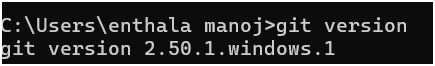
**WEEK 8 HANDSON**

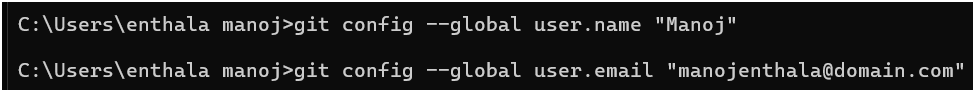
**EX 1:**

**Step 1: Setup Git Configuration**

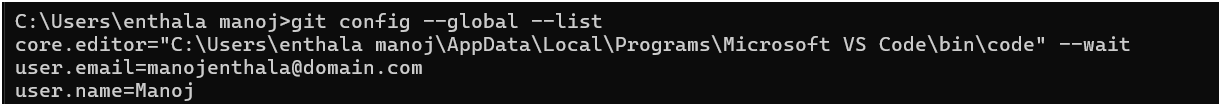
**1. Check Git installation**

****

**2. Configure Git (Global)**

****

**3. Verify Git configuration**

****

**Step 2: Set Notepad++ as Default Git Editor**

**1. Check if Notepad++ is recognized in Git Bash**

**notepad++**

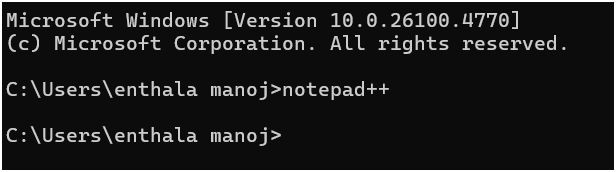
**If you see:  
bash: notepad++: command not found,  
then add Notepad++ to system path:**

**➕ Add Notepad++ to PATH:**

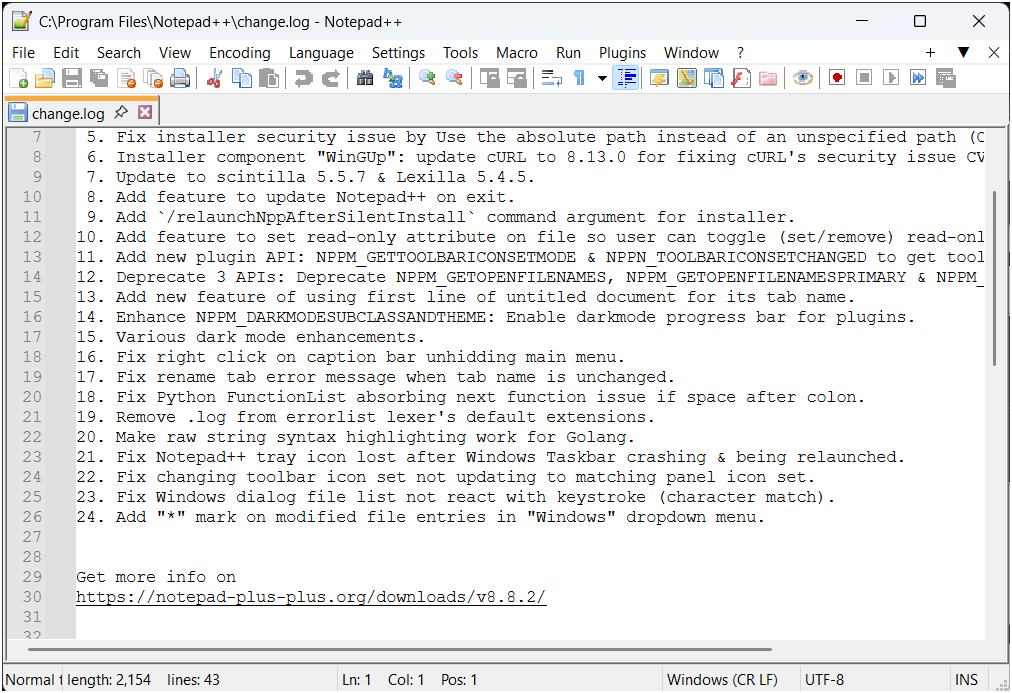
1. **Go to Control Panel → System → Advanced System Settings**
2. **Click Environment Variables**
3. **Under User Variables, edit Path**
4. **Add path like: C:\Program Files\Notepad++\ (depends on your installation)**

