Hands-On Lab (HOL): Git Merge Conflict – Manual Conflict Resolution (Using Feature Branches)

Author

Dr. Sandeep Kumar Sharma

Learning Objective

The objective of this Hands-On Lab is to help learners **create, understand, and manually resolve Git merge conflicts using two feature branches**, exactly as it happens in real-world projects. This lab is a natural continuation of **HOL-05 (Git Branching & Merging)** and reuses the same branching mindset.

Learning Outcome

After completing this lab, learners will be able to:

- Understand why merge conflicts occur during feature branch integration
- Create merge conflicts intentionally for learning purposes
- Identify conflict markers inside files
- Manually resolve conflicts step by step
- Successfully complete a merge after conflict resolution
- Confidently handle merge conflicts in real projects and interviews

---

Pre-requisite Concept (Very Important)

A **merge conflict occurs when**: - Two branches modify the **same file** - On the **same line(s)** - And Git cannot automatically decide which change to keep

Git will **pause the merge** and ask the developer to resolve it manually.

Lab Context (Reference from HOL-05)

In **HOL-05**, we worked with: - `master` branch - `feature-1` branch - `feature-2` branch

In this lab: - Both **feature-1** and **feature-2** will modify the **same file** - This will intentionally create a **merge conflict** when merging into `master`

Initial Lab Assumption

- Git repository already initialized
- `master`, `feature-1`, and `feature-2` branches already exist
- Working tree is clean

Step-by-Step Hands-On Lab: Git Merge Conflict (Feature Branch Based)

STEP 1: Create a Common File on Master Branch

```
git switch master
vi conflict.txt
```

Add the following content:

```
This line is from master branch
```

Save and exit (`Esc :wq`)

```
git add conflict.txt
git commit -m "base commit for merge conflict" conflict.txt
```

STEP 2: Modify the Same File in feature-1 Branch

```
git switch feature-1
vi conflict.txt
```

Modify the same line:

```
This line is modified in feature-1 branch
```

Save and exit

```
git add conflict.txt
git commit -m "feature-1 modifies conflict.txt" conflict.txt
```

STEP 3: Modify the Same File in feature-2 Branch

```
git switch feature-2
vi conflict.txt
```

Modify the same line differently:

```
This line is modified in feature-2 branch
```

Save and exit

```
git add conflict.txt
git commit -m "feature-2 modifies conflict.txt" conflict.txt
```

STEP 4: Merge feature-1 into Master (No Conflict Yet)

```
git switch master
git merge feature-1
```

Explanation: - Merge succeeds - `master` now contains feature-1 changes

---

STEP 5: Merge feature-2 into Master (Conflict Occurs)

```
git merge feature-2
```

Result: - Git detects conflicting changes - Merge process pauses

```
CONFLICT (content): Merge conflict in conflict.txt
```

STEP 6: Check Status During Conflict

```
git status
```

Explanation: - File shows as **both modified** - Git is waiting for manual resolution

---

STEP 7: Open the Conflicted File

```
vi conflict.txt
```

You will see conflict markers:

```
<<<<<<< HEAD
This line is modified in feature-1 branch
=======
This line is modified in feature-2 branch
>>>>>>> feature-2
```

STEP 8: Understand Conflict Markers

- `<<<<<<< HEAD` → Current branch (`master`, which already includes feature-1)
- `=======` → Separator
- `>>>>>>> feature-2` → Incoming branch changes

Git is asking:

> "Which change should I keep?"

STEP 9: Manually Resolve the Conflict

Edit the file and resolve the conflict (example resolution):

```
This line is resolved after merging feature-1 and feature-2
```

Remove **all conflict markers**, save and exit.

STEP 10: Mark Conflict as Resolved

```
git add conflict.txt
```

Internal Meaning: - Git understands the conflict is resolved

STEP 11: Complete the Merge

```
git commit -m "merge feature-2 after resolving conflict"
```

STEP 12: Verify Final State

```
git log --oneline
ls
```

Explanation: - Merge commit created - Conflict successfully resolved - Both feature branch changes are integrated

---

Visual Commit Flow (Conceptual)

```
            feature-1
               |
A --- B ------- C (master)
 \            /
  D ----------   (feature-2)
```

Common Interview Questions

Q1: Why did conflict happen only with feature-2 merge?
➡️Because feature-1 was already merged and modified the same line

Q2: Does Git lose code during conflicts?
➡️No, Git preserves both versions

Q3: How do you tell Git that conflict is resolved?
➡️By running `git add` on the resolved file

Conclusion (Trainer Note)

Merge conflicts are **normal in collaborative development**. Git stops and asks the developer to decide rather than guessing. Once learners understand **branch context + conflict markers**, resolving conflicts becomes a controlled and logical process instead of a panic situation.

✅**End of Hands-On Lab – Git Merge Conflict (Feature Branch Based)**