**WEEK-08 HANDSON SOLUTIONS**

**GIT**

**EXERCISE-01** 1. Git-HOL

**Question:**

Familiar with Git commands like git init, git status, git add, git commit, git push, and git pull.

In this hands-on lab, you will learn how to

· Setup your machine with Git Configuration

· Integrate notepad++.exe to Git and make it a default editor

· Add a file to source code repository

Prerequisites

· Install Git Bash client in your machine

Notes:

Please follow the below steps for creating a free account in GitHub. Don’t use cognizant credentials to login to GitHub.

Estimated time to complete this lab: 30 minutes.

Step 1: Setup your machine with Git Configuration

To create a new repository, signup with GitLab and register your credentials

Login to GitLab and create a “GitDemo” project

1. To check if Git client is installed properly: Open Git bash shell and execute

If output shows Git with its version information that indicates, that Git Client installs properly.

2. To configure user level configuration of user ID and email ID execute

3. To check if the configuration is properly set, execute the following command.

Step 2: Integrate notepad++.exe to Git and make it a default editor

1. To check, if notepad++.exe execute from Git bash

If Git bash could not able to recognize notepad++ command that implies notepad++.exe is note added to the environment path variable.

To add path of notepad++.exe to environment variable, go to control panel -> System -> Advanced System settings. Go to Advanced tab -> Environment variables -> Add path of notepad++.exe to the path user variable by clicking on “Edit

2. Exit Git bash shell, open bash shell and execute

Now, notepad++ will open from Git bash shell

3. To create an alias command for notepad++.exe, execute

It will open notepad++ from bash shell, and create a user profile by adding the line in notepad++

4. To configure the editor, execute the command

5. To verify if notepad++ is the default editor, execute the command

Here ‘-e’ option implies editor

It will show the entire global configuration as shown below,

Step 3: Add a file to source code repository

1. Open Git bash shell and create a new project “GitDemo” by executing the command

2. Git bash initializes the “GitDemo” repository. To verify, execute the command

It will display all the hidden files in the Git “working directory”.

3. To create a file “welcome.txt” and add content to the file, execute the command

4. To verify if the file “welcome.txt” is created, execute

5. To verify the content, execute the command

6. Check the status by executing

Now the file “welcome.txt” is available in Git “working directory”

7. To make the file to be tracked by Git repository, execute the command

8. To add multi line comments, we are opening default editor to comment. Execute the command

Notepad++ editor will open and to add multi-line comment with default editor

9. To check if local and “Working Directory” git repository are same, execute git status

welcome.txt is added to the local repository.

10. Signup with GitLab and create a remote repository “GitDemo”

11. To pull the remote repository, execute

git pull origin master

12. To push the local to remote repository, execute

git push origin master

IMPLEMENTATION:

In this handson lab, the process of configuring Git, integrating a custom text editor, and managing both local and remote repositories was successfully implemented. I initiated by setting up the local Git environment. A dedicated project folder named GitDemo was created, and the git init command was executed to initialize it as a Git repository. User credentials were configured globally using git config --global user.name and git config --global user.email, ensuring that all commits are properly attributed to my name and email.

Name-gitsish

[Email-22501a4401@pvpsit.ac.in](mailto:Email-22501a4401@pvpsit.ac.in)

A new text file, welcome.txt, containing the content “Welcome to the version control” was created. This file was staged using git add welcome.txt and committed to the local repository with git commit -m "Initial commit: Added welcome.txt". This process demonstrated the fundamental Git workflow of creating, staging, and committing changes.

Next, I focused on configuring **Notepad++** as the default Git editor. The installation directory of Notepad++ was verified and added to the system PATH variable to enable execution from Git Bash. Git’s core editor setting was updated using the command:

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

This allowed Git to automatically open commit messages and configuration files in Notepad++, streamlining the commit editing process.

A remote repository was then created in **GitLab** under the *Personal* namespace, named GitDemo. The repository was intentionally left empty to allow a clean push from the local environment. The HTTPS URL of the GitLab repository was obtained and linked to the local repository using the git remote add origin <URL> command.

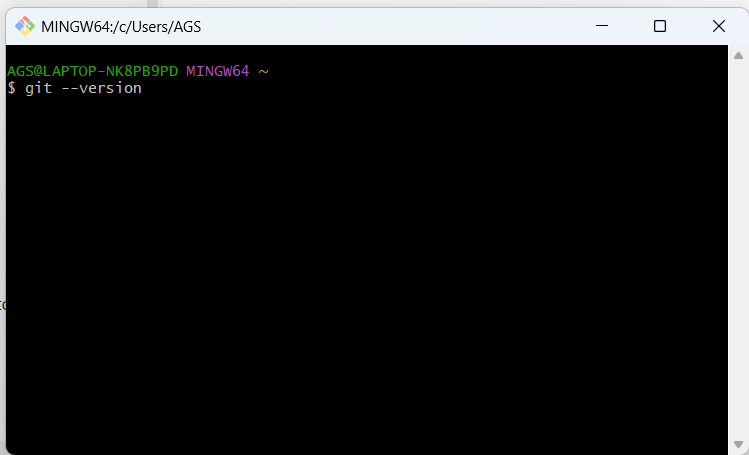
Before pushing, the current branch name was verified with git branch, as newer Git installations may default to main instead of master. The local repository was then pushed to the remote using either git push -u origin main or git push -u origin master, depending on the branch name. The -u flag ensured that the remote branch was set as the upstream branch, allowing simplified push and pull commands in future operations.

