## **4.Git-HOL**

## **Objectives**

* Explain how to resolve the conflict during merge.

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

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

## **Prerequisites**

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

* Hands-on ID: **“Git-T03-HOL\_001”**

Notes\*:

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

Estimated time to complete this lab: **30 minutes.**

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

1. Verify if master is in clean state.
2. Create a branch **“GitWork”.** Add a file “hello.xml”.
3. Update the content of “hello.xml” and observe the status
4. Commit the changes to reflect in the branch
5. Switch to master.
6. Add a file **“hello.xml”** to the master and add some different content than previous.
7. Commit the changes to the master
8. Observe the log by executing **“git log –oneline –graph –decorate –all”**
9. Check the differences with Git diff tool
10. For better visualization, use P4Merge tool to list out all the differences between master and branch
11. Merge the bran to the master
12. Observe the git mark up.
13. Use 3-way merge tool to resolve the conflict
14. Commit the changes to the master, once done with conflict
15. Observe the git status and add backup file to the .gitignore file.
16. Commit the changes to the .gitignore
17. List out all the available branches
18. Delete the branch, which merge to master.
19. Observe the log by executing **“git log –oneline –graph –decorate”**

# A Detailed Guide to Your First Git Repository

This document provides a comprehensive, step-by-step walkthrough of the fundamental Git commands. It expands on the provided lab instructions to ensure a deep understanding of each concept, from local repository initialization to pushing your work to a remote server on GitLab.

## Objective: Core Git Competency

By the end of this tutorial, you will be proficient in the essential Git workflow. You will understand and be able to use the following commands and concepts:

* **Configuration:** git config
* **Repository Initialization:** git init
* **Status Checking:** git status
* **Staging Files:** git add
* **Committing Changes:** git commit
* **Pushing to a Remote:** git push
* **Pulling from a Remote:** git pull
* **Ignoring Files:** .gitignore
* **Branching and Merging:** git branch, git checkout, git merge
* **Conflict Resolution:** Manually and with merge tools

## Step 1: Initial Git Setup and Configuration

Before you can start versioning your projects, you need to introduce yourself to Git. This configuration step associates your name and email with every commit you make, which is crucial for tracking contributions in collaborative environments.

### 1.1. Verify Git Installation

First, ensure that Git is correctly installed on your machine.

* **Action:** Open the Git Bash terminal.
* **Command:**  
  git --version
* **Explanation:** This command asks Git to report its installed version.
* **Expected Output:** You should see a line of text indicating the version number, for example: git version 2.37.2.windows.1. If you see this, Git is installed correctly. If you get an error like "command not found," you must install the Git client before proceeding.

### 1.2. Create a GitLab Account and Project

A remote repository acts as a central storage location for your project, accessible from any computer. We will use GitLab for this lab.

* **Action:**

1. Go to [GitLab.com](https://gitlab.com) and sign up for a new, free account. **Note:** As per the instructions, use a personal email, not a corporate one.
2. After logging in, create a new project. Click the "New project" button.
3. Choose "Create blank project."
4. Name your project **GitDemo**.
5. You can leave the other settings as default for now (e.g., keep it "Private").
6. Click "Create project." You will be taken to your new, empty project page.

### 1.3. Configure Your Git Identity

Now, let's configure your local Git installation with the same identity you used for GitLab.

* **Action:** In your Git Bash terminal, execute the following commands one by one, replacing the placeholder text with your actual name and email.
* **Commands:**  
  git config --global user.name "Your Name"  
  git config --global user.email "youremail@example.com"
* **Explanation:**
* git config: This is the command for setting Git configuration variables.
* --global: This flag tells Git to apply this configuration to every repository on your system. You can also set configurations on a per-repository basis by omitting this flag.
* user.name and user.email: These are the specific variables we are setting. They are used to label your commits.

### 1.4. Verify Your Configuration

Let's double-check that the configuration was set correctly.