**2. Reopen Git Bash and check again:**

**notepad++**

****

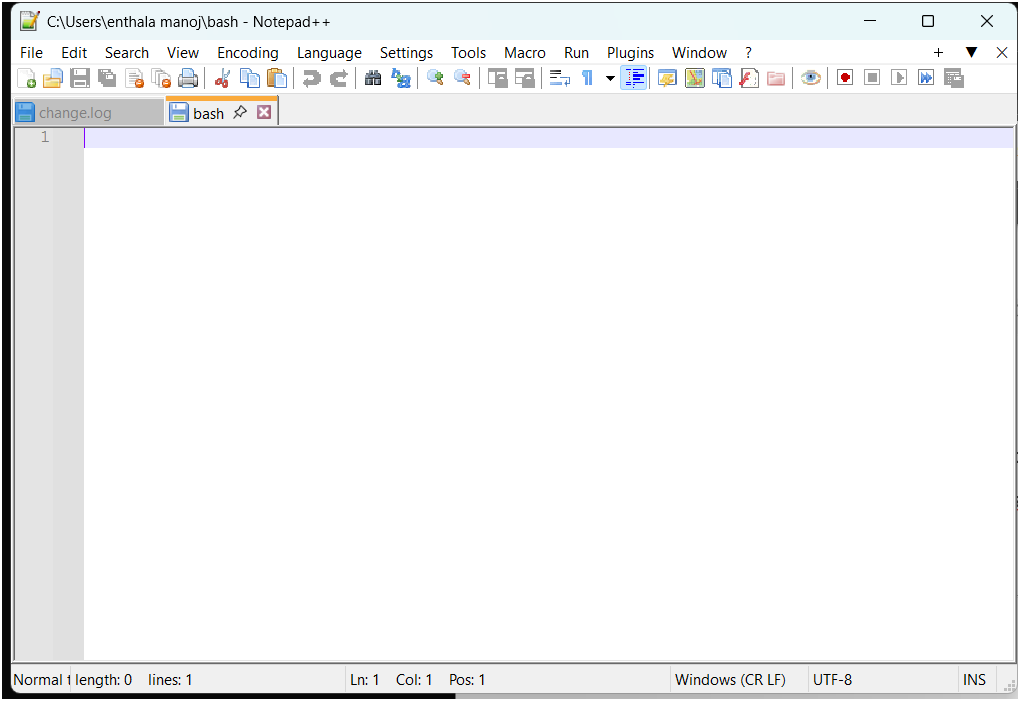
**It should now open Notepad++.**

****

**3. Create alias for Notepad++**

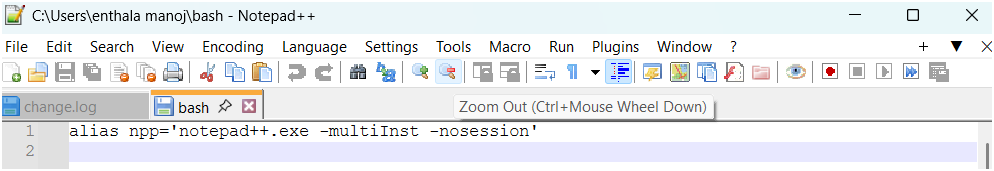
**notepad++.exe bash -profile**

****

****

**This opens Notepad++ in bash profile. Add this line to the file:**

**alias npp='notepad++.exe -multiInst -nosession'**

****

**Save and close.**

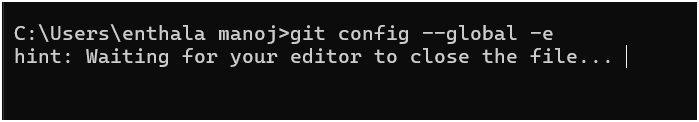
**4. Set Notepad++ as Git editor**

**git config --global core.editor "notepad++.exe -multiInst -nosession"**

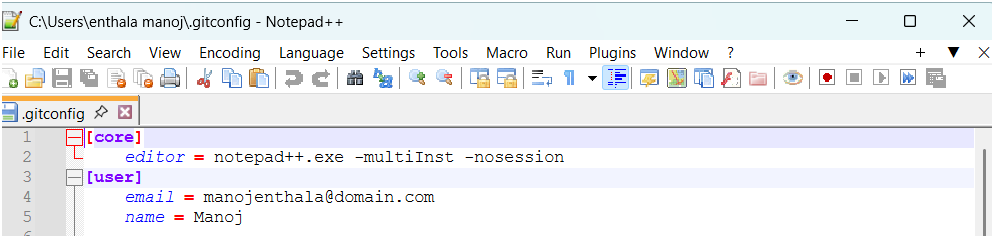
****

**5. Test editor configuration**

**git config --global -e**

****

**Notepad++ will open the config file.**

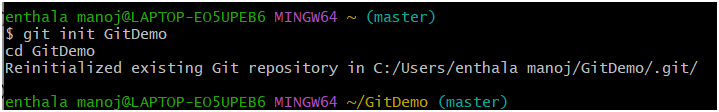
****

**Step 3: Add a File to Git Repository**

**1. Create new local Git repo**

**git init GitDemo**

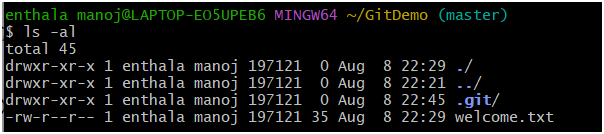