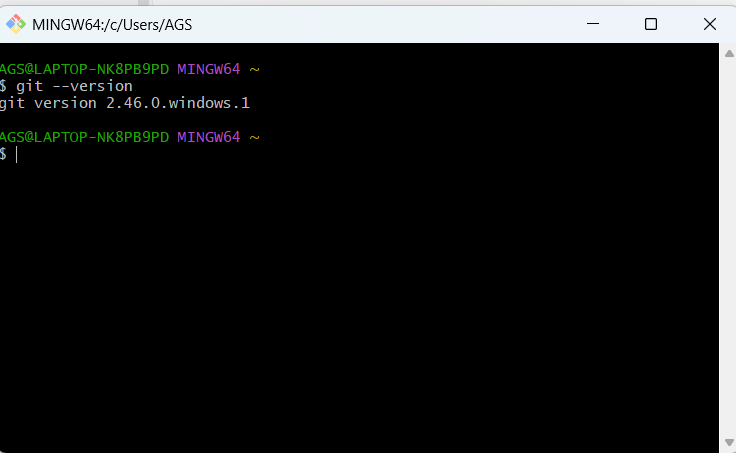
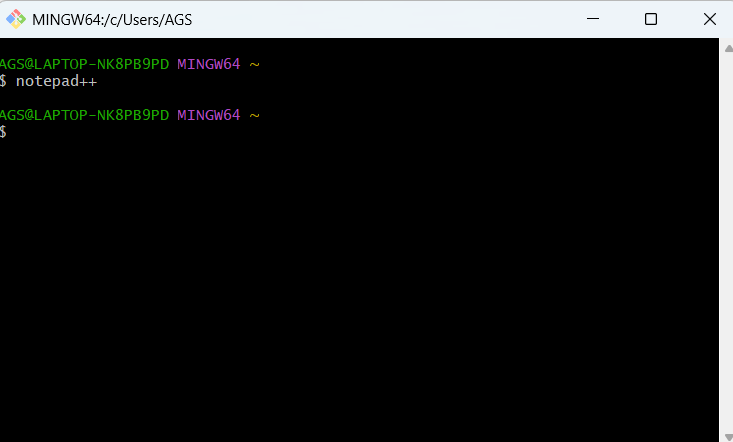
Finally, a git pull command was executed to confirm connectivity between the local and remote repositories. As expected, the output indicated *“Already up to date”* because the remote had just been updated from the local push.

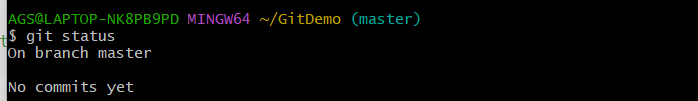
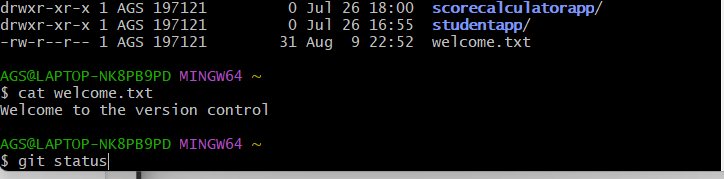
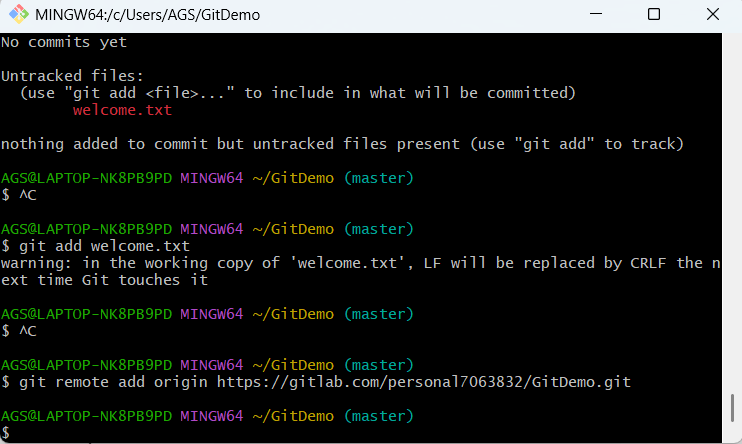
By the end of this exercise, the Git environment was fully configured, Notepad++ was successfully integrated as the default editor, and the complete workflow of initializing a repository, committing changes, linking to a remote repository, and synchronizing via push and pull operations was demonstrated.

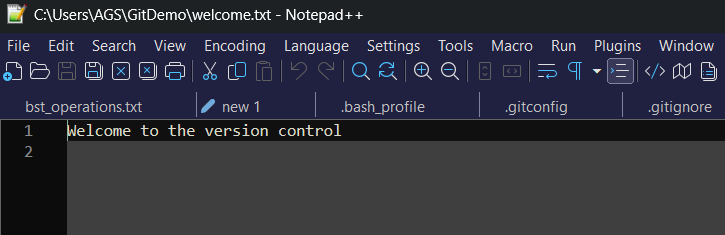
This practical implementation reinforced proficiency with essential Git commands including git init, git config, git add, git commit, git remote add, git push, and git pull, as well as provided hands-on experience with editor integration and remote repository management.

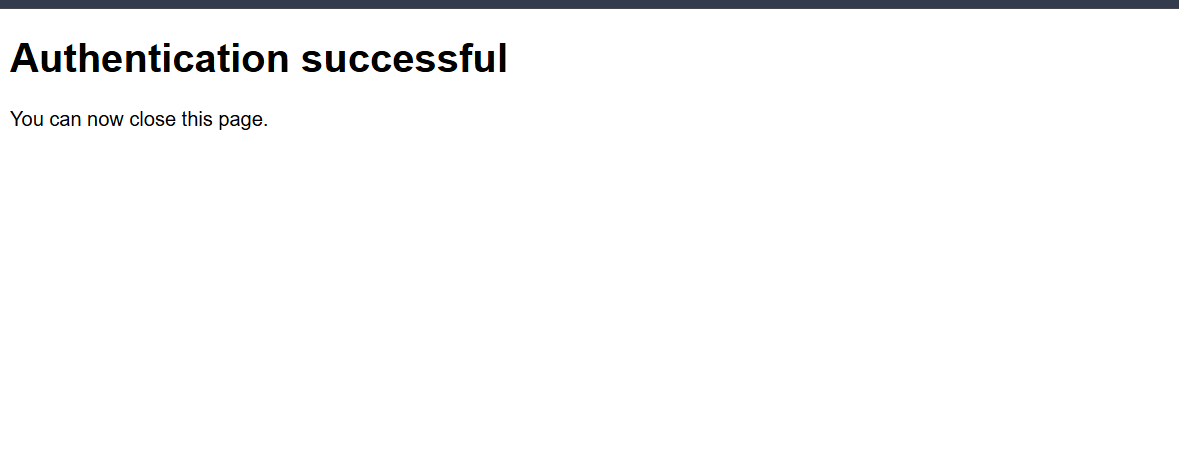
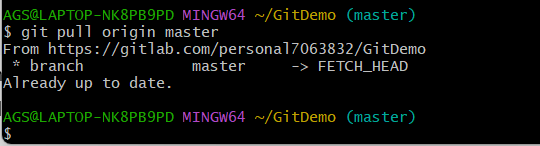
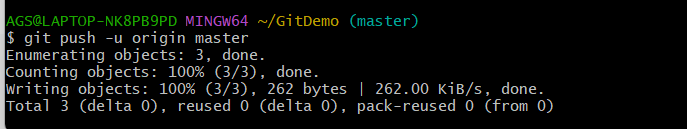
OUTPUT:











**EXERCISE-02** 2. Git-HOL

· Explain git ignore

· Explain how to ignore unwanted files using git ignore

In this hands-on lab, you will learn how to:

· Implement git ignore command to ignore unwanted files and folders

Prerequisites

The following are the pre-requisites to complete this hands-on lab:

· Setting up Git environment

· Integrate notepad++ as a default editor

· A Git repository in the local system and a remote repository in GitLab

Notes\*:

Please follow the below steps for creating a free account in GitHub. Do not use cognizant credentials to login to GitHub.

Estimated time to complete this lab: 20 minutes.

Create a “.log” file and a log folder in the working directory of Git. Update the .gitignore file in such a way that on committing, these files (.log extensions and log folders) are ignored.

Verify if the git status reflects the same about working directory, local repository and git repository.

IMPLEMENTATION:

In this hands-on lab, the objective was to understand the concept of .gitignore in Git and apply it to ignore specific unwanted files and folders from being tracked in the repository. The .gitignore file allows to define patterns that match files or directories which Git should ignore, thereby keeping the repository clean and avoiding unnecessary or sensitive files from being committed.

The requirement specified was to create a .log file and a log folder in the Git working directory, then configure the .gitignore file such that all files with the .log extension and the entire log folder would be ignored by Git. This needed to be tested and verified by checking the git status output to confirm that these files and folders were not tracked after the .gitignore configuration.

The implementation began by ensuring that the Git environment was properly configured, with a local repository initialized and linked to a remote repository on GitLab. A sample log file and a log folder were created in the repository using the following commands:

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

mkdir log

echo "log data" > log/app.log

A .gitignore file was then created in the root of the repository and edited using Notepad++ to include the following entries:

\*.log

log/

The \*.log pattern instructs Git to ignore all files ending with the .log extension across the repository, while log/ tells Git to ignore the entire log directory and its contents.

After saving the .gitignore file, its presence as an untracked file was confirmed using git status. The .gitignore file was then staged and committed with:

git add .gitignore

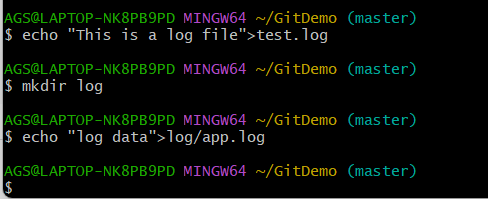
git commit -m "Added .gitignore to ignore log files and log folder"

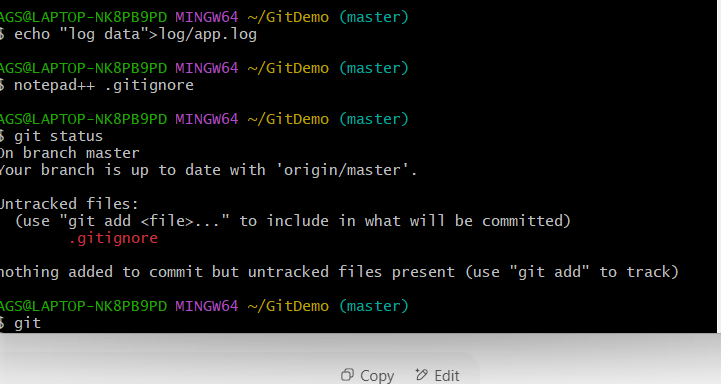
Finally, the changes were pushed to the remote repository using git push.

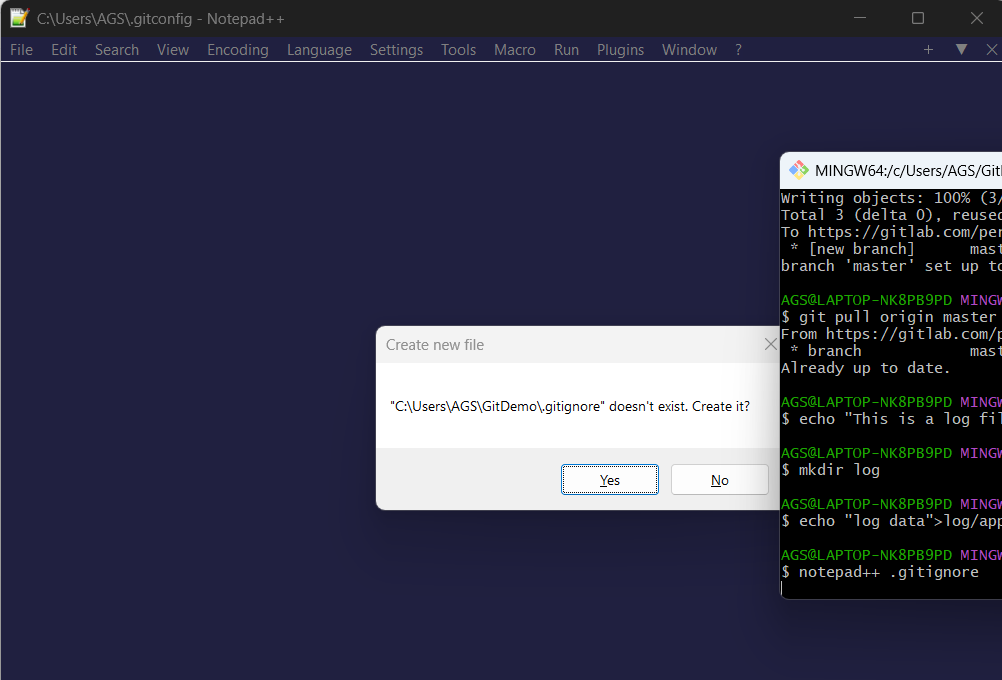
The verification step involved running git status again, which displayed “nothing to commit, working tree clean”, confirming that the .log file and the log folder were successfully excluded from tracking. Only the .gitignore file itself was added to version control.