* **Action:** Run the following command in Git Bash.
* **Command:**  
  git config --list
* **Explanation:** This command displays all the global and local configuration settings.
* **Expected Output:** You will see a list of all configurations. Scroll through it and confirm that user.name and user.email are set to the values you provided in the previous step.

## Step 2: Integrating a Default Text Editor (Notepad++)

When you perform certain Git commands, like git commit, Git will open a text editor to allow you to type in a commit message. By default, this might be a command-line editor like Vim, which can be confusing for new users. This step shows you how to set a more user-friendly editor, like Notepad++, as the default.

### 2.1. Make Notepad++ Accessible from Git Bash

Git Bash needs to know where to find the Notepad++ program. This is done by adding its location to the system's PATH variable.

* **Action:**

1. Find the installation directory for Notepad++. It's usually C:\Program Files\Notepad++.
2. Go to Control Panel -> System -> Advanced System Settings.
3. Click on the "Environment Variables..." button.
4. In the "User variables" section, find the Path variable and click "Edit...".
5. Click "New" and paste the path to your Notepad++ directory (e.g., C:\Program Files\Notepad++).
6. Click OK on all the windows to save the changes.

* **Verification:** Close your Git Bash terminal and open a new one. This is important, as terminals only load the PATH variable on startup. Type notepad++ and press Enter. Notepad++ should launch.

### 2.2. Configure Git to Use Notepad++

Now, explicitly tell Git to use Notepad++ as its core editor.

* **Action:** Execute the following command in Git Bash.
* **Command:**  
  git config --global core.editor "notepad++ -multiInst -notabbar -nosession -noPlugin"
* **Explanation:**
* core.editor: This is the configuration variable for the default text editor.
* "notepad++ ...": We are setting the value to be the command to run Notepad++. The flags (-multiInst, -notabbar, etc.) are recommended settings that ensure Git can open and close the editor smoothly for operations like writing commit messages.

### 2.3. Verify the Editor Configuration

Let's check that Git now recognizes Notepad++ as its editor.

* **Action:** Run the configuration check command again, but this time specifically for the editor.
* **Command:**  
  git config --global -e
* **Explanation:** The -e flag (short for --edit) opens the global configuration file in the default editor you just set.
* **Expected Outcome:** Your global .gitconfig file should open in Notepad++. This confirms that the integration was successful. The file will contain the [user] and [core] sections you just configured. You can simply close Notepad++ without making any changes.

## Step 3: Creating and Tracking Your First File

This is the core of the Git workflow: creating a local repository, adding a file, and committing it to the project's history.

### 3.1. Initialize a Local Repository

First, we need a directory on our computer for the project. Then, we'll turn that directory into a Git repository.

* **Action:** In Git Bash, create a new folder for your project and navigate into it.
* **Commands:**  
  mkdir GitDemo  
  cd GitDemo
* **Explanation:**
* mkdir GitDemo: Creates a new directory named GitDemo.
* cd GitDemo: Changes the current location of the terminal into the GitDemo directory.
* **Action:** Now, initialize the repository.
* **Command:**  
  git init
* **Explanation:** This command creates a new, hidden .git subdirectory within your GitDemo folder. This .git folder contains all the necessary files and metadata for Git to track history, manage branches, and store your commits. Your GitDemo folder is now a Git "working directory."

### 3.2. Create and Add Content to a File

Let's create a simple text file to work with.

* **Action:** Create a file named welcome.txt with some content.
* **Command:**  
  echo "Hello, welcome to our Git project!" > welcome.txt
* **Explanation:**
* echo: A standard command-line utility that prints text.
* >: A redirection operator. It takes the output of the echo command and writes it into the file welcome.txt, creating the file if it doesn't exist.

### 3.3. Check the Status (The Untracked File)

The git status command is your most-used Git command. It tells you the current state of your repository.

* **Action:** Check the status of your new repository.
* **Command:**  
  git status
* **Expected Output:**  
  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)
* **Explanation:** Git sees that welcome.txt exists in the working directory, but it is "untracked." This means the file is not part of Git's version history yet. Git is waiting for you to decide if this file should be included in the next project snapshot (commit).

