

Week: 08

Name: Khushi

Superset id: 6390178

Git Hands-On

1.Objectives

Be familiar with Git commands like:

- git init
- git status
- git add
- git commit
- git push
- git pull

Prerequisites

- Git Bash installed on your machine
- Notepad++ installed (for editor integration)
- GitLab account created (don't use company credentials)

Step 1: Setup Git Configuration on Your Machine

1. Check if Git is installed:

```
git --version
```

2. Set Git username and email:

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

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

3. Verify Git configuration:

```
git config --list
```

Step 2: Integrate Notepad++ with Git as Default Editor

1. Check if Notepad++ opens from Git Bash:

```
notepad++
```

If not recognized, add Notepad++ path to Environment Variables.

2. Restart Git Bash and try again:

```
notepad++
```

3. Create alias for Notepad++:

```
alias np='notepad++'
```

To make it permanent, run:

```
notepad ~/.bashrc
```

And add: `alias np='notepad++'`

4. Set Notepad++ as default editor:

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

5. Verify editor configuration:

```
git config --global -e
```

Step 3: Add File to Source Code Repository

1. Create a new folder and initialize Git:

```
mkdir GitDemo
```

```
cd GitDemo
```

```
git init
```

2. Verify initialization:

```
ls -a
```

3. Create and add content to welcome.txt:

```
echo "Welcome to Git Demo!" > welcome.txt
```

4. Verify file creation:

```
ls
```

5. View file content:

```
cat welcome.txt
```

6. Check Git status:

```
git status
```

7. Add file to staging area:

```
git add welcome.txt
```

8. Commit the change (opens Notepad++):

```
git commit
```

Add multi-line commit message and save.

9. Check Git status again:

```
git status
```

Step 4: Connect to Remote GitLab Repository

1. Create a new GitLab project named GitDemo.

2. Link local repo with remote:

```
git remote add origin https://gitlab.com/your-username/GitDemo.git
```

3. Pull from remote (if needed):

```
git pull origin master --allow-unrelated-histories
```

4. Push local repo to GitLab:

```
git push -u origin master
```

Summary

Commands Covered:

- git init

- git status

- git add

- git commit
- git config
- git push
- git pull

OUTPUT:

Output	Clear
<pre>git version 2.42.0.windows.1 user.name=Your Name user.email=youremail@example.com Initialized empty Git repository in C:/Users/YourName/GitDemo/.git/ On branch master No commits yet Untracked files: (use "git add <file>..." to include in what will be committed) welcome.txt nothing added to commit but untracked files present (use "git add" to track) On branch master No commits yet Changes to be committed: (use "git rm --cached <file>..." to unstage) new file: welcome.txt</pre>	

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: welcome.txt

[master (root-commit) 3a3b3c4] Initial commit

1 file changed, 1 insertion(+)

create mode 100644 welcome.txt

Enumerating objects: 3, done.

Counting objects: 100% (3/3), done.

Writing objects: 100% (3/3), 258 bytes | 258.00 KiB/s, done.

Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

To <https://gitlab.com/your-username/GitDemo.git>

* [new branch] master -> master

2.Objectives

- Explain .gitignore
- Explain how to ignore unwanted files using .gitignore
- Implement .gitignore in a local Git repository

Prerequisites

- Git installed and configured
- Notepad++ integrated as default editor
- Local Git repository already created and linked to GitLab

Estimated Time

20 minutes

Step-by-Step Guide

1. Navigate to your Git project:

```
cd GitDemo
```

2. Create files and folders to be ignored:

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

```
mkdir log
```

```
echo "Another log file" > log/app.log
```

3. Check Git status before ignoring:

```
git status
```

Expected Output:

Untracked files:

(use "git add <file>..." to include in what will be committed)

```
debug.log
```

log/

4. Create a .gitignore file and add ignore rules:

notepad++ .gitignore

Add the following lines:

*.log

log/

Save and close the file.

5. Check git status again:

git status

Expected Output:

Untracked files:

(use "git add <file>..." to include in what will be committed)

.gitignore

(debug.log and log/ are now ignored)

6. Add and commit the .gitignore file:

git add .gitignore

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

7. Push to remote repository:

git push origin master

Summary

- .gitignore allows you to exclude specific files or folders from being tracked by Git.
- Useful for ignoring logs, temp files, build artifacts, etc.
- Patterns like *.log or folder/ ensure they are excluded from version control.

OUTPUT:

```
Output Clear
On branch master

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

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

*.log
log/
On branch master

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

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

[master 84b3e21] Added .gitignore to exclude .log files and log/ directory
1 file changed, 2 insertions(+)
create mode 100644 .gitignore
```

```
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 265 bytes | 265.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://gitlab.com/your-username/GitDemo.git
* [new branch]      master -> master
```


3.Objectives

- Explain branching and merging
- Explain creating a branch request in GitLab
- Explain creating a merge request in GitLab

Prerequisites

- Git environment setup
- P4Merge tool installed and configured on Windows
- GitLab account created (do not use cognizant credentials)

Step-by-Step Instructions

Branching:

1. Create a new branch "GitNewBranch":

```
git branch GitNewBranch
```

2. List all branches (local and remote):

```
git branch -a
```

Observe the "*" mark next to the current branch.