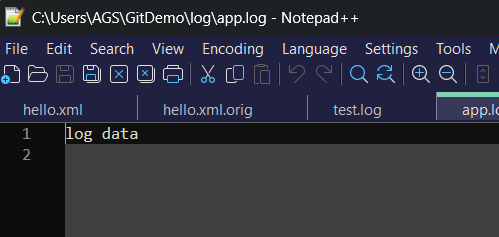
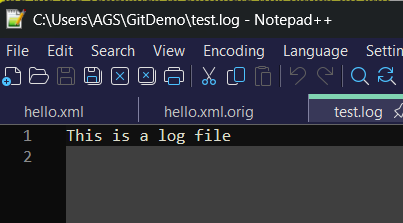
Through this implementation, the functionality of .gitignore was demonstrated effectively, meeting the lab’s objectives. This exercise reinforced the importance of ignoring temporary, generated, or sensitive files to maintain a clean and efficient repository.

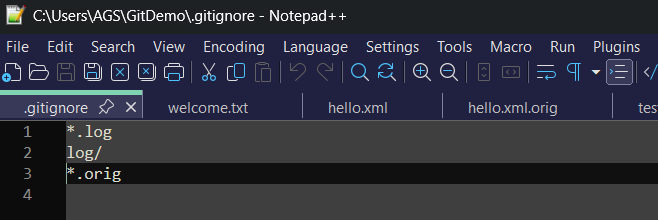
OUTPUT:

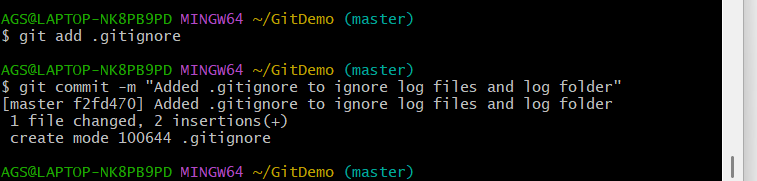


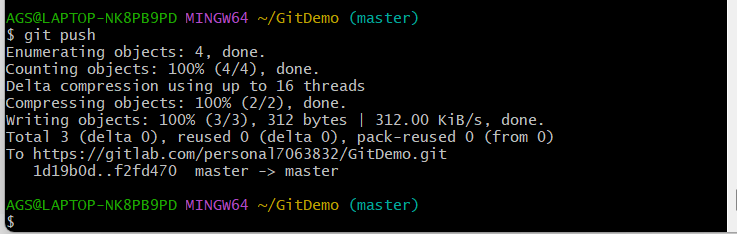












**EXERCISE-03** 1. Git-HOL

Explain branching and merging

· Explain about creating a branch request in GitLab

· Explain about creating a merge request in GitLab

In this hands-on lab, you will learn how to:

· Construct a branch, do some changes in the branch, and merge it with master (or trunk)

Prerequisites

The following are the pre-requisites to complete this hands-on lab:

· Setting up Git environment with P4Merge tool for Windows

Notes\*:

Please follow the below steps for creating a free account in GitHub. Do not use cognizant credentials to login to GitHub.

Estimated time to complete this lab: 30 minutes.

Please follow the instruction to complete the hands-on. Each instruction expects a command for the Git Bash.

Branching:

1. Create a new branch “GitNewBranch”.

2. List all the local and remote branches available in the current trunk. Observe the “\*” mark which denote the current pointing branch.

3. Switch to the newly created branch. Add some files to it with some contents.

4. Commit the changes to the branch.

5. Check the status with “git status” command.

Merging:

1. Switch to the master

2. List out all the differences between trunk and branch. These provide the differences in command line interface.

3. List out all the visual differences between master and branch using P4Merge tool.

4. Merge the source branch to the trunk.

5. Observe the logging after merging using “git log –oneline –graph –decorate”

6. Delete the branch after merging with the trunk and observe the git status.

IMPLEMENTATION:

In this hands-on lab, the objective was to understand and implement branching and merging in Git, and to utilize the **P4Merge** visual tool for identifying and reviewing differences between branches.

Branching allows developers to create independent lines of development, while merging integrates those changes back into a primary branch (commonly master or main).

P4Merge serves as a graphical diff/merge tool, enabling a clear, side-by-side visualization of file changes.

The lab commenced in the existing GitDemo repository.

A new branch named GitNewBranch was created using the git branch GitNewBranch command and was activated with git checkout GitNewBranch.

Inside this branch, a new file named branchfile.txt was created and populated with the text *“Added in GitNewBranch”*.

This file was staged using git add branchfile.txt and committed with the message *“Added branchfile.txt in GitNewBranch”*.

To enable visual comparison, **P4Merge** was installed and configured as the default diff and merge tool for Git using:

git config --global merge.tool p4merge

git config --global mergetool.p4merge.path "C:/Program Files/Perforce/p4merge.exe"

git config --global diff.tool p4merge

git config --global difftool.p4merge.path "C:/Program Files/Perforce/p4merge.exe"

Switching back to the master branch with git checkout master, the differences between master and GitNewBranch were displayed visually using the command:

git difftool master GitNewBranch

This launched the P4Merge graphical interface, clearly showing the newly added file in the branch compared to master.

The P4Merge visual interface displayed the difference clearly: on the left side (master) there was no branchfile.txt, while on the right side (GitNewBranch) the new file and its content “Added in GitNewBranch” were present.

This visual side-by-side view made it easy to identify that the only change between the branches was the addition of this new file.