### 3.4. Stage the File

Before you commit a file, you must first add it to the "staging area." The staging area is an intermediate step where you gather all the changes you want to include in your next commit.

* **Action:** Add welcome.txt to the staging area.
* **Command:**  
  git add welcome.txt
* **Explanation:** This command takes the current version of welcome.txt and places it in the staging area, marking it for inclusion in the next commit.
* **Action:** Check the status again.
* **Command:**  
  git status
* **Expected Output:**  
  On branch master  
    
  No commits yet  
    
  Changes to be committed:  
    (use "git rm --cached <file>..." to unstage)  
          new file:   welcome.txt
* **Explanation:** The message has changed. Git now shows welcome.txt under "Changes to be committed." The file is now staged.

### 3.5. Commit the Staged File

A "commit" is a snapshot of your staged changes at a specific point in time. Each commit has a unique ID and a descriptive message.

* **Action:** Commit the staged file with a message.
* **Command:**  
  git commit -m "Add initial welcome file"
* **Explanation:**
* git commit: The command to create a new commit.
* -m "...": A flag that allows you to provide a short commit message directly on the command line. If you run git commit without the -m flag, Git will open the default editor (Notepad++) for you to write a more detailed, multi-line message.
* **Expected Output:** You will see a message confirming the commit, showing the branch, the commit hash (a unique ID), and a summary of the changes.
* **Action:** Check the status one last time.
* **Command:**  
  git status
* **Expected Output:**  
  On branch master  
  nothing to commit, working tree clean
* **Explanation:** This message confirms that your working directory is clean. All changes have been successfully saved (committed) into the local repository's history.

## Step 4: Connecting Local and Remote Repositories

Your project now exists on your local machine, but no one else can see it. The final step is to connect your local repository to the remote one you created on GitLab and "push" your commits there.

### 4.1. Link the Remote

You need to tell your local repository the URL of the remote one on GitLab.

* **Action:** Go to your GitDemo project page on GitLab. Click the "Clone" button and copy the "Clone with HTTPS" URL. It will look something like https://gitlab.com/your-username/gitdemo.git.
* **Command:** In Git Bash, run the following command, pasting the URL you just copied. We give the remote a nickname, traditionally origin.  
  git remote add origin https://gitlab.com/your-username/gitdemo.git
* **Explanation:**
* git remote add: The command to add a new remote connection.
* origin: The standard nickname for the primary remote repository.
* https://...: The URL of the remote.

### 4.2. Push Your Local Commits to the Remote

"Pushing" is the act of uploading your local commits to the remote repository.

* **Action:** Push your master branch to origin.
* **Command:**  
  git push -u origin master
* **Explanation:**
* git push: The command to upload commits.
* origin: The destination remote (the nickname we just set).
* master: The local branch you are pushing.
* -u: (short for --set-upstream) This is a one-time flag that creates a link between your local master branch and the master branch on origin. In the future, you can simply run git push from this branch without specifying the remote and branch again.
* **Action:** You may be prompted for your GitLab username and password. Enter them to authorize the push.
* **Expected Outcome:** After the push completes, go back to your GitLab project page in your web browser and refresh it. You will now see your welcome.txt file and the commit message "Add initial welcome file" listed in the project.

## Step 5: Ignoring Files with .gitignore

As your project grows, you will inevitably have files and folders that you do not want to include in your Git repository. These are often files that are generated automatically, contain sensitive information, or are specific to your local machine. Git provides a simple and powerful mechanism to handle this: the .gitignore file.

