Experiment 1: Introduction to Git and Local Repository Management

Aim:

To understand the basics of Git and to learn how to create and manage a local Git repository by performing fundamental operations such as initializing a repository, adding files, committing changes, and viewing commit history.

Theory:

Git is a **distributed version control system** used to track changes in source code and files during software development. It allows developers to maintain multiple versions of a project, collaborate efficiently, and revert to previous states if required.

In Git, a **repository** is a storage location that contains all project files along with their complete change history. A **local repository** exists on the user's machine and enables version control without requiring an internet connection.

The basic workflow of Git consists of three main stages:

1. **Working Directory**
   This is where files are created, modified, or deleted by the user.
2. **Staging Area (Index)**
   The staging area temporarily holds files that are marked to be included in the next commit.
3. **Repository (Commit History)**
   The repository permanently stores snapshots of the project in the form of commits.

Steps:

1.Create a Folder

Create a folder for the experiment: mkdir Git_Lab_Experiment1

Move into the directory: cd Git_Lab_Experiment1

2. Initialize a Git Repository

>>git init- Initializes a new Git repository.

This creates a hidden .git folder which tracks all version control information.

3.To List Files and Folder Inside the Current Directory: ls

To List All the Files, including hidden Files: **ls -a**

4. Configure Git (First-Time Setup)

>> git config --global user.name "Your Name"

>>git config --global user.email your_email@example.com

5. Create a File inside the Directory:

>>echo "This is my first Git experiment" > README.txt

6. Check file status: Displays the current state of the working directory and staging area.

>>git status

```
ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1
$ git init
Initialized empty Git repository in E:/Git_Lab_Experiment1/.git/

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ ls

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ ls -a
./  ../  .git/

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ git config --global user.name "Srusti20"

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ git config --global user.email srustishettym@gmail.com

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ echo "This is my first Git experiment" > README.txt

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ git status
On branch master

No commits yet

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

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

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$
```

7. Add Files to Staging Area:

>>git add README.txt or git add . (To add all the files)- Adds files to the staging area.

8.Check Status Again:

>>git status

```
ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ git add .

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ git status
On branch master

No commits yet

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


ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$
```

9. Commit the Files: Saves changes permanently to the repository with a descriptive message.

>>git commit -m "Initial commit"

```
ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ git commit -m "Initial commit"
[master (root-commit) e2aa0bf] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
```

10. Modify the File

>>echo "Git tracks file changes" >> README.txt

Check Status again: >>git status

```
ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ echo "Git tracks file changes" >> README.txt

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Add and commit changes:

>>git add README.txt

>>git commit -m "Updated README file"

```
ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ git add .

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ git commit -m "Updated README file"
[master 714b259] Updated README file
 1 file changed, 1 insertion(+)
```

11. View Commit History: Shows the commit history of the repository.

>>git log

```
ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1 (master)
$ git log
commit 714b259a1df56293d8b071acc6767f3aeadd0448 (HEAD -> master)
Author: Srusti20 <srustishettym@gmail.com>
Date:   Mon Jan 12 21:26:17 2026 +0530

    Updated README file

commit e2aa0bf00200517701b6aa35e0b715bcca6523c1
Author: Srusti20 <srustishettym@gmail.com>
Date:   Mon Jan 12 21:24:52 2026 +0530

    Initial commit
```

Experiment 2: Working with Git Branches

Aim:

To understand Git branching and learn how parallel development is managed using branches.

Objective: Understand branching.
Tasks: Create feature branch, commit changes, merge branches, resolve conflicts.
Outcome: Learn parallel development

Theory:

Git is a distributed version control system that allows developers to track changes in source code and collaborate efficiently. One of the most powerful features of Git is **branching**. A branch in Git represents an independent line of development, enabling developers to work on new features, bug fixes, or experiments without affecting the main (master/main) codebase.

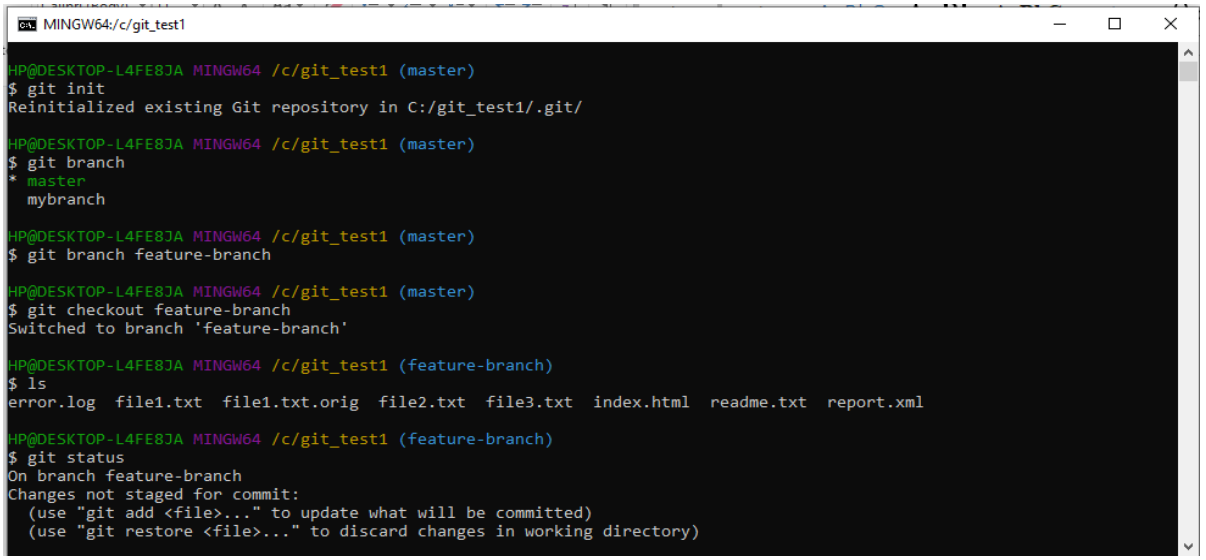
By default, every Git repository starts with a main branch (commonly called `main` or `master`). When a new branch is created, it points to the same commit as the current branch. Any changes made in the new branch are isolated from other branches until they are merged. This isolation helps in safe development and easy experimentation.

Git allows users to **create branches** to develop features independently, **switch between branches** to work on different tasks, and **merge branches** to integrate completed work back into the main branch. During merging, Git automatically combines changes, but if conflicting changes occur, the user must resolve these conflicts manually.

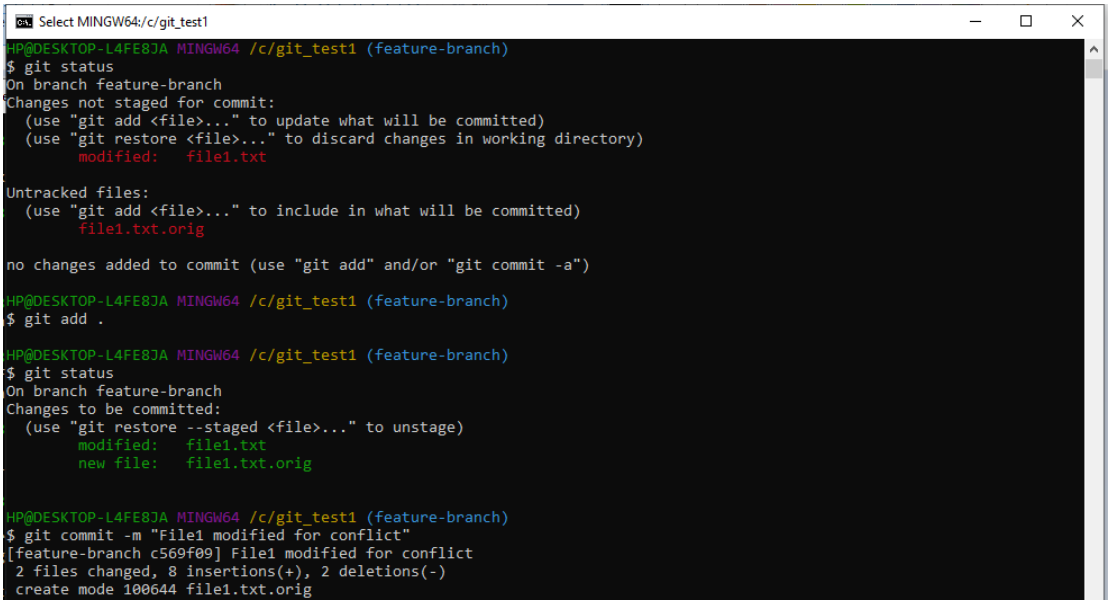
Branching is especially useful in team environments, where multiple developers work on the same project. Each developer can work on their own branch, reducing the risk of overwriting others' work. Once the changes are tested and reviewed, the branch can be merged into the main branch and deleted if no longer needed.

Steps / Procedure

1. **Initialize or Open a Git Repository**
    - Open the terminal.
    - Navigate to the project directory.
    - Initialize Git (if not already initialized):
    - `git init`
2. **Check the Current Branch**
    - Verify the active branch (usually `main` or `master`):
    - `git branch`
3. **Create a Feature Branch**
    - Create a new branch named `feature-branch`:
    - `git branch feature-branch`
4. **Switch to the Feature Branch**
    - Move to the newly created branch:
    - `git checkout feature-branch`
5. **Make Changes in the Feature Branch**
    - Edit or add files to implement a new feature.
    - Check the status:
    - `git status`



6. **Commit Changes in the Feature Branch**
    - Stage the changes:
    - `git add .`
    - Commit the changes:
    - `git commit -m "Added new feature in feature branch"`



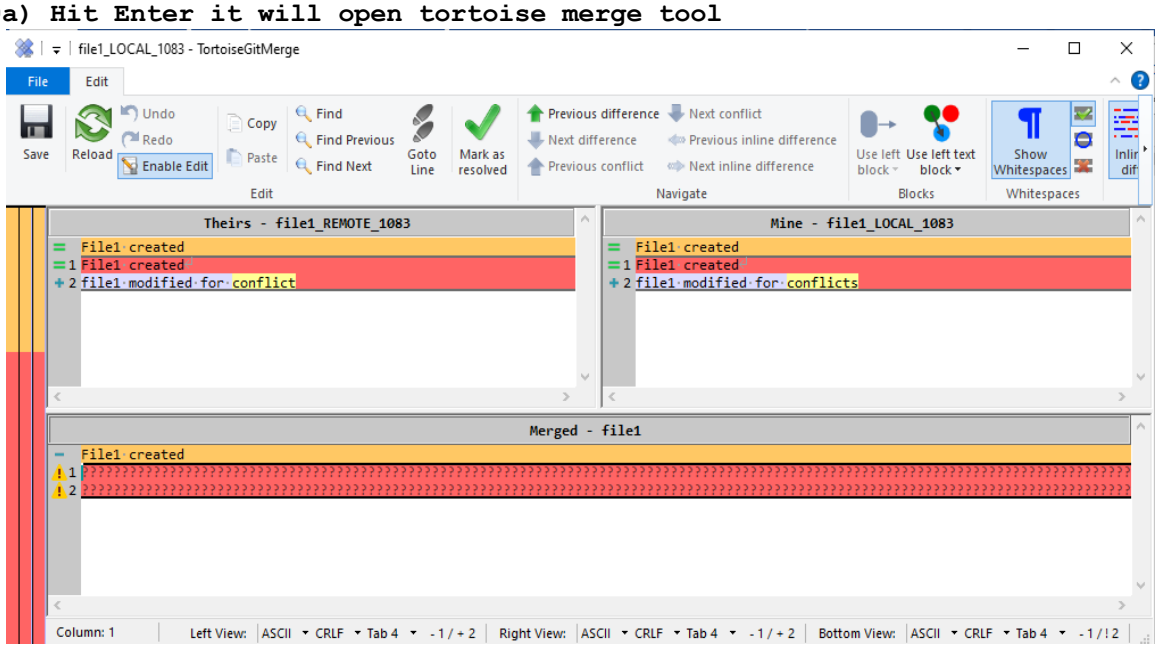
7. **Switch Back to the Main Branch**
    - Return to the main branch:
    - `git checkout main`
8. **Make Changes in the Main Branch (for Conflict Practice)**
    - Modify the same file that was edited in the feature branch.
    - Commit the changes:
    - `git add .`
    - `git commit -m "Updated file in main branch"`
9. **Merge Feature Branch into Main Branch**
    - Merge the feature branch:
    - `git merge feature-branch`

```
HP@DESKTOP-L4FE8JA MINGW64 /c/Git_Experiment2 (feature-branch)
$ git checkout master
Switched to branch 'master'

HP@DESKTOP-L4FE8JA MINGW64 /c/Git_Experiment2 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1

no changes added to commit (use "git add" and/or "git commit -a")

HP@DESKTOP-L4FE8JA MINGW64 /c/Git_Experiment2 (master)
$ git add .

