**Digital Nurture 7.0 Deep Skilling**

**Week 8**

**Hands on 1**

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

**GIT BASH :**

**A screen shot of a computer

Description automatically generated**

**OUTPUT :**

**In Gitlab :**

**A screenshot of a computer

Description automatically generated**

**A black rectangular object with a white border

Description automatically generated**

**Hands on 2**

**1: What is .gitignore in Git?**

**Answer:**

* .gitignore is a text file that tells Git which files or folders to **ignore** in a repository.
* Files listed in .gitignore are **not tracked** by Git, meaning they won’t be staged, committed, or pushed to the remote repository.
* It’s typically used to ignore temporary files, build outputs, logs, environment configuration files, or anything that should stay local and not be shared with others.

**Q2: How do you ignore unwanted files using .gitignore?**

**Answer:**

1. Create or edit a file named .gitignore in the root of your repository.
2. Add patterns for files/folders you want Git to ignore. For example:

# Ignore all .log files

\*.log

# Ignore the log folder

log/

1. Save the. gitignore file.
2. If any files you want to ignore were **already tracked** by Git, untrack them using:

CopyEdit

git rm --cached filename

1. Commit the. gitignore file so it’s shared with the project:

CopyEdit

git add .gitignore

git commit -m "Add .gitignore to ignore unwanted files"

git push

1. From now on, files matching those patterns will be ignored by Git.

Implement git ignore command to ignore unwanted files and folders

**A screen shot of a computer

Description automatically generated**

**A black screen with white text

Description automatically generated**

**Output :**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Notepad++**

**A screenshot of a computer

Description automatically generated**

**Hand on 3**

Objectives :

**1: What is branching and merging in Git?**

**Answer:**

* **Branching** is the process of creating an independent line of development from the main codebase. It allows developers to work on new features, bug fixes, or experiments without affecting the stable main branch (e.g., main or master).
* **Merging** is the process of integrating changes from one branch into another. Once work in a branch is complete and tested, it can be merged back into the main branch so the new changes become part of the project.

**2: How do you create a branch request (Merge Request) in GitLab?**

**Answer:**  
In GitLab, a **branch request** usually means pushing a branch to the remote repository so it can be reviewed and merged.  
Steps:

1. **Create a branch locally:**

bash

CopyEdit

git checkout -b feature-branch

1. **Make changes, commit, and push the branch to GitLab:**

bash

CopyEdit

git add .

git commit -m "My new feature"

git push origin feature-branch

1. **Go to GitLab → Project → Repository → Branches** — you’ll see your new branch listed.
2. This pushed branch can now be used to create a **Merge Request**.

**3: How do you create a Merge Request in GitLab?**

**Answer:**  
A **Merge Request (MR)** in GitLab is a request to merge changes from one branch into another, often from a feature branch into main or master.  
Steps:

1. In GitLab, after pushing your branch, you’ll see a banner suggesting **"Create merge request"** — click it.
2. Or go to **Merge Requests** → **New merge request**.
3. Select:
   * **Source branch** → your new feature branch.
   * **Target branch** → the branch you want to merge into (usually main).
4. Add a title and description for your changes.
5. Assign reviewers if required.
6. Click **"Create merge request"**.
7. After review and approval, click **"Merge"** to complete the process.

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

**A computer screen shot of a black screen

Description automatically generatedA computer screen shot of text

Description automatically generated**

**Output :**

A screenshot of a computer

Description automatically generated

A black screen with white text

Description automatically generated

**Hands on 4**

### ****1: How do you resolve a conflict during a Git merge?****

**Answer:**  
When two branches change the same part of a file, Git cannot automatically decide which change to keep — this creates a **merge conflict**.  
To resolve it:

1. **Start the merge**

bash

Copy code

git merge branch-name

If a conflict occurs, Git shows:

pgsql

Copy code

CONFLICT (content): Merge conflict in filename

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

1. **Check which files have conflicts**

bash

Copy code

git status

Files with conflicts will be marked as **both modified**.

1. **Open the conflicting file**  
   Inside the file you’ll see conflict markers:

pgsql

Copy code

<<<<<<< HEAD

Code from the current branch

=======

Code from the branch being merged

>>>>>>> branch-name

1. **Edit the file**
   * Choose one version,
   * Combine both,
   * Or rewrite the section entirely.
2. **Mark the conflict as resolved**

bash

Copy code

git add filename

1. **Commit the merge result**

bash

Copy code

git commit -m "Resolve merge conflict in filename"

1. **Push to remote (if needed)**

bash

Copy code

git push origin branch-name

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.

GitBash :

A screen shot of a computer

Description automatically generated

**A screenshot of a computer

Description automatically generated**

**A computer screen shot of a black screen

Description automatically generated**

**Output :**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Hands on 5**

### ****1: Explain how to clean up and push back to remote Git****

**Answer:**  
Cleaning up and pushing back to remote Git means making sure your local repository is in a tidy, consistent state before sending changes to the remote repository. The steps are:

1. **Check your working tree**

CopyEdit

git status

* + Make sure you know which files are staged, unstaged, or untracked.

1. **Remove or ignore unwanted files**
   * Delete temporary or backup files you don’t want in Git:

CopyEdit

rm filename

* + Or add them to .gitignore to prevent tracking.

1. **Stage and commit valid changes**

CopyEdit

git add .

git commit -m "Your commit message"

1. **Sync with remote before pushing**

CopyEdit

git fetch origin

git pull --rebase origin main # or master

This ensures you integrate any remote changes into your local branch.

1. **Push your clean branch to remote**

CopyEdit

git push origin main

This updates the remote repository with your local commits.

1. **Optional cleanup of branches**
   * Delete merged local branches:

CopyEdit

git branch -d branch-name

* + Delete remote branches no longer needed:

CopyEdit

git push origin --delete branch-name

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

**A screen shot of a computer

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

**Output :**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**