### 5.1. Why Ignoring Files is Essential

Committing every file to your repository is a bad practice that can lead to numerous problems:

* **Repository Bloat:** Committing large, generated files (like compiled code or software packages) makes your repository unnecessarily large, slowing down cloning and fetching operations for everyone on the team.
* **Merge Conflicts:** Auto-generated files, like logs, change constantly. If two developers commit their own versions of a log file, it will create a pointless merge conflict that needs to be resolved.
* **Security Risks:** The most critical reason to ignore files is to prevent sensitive data—such as API keys, database passwords, or private configuration files (e.g., .env)—from ever being committed. Once something is in the commit history, it is difficult to remove completely and can be exposed if the repository is public or compromised.
* **Clutter:** Your git status output should be clean, showing you only the relevant source code changes you've made. If it's cluttered with dozens of untracked system files (.DS\_Store, Thumbs.db) or build artifacts, it becomes hard to see what's important.

The .gitignore file solves all these problems by telling Git, "Do not track these files or directories, and don't even show them to me as 'untracked'."

### 5.2. Hands-On: Implementing .gitignore

Let's walk through the lab exercise. We will create a log file and a log directory, see that Git tracks them, and then use .gitignore to make Git ignore them permanently.

**Prerequisite:** Ensure you are still inside your GitDemo project directory in the Git Bash terminal.

#### 1. Create Files to be Ignored

First, let's create the files and folders that we want to ignore.

* **Action:** Execute the following commands in Git Bash.
* **Commands:**  
  # Create a file with a .log extension  
  echo "Application started successfully at $(date)" > app.log  
    
  # Create a directory named 'logs'  
  mkdir logs  
    
  # Create a file inside the 'logs' directory  
  echo "User session 12345 logged out." > logs/session.log
* **Explanation:** We have now created app.log in our root directory and a logs folder which contains session.log. These are typical examples of files you would want to ignore.

#### 2. Check the Status Before Ignoring

Now, let's ask Git what it thinks about our new files.

* **Action:** Run git status.
* **Command:**  
  git status
* **Expected Output:**  
  On branch master  
  Your branch is up to date with 'origin/master'.  
    
  Untracked files:  
    (use "git add <file>..." to include in what will be committed)  
          app.log  
          logs/  
    
  nothing added to commit but untracked files present (use "git add" to track)
* **Explanation:** As expected, Git sees app.log and the logs/ directory as new, "untracked" items. It's waiting for us to either git add them or tell it to ignore them.

#### 3. Create and Configure the .gitignore File

Now for the main event. We will create a special file named .gitignore and add patterns to it.

* **Action:** Create the .gitignore file and open it in your default editor (Notepad++).
* **Command:**  
  notepad++ .gitignore
* **Action:** Inside Notepad++, add the following lines to the file. Then save the file and close the editor.  
  # Ignore all files that end with the .log extension  
  \*.log  
    
  # Ignore the entire 'logs' directory and all its contents  
  logs/
* **Explanation of .gitignore Patterns:**
* #: Lines beginning with a hash are comments, which are ignored by Git but are useful for explaining your rules.
* \*.log: The asterisk \* is a wildcard that matches any sequence of characters. This pattern tells Git to ignore any file, in any directory, that ends with .log. This will match app.log and logs/session.log.
* logs/: Adding a name followed by a forward slash / tells Git to ignore a directory and everything inside it, recursively.

#### 4. Verify the Result

With our .gitignore file saved, let's check the status again.

* **Action:** Run git status.
* **Command:**  
  git status
* **Expected Output:**  
  On branch master  
  Your branch is up to date with 'origin/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)
* **Analysis:** This is a perfect result! Notice two things:

1. The app.log file and the logs/ directory have completely disappeared from the output. Git is now successfully ignoring them.
2. The .gitignore file itself is now listed as an "untracked file." This is correct—we have created a new file that should be part of our project's history.

#### 5. Commit the .gitignore File

The final step is to add the .gitignore file to the repository. This ensures that these ignore rules are shared with every other developer who clones the project, providing a consistent environment for everyone.

* **Action:** Stage and commit the .gitignore file.
* **Commands:**  
  # Stage the .gitignore file  
  git add .gitignore  
    
  # Commit the staged file with a clear message  
  git commit -m "Add .gitignore to exclude log files and directories"
* **Action:** Push this change to your remote repository on GitLab.
* **Command:**  
  git push

You have now successfully implemented .gitignore to keep your repository clean and professional. If you refresh your GitLab project page, you will see the .gitignore file has been added to your repository.