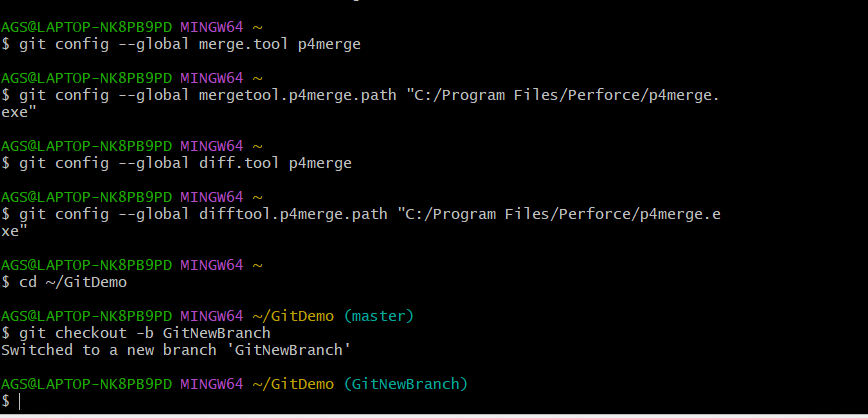
Merging was then performed by executing git merge GitNewBranch. Since master had not diverged from the branch, the result was a fast-forward merge, bringing the commit from GitNewBranch directly into master.

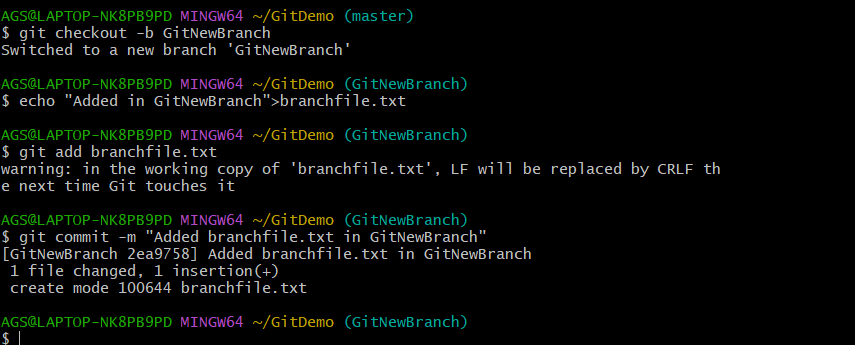
The command git log --oneline --graph --decorate confirmed the updated commit history, showing the branch commit now part of the master branch. Finally, the merged branch was deleted locally using git branch -d GitNewBranch.

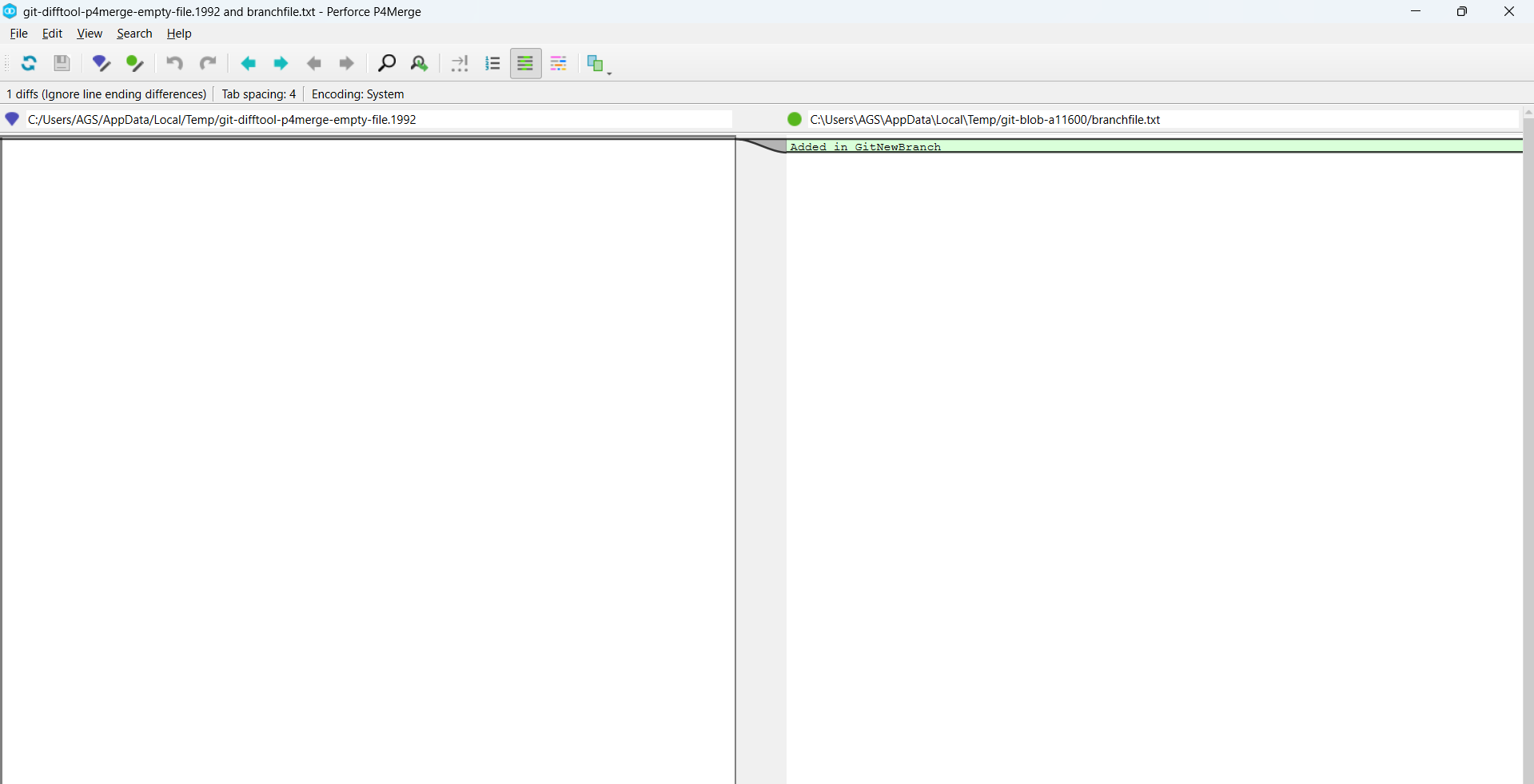
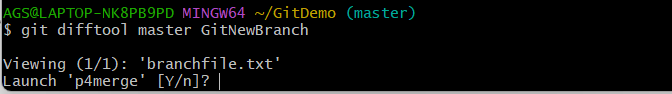
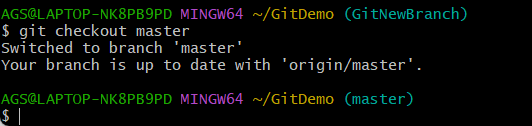
This exercise successfully demonstrated the complete workflow of creating a branch, making independent changes, visually comparing differences using P4Merge, and merging changes back into the main development branch.