3. Switch to the new branch:

```
git checkout GitNewBranch
```

4. Add files with content:

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

5. Stage and commit the changes:

```
git add feature.txt
```

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

6. Check the status:

```
git status
```

Merging:

1. Switch back to master branch:

git checkout master

2. List differences between master and GitNewBranch:

git diff master GitNewBranch

3. Visual diff using P4Merge (if configured):

git difftool master GitNewBranch

4. Merge GitNewBranch into master:

git merge GitNewBranch

5. View merged history visually:

git log --oneline --graph --decorate

6. Delete the branch after merge:

git branch -d GitNewBranch

7. Check status again:

git status

OUTPUT:

```
GitNewBranch
* master
  GitNewBranch
  remotes/origin/master
  remotes/origin/GitNewBranch
Switched to branch 'GitNewBranch'
[GitNewBranch abc1234] Added feature.txt in GitNewBranch
 1 file changed, 1 insertion(+)
 create mode 100644 feature.txt
On branch GitNewBranch
nothing to commit, working tree clean
```

Output

Clear

```
Switched to branch 'master'
diff --git a/feature.txt b/feature.txt
new file mode 100644
index 0000000..e69de29
--- /dev/null
+++ b/feature.txt
@@ -0,0 +1 @@
+This is a new feature
Viewing differences using P4Merge...
Updating 1234567..abc1234
Fast-forward
 feature.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 feature.txt
* abc1234 (HEAD -> master, origin/master) Added feature.txt in GitNewBranch
* 1234567 Initial commit
Deleted branch GitNewBranch (was abc1234).
On branch master
nothing to commit, working tree clean
```

4.Objectives

- Explain how to resolve conflicts during a merge.

Prerequisites:

- Install [Git](#)
- Install [P4Merge](#)
- Configure Git and P4Merge:

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

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

```
git config --global merge.tool p4merge
```

```
git config --global mergetool.p4merge.cmd "C:/Program Files/Perforce/p4merge.exe"
"$BASE" "$LOCAL" "$REMOTE" "$MERGED"
```

(Adjust P4Merge path based on your OS)

Step 1: Verify if master is in clean state

```
git checkout master
```

```
git status
```

Ensure it shows: nothing to commit, working tree clean

Step 2: Create a branch "GitWork". Add a file "hello.xml".

git checkout -b GitWork

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

Step 3: Update the content of "hello.xml" and observe the status

echo "<message>Updated in GitWork branch</message>" > hello.xml

git status

Step 4: Commit the changes to reflect in the branch

git add hello.xml

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

Step 5: Switch to master

git checkout master

Step 6: Add a file "hello.xml" to the master and add some different content

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

Step 7: Commit the changes to the master

git add hello.xml

git commit -m "Added hello.xml in master with different content"

Step 8: Observe the log

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

Step 9: Check the differences with Git diff tool

git diff GitWork

Step 10: Use P4Merge tool to visualize differences

git difftool GitWork

Step 11: Merge the branch to master

git merge GitWork

This should trigger a merge conflict.

Step 12: Observe the Git markup

cat hello.xml

You'll see:

```
<<<<<<< HEAD
```

```
<message>Hello from master branch</message>
```

```
=====
```

```
<message>Updated in GitWork branch</message>
```

```
>>>>>>> GitWork
```

Step 13: Use 3-way merge tool (P4Merge) to resolve the conflict

```
git mergetool
```

P4Merge opens, resolve and save the merged file.

Step 14: Commit the changes to master, once done

```
git add hello.xml
```

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

Step 15: Observe the status and add backup file to .gitignore

```
git status
```

```
echo "*.orig" >> .gitignore
```

Step 16: Commit the changes to .gitignore

```
git add .gitignore
```

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

Step 17: List out all the available branches

```
git branch
```

Step 18: Delete the branch which merged to master

```
git branch -d GitWork
```

Step 19: Observe the log again

```
git log --oneline --graph --decorate
```

OUTPUT:

Output

```
[Hands-On 4 Output]  
Created conflicting changes in hello.xml  
Merge conflict detected  
Conflict resolved using mergetool  
Committed resolved file  
Ignored *.orig backup files
```

5.Objectives

- Clean up local changes.
- Push the updated work to the remote repository.
- Confirm that changes reflect in GitHub.

Prerequisites

- Git installed and configured (git config)
- GitHub account and a remote repo created
- Hands-on ID: **Git-T03-HOL_002**
- You've already worked locally and committed changes

1. Verify if master is in clean state

git status

If clean:

nothing to commit, working tree clean

If not clean:

git add .

git commit -m "Committing pending changes for Git-T03-HOL_002"

2. List out all the available branches

git branch -a

Expected Output:

* master

feature-branch

remotes/origin/master

remotes/origin/feature-branch

3. Pull the remote git repository to the master

git pull origin master

This fetches the latest changes from the remote master branch and merges into your local master.

4. Push the changes to the remote repository

```
git push origin master
```

If pushing for the first time or the branch is not tracked yet:

```
git push -u origin master
```

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

- Go to your GitHub repository in your browser.
- Confirm if the latest commits and files are visible.

Optional Cleanup:

To remove merged branches locally:

```
git branch --merged
```

```
git branch -d feature-branch-name
```

To delete a branch from remote (if needed):

```
git push origin --delete feature-branch-name
```

OUTPUT:

Output

```
[Hands-On 5 Output]  
Working tree clean  
Already up to date with remote  
Pushed to origin/master successfully
```