## Step 6: Branching and Merging - The Core of Collaboration

Branching is arguably the most important feature of Git. It allows you to diverge from the main line of development and work on new features, bug fixes, or experiments in an isolated environment without affecting the stable master branch. Once your work is complete and tested, you can "merge" it back into master.

### 6.1. Why Branch?

Imagine a team working on a single master branch. If one developer is working on a complex, half-finished feature and commits it, the main project could be broken for everyone else. Branching solves this by providing a safe, independent workspace.

* **Isolation:** Changes in your branch don't affect others until you are ready to merge.
* **Parallel Development:** Multiple developers can work on different features simultaneously, each in their own branch.
* **Organization:** It creates a clean, linear history. The master branch only contains stable, completed work, while feature development happens in separate, temporary branches.

### 6.2. Hands-On: The Branch and Merge Workflow

This lab will walk you through creating a branch, making changes, and then merging those changes back into the master branch.

**Prerequisite:** Ensure you are in your GitDemo directory in Git Bash. The master branch should be clean. You can verify this with git status.

#### 1. Create a New Branch

* **Action:** Create a new branch called GitNewBranch.
* **Command:**  
  git branch GitNewBranch
* **Explanation:** This command creates a new pointer (a branch) to the exact same commit you are currently on. It does *not* switch you to the new branch.

#### 2. List and Verify Branches

* **Action:** List all the branches in your local repository.
* **Command:**  
  git branch
* **Expected Output:**  
    GitNewBranch  
  \* master
* **Explanation:** The command lists all local branches. The asterisk \* indicates the branch you are currently on, which is still master.

#### 3. Switch to the New Branch

* **Action:** Switch your working directory to the GitNewBranch.
* **Command:**  
  git checkout GitNewBranch
* **Explanation:** The checkout command updates the files in your working directory to match the state of the branch you are checking out and moves the HEAD pointer to that branch.
* **Expected Output:** Switched to branch 'GitNewBranch'
* **Note:** You can create and switch to a new branch in a single step with git checkout -b GitNewBranch.

#### 4. Make Changes in the New Branch

Now that we are safely on our new branch, let's make some changes.

* **Action:** Create a new file called feature.txt.
* **Command:**  
  echo "This is a new feature." > feature.txt
* **Action:** Stage and commit this new file *to the current branch*.
* **Commands:**  
  git add feature.txt  
  git commit -m "Implement new feature"
* **Explanation:** This commit now exists *only* on GitNewBranch. The master branch is unaware of feature.txt.
* **Action:** Check the status to confirm everything is clean on this branch.
* **Command:** git status
* **Expected Output:** On branch GitNewBranch\nnothing to commit, working tree clean

#### 5. Switch Back to Master and See the Difference

* **Action:** Switch back to the master branch.
* **Command:**  
  git checkout master
* **Expected Output:** Switched to branch 'master'
* **Observation:** Look at the files in your GitDemo directory. The feature.txt file has vanished! This is because it doesn't exist on the master branch. This demonstrates the powerful isolation that branches provide.

#### 6. Compare the Branches (Diff)

Before merging, it's good practice to see what changes the merge will introduce.

* **Action:** View the differences between master and GitNewBranch.
* **Command:**  
  git diff GitNewBranch
* **Explanation:** git diff <branch-name> shows the changes between your current branch and the specified branch.
* **Expected Output:** You will see a "diff" output indicating that feature.txt is a new file and showing its content.
* **(Optional) Visual Diff with P4Merge:** For more complex changes, a visual tool is invaluable.
* **Prerequisite:** You must have P4Merge installed and configured. To configure it:  
  git config --global diff.tool p4merge  
  git config --global difftool.p4merge.path 'C:/Program Files/Perforce/p4merge.exe' #<-- Adjust this path  
  git config --global difftool.prompt false
* **Action:** Launch the visual diff tool.
* **Command:** git difftool GitNewBranch
* **Expected Outcome:** P4Merge will launch, showing a side-by-side comparison of the master and GitNewBranch states.