**cd GitDemo**

****

**2. Confirm .git folder was created**

**ls -al**

**You should see .git/ listed.**

****

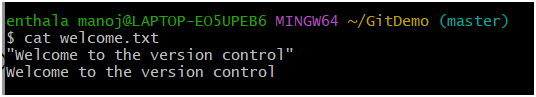
**3. Create a file**

**echo "Welcome to the version control" >> welcome.txt**

****

**4. Check file content**

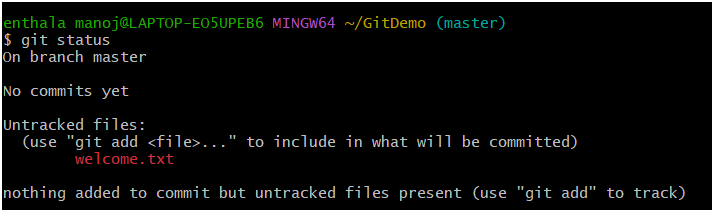
**cat welcome.txt**

****

**5. Check Git status**

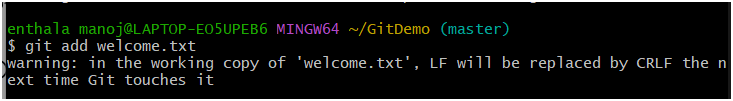
**git status**

**You'll see:**

****

**6. Track the file**

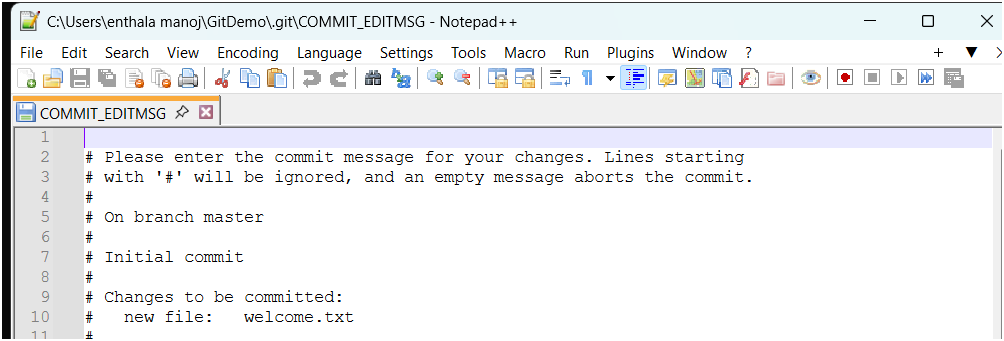
**git add welcome.txt**

**7. Commit with Notepad++ as editor**

**git commit**

****

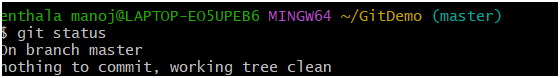
**This opens Notepad++. Add multi-line commit message. Save and close.**

****

**8. Confirm commit**

**git status**

**Output:**

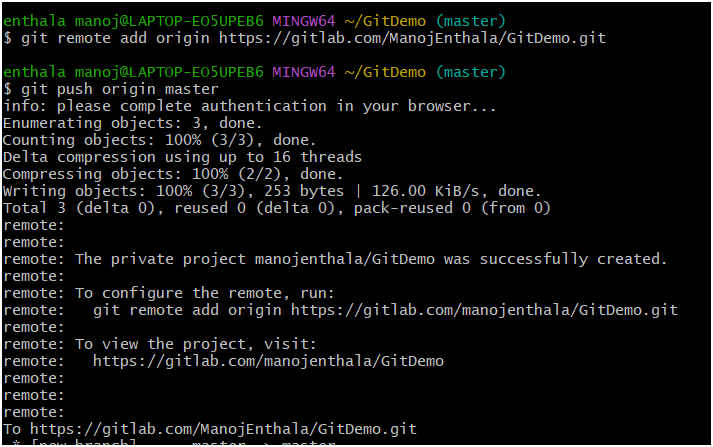
**☁️ Step 4: Push to GitLab (Remote Repository)**

**1. Create a free GitLab account (use personal email)**

* **Go to** [**https://gitlab.com**](https://gitlab.com)
* **Create a project named GitDemo**