HP@DESKTOP-L4FE8JA MINGW64 /c/Git_Experiment2 (master)
$ git commit -m "File1 modified for conflicts"
[master 5a5ecbd] File1 modified for conflicts
 1 file changed, 2 insertions(+), 1 deletion(-)

HP@DESKTOP-L4FE8JA MINGW64 /c/Git_Experiment2 (master)
$ git status
On branch master
nothing to commit, working tree clean

HP@DESKTOP-L4FE8JA MINGW64 /c/Git_Experiment2 (master)
$ git merge feature-branch
Auto-merging file1
CONFLICT (content): Merge conflict in file1
Automatic merge failed; fix conflicts and then commit the result.
```

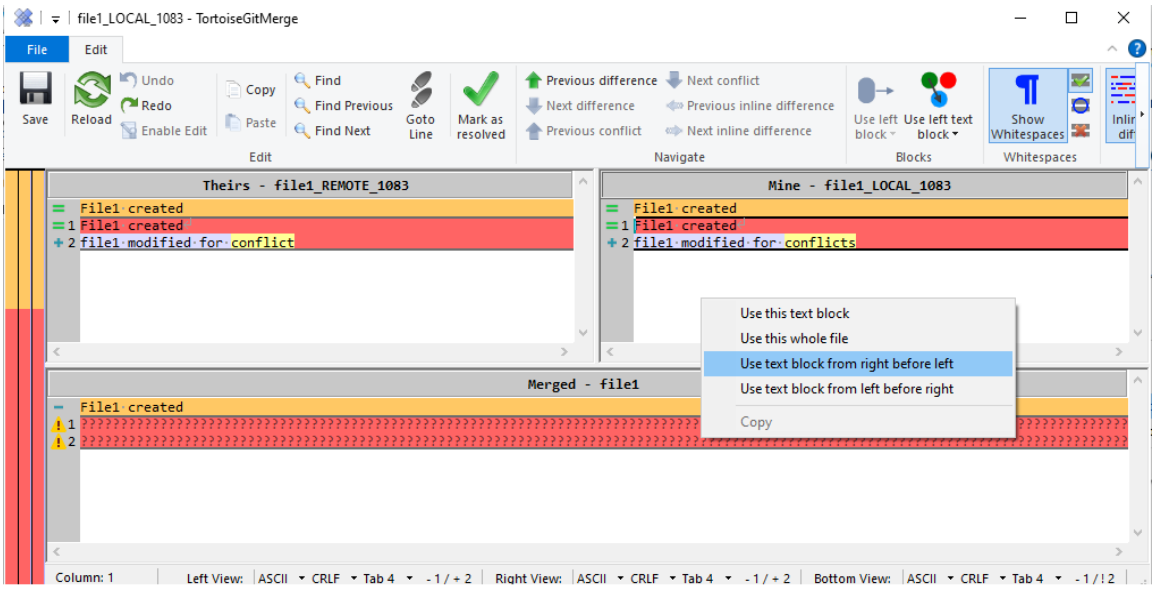
10. **Resolve Merge Conflicts (If Any)**
    - Open conflicted files and manually resolve differences.
    - After resolving, stage the file:
    - `git add <filename>`
    - Complete the merge:
    - `git commit`

```
HP@DESKTOP-L4FE8JA MINGW64 /c/Git_Experiment2 (master|MERGING)
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc codecompare smerge emerge vimdiff nvimdiff
Merging:
file1

Normal merge conflict for 'file1':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (tortoisemerge):
```

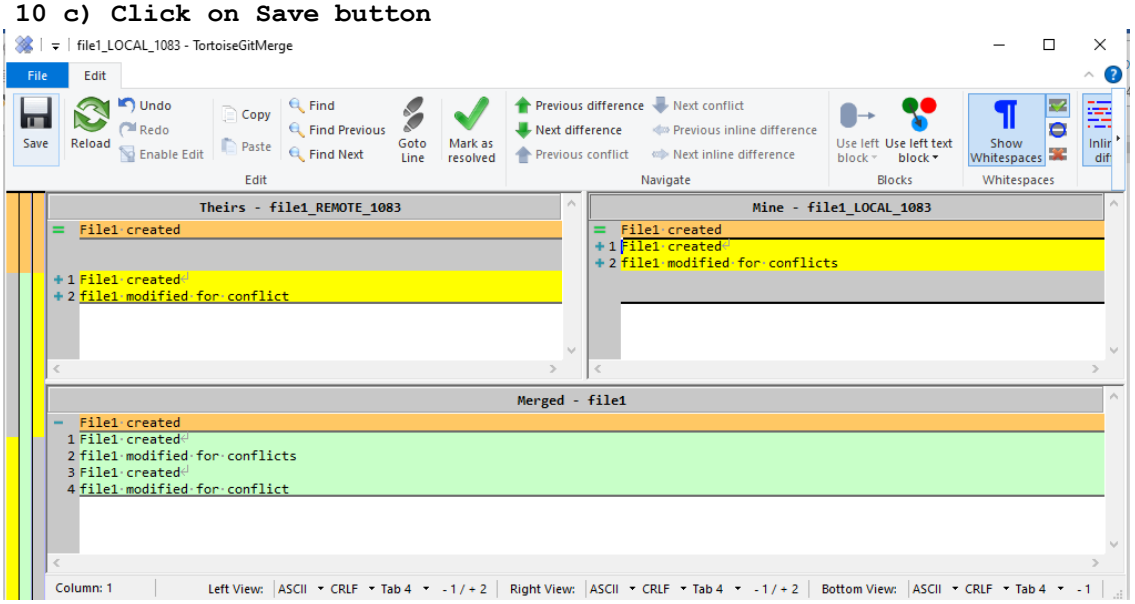
10a) Hit Enter it will open tortoise merge tool



10 b) Right click on Mine- File1_local_1083 and select 3rd option (use block from right before left)



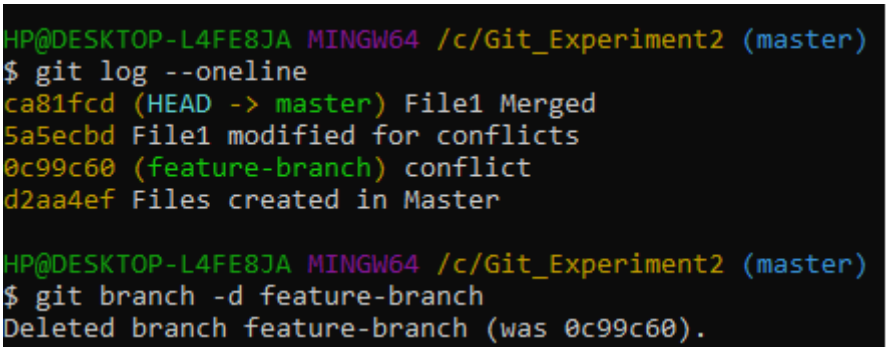
10 c) Click on Save button



10 d)

```
HP@DESKTOP-L4FE8JA MINGW64 /c/Git_Experiment2 (master|MERGING)
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        modified:   file1

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.orig


HP@DESKTOP-L4FE8JA MINGW64 /c/Git_Experiment2 (master|MERGING)
$ git add .

HP@DESKTOP-L4FE8JA MINGW64 /c/Git_Experiment2 (master|MERGING)
$ git commit -m "File1 Merged"
[master ca81fcd] File1 Merged
```

11. **Verify the Merge**
    - Check commit history:
    - `git log --oneline`
12. **Delete the Feature Branch (Optional)**
    - Remove the feature branch after successful merge:
    - `git branch -d feature-branch`



Result / Outcome

Git branches were successfully created, merged, and conflicts were resolved. This experiment demonstrates how parallel development is managed using Git branches.

Experiment 3: GitHub Repository Creation and Push

Aim:

To understand remote repository management using GitHub by creating a GitHub repository and performing operations such as pushing local code, pulling updates from the remote repository, and cloning a repository.

Theory:

GitHub is a remote repository hosting platform that works with Git to enable distributed version control. While Git manages versions locally, GitHub stores repositories on a remote server, allowing collaboration, backup, and synchronization of code.

A remote repository is an online repository that can be accessed over the internet. Developers push their local commits to the remote repository and pull updates made remotely to keep their local repository synchronized.

The major operations involved in remote repository management are:

- Push (git push)
  Transfers commits from the local repository to the remote GitHub repository.
- Pull (git pull)
  Fetches changes from the remote repository and merges them into the local repository.
- Clone (git clone)
  Creates a local copy of an existing remote repository, including its entire history.
- Remote (origin)
  The default name assigned to the remote GitHub repository URL.
- Branch (main / master)
  Represents a line of development. The main branch contains the stable version of the project.

Step 1 : Setup Git Global Configuration

```
>>> git config --global user.name "Your Name"
```

```
>>> git config --global user.email "your_email@example.com"
```

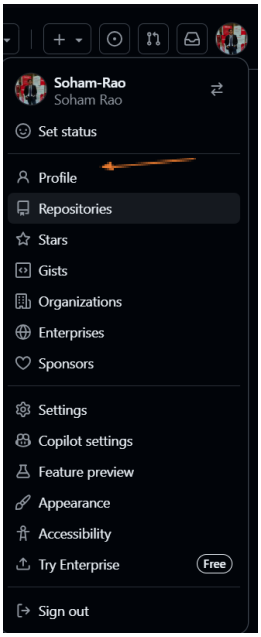
```
soham@Legionnaire MINGW64 /c/users/soham/OneDrive/Desktop/git-demo (master)
$ git config --global user.name "Your name"
```

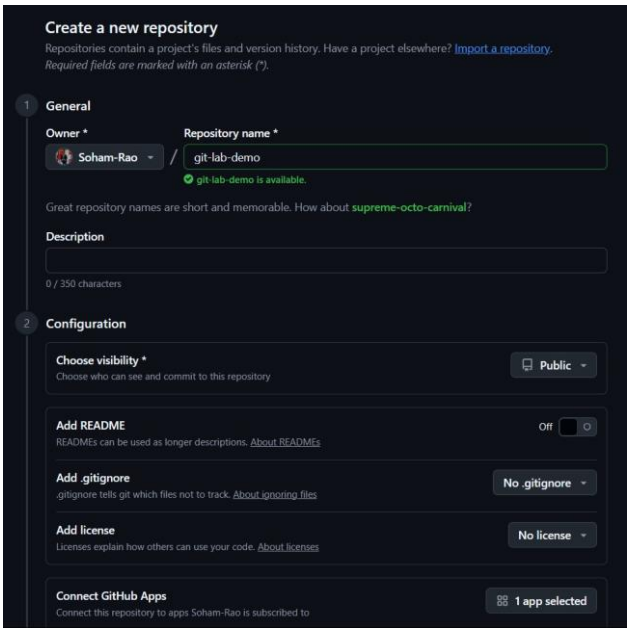
```
soham@Legionnaire MINGW64 /c/users/soham/OneDrive/Desktop/git-demo (master)
$ git config --global user.email "Your email"
```

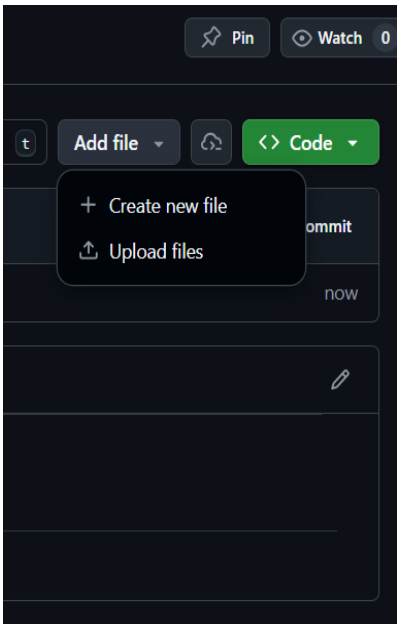
Step 2 : Sign in to GitHub

- Open https://github.com
- Sign in → Click **New Repository**
- Name it **git-lab-demo**
- Keep it *public*

- Click **Create Repository**







Step 3 : Clone the Repository

- On GitHub, **Copy HTTPS link**, then in Git Bash:

```
>>> cd Desktop

>>> git clone https://github.com/YourUsername/git-lab-demo.git

>>> cd git-lab-demo
```

```
soham@Legionnaire MINGW64 /c/users/soham/OneDrive/Desktop
$ git clone https://github.com/Soham-Rao/git-lab-demo.git
Cloning into 'git-lab-demo'...
warning: You appear to have cloned an empty repository.

soham@Legionnaire MINGW64 /c/users/soham/OneDrive/Desktop
$ cd git-lab-demo

soham@Legionnaire MINGW64 /c/users/soham/OneDrive/Desktop/git-lab-demo (master)
$
```

Step 4 : Add a Remote (if not cloned)

- If you started in a new folder instead of cloning:

```
>>> git init

>>> git remote add origin https://github.com/YourUsername/git-lab-demo.git

>>> git pull origin main
```

```
soham@Legionnaire MINGW64 /c/users/soham/OneDrive/Desktop/git-lab-demo (master)
$ git remote add origin https://github.com/Soham-Rao/git-lab-demo
```

```
soham@Legionnaire MINGW64 /c/users/soham/OneDrive/Desktop/git-lab-demo (master)
$ git pull origin master
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 857 bytes | 171.00 KiB/s, done.
From https://github.com/Soham-Rao/git-lab-demo
 * branch            master     -> FETCH_HEAD
 * [new branch]      master     -> origin/master
```

Step 5 : Add a Local File

- Create a simple Python file:

```
# main.py

print("This file is from local machine.")
```

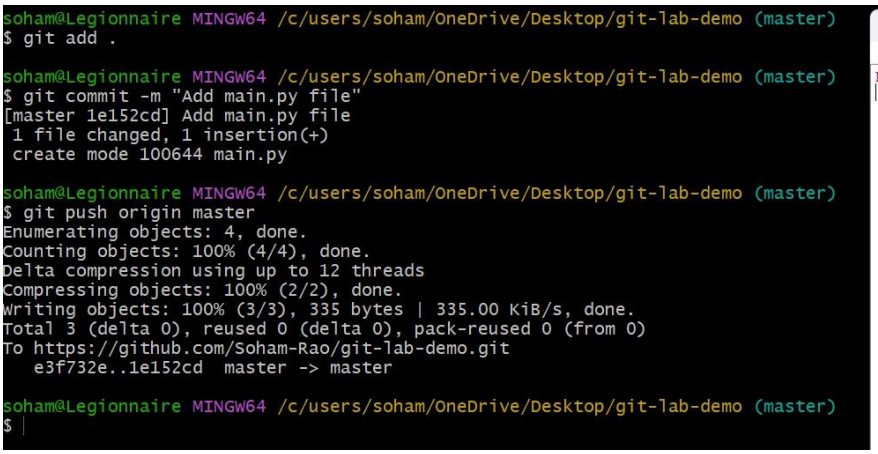
- Then run:

```
>>> git add main.py

>>> git commit -m "Add main.py file"

>>> git push origin main
```

- NOTE: if its your first time pushing, you need to run the following:

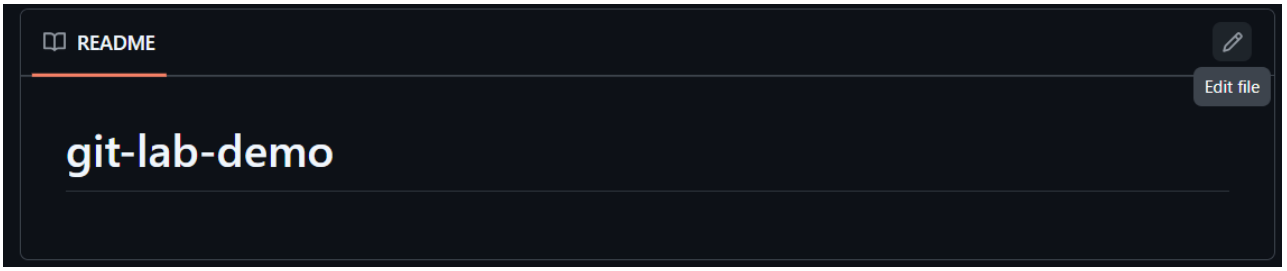
```
>>> git push -u origin main
```

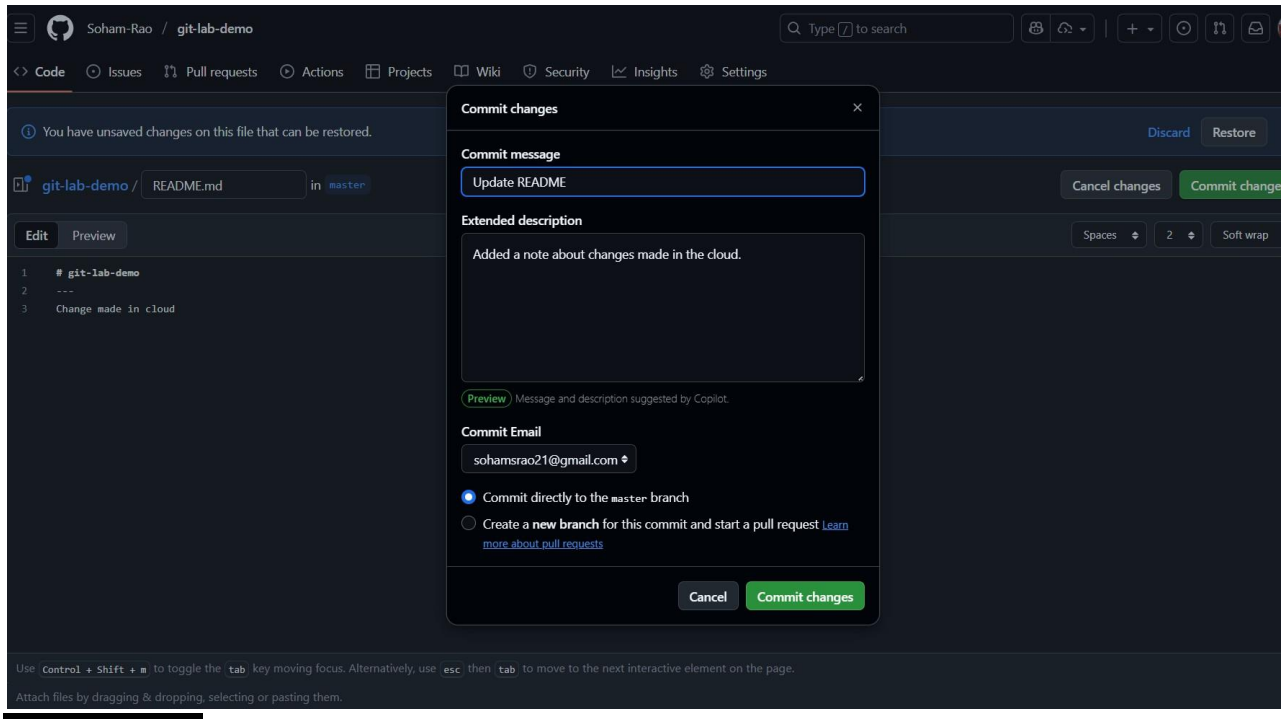
- Now the file will appear on GitHub.



Step 6 : Demonstrate Fetch and Pull

- Make a small change directly on GitHub (edit README.md online, e.g., add “Updated via GitHub”).
- Then on Git Bash:





>>> git fetch origin

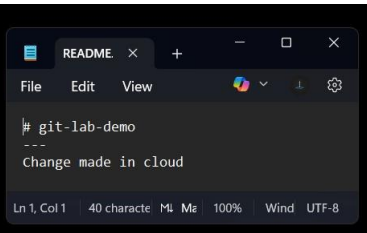
>>> git merge origin/main

(OR)

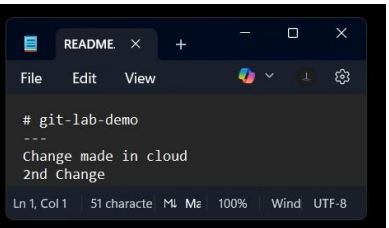
>>> git pull origin main

- git fetch download the latest updates from GitHub but make no changes
- git pull git fetch + update changes to local repository
- This downloads the updated version from GitHub.

```
soham@Legionnaire MINGW64 /c/users/soham/OneDrive/Desktop/git-lab-demo (master)
$ git fetch origin
From https://github.com/Soham-Rao/git-lab-demo
 * [new branch]      master     -> origin/master

soham@Legionnaire MINGW64 /c/users/soham/OneDrive/Desktop/git-lab-demo (master)
$ git merge origin/master
Updating 1e152cd..c68e29f
Fast-forward
 README.md | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)

soham@Legionnaire MINGW64 /c/users/soham/OneDrive/Desktop/git-lab-demo (master)
$
```




```
soham@Legionnaire MINGW64 /c/users/soham/OneDrive/Desktop/git-lab-demo (master)
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 991 bytes | 141.00 KiB/s, done.
From https://github.com/Soham-Rao/git-lab-demo
   c68e29f..659b241  master     -> origin/master
Updating c68e29f..659b241
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
```



Experiment 4: GitHub Collaboration Using Pull Requests

Aim:

To understand and implement team-based software development using GitHub by forking a repository, creating a feature branch, making changes, raising a pull request, reviewing the changes, and merging them into the main branch. This experiment helps in learning collaborative development practices followed in real-world software projects.

Theory:

GitHub Collaboration

GitHub is a distributed version control platform built on Git that allows multiple developers to work on the same project simultaneously. To avoid conflicts and maintain stability, developers do not work directly on the master branch. Instead, they use separate branches or forked repositories.

Forking a Repository

Forking creates a personal copy of an existing repository in a user's GitHub account. It allows developers to make changes independently without affecting the original repository. Forking is commonly used in team projects and open-source development.

Branches

A branch is an independent line of development. Feature branches such as feature-1 are used to develop new features or fix bugs while keeping the master branch stable.

Pull Request (PR)

A pull request is a request to merge changes from a feature branch into the master branch. It enables code review, discussion, and verification before merging the changes.

Code Review and Merging

Before merging, team members review the pull request to check code correctness, standards, and possible conflicts. After approval, the changes are merged into the master branch, preserving version history.

Importance of Pull Requests

Pull requests prevent direct changes to the master branch, support team collaboration, improve code quality, and follow industry-standard development practices.

Steps:

1.Create a New Repository on GitHub

1. Login to GitHub
2. Click **New Repository**
3. Enter repository name (example: Git_PullRequests)
4. Select **Public**
5. Check **Add README.md**
6. Click **Create Repository**

2. Clone Repository to Local System

>>git clone <repository-url>

>>cd Git_PullRequests

3. Create Files in Master Branch

Create a file:

touch file1.txt

Add content:

echo "This file is created in master branch" > file1.txt

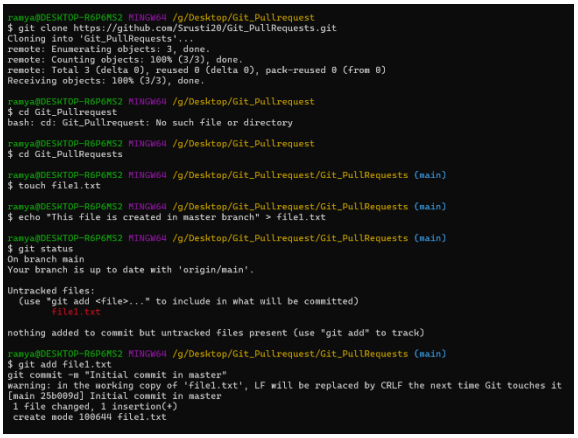
Check status:

git status

Add and Commit in Master

git add file1.txt

git commit -m "Initial commit in master"



4.Push Changes to Master/main

git push origin main

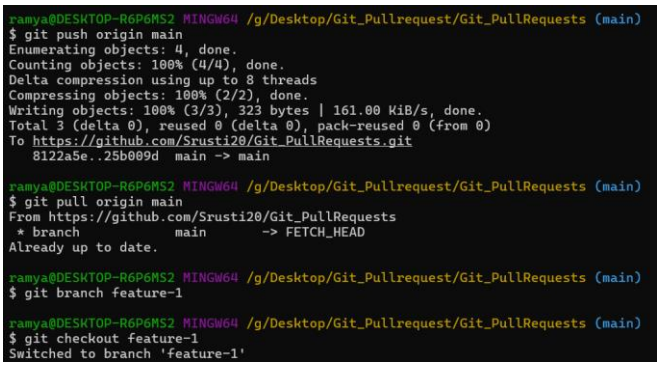
5. Pull Latest Changes (Safe Practice)

git pull origin main

6. Create Feature Branch

git branch feature-1

git checkout feature-1



Step 7: Modify Files in Feature Branch

Edit existing file:

```
echo "This change is from feature-1 branch" >> file1.txt
```

Create new file:

```
touch feature.txt
```

```
echo "Feature branch file" > feature.txt
```

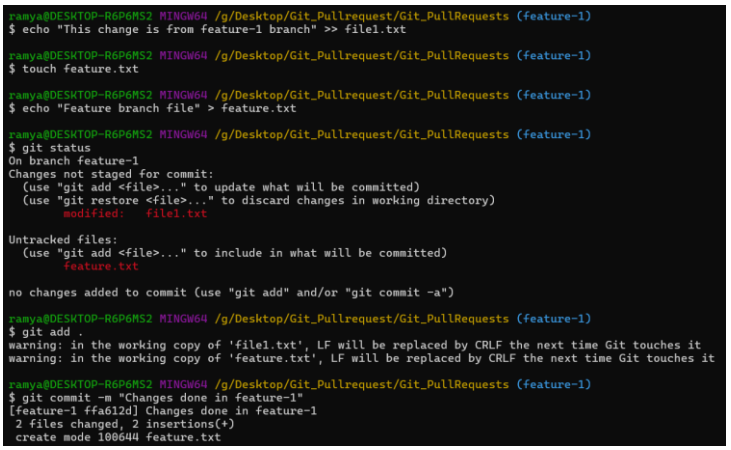
Check status:

```
git status
```

Add and Commit in Feature Branch

```
git add .
```

```
git commit -m "Changes done in feature-1"
```



Step 8: Push Feature Branch to GitHub

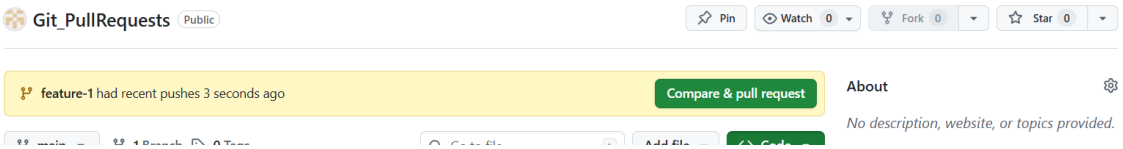
```
git push origin feature-1
```

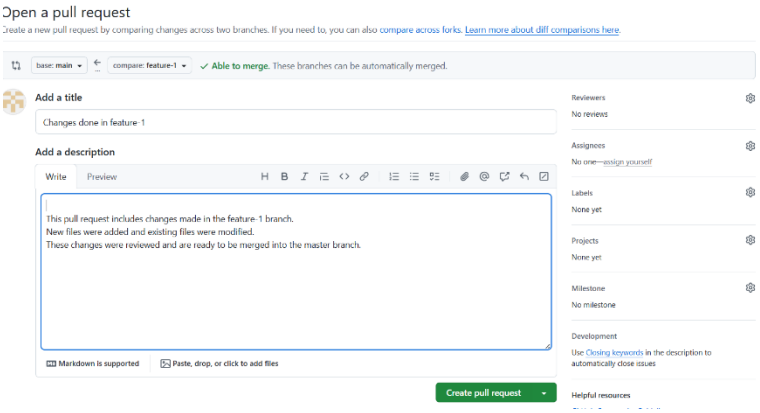
```
ramya@DESKTOP-R6P6MS2 MINGW64 /g/Desktop/Git_Pullrequest/Git_PullRequests (feature-1)
$ git push origin feature-1
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 411 bytes | 137.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
remote:
remote: Create a pull request for 'feature-1' on GitHub by visiting:
remote:      https://github.com/Srusti20/Git_PullRequests/pull/new/feature-
remote:
To https://github.com/Srusti20/Git_PullRequests.git
 * [new branch]      feature-1 -> feature-1
```

Step 9: Create Pull Request (GitHub UI)

1. Open GitHub repository
2. Click **Compare & Pull Request**
3. Base branch → master
4. Compare branch → feature-1
5. Add description
6. Click **Create Pull Request**

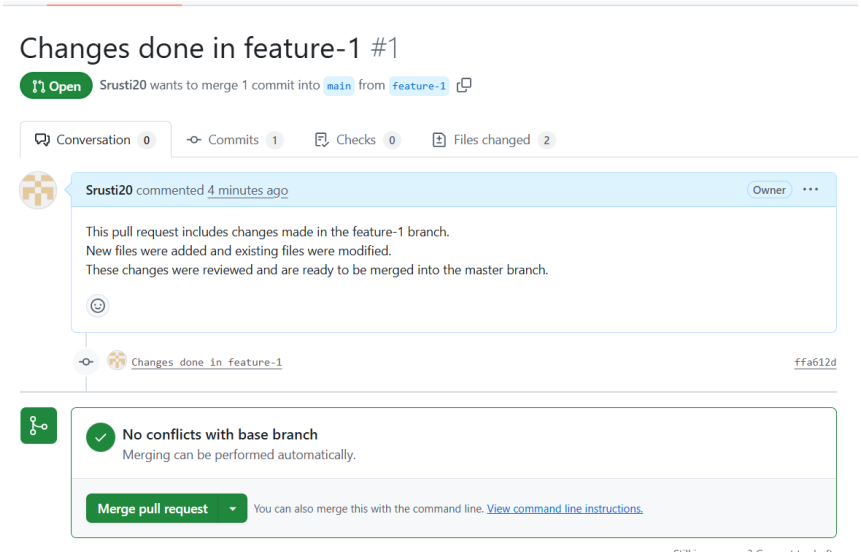
This step is required because it ensures that changes made in a separate branch (feature-1) are **reviewed and approved** before being merged into the main branch (master). It helps prevent errors, maintain code quality, and allows collaboration by letting others see, comment on, or suggest improvements to your changes. Without a pull request, changes could be merged directly, which might introduce bugs or conflicts in the main code.





Step 10: Review Pull Request

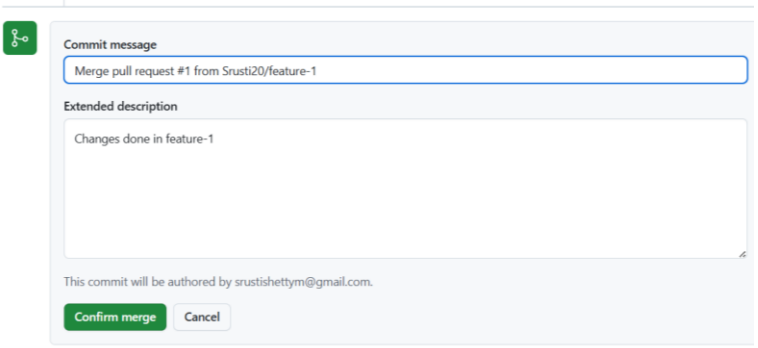
- Reviewer checks code
- Approves changes

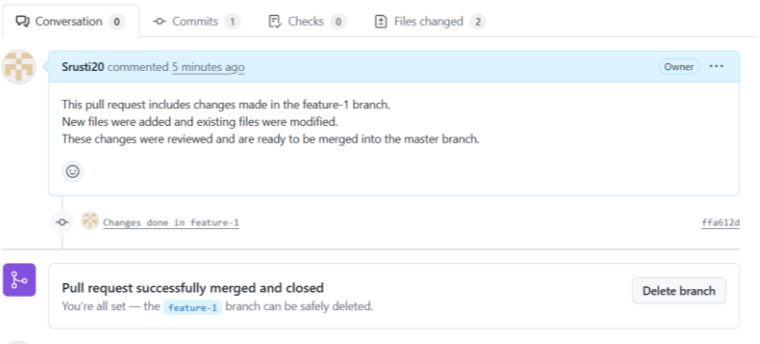


Step 11: Merge Pull Request

Click **Merge Pull Request**
Confirm merge

Feature-1 code is merged into master.





Merge Pull Request in GitHub means you are taking the changes from a feature branch (like feature-1) and combining them into the **base branch** (usually master or main).

Step 12: Update Local Master After Merge

git checkout master

git pull origin master

Step 13: Verify Commit History

git log --oneline

Step 14: Delete Feature Branch (Optional but Best Practice)

Local:

git branch -d feature-1

Remote:

git push origin --delete feature-1

```
ramya@DESKTOP-R6P6MS2 MINGW64 /g/Desktop/Git_Pullrequest/Git_PullRequests (feature-1)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

ramya@DESKTOP-R6P6MS2 MINGW64 /g/Desktop/Git_Pullrequest/Git_PullRequests (main)
$ git pull origin main
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (1/1), 897 bytes | 6.00 KiB/s, done.
From https://github.com/Srusti20/Git_PullRequests
 * branch            main       -> FETCH_HEAD
   25b009d..2b3bbd7  main       -> origin/main
Updating 25b009d..2b3bbd7
Fast-forward
 feature.txt | 1 +
 file1.txt   | 1 +
 2 files changed, 2 insertions(+)
 create mode 100644 feature.txt

ramya@DESKTOP-R6P6MS2 MINGW64 /g/Desktop/Git_Pullrequest/Git_PullRequests (main)
$ git log --oneline
2b3bbd7 (HEAD -> main, origin/main, origin/HEAD) Merge pull request #1 from
Srusti20/feature-1
ffa612d (origin/feature-1, feature-1) Changes done in feature-1
25b009d Initial commit in master
8122a5e Initial commit

ramya@DESKTOP-R6P6MS2 MINGW64 /g/Desktop/Git_Pullrequest/Git_PullRequests (main)
$ git branch -d feature-1
Deleted branch feature-1 (was ffa612d).

ramya@DESKTOP-R6P6MS2 MINGW64 /g/Desktop/Git_Pullrequest/Git_PullRequests (main)
$ git push origin --delete feature-1
To https://github.com/Srusti20/Git_PullRequests.git
 - [deleted]         feature-1
```

Experiment 5: Git Tagging and Release Creation

Aim:

To understand release management in Git by creating annotated tags, pushing tags to a remote repository, and creating releases on GitHub.

Theory:

Release management is the process of identifying, labeling, and distributing stable versions of a software project. Git provides a feature called **tagging** to mark specific points in a repository's commit history, usually to indicate version releases such as *v1.0*, *v2.0*, etc.

A **Git tag** is a reference that points to a specific commit and remains constant, unlike branches which move as new commits are added. Tags are mainly used to mark important milestones like software releases.

There are two main types of tags in Git:

- **Lightweight tags**: Simple references to a commit.
- **Annotated tags**: Full objects that store metadata such as tag name, author, date, and a descriptive message. Annotated tags are recommended for release management.

To share tags with others, they must be **pushed to the remote repository** using Git commands. Once tags are available on GitHub, they can be used to create **GitHub releases**, which provide a user-friendly way to publish software versions along with release notes and downloadable assets.

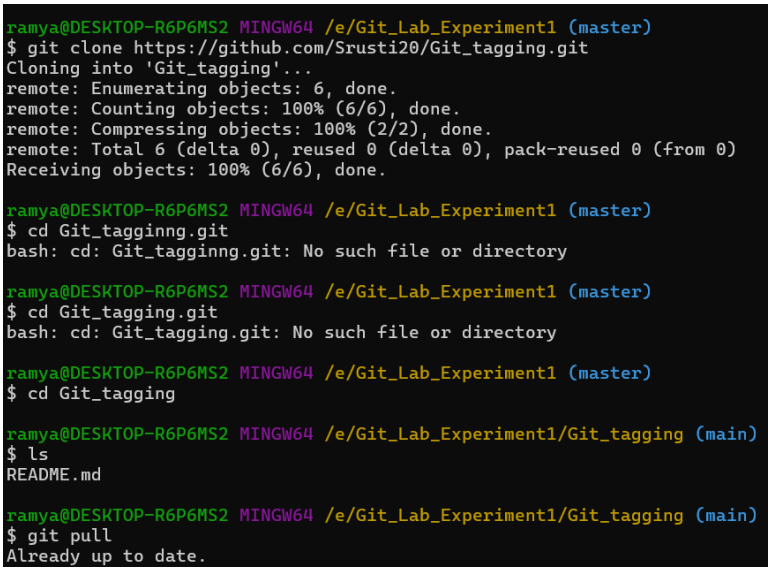
Using Git tagging and GitHub releases helps in:

- Identifying stable versions of software
- Maintaining clear version history
- Distributing software efficiently
- Improving project organization and collaboration

Steps:

1.Move to the Master/Main Branch (If Already in the Master Branch Ignore this Step)

2.git pull: To ensure that the local repository is fully up to date with the remote repository.



3.To Create a tag:

>>git tag v1.0

This Create the tag with the name v1.0 Locally

4.To display the tags Created and to Check whether the tag is created or Not

>>git tag

5.To add Message to the tag

>>git add -a v1.2 -m “v1.2 Created”

6.git tag : **To Check whether the v1.2 created or Not**

```
ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1/Git_tagging (main)
$ git tag v1.0

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1/Git_tagging (main)
$ git tag
v1.0

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1/Git_tagging (main)
$ git tag -a v1.2 -m "v1.2 Tag Created"

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1/Git_tagging (main)
$ git tag
v1.0
v1.2
```

7.To List all the recent activities happened in the v1.0

>>git show v1.0

```
ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1/Git_tagging (main)
$ git show v1.0
commit 66fead520be34e29316a4cf583d713f23d66b17d (HEAD -> main, tag: v1.2, ta
g: v1.0, origin/main, origin/HEAD)
Author: Srusti M <srustishettym@gmail.com>
Date:   Mon Jan 12 23:00:35 2026 +0530

    Update README.md

diff --git a/README.md b/README.md
index 602cb5c..e96042d 100644
--- a/README.md
+++ b/README.md
@@ -1 +1,2 @@
-# Git_tagging
\ No newline at end of file
+# Git_tagging
+To Mark the release Points'
```

10.To list all the Versions

>>git tag -l “v1.*”

```
ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1/Git_tagging (main)
$ git tag -l "v1.*"
v1.0
v1.2
```

To Check Created tags in the Remote Github repository

1.Pull these tags to the Remote Github Repository

>>git push origin v1.0

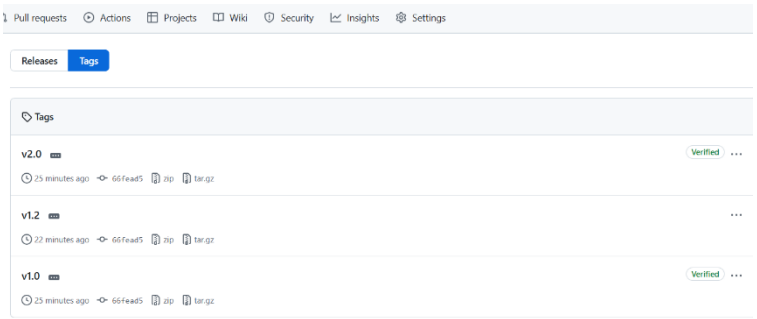
Now In the Github Repository the tags Number is Increased

2.git tag v2.0: **This Creates the one More tag**

3.To Push all the Created tags at a time

>>git push --tags

```
ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1/Git_tagging (main)
$ git push origin v1.0
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/Srusti20/Git_tagging.git
 * [new tag]         v1.0 -> v1.0

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1/Git_tagging (main)
$ git tag v2.0

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1/Git_tagging (main)
$ git push --tags
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 168 bytes | 18.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/Srusti20/Git_tagging.git
 * [new tag]         v1.2 -> v1.2
 * [new tag]         v2.0 -> v2.0
```



To delete the tags

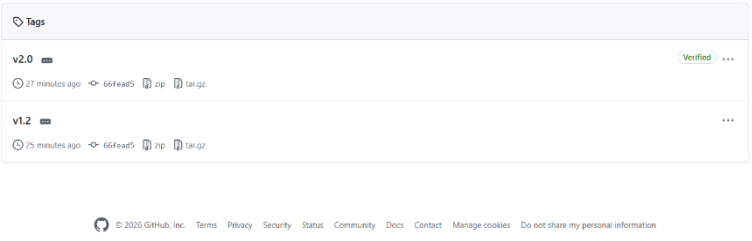
1.To delete the tags Locally:

>>git tag -d v1.0(But still the v1.0 tag exists in the Remote repository)

2.To delete the tags remotely

>>git push origin -d v1.0

