**SUPERSET ID : 6364309 (Aryabrat Mishra)**

**Complete Git Hands-On Lab Solutions**

**LAB 1 : GitHub Setup and Configuration**

**Step 1: Setup your machine with Git Configuration**

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

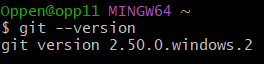
Login to GitLab and create a “GitDemo” project

**Step 1: Setting up Git Configuration**

**1.1 Verifying Git Installation**

git --version

*Output:*



**1.2 Configuring User Information**

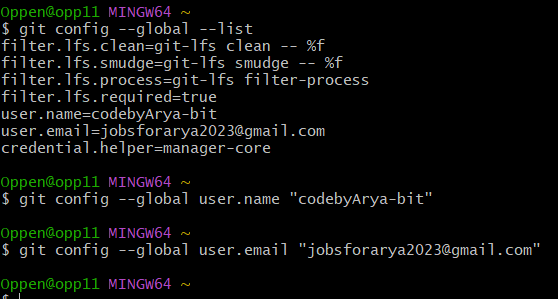
git config --global user.name "codebyArya-bit"

git config --global user.email "jobsforarya2023@gmail.com”

**1.3 Verifying Configuration**

git config --global --list

*Output:*



**Step 2: Configuring Notepad++ as Default Editor**

**2.1 Testing Notepad++ Command**

notepad++

*Output:*



**2.2 Creating Alias for Notepad++**

alias npp='notepad++.exe'