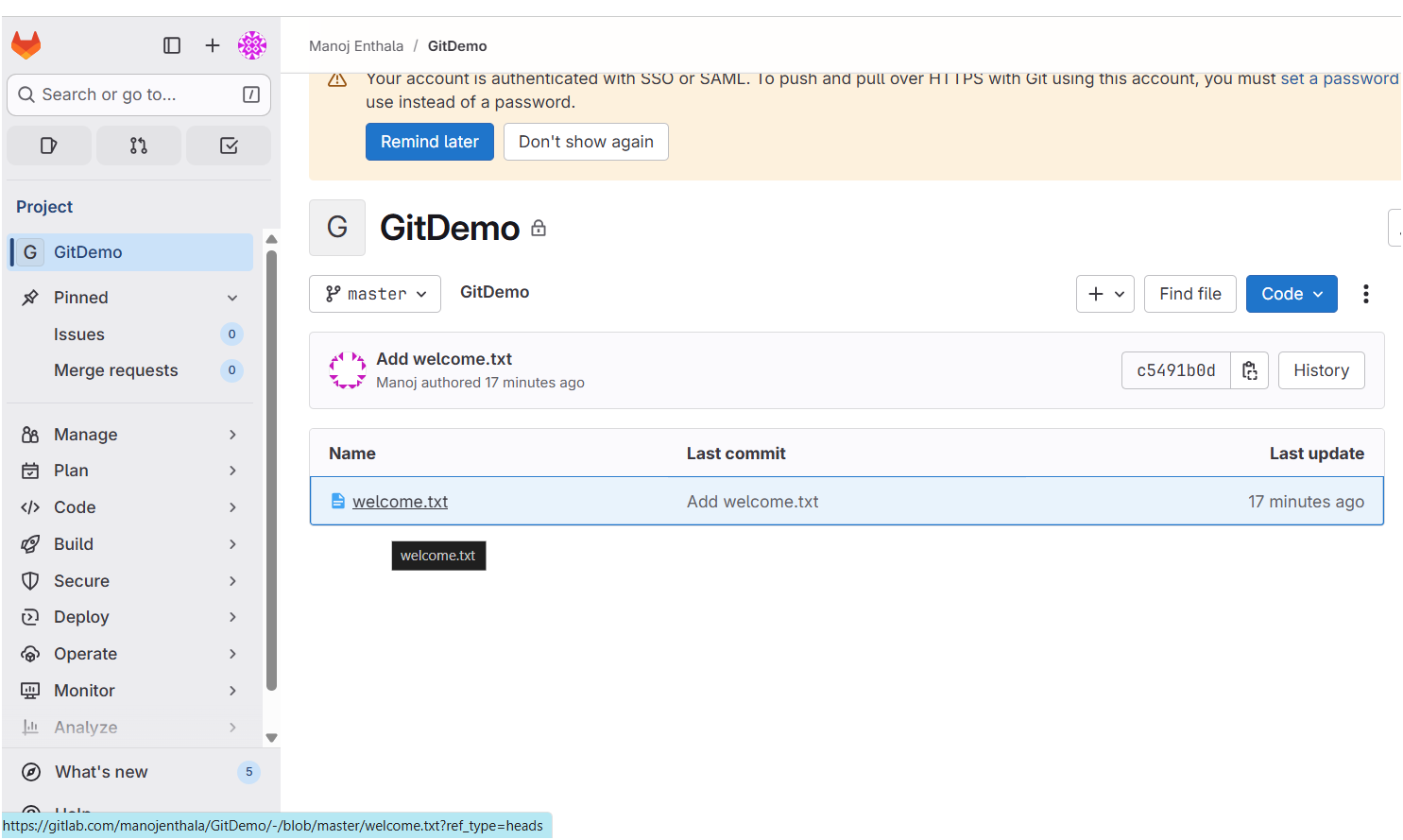
**2. Link your local repo to remote**

**git remote add origin https://gitlab.com/yourusername/GitDemo.git**

****

**4. Push local to remote**

**git push origin master**

**EX 2:** **Explain git ignore**

**Explain how to ignore unwanted files using git ignore**

**1. What is .gitignore?**

* **.gitignore is a text file that tells Git which files or folders it should ignore when tracking changes.**
* **This means the files listed inside .gitignore won’t be staged, committed, or pushed to the repository.**
* **Useful for:**
  + **Temporary files (.log, .tmp, etc.)**
  + **Build outputs (dist/, bin/)**
  + **Sensitive files (.env, credentials)**
  + **System files (Thumbs.db, .DS\_Store)**

**2. How Git Decides What to Ignore**

**Git looks for .gitignore rules from top to bottom in:**

1. **The repository’s root .gitignore file.**
2. **Any .gitignore files inside subdirectories.**
3. **Global .gitignore settings (affecting all repos on your computer).**

**3. Syntax Rules for .gitignore**

* **filename.ext → ignore a specific file.**
* **\*.ext → ignore all files with that extension.**
* **/foldername/ → ignore a specific folder.**
* **!filename.ext → don’t ignore a file (exception rule).**
* **# comment → comment inside .gitignore.**