#### 7. Merge the Branch

Now, let's bring the work from our feature branch into master.

* **Action:** Merge GitNewBranch into your current branch (master).
* **Command:**  
  git merge GitNewBranch
* **Explanation:** This command takes the changes from GitNewBranch and integrates them into master. In this case, since master hasn't changed since we branched, Git performs a simple "fast-forward" merge by just moving the master pointer up to the same commit as GitNewBranch.
* **Expected Output:** You will see a message about the merge, indicating a "fast-forward" and listing the files that were changed. Now, if you check your files, feature.txt will be present in the master branch.

#### 8. View the History

* **Action:** View the commit history as a graph.
* **Command:**  
  git log --oneline --graph --decorate
* **Explanation:** This is a powerful log command that shows a compact, graphical view of your commit history, making it easy to see branches and merges. You will see that master, GitNewBranch, and HEAD are all pointing to the same latest commit.

#### 9. Clean Up: Delete the Branch

Once a feature branch has been successfully merged, it's common practice to delete it to keep the repository tidy.

* **Action:** Delete the GitNewBranch.
* **Command:**  
  git branch -d GitNewBranch
* **Explanation:** The -d (or --delete) flag deletes the specified branch. Git will prevent you from deleting a branch that has unmerged changes, which is a safety feature.
* **Expected Output:** Deleted branch GitNewBranch (was <commit-hash>).
* **Verification:** Run git branch again, and you will see that only \* master remains.

### 6.3. Local Merge vs. GitLab Merge Request

What we just did was a *local merge*. In a professional team environment, you typically don't merge directly into master on your own machine. Instead, the workflow is:

1. Push your feature branch to the remote repository: git push origin GitNewBranch
2. Go to the GitLab UI.
3. Click the "Create merge request" button.
4. Fill out the title and description, assign reviewers, and submit.
5. This creates a **Merge Request (MR)**, which is a platform for discussing the changes, running automated tests, and getting code review approvals.
6. Once the MR is approved, a project maintainer will click the "Merge" button on GitLab, which performs the merge on the remote repository. This provides a crucial layer of quality control and collaboration.

## Step 7: Handling Merge Conflicts

So far, we have performed a "fast-forward" merge, where the master branch hadn't changed. But what happens when you try to merge two branches that have both changed the *same part of the same file*? This is a **merge conflict**, and Git will stop and ask you, the developer, to resolve it.

### 7.1. Why Conflicts Happen

A merge conflict occurs when Git cannot make an automatic decision about how to combine changes. This happens when two separate branches, starting from the same commit, make conflicting edits. For example:

* **Branch A:** Changes line 5 of config.txt.
* **Branch B:** Also changes line 5 of config.txt.

When you try to merge Branch B into Branch A, Git doesn't know which version of line 5 is correct. It cannot read your mind, so it pauses the merge and hands control over to you.

### 7.2. Hands-On: Creating and Resolving a Conflict

This lab will intentionally create a conflict to teach you how to resolve it.

**Prerequisite:** You should be in your GitDemo directory, on a clean master branch.

#### 1. Create a New Branch and Make a Change

* **Action:** Create a new branch called GitWork and switch to it.
* **Command:**  
  git checkout -b GitWork
* **Action:** Create a new file named hello.xml with some content.
* **Command:**  
  echo "<message>Hello from the GitWork branch!</message>" > hello.xml
* **Action:** Stage and commit this change to the GitWork branch.
* **Commands:**  
  git add hello.xml  
  git commit -m "Add hello.xml from GitWork"

#### 2. Switch to Master and Create a Conflicting Change

Now, we will go back to master and make a change that will conflict with the work we just did.