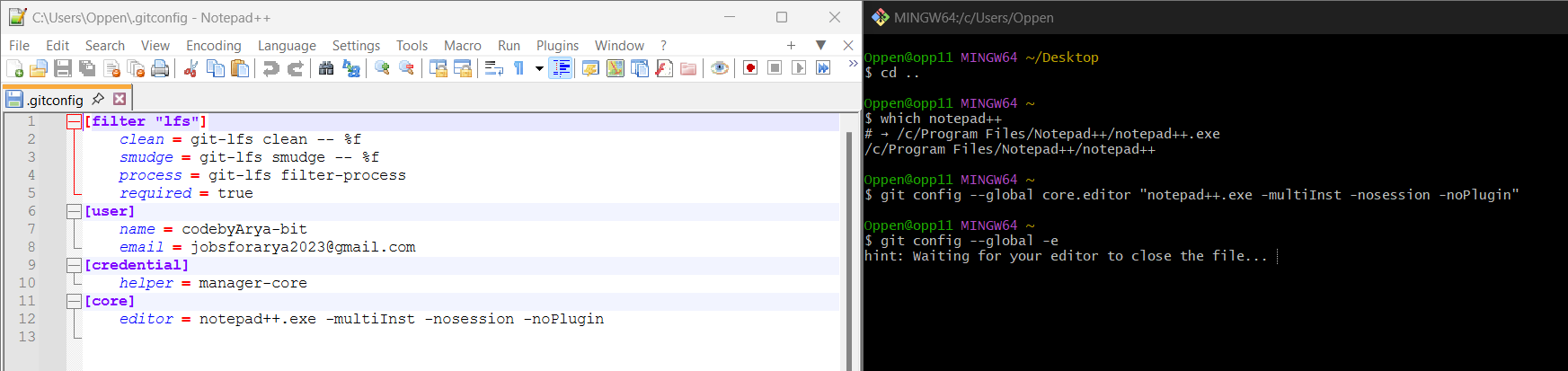
**2.3 Setting Notepad++ as Git Editor**

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

**2.4 Testing Editor Configuration**

git config --global -e

*Result:*



**Step 3: Creating and Managing a Repository**

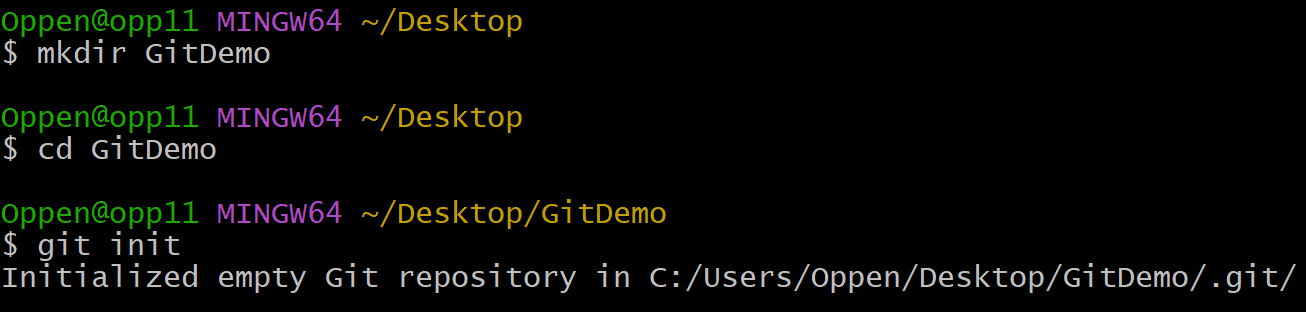
**3.1 Initializing Repository**

mkdir GitDemo

cd GitDemo

git init

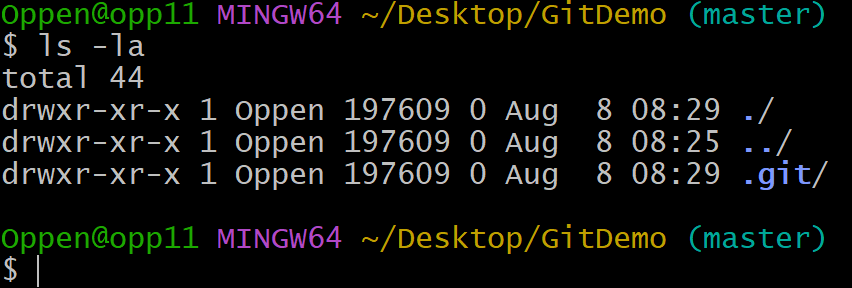
*Output:*



**3.2 Verifing Repository Structure**

ls -la

*Output:*



**3.3 Creating and Adding Content to File**

echo "Welcome to Git Demo" > welcome.txt

*Output:*

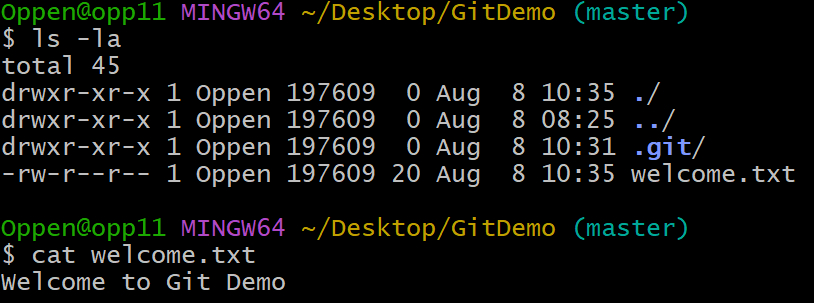


**3.4 Verifying File Creation**

ls -la

cat welcome.txt

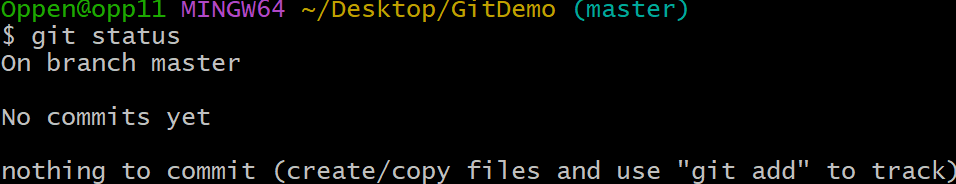
*Output:*



**3.5 Checking Git Status**

git status

*Output:*



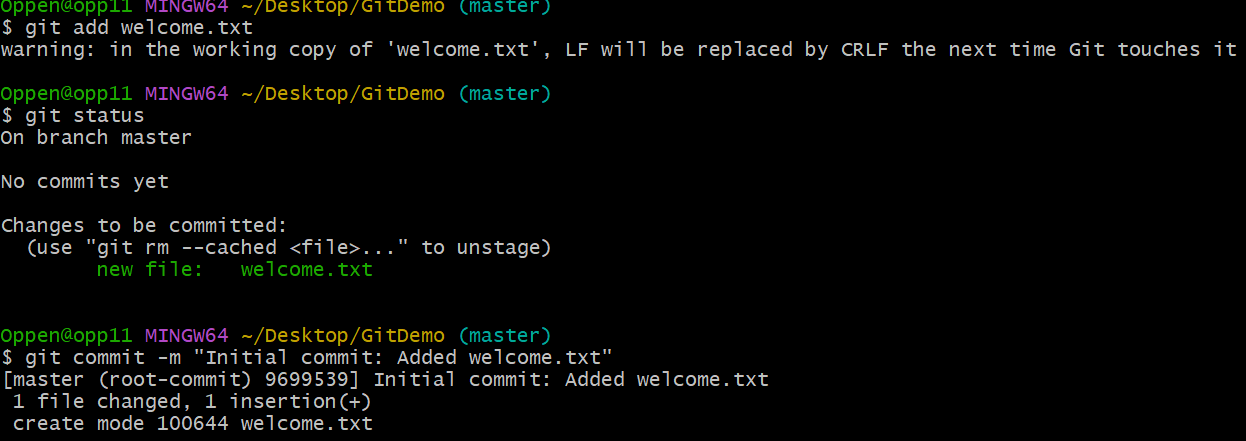
**3.6 Adding File to Staging Area**

git add welcome.txt

**3.7 Commiting Changes**

git commit -m "Initial commit: Added welcome.txt"

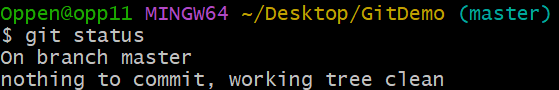
*Output:*



**3.8 Verifying Status After Commit**

git status

*Output:*



**LAB 2 : Git Ignore Implementation**

## **Objectives**

.gitignore is a plain-text file that tells Git which **unwanted** files or paths to ignore—so they don’t clutter your commits or repository history. Here’s how it works and how to use it:

## 1. Create a .gitignore file

At the root of your repo, add a file named exactly .gitignore:

cd your-repo/

touch .gitignore

## 2. Basic syntax & rules

In .gitignore, each line is a pattern:

**Blank lines** or lines beginning with # are ignored (comments).

**Literal filenames**:

secret.txt # ignore a single file