**# Ignore all log files**

**\*.log**

**# Ignore the log folder**

**/logs/**

**# But track important.log**

**!important.log**

**4. Steps to Ignore Unwanted Files**

**In Git Bash**

**# Step 1: Create or edit the .gitignore file**

**nano .gitignore**

**Add your ignore patterns inside, e.g.:**

**\*.log**

**/log/**

**# Step 2: Save and close the file (Ctrl+O, Enter, Ctrl+X in nano)**

**# Step 3: Stage the .gitignore file**

**git add .gitignore**

**# Step 4: Commit the change**

**git commit -m "Add .gitignore to ignore .log files and log folder"**

**5. Important Note**

**If a file is already tracked by Git, adding it to .gitignore will not stop Git from tracking it.  
You must first remove it from the index:**

**bash**

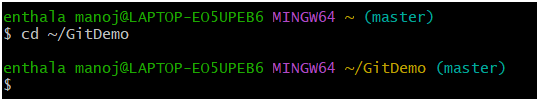
**CopyEdit**

**git rm --cached filename.log**

**Then commit, and .gitignore will take effect.**

**Go to your working directory**

**cd ~/GitDemo**

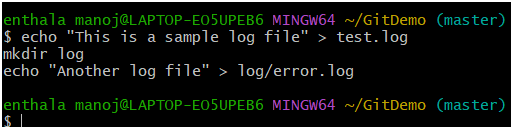
****

**Create a .log file and a log folder**

**echo "This is a sample log file" > test.log**

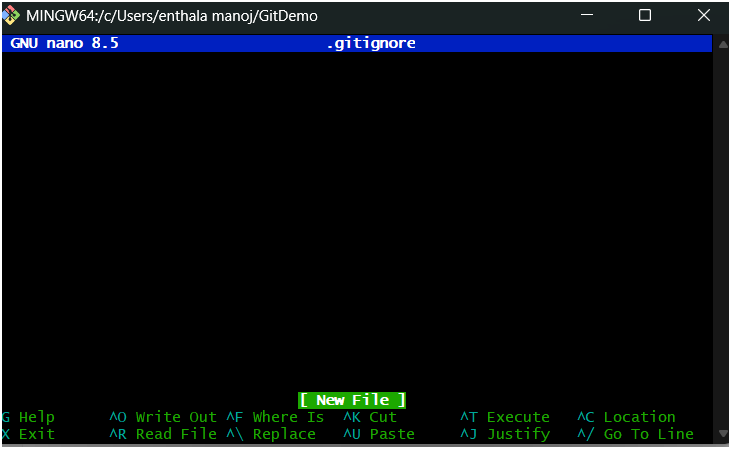
**mkdir log**

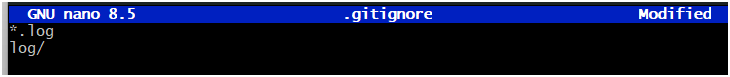
**echo "Another log file" > log/error.log**

****

**Create/Edit .gitignore**

**nano .gitignore**

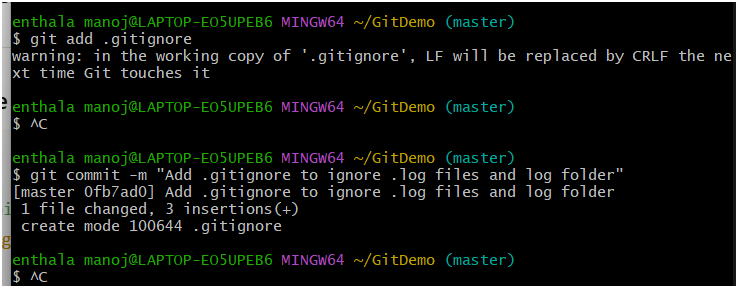
****

****

**Stage and commit .gitignore**

**git add .gitignore**

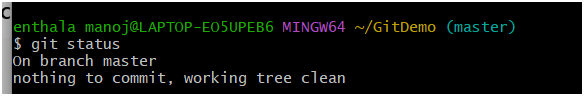
**git commit -m "Add gitignore to ignore .log files and log folder"**

****

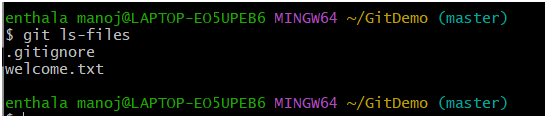
**Verify Git ignores unwanted files**

**git status**

**You should see something like:**

**Quick check**

**git ls-files**

****

**EX 3:** **Explain branching and merging**

* **Explain about creating a branch request in GitLab**
* **Explain about creating a merge request in GitLab**

**1. Branching and Merging in Git**

**Branching**

* **A branch is a separate line of development.**
* **The master (or main) branch is usually your stable production-ready code.**
* **You create a branch to work on new features, bug fixes, or experiments without affecting the main code.**

**Commands in Git Bash:**

**# Create a new branch**

**git branch feature-login**

**# Switch to the branch**

**git checkout feature-login**

**# or in one step:**

**git checkout -b feature-login**

**Merging**

* **Merging combines changes from one branch into another.**
* **You usually merge your feature branch into master when your work is ready.**