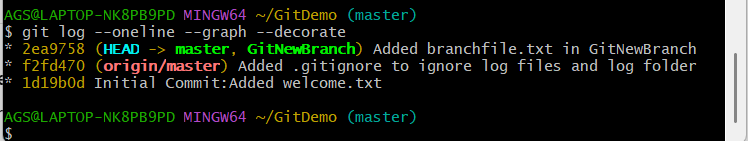
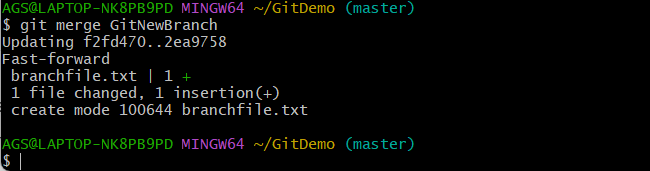
The process reinforced key Git operations including git branch, git checkout, git add, git commit, git merge, and git log, as well as the integration of an external tool (P4Merge) for enhanced version control visualization.

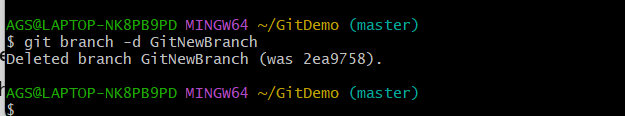
OUTPUT:





****

****

****

**EXERCISE-04-** 4. Git-HOL

· Explain how to resolve the conflict during merge.

In this hands-on lab, you will learn how to:

· Implement conflict resolution when multiple users are updating the trunk (or master) in such a way that it results into a conflict with the branch’s modification.

Prerequisites

The following are the pre-requisites to complete this hands-on lab:

· Hands-on ID: “Git-T03-HOL\_001”

Notes\*:

Please follow the below steps for creating a free account in GitHub. Do not use cognizant credentials to login to GitHub.

Estimated time to complete this lab: 30 minutes.

Please follow the instructions to complete the hands-on. Each instruction expect a command for the Git Bash.

1. Verify if master is in clean state.

2. Create a branch “GitWork”. Add a file “hello.xml”.

3. Update the content of “hello.xml” and observe the status

4. Commit the changes to reflect in the branch

5. Switch to master.

6. Add a file “hello.xml” to the master and add some different content than previous.

7. Commit the changes to the master

8. Observe the log by executing “git log –oneline –graph –decorate –all”

9. Check the differences with Git diff tool

10. For better visualization, use P4Merge tool to list out all the differences between master and branch

11. Merge the bran to the master

12. Observe the git mark up.

13. Use 3-way merge tool to resolve the conflict

14. Commit the changes to the master, once done with conflict

15. Observe the git status and add backup file to the .gitignore file.

16. Commit the changes to the .gitignore

17. List out all the available branches

18. Delete the branch, which merge to master.

19. Observe the log by executing “git log –oneline –graph –decorate

IMPLEMENTATION:

In this hands-on exercise, I was tasked with resolving a merge conflict between two Git branches using **P4Merge** as the visual merge tool, while also following best practices for repository maintenance.

The objective was to simulate a real-world conflict scenario, visually inspect the changes, decide on the final merged version, and ensure that unnecessary files (such as merge backups) were ignored in future commits.

To start, I created and worked on a separate branch named GitWork alongside the existing master branch.

In both branches, I created a file named hello.xml but with different <message> content, ensuring a merge conflict would occur.

Before merging, I ran the command git difftool master GitWork to launch **P4Merge** and visually analyze the differences — the GitWork branch displayed <message>Hello from GitWork branch -updated</message> while the master branch displayed <message>Hello form Master branch</message>.

The visual interface made it clear that the only difference was in the <message> line, helping me identify which version to keep.

When I executed git merge GitWork, Git detected a conflict in hello.xml.

To resolve it, I ran git mergetool, which opened P4Merge in three-way merge mode showing the base version, the local version (master), and the remote version (GitWork).

I edited the merged output panel to keep the intended final content from GitWork, saved the file, and closed P4Merge.

This action signaled to Git that the conflict was resolved, and I committed the changes with git commit -m "Resolved merge conflict in hello.xml".