**Directory names** (trailing slash):

logs/ # ignore the entire logs/ folder

**Wildcards**:

\*.log # ignore all “.log” files

build/\*\*/\*.o # ignore all “.o” files under build/

**Negation** (unstage a sub-path inside an ignored folder):

node\_modules/ # ignore everything in node\_modules/

!node\_modules/keep-me.js

## 3. Typical use-cases

# OS artifacts

.DS\_Store

Thumbs.db

# Editor / IDE settings

.vscode/

\*.suo

\*.swp

# Build outputs

/dist/

/build/

/\*.exe

\*.class

# Secrets & credentials

.env

## 4. Applying changes

**New files** matching your patterns will now be **untracked**.

To stop tracking a file that was already committed:

git rm --cached path/to/file

git commit -m "Stop tracking file and add to .gitignore"

## 5. Check what Git will ignore

git status

git check-ignore -v path/to/suspected.file

## 6. Global ignores (for your machine)

If you have OS or editor files you never want to track across all repos:

# 1) Create a global ignore file

touch ~/.gitignore\_global

# 2) List patterns there, e.g. \*.log, .DS\_Store

# 3) Tell Git to use it

git config --global core.excludesfile ~/.gitignore\_global

### Best Practices

Commit your project‐specific .gitignore to version control so everyone on the team shares the same ignore rules.

Keep secrets out of Git—use environment variables or a secrets manager instead of committing credentials.

With these rules in place, Git will automatically skip unwanted files, keeping your repository clean and focused on source code.

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.

**2.1 Creating Files to Ignore**

echo "Debug information" > debug.log

mkdir logs

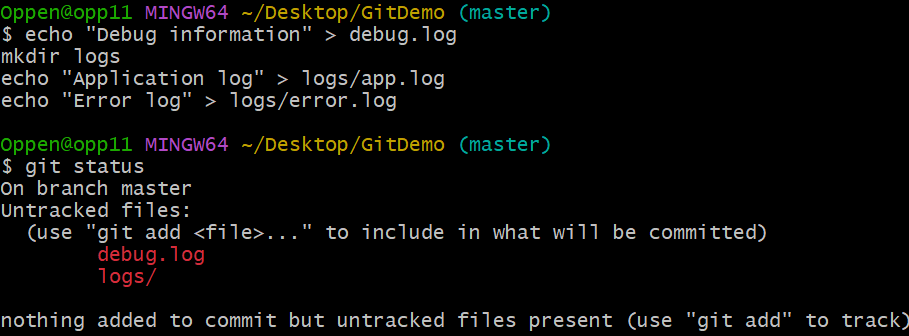
echo "Application log" > logs/app.log

echo "Error log" > logs/error.log

**2.2 Checking Status Before .gitignore**

git status

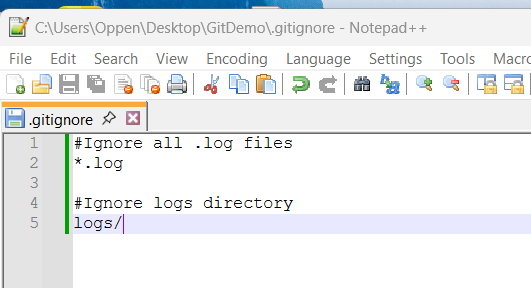
*Output:*



**2.3 Creating .gitignore File**

notepad++.exe .gitignore

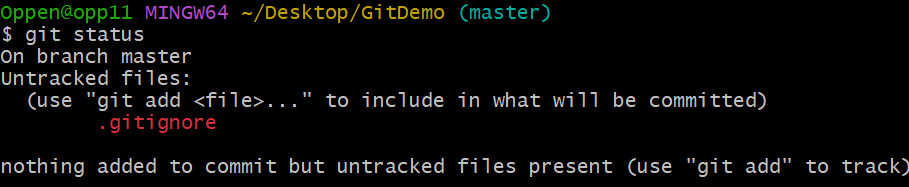
Adding/Editing:



**2.4 Verifying .gitignore Effect**

git status

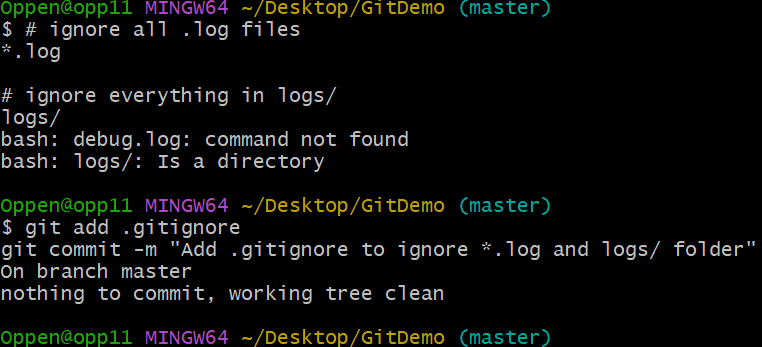
*Output:*



**2.5 Commiting .gitignore**

git commit -m “Added .gitignore to exclude log files and folders”

*Output:*



**Lab 3: Branching and Merging**

## **Objectives**

Here’s an overview of how branching & merging works in Git, plus the GitLab-specific steps for creating a new branch and then opening a merge request.

## 1. Branching & Merging in Git

### What is a branch?

A **branch** is simply a movable pointer to a commit.

By default every repo has a main (or master) branch.

Creating a new branch lets you work in isolation without touching main.

### Common branch commands

# List all local branches

git branch

# Create & switch to a new branch “feature-xyz”

git checkout -b feature-xyz

# (or with Git ≥2.23)

git switch -c feature-xyz

# Push your new branch up to the remote

git push -u origin feature-xyz

### What is a merge?

A **merge** takes the commits from one branch and integrates them into another (usually main).

Git will fast-forward if no divergent commits exist, otherwise it performs a three-way merge.

# From main, merge in your feature branch

git checkout main

git pull # ensure main is up to date

git merge feature-xyz # integrate changes

# If you see conflict markers (<<<<<<<), fix them in your editor, then:

git add <resolved-files>

git commit # completes the merge

git push

## 2. Creating a Branch in GitLab

You have two common ways to create a branch in your GitLab project:

### A) Via the GitLab Web UI

**Go to** your project on GitLab.

Click **Repository → Branches** in the left sidebar.

Click **New branch**.

Enter your branch name (e.g. feature-xyz) and select the base branch (usually main).

Click **Create branch**.

Your new branch now exists on GitLab—anyone can check it out or create a merge request from it.

### B) Via Git CLI (push a local branch)

git checkout -b feature-xyz # create & switch locally

git push -u origin feature-xyz # publish to GitLab

Once pushed, GitLab will show your branch under **Repository → Branches**, and often suggest you open a merge request right away.

## 3. Creating a Merge Request in GitLab

A **Merge Request** (MR) is how you ask to merge your branch into another branch (e.g. main):

**Go to** your project in GitLab.

Click **Merge Requests** in the sidebar, then **New merge request**.

**Select source branch** (your feature branch) **and target branch** (main).

Click **Compare branches and continue**.

Fill in:

**Title**: short summary (e.g. “Add user‐profile endpoint”).

**Description**: detailed notes, screenshots, links to issues.

**Assignee(s)** or **Reviewer(s)** if your workflow requires.

**Labels**, **Milestone**, **Pipeline trigger** settings as needed.

Click **Create merge request**.

Once created, reviewers can comment, request changes, and—when everything’s approved—click **Merge** to bring your work into main.

**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.

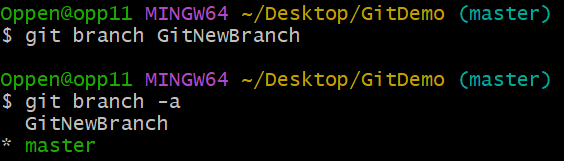
**Objective:** Practice branch creation, switching, and merging.

**3.1 Creating New Branch**

git branch GitNewBranch

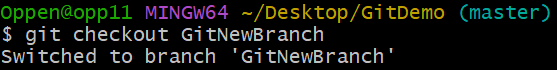
**3.2 Listing All Branches**

git branch -a



**3.3 Switching to New Branch**

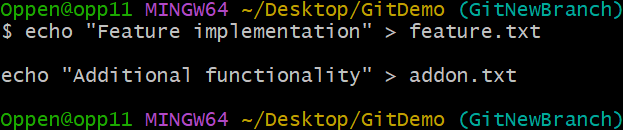
git checkout GitNewBranch



**3.4 Adding Files to Branch**

echo "Feature implementation" > feature.txt

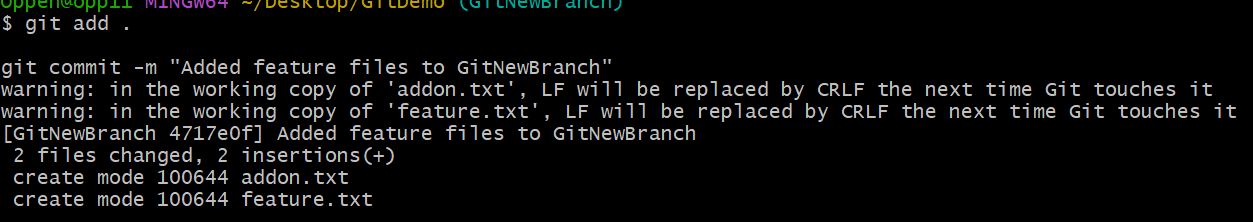
echo "Additional functionality" > addon.txt



**3.5 Adding and Commiting Changes**

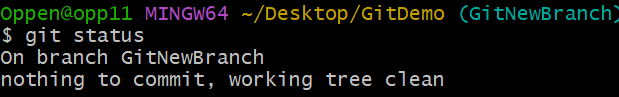
git add .

git commit -m "Added feature files to GitNewBranch"



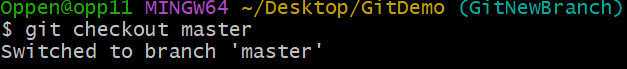
**3.6 Checking Status**

git status



**3.7 Switching Back to Master**

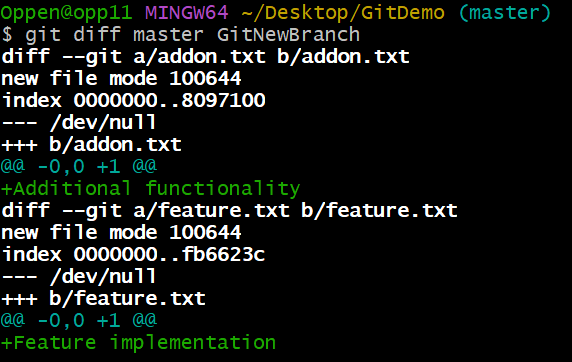
git checkout master



**3.8 Comparing Branches**

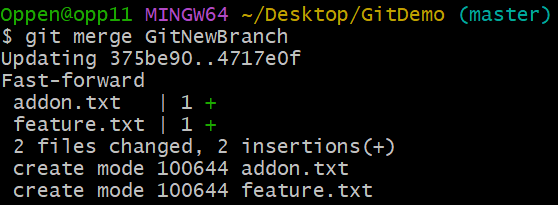
git diff master GitNewBranch

*Output:*



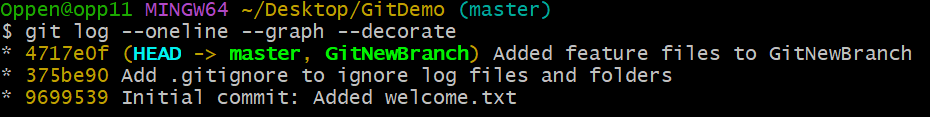
**3.9 Merging Branch to Master**

git merge GitNewBranch



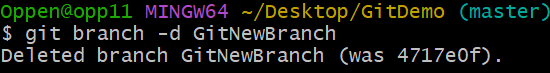
**3.10 Viewing Merge History**

git log --oneline --graph --decorate



**3.11 Deleting Merged Branch**

git branch -d GitNewBranch



**Lab 4: Conflict Resolution**

## **Objectives**

Explain how to resolve the conflict during merge.

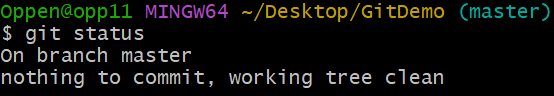
Please follow the instructions to complete the hands-on. Each instruction expect a command for the Git Bash.

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”**

**Objective:** Learn to handle merge conflicts.

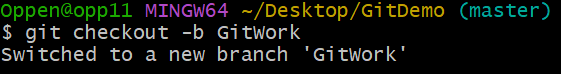
**4.1 Verifying Clean Master State**

git status



**4.2 Creating and Switching to Work Branch**

git checkout -b GitWork



**4.3 Creating hello.xml in Branch**

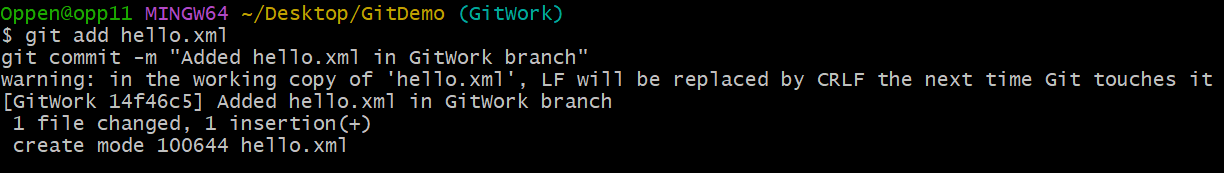
echo "<message>Hello from branch</message>" > hello.xml



**4.4 Commiting in Work Branch**

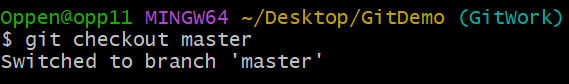
git add hello.xml

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



**4.5 Switching to Master**

git checkout master



**4.6 Create Conflicting hello.xml in Master**

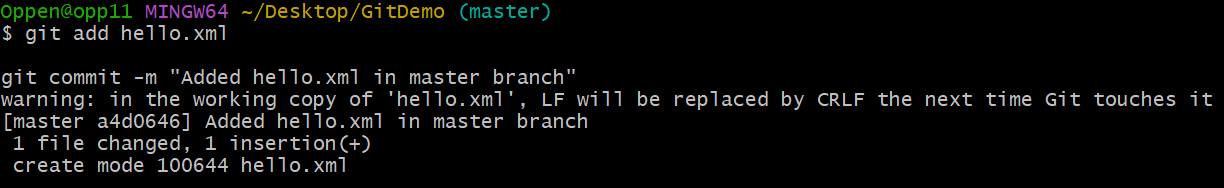
echo "<message>Hello from master</message>" > hello.xml



**4.7 Commiting in Master**

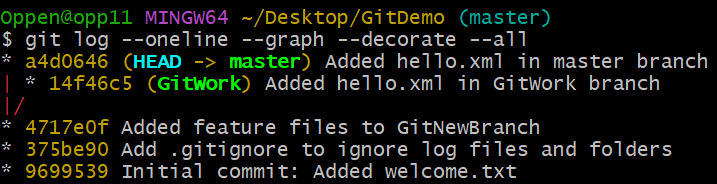
git add hello.xml

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



**4.8 Viewing History**

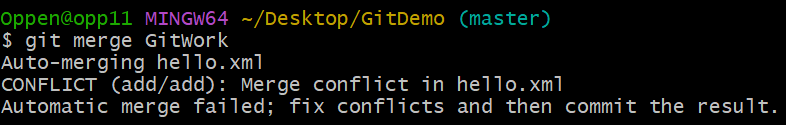
git log --oneline --graph --decorate --all



**4.9 Attempting to Merge**

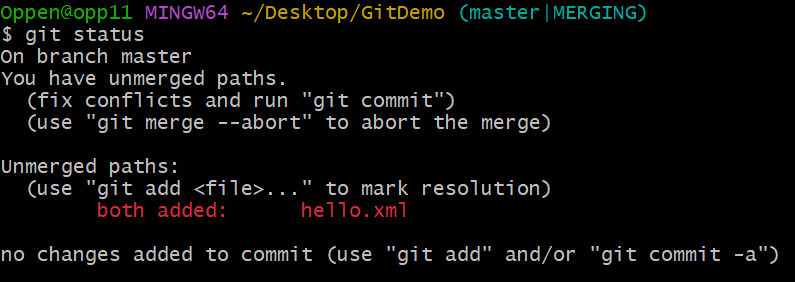
git merge GitWork

*Output:*



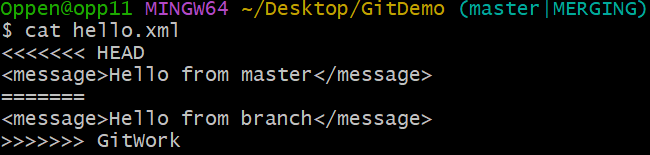
**4.10 Checking Conflict Status**

git status



**4.11 Viewing Conflict Markers**

cat hello.xml



**4.12 Resolving Conflict Manually**

Edit hello.xml:

notepad++.exe hello.xml

--message to edit : <message>Hello from both master and branch</message>



**4.13 Adding Resolved File and Commit**

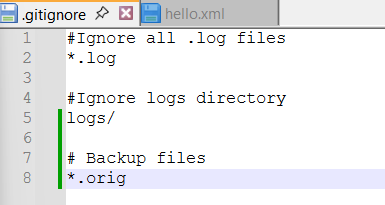
git add hello.xml

git commit -m "Resolved merge conflict in hello.xml"



**4.14 Updating .gitignore for Backup Files**





Adding to .gitignore:

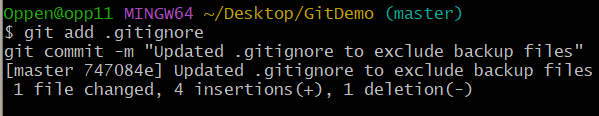
# Backup files

\*.orig

**4.15 Commiting .gitignore Update**

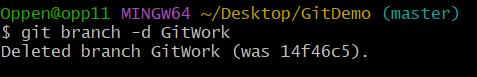
git add .gitignore

git commit -m "Updated .gitignore to exclude backup files"



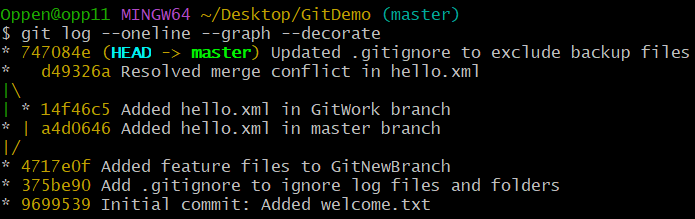
**4.16 Deleting Work Branch**

git branch -d GitWork



**4.17 Viewing Final History**

git log --oneline --graph --decorate



**Lab 5: Remote Repository Operations**

## **Objectives**

Here’s a typical “clean up and push” workflow once you’ve finished working locally and want your repo to be tidy and fully synced with the remote:

## 1. Review & stage your changes

**See what’s changed**

git status

**Stage only the files you intend**

git add path/to/file1 path/to/file2

Or stage everything new/modified:

git add .

## 2. Commit with a clear message

git commit -m "Short summary of what you added/fixed"

If you’ve got leftover uncommitted work you’re not ready to include, you can **stash** it:

git stash save "WIP: some notes"

## 3. Clean untracked files (optional)

To remove files Git hasn’t seen before (be **very** sure—you can’t undo this):

git clean -fd # -f = force, -d = include directories

To preview what would be deleted:

git clean -nfd

## 4. Update your main branch

If you’ve been working on a feature branch, switch back and bring main up to date:

git checkout main

git fetch origin

git pull --rebase origin main

## 5. Merge your feature (if you haven’t already)

git merge --no-ff feature-branch-name

Resolve any conflicts (look for <<<<<<< markers), then

git add <resolved-files>

git commit # completes the merge

## 6. Push your changes to the remote

git push origin main

If you’re still on a feature branch and want to publish it:

git push -u origin feature-branch-name

## 7. Clean up merged branches

Once your feature is merged and pushed, delete the branch **locally**:

git branch -d feature-branch-name

—and **remotely**:

git push origin --delete feature-branch-name

## 8. Prune stale remotes (optional)

To remove references to branches that no longer exist on origin:

git fetch --prune

### Putting it all together (example)

# 1. Stage & commit your work

git add .

git commit -m "Implement X feature"

# 2. Clean up any stray untracked files

git clean -nfd # preview

git clean -fd # actually delete

# 3. Switch to main and update it

git checkout main

git pull --rebase

# 4. Merge your feature branch

git merge --no-ff feature/X

# 5. Push main to remote

git push origin main

# 6. Delete your feature branch

git branch -d feature/X

git push origin --delete feature/X

# 7. Prune any stale remote‐tracking refs

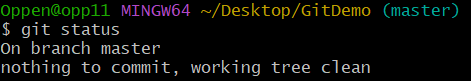
git fetch --prune

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.

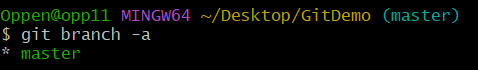
**5.1 Verifying Clean Master State**

git status



**5.2 Listing Branches**

git branch -a



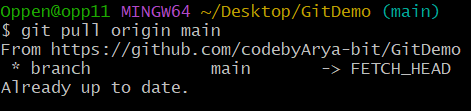
**5.3 Adding Remote Repository**

git remote add origin https://github.com/codebyArya-bit/GitDemo.git



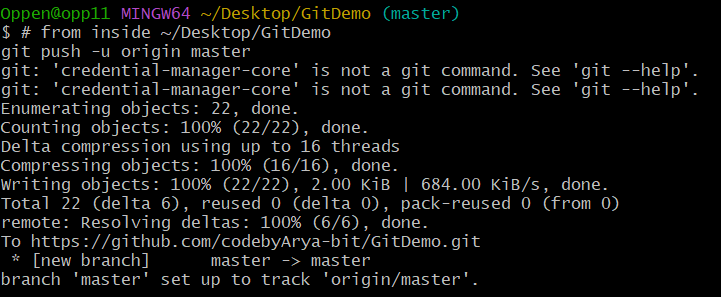
**5.4 Pulling from Remote**

git pull origin main



**5.5 Push Changes to Remote**

git push -u origin main



**5.6 Verifying**

**Online**