**Commands:**

**# Switch to master**

**git checkout master**

**# Merge feature branch into master**

**git merge feature-login**

* **If there’s no conflict, Git merges automatically.**
* **If there’s a conflict, you must resolve it manually (like I explained earlier).**

**2. Creating a Branch Request in GitLab**

**In GitLab, a "branch request" isn’t a Git term — it’s basically pushing your branch to the remote repository so others can see it.**

**Steps:**

1. **Create a branch locally in Git Bash:**

**git checkout -b GitNewBranch**

1. **Push the branch to GitLab:**

**git push origin GitNewBranch**

1. **This makes the branch available in GitLab’s web interface.**

**3. Creating a Merge Request in GitLab**

**In GitLab, a Merge Request (MR) is the web-based way to review and merge changes.**

**Steps:**

1. **Go to your GitLab repository in a browser.**
2. **GitLab will usually show a banner like “Create merge request” after you push a branch.**
3. **Click Create merge request.**
4. **Fill in:**
   * **Source branch = your feature branch (e.g., GitNewBranch)**
   * **Target branch = master (or main)**
   * **Description of changes**
5. **Submit the MR.**
6. **Team members can review, comment, and approve before merging.**

**Branching**

**Create a new branch “GitNewBranch”**

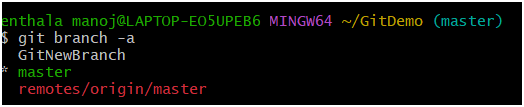
**git branch GitNewBranch**

****

**List all local and remote branches**

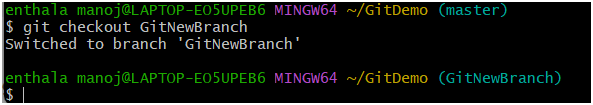
**git branch -a**

* **The \* shows your current branch.**
* **Right now, it should still be master.**

****

**Switch to the new branch**

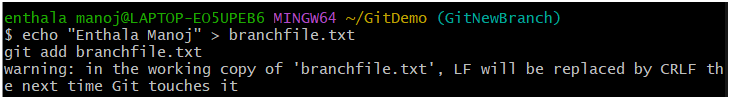
**git checkout GitNewBranch**

****

**Add files to the new branch**

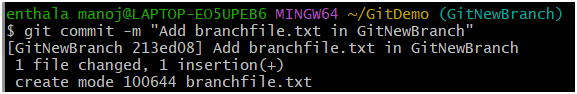
**echo "This file is for the new branch" > branchfile.txt**

**git add branchfile.txt**

****

**Commit changes to the branch**

**git commit -m "Add branchfile.txt in GitNewBranch"**

****

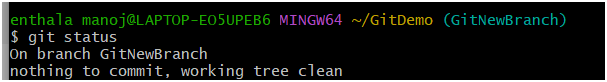
**Check status**

**git status**

**It should say:**

**On branch GitNewBranch**

**nothing to commit, working tree clean**

****

**Merging**

**Switch back to master**

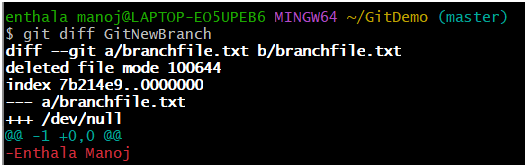
**git checkout master**

****

**See differences between master and the branch**

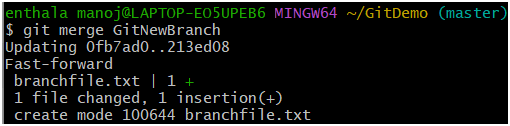
**git diff GitNewBranch**

**This shows text-based differences.**

****

**Merge the branch into master**

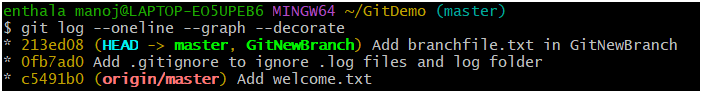
**git merge GitNewBranch**

****

**Observe merge log**

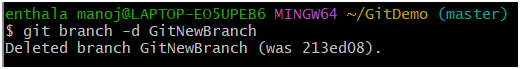
**git log --oneline --graph --decorate**

**This will show a visual graph of commits, with decorations for branches.**

****

**Delete the branch after merging**

**git branch -d GitNewBranch**

****

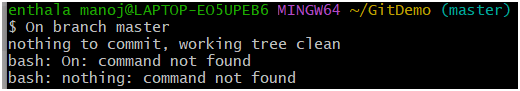
**Check status**

**git status**

**It should say:**

**On branch master**

**nothing to commit, working tree clean**

****

**Ex 4:** **Explain how to resolve the conflict during merge.**

**When two branches change the same part of a file differently, Git can’t decide which change to keep.  
That’s a merge conflict — and you must resolve it manually.**