```
ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1/Git_
$ git tag -d v1.0
Deleted tag 'v1.0' (was 66fead5)

ramya@DESKTOP-R6P6MS2 MINGW64 /e/Git_Lab_Experiment1/Git_
$ git push origin -d v1.0
To https://github.com/Srusti20/Git_tagging.git
 - [deleted]         v1.0
```



Experiment 6: Jenkins Installation and Configuration

Jenkins is an **open-source automation server** used to implement **Continuous Integration (CI)** and **Continuous Delivery (CD)** in software development. It automates the process of building, testing, and deploying applications whenever changes are made to the source code.

Steps

1. System Preparation
- Ensure the system has a stable internet connection.
- Install **Java (JDK 8 or JDK 11)** since Jenkins is Java-based.
- Verify Java installation using the command:
  java -version

2. Download and Install Jenkins
- Visit the official Jenkins website: https://www.jenkins.io
- Download the Jenkins package suitable for the operating system (Windows/Linux/macOS).
- Install Jenkins using the installer or package manager:
  - On Linux: sudo apt install jenkins (Debian/Ubuntu)
  - On Windows: Run the .msi installer

3. Start Jenkins Service
- Start the Jenkins service:
  - Linux: sudo systemctl start jenkins
  - Windows: Jenkins starts automatically after installation
- Check Jenkins service status:
  - sudo systemctl status jenkins

4. Access Jenkins Web Interface
- Open a web browser.
- Enter the Jenkins URL:
  http://localhost:8080
- Jenkins dashboard will appear.

5. Unlock Jenkins
- Locate the **initial admin password**:
  - Linux: /var/lib/jenkins/secrets/initialAdminPassword
  - Windows: Found in the Jenkins installation directory
- Copy the password and paste it into the web interface.

6. Install Suggested Plugins
- Select **"Install suggested plugins"** option.
- Jenkins automatically installs essential plugins required for CI/CD.

7. Create Admin User
- Enter admin user details (username, password, email).
- Save and continue.

8. Configure Jenkins Environment
- Set the Jenkins URL.
- Configure global tools:
  - JDK
  - Git
  - Maven (if required)
- Save the configuration.

9. Create a Sample Job
- Click **New Item** on Jenkins dashboard.
- Enter job name and select **Freestyle Project**.
- Configure source code management (Git repository).
- Add build steps (e.g., Execute shell or batch command).
- Save and build the job.

10. Verify Jenkins Setup

- Run the job manually using **Build Now**.
- Check **Console Output** for successful execution.
- Confirm Jenkins is working properly.

11. Stop Jenkins (Optional)

- Stop Jenkins service if required:
    - Linux: sudo systemctl stop jenkins
    - Windows: Stop Jenkins from Services panel

Result

Jenkins was successfully installed, configured, and verified by executing a sample build job.

Experiment 7: Jenkins Freestyle Job

Aim:

To create and execute a basic Jenkins freestyle job by configuring build settings and running a simple build command in order to understand automated build execution in Jenkins.

Theory:

Jenkins is an open-source automation server used to automate various stages of the software development process such as building, testing, and deployment. It supports Continuous Integration (CI) by automatically executing predefined jobs whenever triggered by the user or by automation mechanisms.

A Jenkins freestyle job is the simplest type of job provided by Jenkins. It allows users to configure build tasks through a graphical interface without using complex scripts or pipelines. Freestyle jobs are mainly used for learning, testing, and performing basic automation tasks.

In this experiment, Jenkins is used without integrating any source code management system. Instead, a basic build step is configured to execute a Windows batch command. This helps in understanding how Jenkins executes commands on the build machine and displays the output in the console.