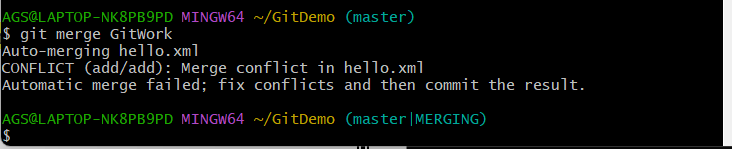
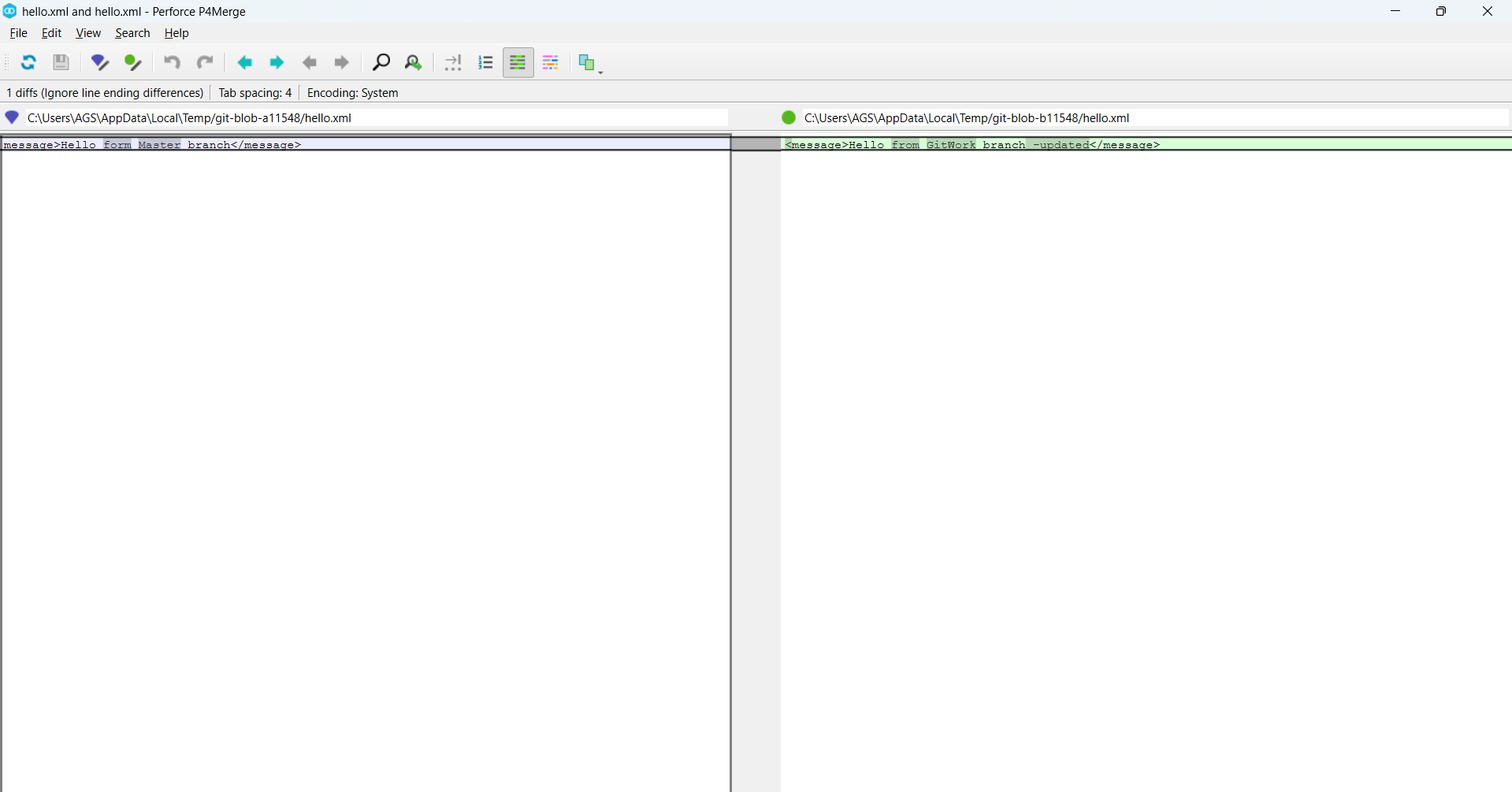
Next, as part of the requirement to configure .gitignore to avoid tracking unwanted merge backup files, I appended the pattern \*.orig to .gitignore and committed it.

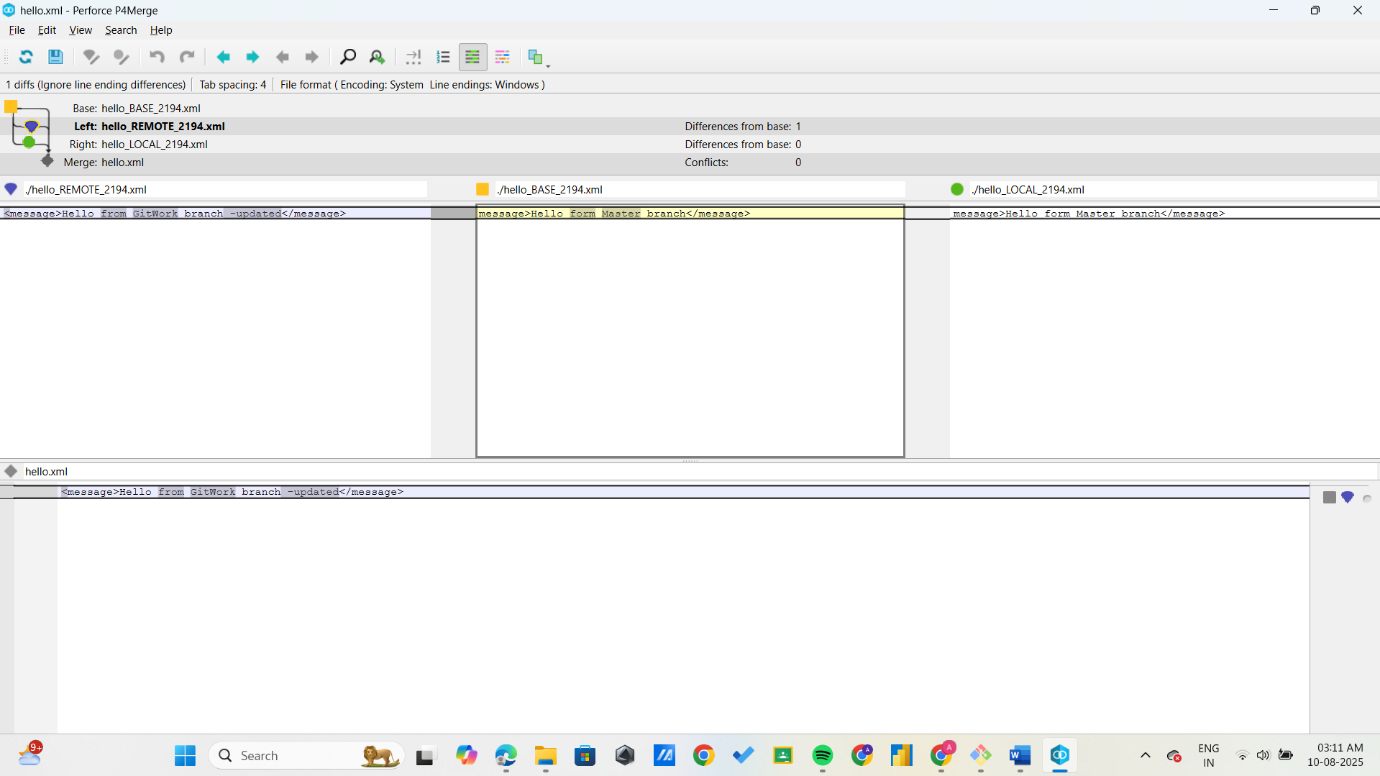
Finally, I deleted the merged branch using git branch -d GitWork and reviewed the repository history using git log --oneline --graph --decorate, confirming that the merge commit, .gitignore update, and previous commits were all present.