**How to resolve a merge conflict**

**1. Understand when conflicts happen**

**A conflict happens during:**

* **git merge <branch>**
* **git pull (which internally does a merge)**
* **git rebase**

**Example:**

* **Branch master: hello.xml → <greeting>Hello from master</greeting>**
* **Branch GitWork: hello.xml → <greeting>Hello from GitWork</greeting>**

**When you merge them, Git doesn’t know which greeting is “correct”.**

**2. How Git shows a conflict**

**When Git detects a conflict, it:**

* **Stops the merge**
* **Marks the file with conflict markers:**

**xml**

**CopyEdit**

**<<<<<<< HEAD**

**<greeting>Hello from master</greeting>**

**=======**

**<greeting>Hello from GitWork</greeting>**

**>>>>>>> GitWork**

* **HEAD = your current branch**
* **After ======= = the changes from the branch you’re merging in**

**3. Resolve the conflict manually**

**You must edit the file and keep the correct/combined version, removing all markers.**

**Example resolution:**

**Xml:**

**<greeting>Hello from both master and GitWork branches</greeting>**

**4. Mark the conflict as resolved**

**After editing:**

**git add hello.xml**

**5. Commit the merge**

**git commit**

**Git will create a merge commit showing that both branches are now combined.**

**6. Verify**

**git status**

**git log --oneline --graph --decorate**

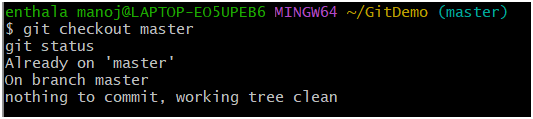
**Git Conflict Resolution Lab Steps**

**1. Verify master is clean**

**git checkout master**

**git status**

**Output should be:**

**2. Create branch GitWork**

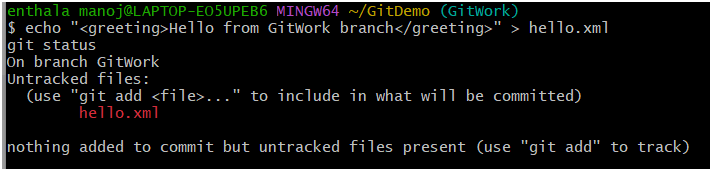
**git checkout -b GitWork**

****

**3. Add hello.xml in GitWork**

**echo "<greeting>Hello from GitWork branch</greeting>" > hello.xml**

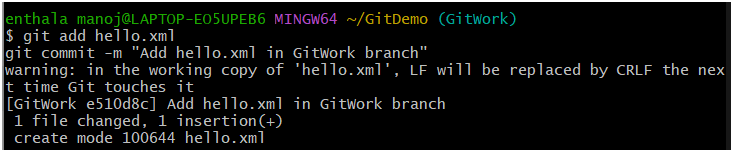
**git status**

****

**4. Commit changes in branch**

**git add hello.xml**

**git commit -m "Add hello.xml in GitWork branch"**

****

**5. Switch to master**

**git checkout master**

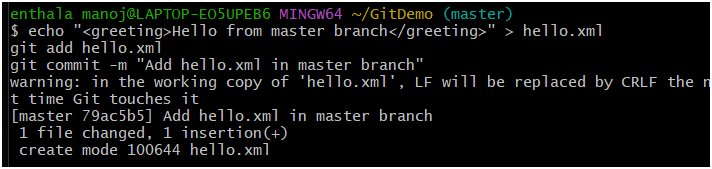
****

**6. Add hello.xml with different content in master**

**echo "<greeting>Hello from master branch</greeting>" > hello.xml**

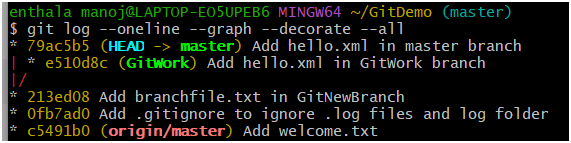
**git add hello.xml**

**git commit -m "Add hello.xml in master branch"**

****

**7. View log graph**

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

****

**8. Check differences before merge**

**git diff GitWork**

****

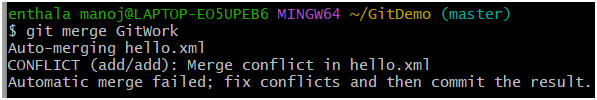
**9. Merge GitWork into master**

**git merge GitWork**

**Here, you should get:**

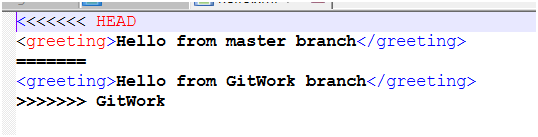
**CONFLICT (add/add): Merge conflict in hello.xml**