* **Action:** Switch back to the master branch.
* **Command:**  
  git checkout master
* **Action:** Create the *same file*, hello.xml, but with *different* content.
* **Command:**  
  echo "<message>Greetings from the master branch!</message>" > hello.xml
* **Action:** Stage and commit this change to the master branch.
* **Commands:**  
  git add hello.xml  
  git commit -m "Add hello.xml from master"

#### 3. View the Diverged History

At this point, our commit history has diverged. The two branches have different, competing versions of hello.xml.

* **Action:** View the graph log to see the split.
* **Command:**  
  git log --oneline --graph --decorate --all
* **Expected Output:** You will see a graph showing the master and GitWork branches have split off from a common ancestor commit.

#### 4. Attempt the Merge and Trigger the Conflict

This is the moment of truth. Let's try to merge GitWork into master.

* **Action:** Attempt the merge.
* **Command:**  
  git merge GitWork
* **Expected Output:**  
  Auto-merging hello.xml  
  CONFLICT (content): Merge conflict in hello.xml  
  Automatic merge failed; fix conflicts and then commit the result.
* **Explanation:** Git failed. It tells you exactly which file (hello.xml) has a conflict. The merge process is now paused, and your repository is in a "merging" state.

#### 5. Inspect and Resolve the Conflict

Git has modified the conflicted file to show you both versions of the change.

* **Action:** Open hello.xml in a text editor.
* **Command:** notepad++ hello.xml
* **Contents of the file:**  
  <<<<<<< HEAD  
  <message>Greetings from the master branch!</message>  
  =======  
  <message>Hello from the GitWork branch!</message>  
  >>>>>>> GitWork
* **Explanation of Conflict Markers:**
* <<<<<<< HEAD: Everything between this marker and ======= is the version from your current branch (master).
* =======: This divides the two conflicting versions.
* >>>>>>> GitWork: Everything between ======= and this marker is the version from the branch you are trying to merge (GitWork).
* **Action:** Resolve the conflict by manually editing the file. You must delete the conflict markers and decide what the final content should be. For this lab, let's combine them. Edit the file to look like this:  
  <message>Greetings from the master branch!</message>  
  <message>Hello from the GitWork branch!</message>
* **Save the file** after you have resolved the conflict.

#### 6. (Optional) Using a Visual Merge Tool

For complex conflicts, a visual tool is much easier.

* **Action:** Launch the configured merge tool.
* **Command:** git mergetool
* **Expected Outcome:** P4Merge (or your configured tool) will open in a "3-way merge" view, showing LOCAL (your master version), BASE (the original version before any changes), and REMOTE (the GitWork version). You can click to select which lines to keep and save the final merged result. The tool will often create a backup file (e.g., hello.xml.orig).

#### 7. Finalize the Merge

Once you have manually saved the resolved file, you must tell Git that the conflict is handled.

* **Action:** Stage the resolved file.
* **Command:**  
  git add hello.xml
* **Explanation:** This git add command tells Git, "I have fixed the conflict in this file; please accept my version as the final one."
* **Action:** Commit the merge. Now that all conflicts are resolved, you can create the "merge commit."
* **Command:**  
  git commit
* **Explanation:** Running git commit while in a merging state will open your text editor with a pre-populated merge commit message. You can keep it as is or edit it. Save and close the editor to finalize the commit.

#### 8. Clean Up

* **Action:** Your merge tool may have created a backup file ending in .orig. We don't want to track this. Add it to your .gitignore file.
* **Commands:**  
  echo "\*.orig" >> .gitignore  
  git add .gitignore  
  git commit -m "Ignore merge tool backup files"
* **Action:** Now that the GitWork branch is successfully merged, we can delete it.
* **Command:**  
  git branch -d GitWork

#### 9. View the Final History

* **Action:** Look at the log one last time.
* **Command:**  
  git log --oneline --graph --decorate
* **Expected Outcome:** You will now see a "merge commit" in your history with two parent commits, visually showing where the two divergent branches were successfully joined back together.

Congratulations! You have successfully navigated and resolved a merge conflict, one of the most important skills for any Git user.