This experiment demonstrates:

- Creation of a Jenkins freestyle job
- Execution of a simple automated build command
- Viewing build output through the Jenkins console

Steps:

1.Search localhost:8085

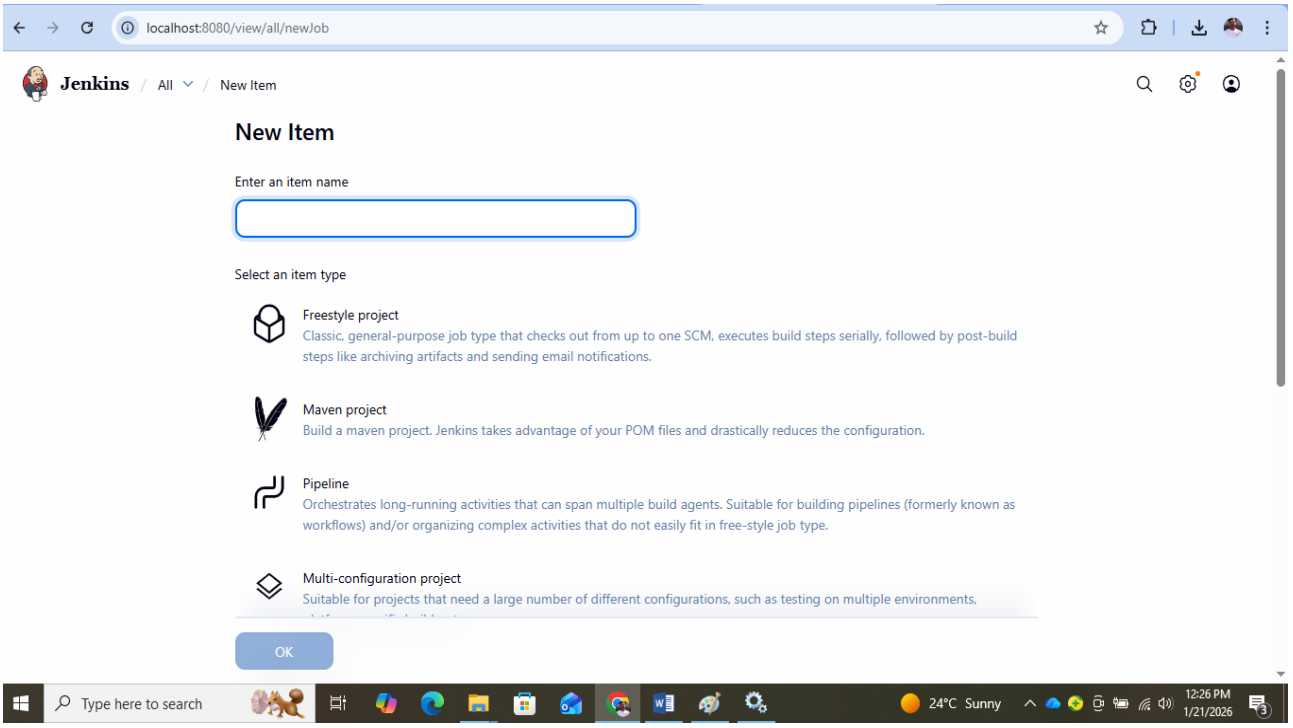
2.Login Using username(jenkins) and Password(jenkins)

3.Simple Build

i. Click on **New Item**

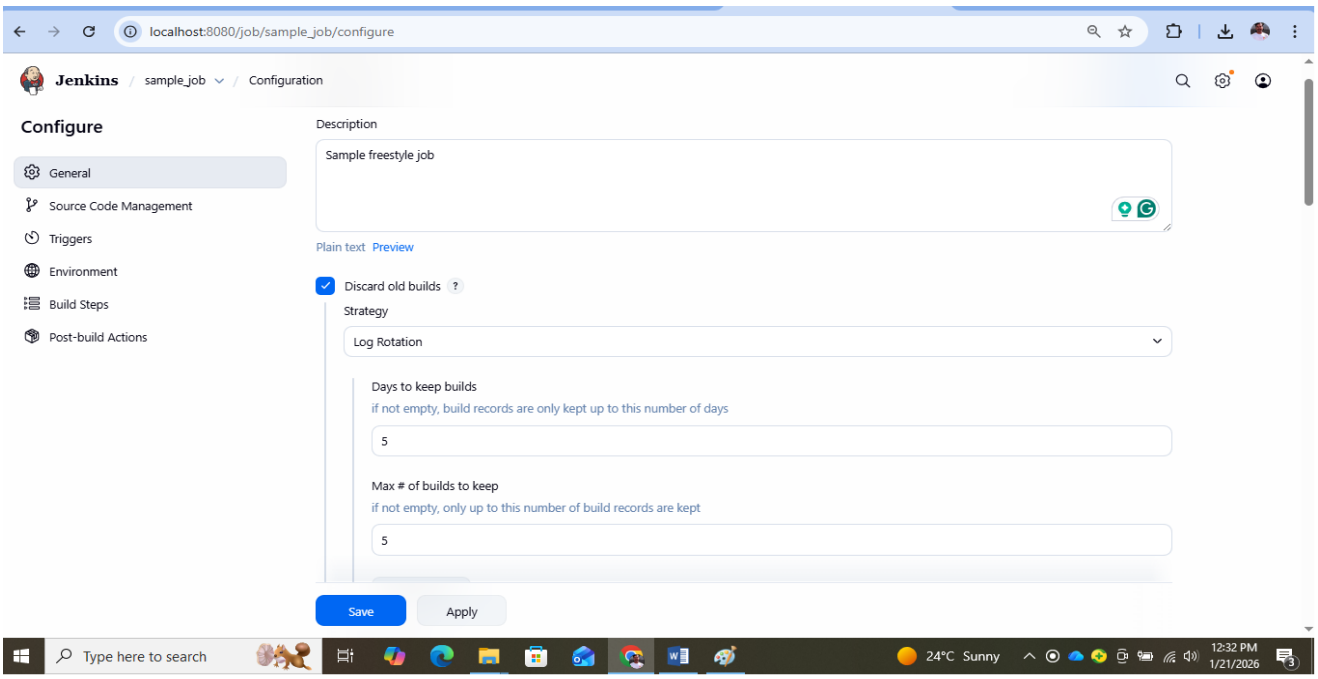
ii. Enter an **item name** (Ex: sample_job)

iii. Select **Freestyle project** and Click **Ok**



iv. Give **Description** (Example: Sample Free Style Job)

v. Select **Discard old Builds**: Enter days to keep builds and Max of Build to Keep

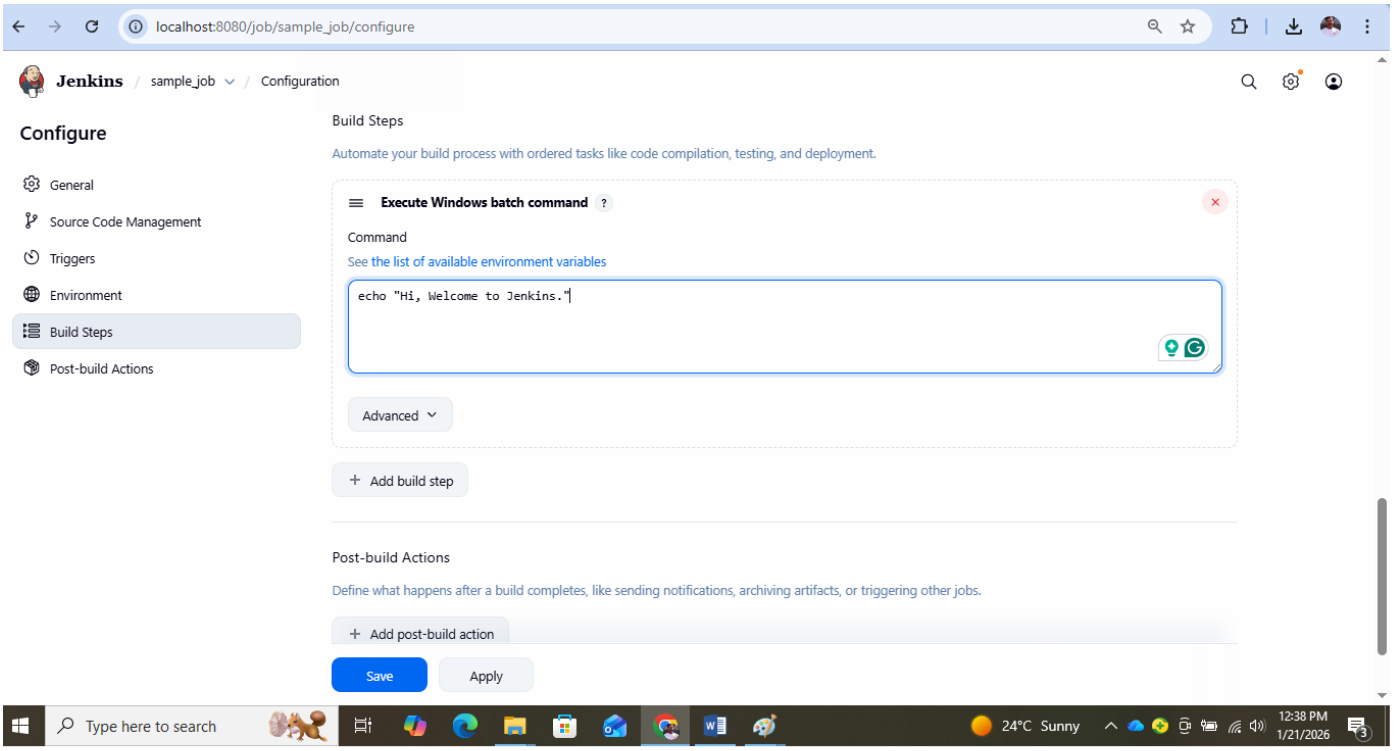


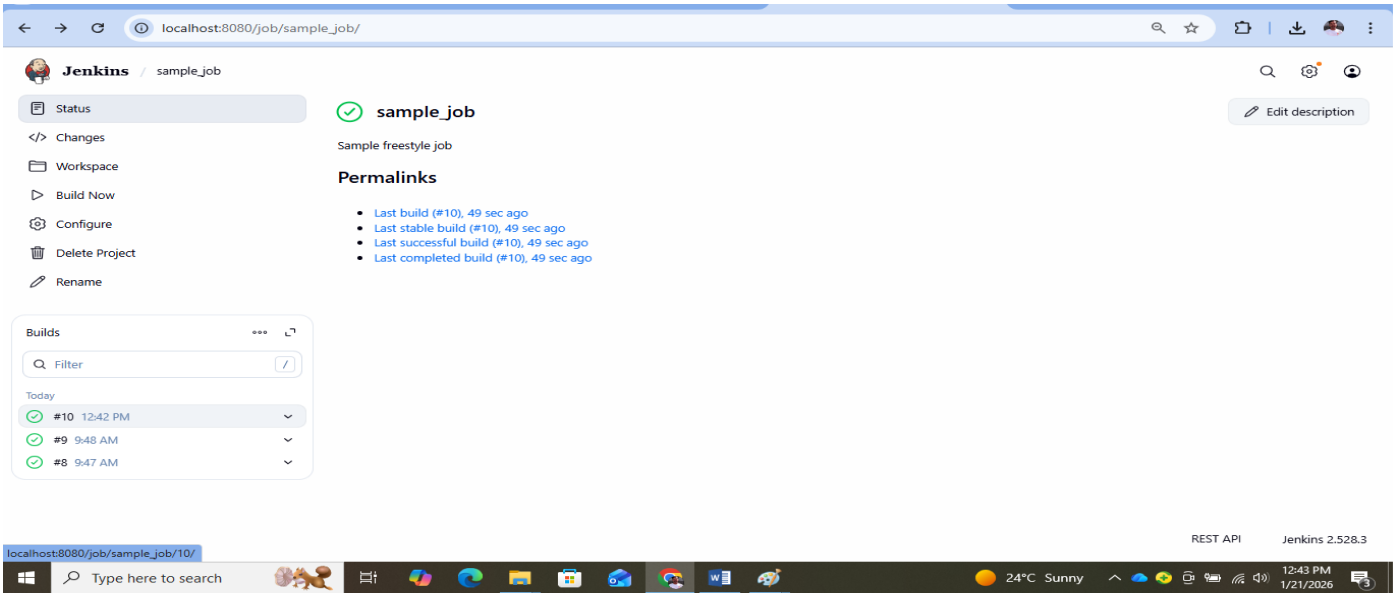
vi. In Build Step click on **Add build step**

vii. Select **Execute Windows Batch Command** and type **echo "Hi, Welcome to Jenkins."**

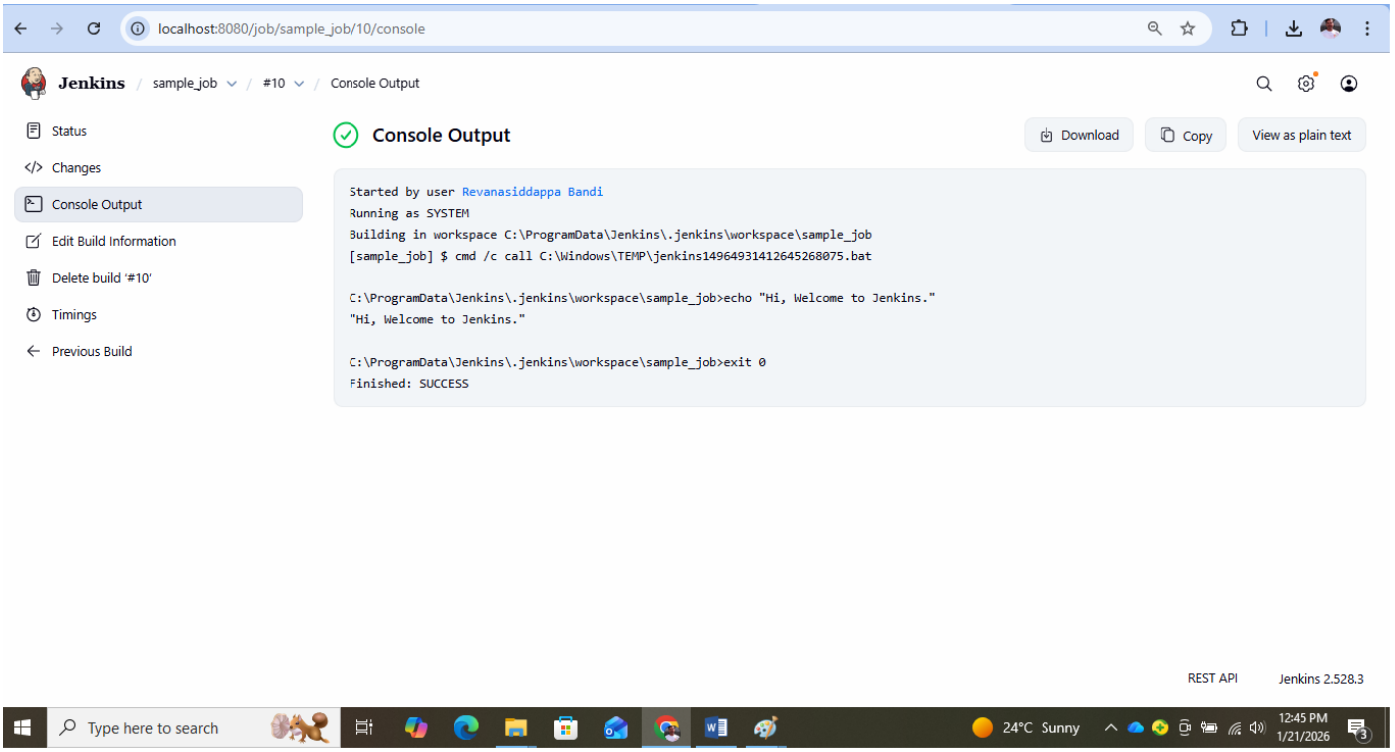
viii. Save and apply

ix. Click on **Build Now**





x. Click On the **Console Output** to Check the Output (The Printed output will be **"Hi, Welcome to Jenkins."**)


localhost:8080/job/sample_job/10/console
Jenkins / sample_job / #10 / Console Output
Status
Changes
Console Output
Edit Build Information
Delete build '#10'
Timings
Previous Build
Console Output
Download
Copy
View as plain text
Started by user Revanasiddappa Bandi
Running as SYSTEM
Building in workspace C:\ProgramData\Jenkins\.jenkins\workspace\sample_job
[sample_job] $ cmd /c call C:\Windows\TEMP\jenkins14964931412645268075.bat
C:\ProgramData\Jenkins\.jenkins\workspace\sample_job>echo "Hi, Welcome to Jenkins."
"Hi, Welcome to Jenkins."
C:\ProgramData\Jenkins\.jenkins\workspace\sample_job>exit 0
Finished: SUCCESS
REST API
Jenkins 2.528.3
Type here to search
24°C Sunny
12:45 PM
1/21/2026

Experiment 8: Jenkins Build Triggers

Aim:

To understand and configure Jenkins build automation by enabling build triggers such as SCM polling and GitHub webhooks in order to automatically trigger builds when changes occur in the source code repository.

Theory:

Jenkins is an open-source automation server that supports Continuous Integration (CI) by automatically building and testing software whenever changes are made to the source code. One of the key features of Jenkins is its ability to trigger builds automatically without manual intervention.

A build trigger defines the condition under which a Jenkins job is executed. Build triggers play a vital role in automation by ensuring that builds run immediately when changes are detected.

In this experiment, two important Jenkins build triggers are used:

SCM Polling

SCM (Source Code Management) polling allows Jenkins to periodically check the repository for any changes. Jenkins compares the current version of the code with the previously built version. If a change is detected, a new build is automatically triggered.

SCM polling is useful when webhooks are not available, but it may consume more server resources due to frequent checking.

GitHub Webhook

A GitHub webhook enables real-time communication between GitHub and Jenkins. Whenever a code change (such as a push or commit) occurs in the GitHub repository, GitHub sends a notification to Jenkins, which immediately triggers a build.

Webhooks are more efficient than polling because builds are triggered instantly and do not require repeated checking.

Outcome

By configuring SCM polling and GitHub webhooks, Jenkins automatically triggers builds whenever source code changes occur, achieving efficient and reliable automation.

Steps:

Configure SCM polling and GitHub webhook with Automatic build Triggering

i. Create the **Repository** with the Name **Github_Bankservice** in Github

ii. Add the **BankServive.java** to this Repository