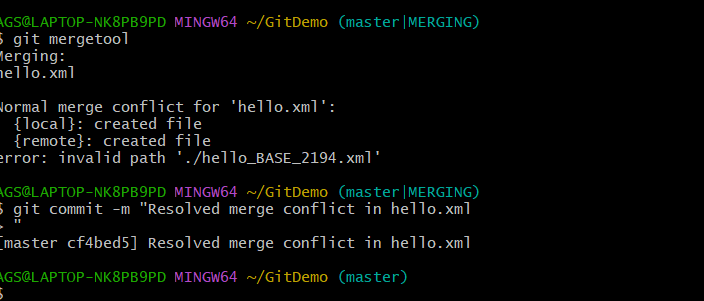
This exercise successfully demonstrated conflict resolution using a visual tool, the ability to interpret side-by-side differences, and proper repository hygiene by excluding unnecessary files and results can be seen in below outputs.

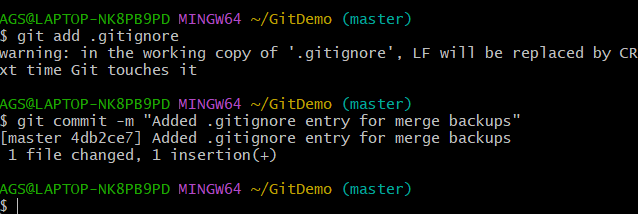
OUTPUT:

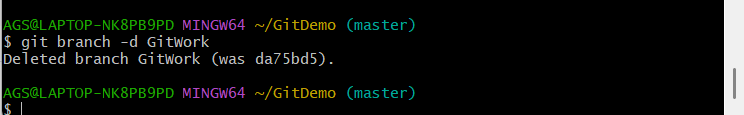


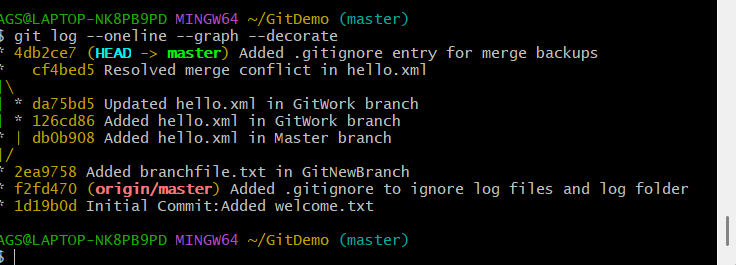


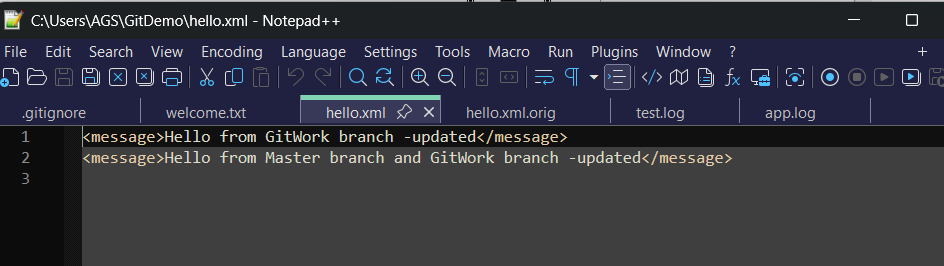


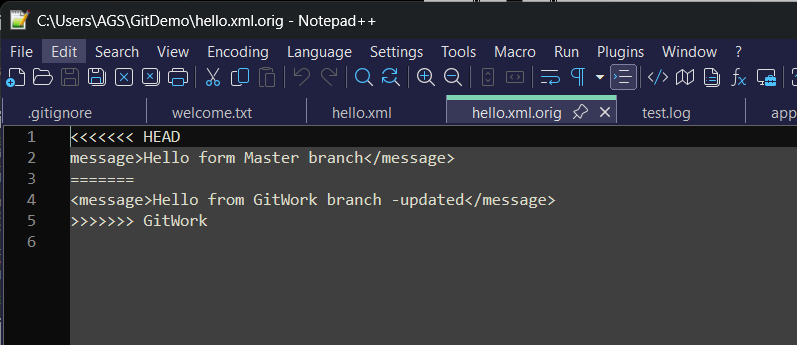
****

****

****

****

****

****

**EXERCISE-05** 5.GIT HOL

· Explain how to clean up and push back to remote Git

In this hands-on lab, you will learn how to:

· Execute steps involving clean up and push back to remote Git.

Prerequisites

The following are the pre-requisites to complete this hands-on lab:

· Hands-on ID: “Git-T03-HOL\_002”

Notes\*:

Please follow the below steps for creating a free account in GitHub. Do not use cognizant credentials to login to GitHub.

Estimated time to complete this lab: 10 minutes.

Please follow the instructions to complete the hands-on. Each instruction expects a command for the Git Bash.

1. Verify if master is in clean state.

2. List out all the available branches.

3. Pull the remote git repository to the master

4. Push the changes, which are pending from “Git-T03-HOL\_002” to the remote repository.

5. Observe if the changes are reflected in the remote repository.

IMPLEMENTATION:

In this hands-on lab, I performed several Git operations to demonstrate how to clean up the local repository and push changes to the remote repository.

**Verifying Clean State of master Branch**  
I began by checking the status of the local master branch using git status. The output confirmed that the working directory was clean with no uncommitted changes. Additionally, the branch was ahead of the remote origin/master by six commits, indicating pending commits to push.

**Listing Available Branches**  
Using git branch -a, I listed all local and remote branches. The output showed only the local master branch and the remote tracking branch origin/master. The branch Git-T03-HOL\_002 specified in the exercise was not present locally or remotely.

**Pulling Latest Changes from Remote**  
To ensure the local master branch was up-to-date, I ran git pull origin master. The response confirmed that the branch was already synchronized with the remote repository.

**Pushing Local Commits to Remote**  
I then pushed the six local commits on master to the remote repository with git push origin master. This successfully updated the remote master branch, confirming the push operation.

**Handling the Git-T03-HOL\_002 Branch**  
Since there were no pending changes or copies of this branch available, I did not create or push from it. Instead, the push operation was demonstrated using the master branch.  
The exercise required demonstrating clean-up and pushing changes back to remote Git. All steps were completed successfully on the master branch, including verifying branch status, pulling updates, and pushing commits and are shown in below outputs.

OUTPUT:

