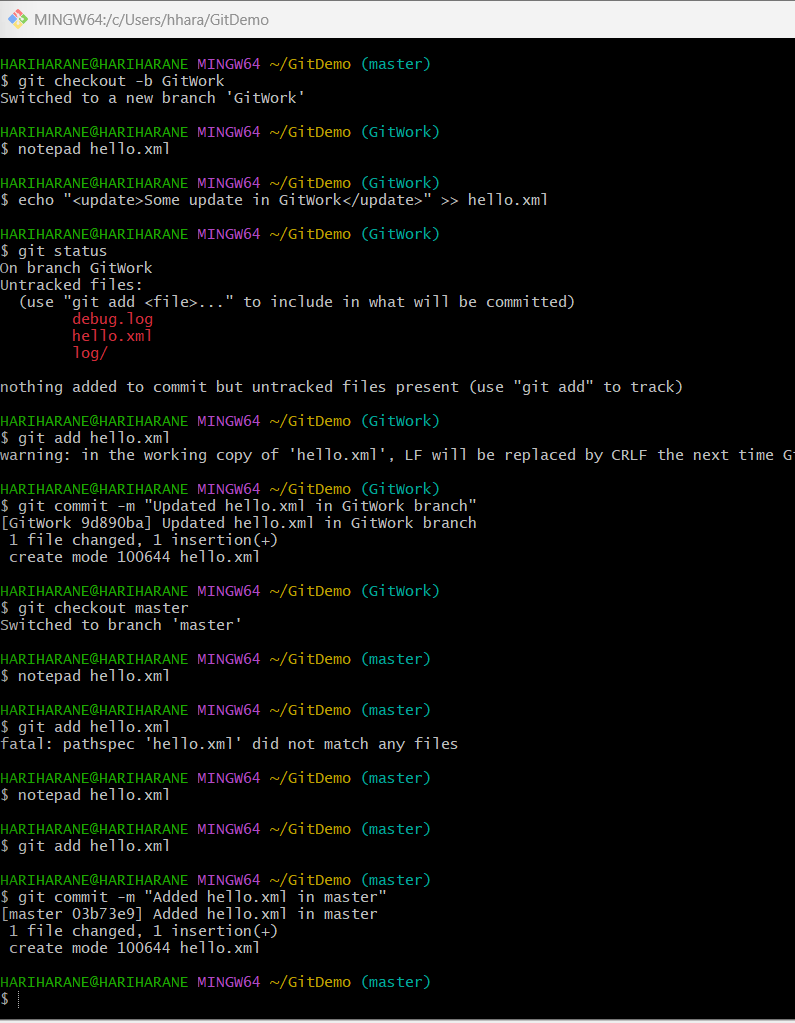
git branch -d GitWork

**Step 16 – Final Log Check**

**Command:**

git log --oneline --graph --decorate

**Explanation:**  
Confirms a clean merge with conflict resolved.



**Task 5 – Clean up and Push Back to Remote Git**

**Objective:**  
To verify the repository state, pull the latest remote changes, push local changes to the remote repository, and confirm updates.

**Step 1 – Verify if master is in a Clean State**

**Command:**

git status

**Explanation:**  
Checks if there are no uncommitted changes before pulling or pushing to remote.

**Step 2 – List all Available Branches**

**Command:**

git branch

**Explanation:**  
Lists all local branches.  
To also see remote branches:

git branch -a

**Step 3 – Pull the Remote Repository to Master**

**Commands:**

git checkout master

git pull origin master

**Explanation:**  
Ensures you’re on master branch and pulls the latest changes from the remote.  
If there are unrelated histories, use:

git pull origin master --allow-unrelated-histories

**Step 4 – Push Local Changes to Remote**

**Command:**

git push origin master

**Explanation:**  
Uploads local commits from master to the remote repository.  
If this is the first push for the branch:

git push -u origin master

**Step 5 – Confirm Changes on Remote**

**Action:**  
Opened GitLab (or GitHub) repository in a browser.

**Check for:**

* The files updated in **Task 4** (hello.xml, .gitignore, etc.).
* The latest commit message is visible.
* Branch is up to date with origin/master.