```
public class BankService {

    // Nested class (can be public or private, doesn't matter for functionality)
    static class BankAccount {
        private double balance;

        public BankAccount(double initialBalance) {
            if (initialBalance < 0) {
                throw new IllegalArgumentException("Initial balance cannot be negative");
            }
```

```
        this.balance = initialBalance;
    }

    public void deposit(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("Deposit must be positive");
        }
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount <= 0 || amount > balance) {
            throw new IllegalArgumentException("Invalid withdrawal");
        }
        balance -= amount;
    }

    public double getBalance() {
        return balance;
    }
  }

  // The main entry point
  public static void main(String[] args) {
    // Create an instance of the BankAccount class
    BankAccount acc = new BankAccount(5000);

    System.out.println("Initial Balance: " + acc.getBalance());

    // Perform operations
    acc.deposit(700);
    System.out.println("Balance after deposit of 500: " + acc.getBalance());

    acc.withdraw(100);
    System.out.println("Balance after withdrawal of 300: " + acc.getBalance());

    // Print final result
    System.out.println("Final Balance: " + acc.getBalance());
  }
}
```

Save the Above Code With the **name BankService.java**

iii. Now in the Jenkins Click on **New Item** and Enter the Item Name **GitHub_BankApplication**

iv. Select **Freestyle Project** and Click on **Ok**

v. Enter the **Description**

vi. In the **Source Code Management Select Git**

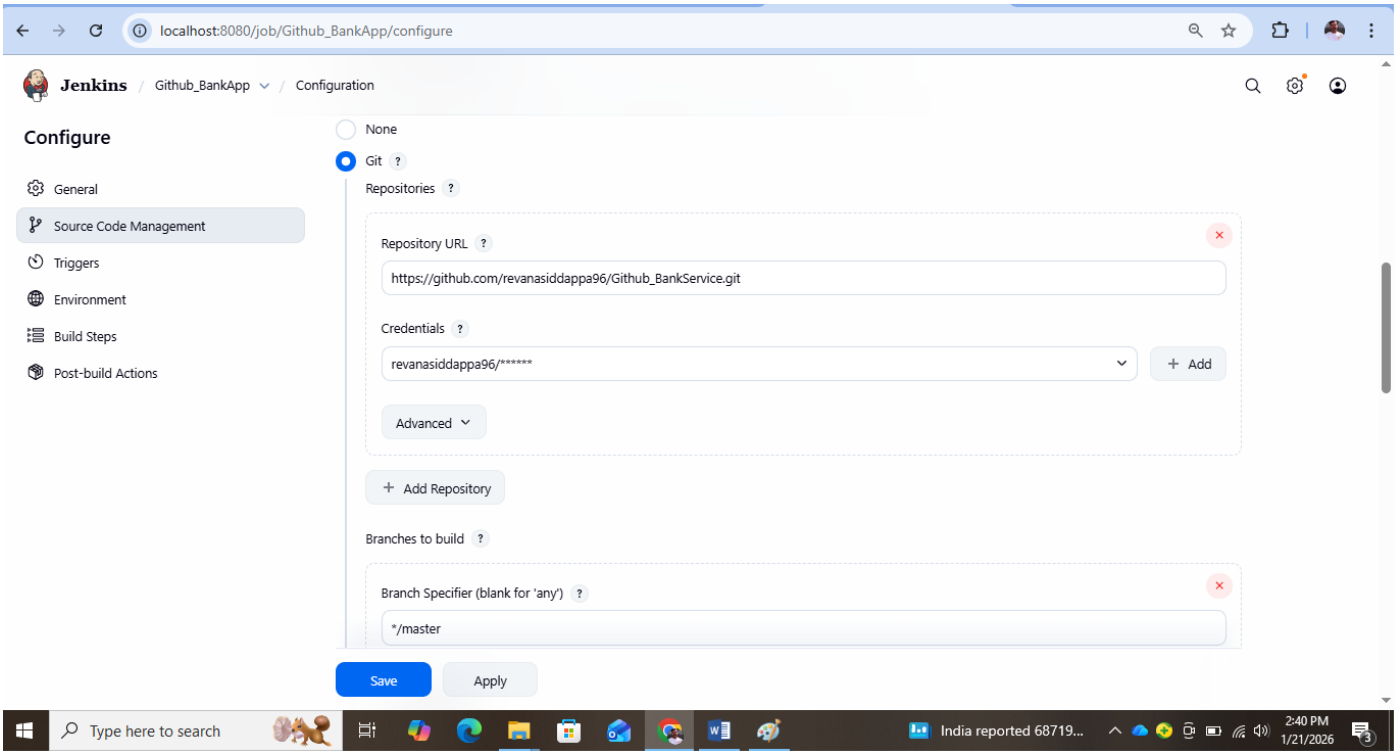
vii. Repository URL : Enter your **Repository URL** (Example: https://github.com/username/Github_Bankservice)

viii. In the Credentials:

- Click on Add
- Add your **Username and Password** of Github

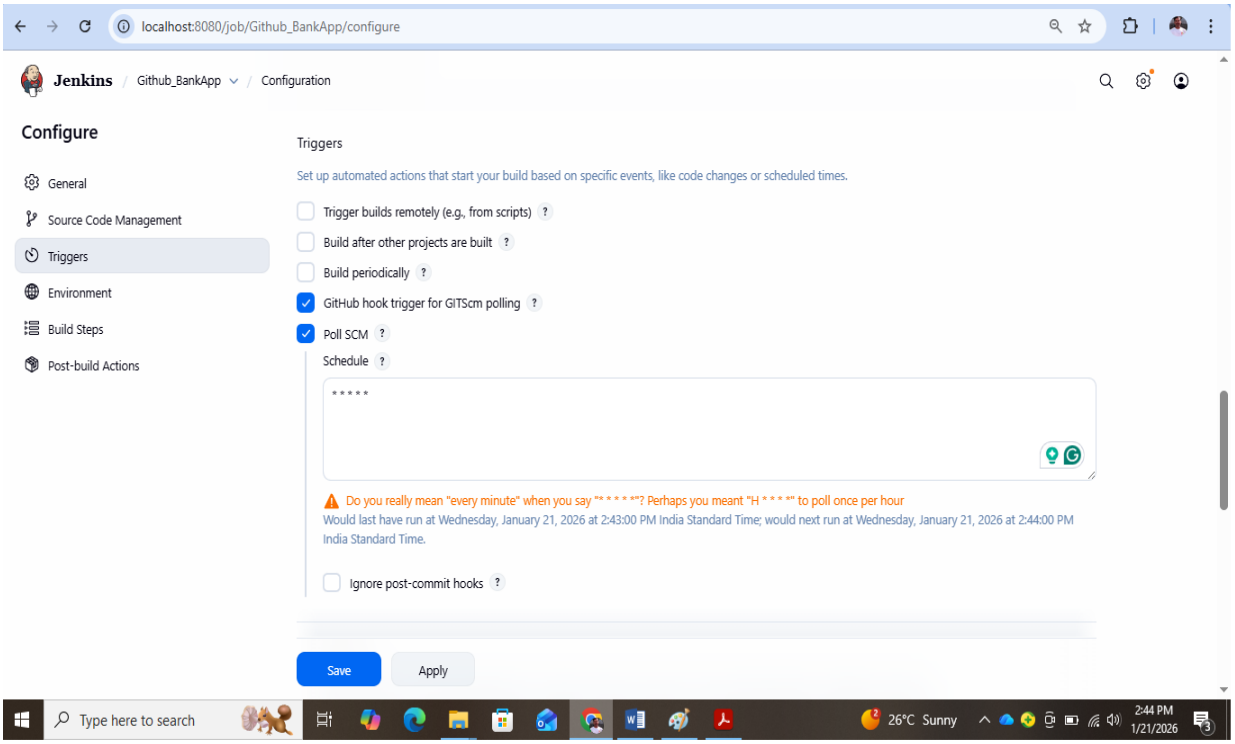
ix. In Branches to Build Section

Specify **the Branch in the Branch Specifier Section (*/main or */master**)



x. In the Triggers Section **Select Github hook trigger for GITScm polling and Poll SCM**

- In the **Schedule Enter * * * * ***

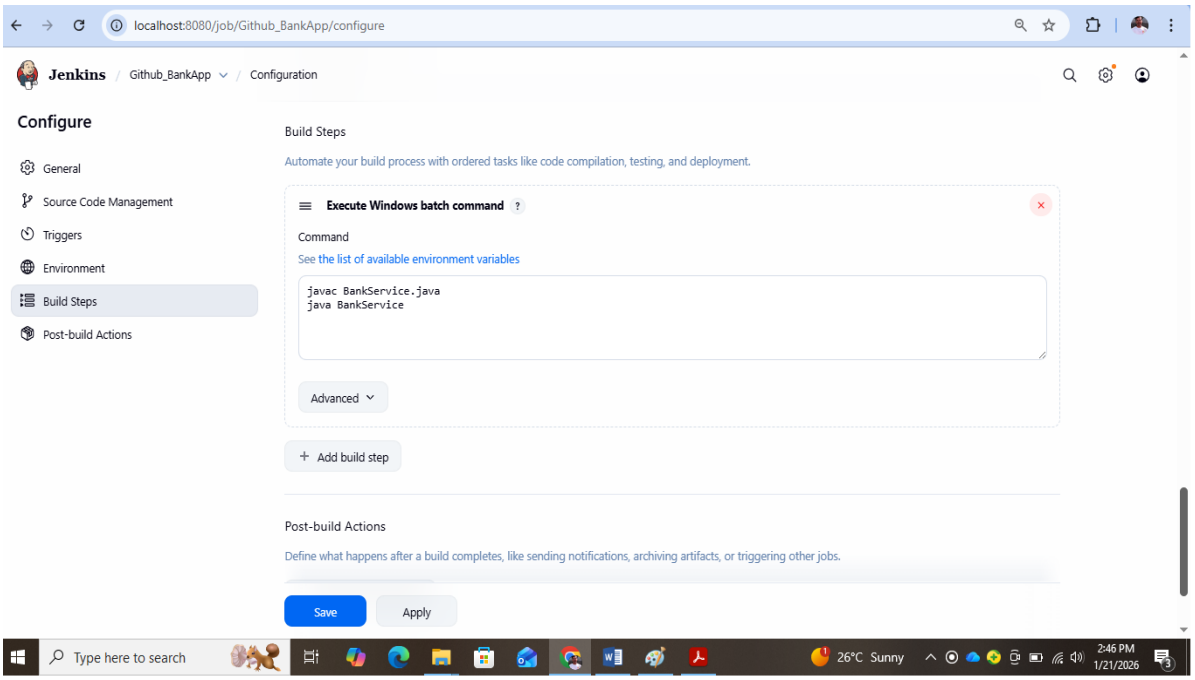


xi. In Build Step Enter **Add build Step** and select the **Execute Windows Batch Command**

xii. **Enter**

- **javac BankService.java**

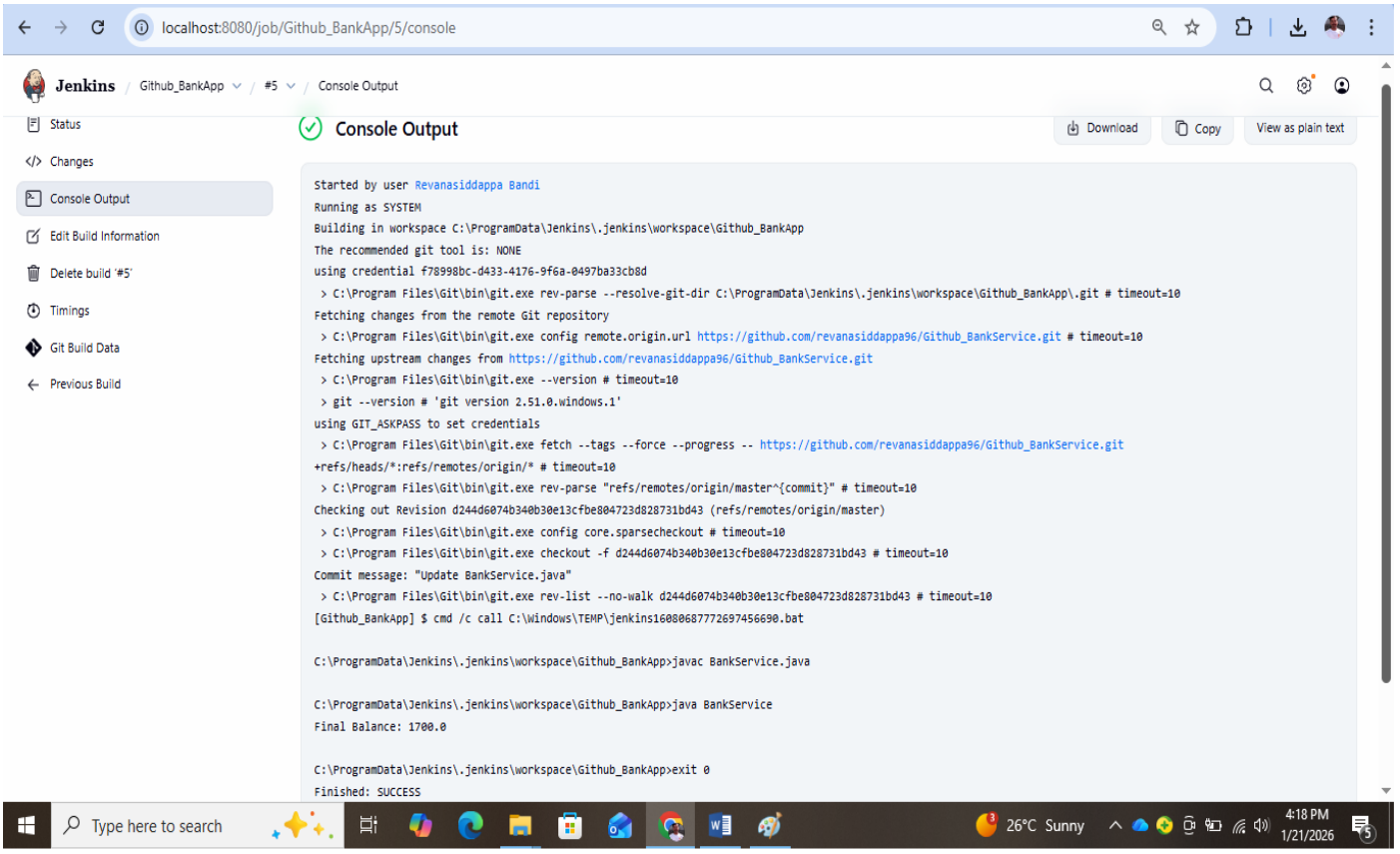
- **java BankService**



xiii. Click on **Save**

xiv. Enter **Build Now**

xv. Click on **Console Output** and **Check the Output**

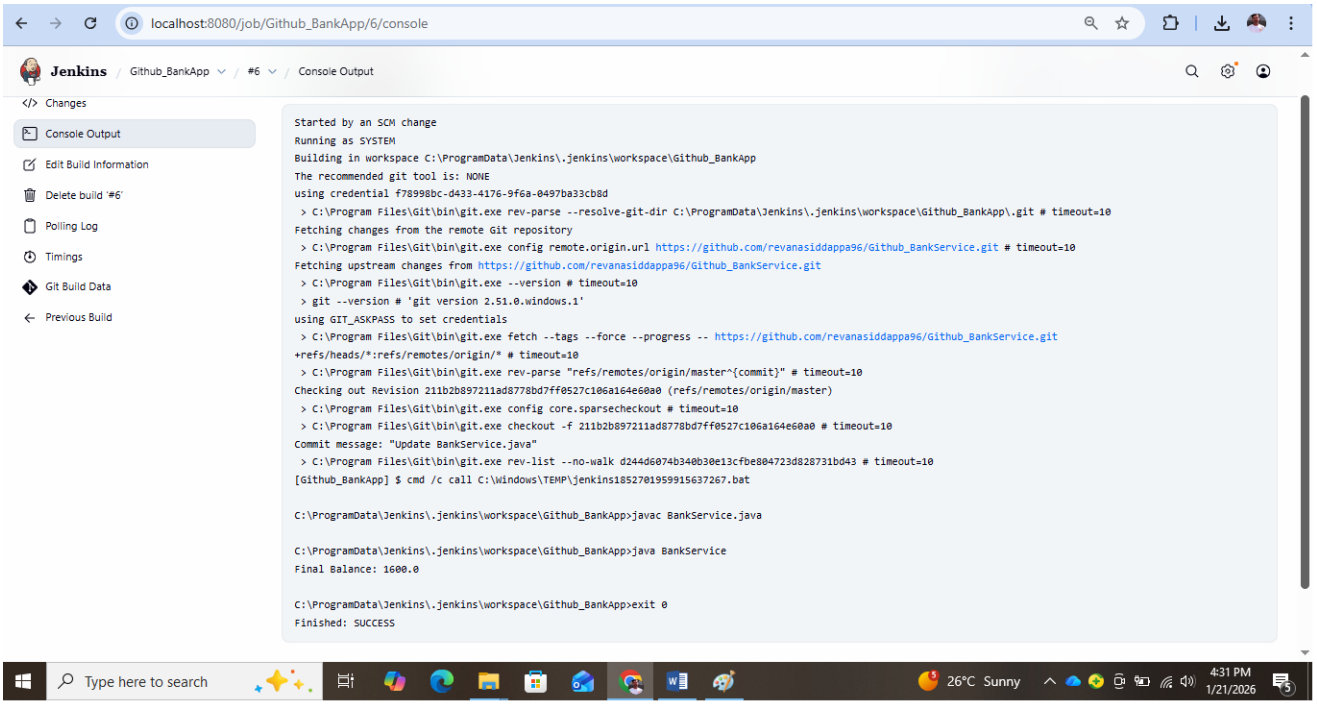


Now Make the Changes to code in the Github (Example: Change Amount from 200 to 100)

i. Now Come to Jenkins **Status Section**

ii. Now the **Build Will Automatically Starts** Since we have **given trigger**

iii. Now Click the **Console Output** and check the output for the **Changed Code**



Experiment 9: Creation and Build of a Java Project Using Apache Maven

Aim:

To understand the use of Apache Maven by creating a Java project using Maven archetypes, building the project using Maven build lifecycle commands, managing dependencies, and generating project documentation.

Theory:

Apache Maven is a build automation and project management tool primarily used for Java-based applications. Maven simplifies the process of project creation, compilation, dependency management, testing, packaging, and documentation using a standardized project structure and configuration.

Maven follows the concept of Project Object Model (POM), which is defined in a file named pom.xml. This file contains all essential project information such as group ID, artifact ID, version, dependencies, and build configurations.

Key Concepts in Maven

Maven Archetype

A Maven archetype is a project template that generates a standard directory structure and configuration files. The command mvn archetype:generate connects to the internet and downloads required templates to create a ready-to-use project skeleton.