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

****

**10. Resolve the conflict**

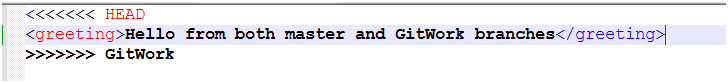
**Open hello.xml in Notepad++ or any editor. You’ll see:**

****

**Edit to resolve:**

**<greeting>Hello from both master and GitWork branches</greeting>**

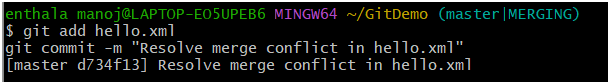
**Save the file.**



**11. Stage and commit the resolved file**

**git add hello.xml**

**git commit -m "Resolve merge conflict in hello.xml"**

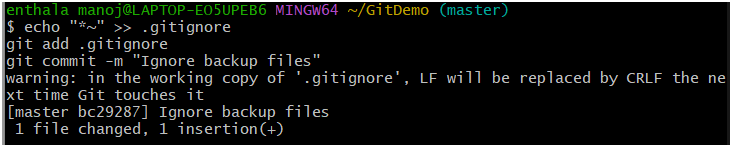
****

**12. Update .gitignore for backup files**

**echo "\*~" >> .gitignore**

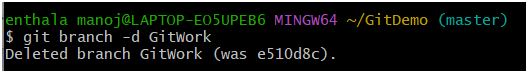
**git add .gitignore**

**git commit -m "Ignore backup files"**

****

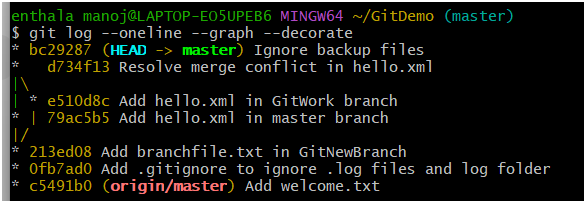
**13. Delete GitWork after merge**

**git branch -d GitWork**

****

**14. Final log view**

**git log --oneline --graph –decorate**

****

**Ex 5:** **Explain how to clean up and push back to remote Git**

**Cleaning up and pushing back to remote Git basically means:**

1. **Making sure your local repository is tidy (no untracked or unnecessary files, no half-done changes).**
2. **Making sure your local branch is synced with the remote repository.**
3. **Pushing your final, committed changes back to the remote so others can see them.**

**Step-by-step explanation**

**1. Verify your branch is clean**

**git status**

* **If you see nothing to commit, working tree clean, your branch has no uncommitted changes.**
* **If you have unwanted untracked files, remove them with:**

**git clean -fd**

***(Be careful—this deletes untracked files and folders.)***

**2. Check your branches**

**git branch -a**

**This shows your local (\* mark = current branch) and remote branches.**

**3. Sync with remote**

**Before pushing, make sure your local branch has the latest changes from the remote:**

**git pull origin master**

**If you’re on a different branch, replace master with your branch name.**

**4. Push your changes**

**If your commits are ready, push them to the remote:**

**git push origin master**

**or for another branch:**

**git push origin <branchname>**

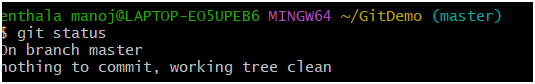
**5. Verify on remote**

* **Go to your GitHub/GitLab repository.**
* **Check the commit history or file changes.**
* **Confirm your updates are there.**

**1. Verify if master is in clean state**

**git status**

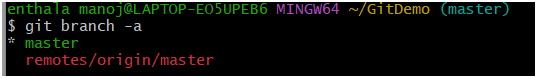
**If it says nothing to commit, working tree clean, you’re good.**

****

**2. List out all the available branches**

**git branch -a**

**This will show both local and remote branches.**

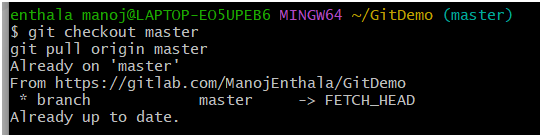
****

**3. Pull the remote git repository to the master**

**git checkout master**

**git pull origin master**

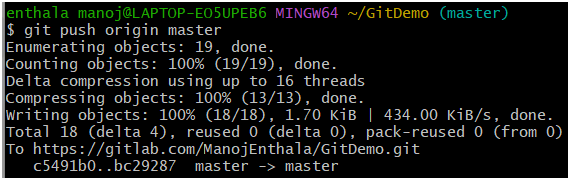
**This ensures your master branch is up to date with the remote.**

****

**4. Push the changes from previous lab to remote**

**If you already committed changes during Git-T03-HOL\_002, push them:**

**git push origin master**

****

**5. Verify changes in remote repository**

* **Go to your repository on GitHub/GitLab in a browser.**
* **Refresh the page and confirm your commits/files appear.**