GroupId and ArtifactId

- groupId represents the organization or domain name (e.g., company or university).
- artifactId is the unique name of the application.
  Together, they uniquely identify a Maven project.

Local Repository (.m2 folder)

Maven downloads all required dependencies and plugins from remote repositories and stores them in the local repository located in the .m2 directory. This avoids repeated downloads.

Maven Build Lifecycle

Maven provides a predefined build lifecycle with important phases:

- **mvn package**: Compiles the source code and packages it into a JAR file, creating a target directory.
- **mvn clean**: Removes previously generated build files by deleting the target directory.
- **mvn site:** Generates project documentation, reports, and HTML pages inside the target/site directory.

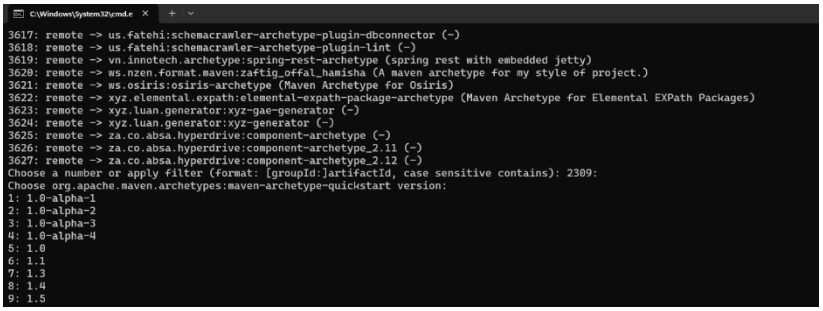
Steps:

1. To **verify the installation of Apache Maven** and display **version and environment details**.

>>mvn --version

2.mvn archetype:generate -> (Create the directory structure and connect to the internet and download the diff packegs/templates)

3.Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 2309:(type enter)



4.**Choose a number: 9:(press enter)**gives the java compliler version

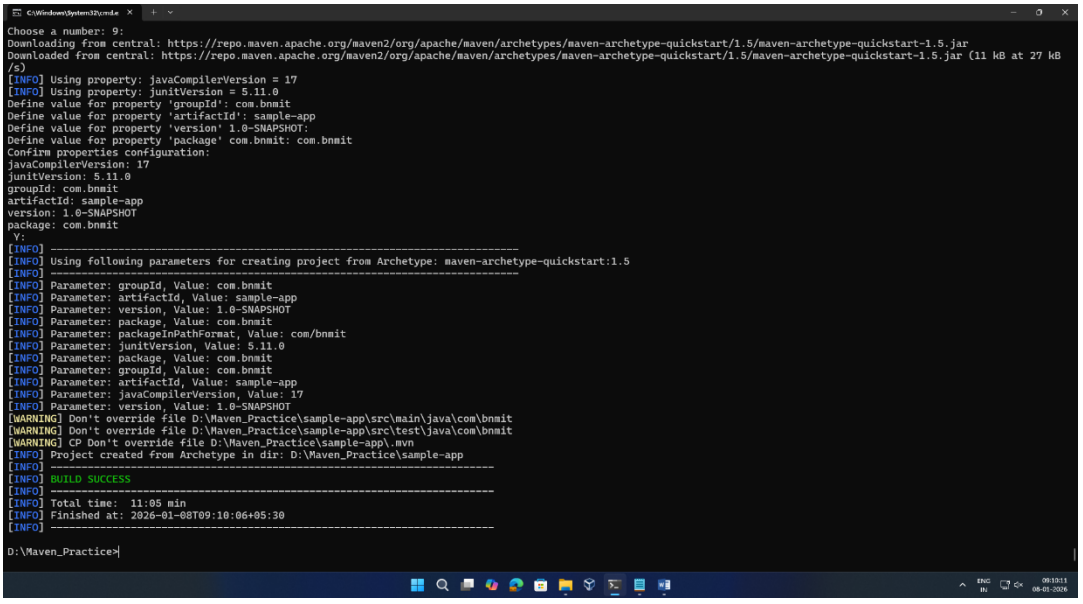
5.groupId(company or university name)=com.bnmit

6.artifactid(unique id for application):sample-app

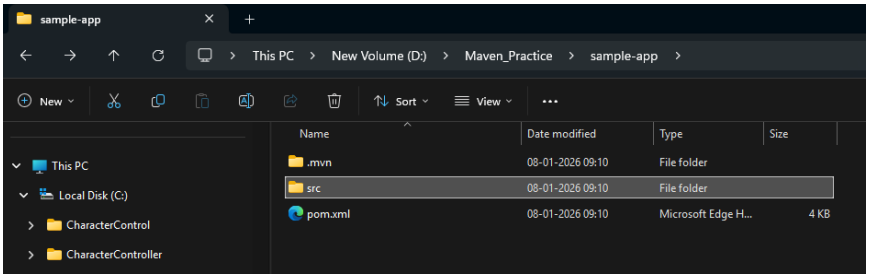
7.'version' 1.0-SNAPSHOT:(press enter)

8.'package' com.bnmit:com.bnmit (list out all configuratons)

9.press "y"

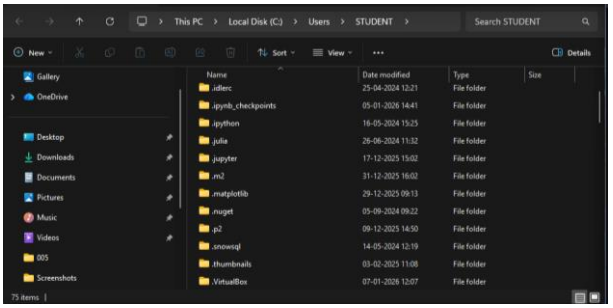


10.check the folder the **sample-app folder** will be created in your directory



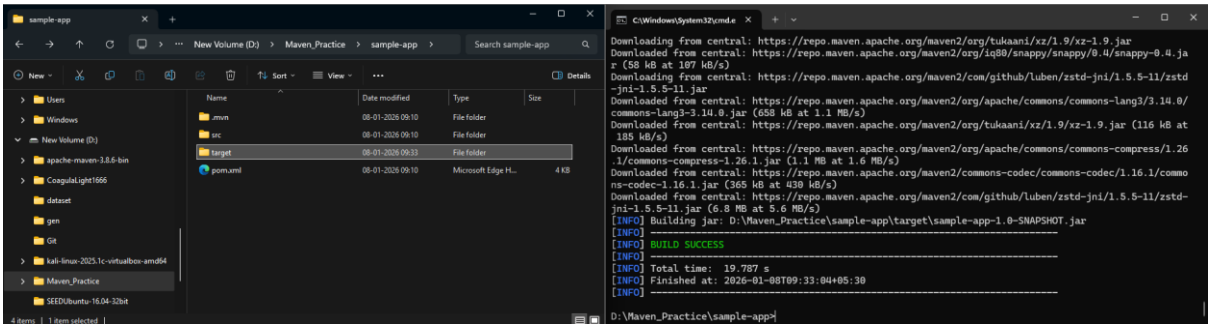
Maven downloads all required dependencies and plugins from remote repositories and stores them in the local repository located in the .m2 directory. This avoids repeated downloads.

11.open **local c->user->student->open .m2 file**

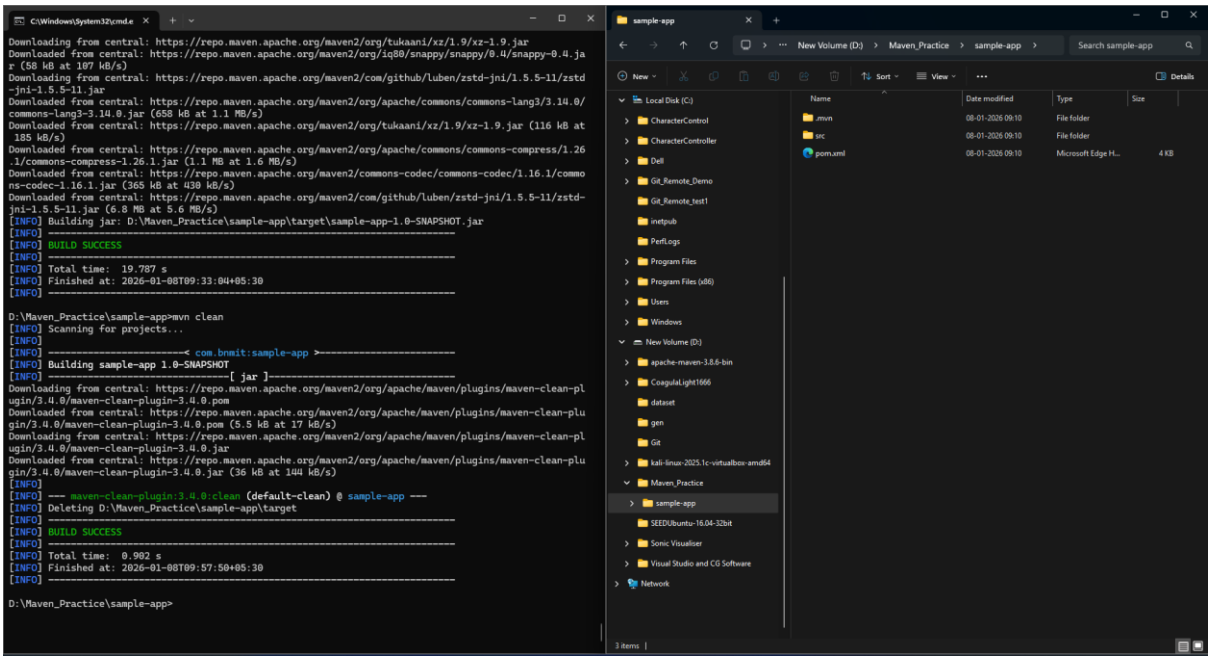


12.mvn package:

It will create the target folder inside the sample-app folder



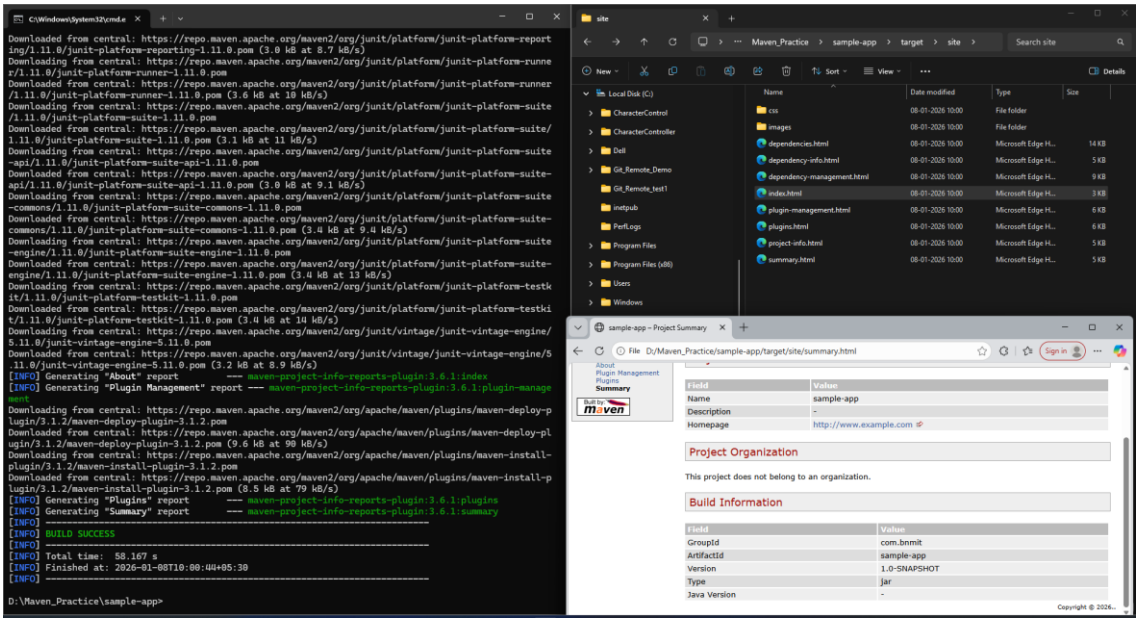
13.To delete all the built application->mvn clean(target folder will be deleted)



14.mvn site :

mvn site is used to **generate project documentation and reports** automatically for a Maven project.

The site folder will be created in the target file,open the **index.html** and explore



Modification and Execution of Java Application in a Maven Project

1.try to change the hello world code and put other code and execute it

sample-app-->src-->main-->java-->com-->bnmit-->app.java-->open file and paste the Bank Application Code

```
class BankAccount {
   private double balance;

   public BankAccount(double initialBalance) {
      if (initialBalance < 0)
         throw new IllegalArgumentException("Initial balance cannot be negative");
      this.balance = initialBalance;
   }
   public void deposit(double amount) {
      if (amount <= 0)
         throw new IllegalArgumentException("Deposit must be positive");
      balance += amount;
   }

   public void withdraw(double amount) {
      if (amount <= 0 || amount > balance)
         throw new IllegalArgumentException("Invalid withdrawal");
      balance -= amount;
   }

  public double getBalance() {
      return balance;
   }
}
public class BankService {
   public static void main(String[] args) {
      BankAccount acc = new BankAccount(1000);
      acc.deposit(500);
      acc.withdraw(300);
      System.out.println("Final Balance: " + acc.getBalance());
   }
}
```

2.change the name of the file app.java->BankService.java

3.mvn clean

4.mvn package

5.class folder -->two class files will be created

6.**cd target**

7.java -cp sample-app-1.0-SNAPSHOT.jar com.bnmit.BankService