DevOps Shack Git Assignment | Task:8

Task 8: Simulating and Resolving Merge Conflicts

8.1 Introduction to Merge Conflicts

In Git, a **merge conflict** happens when **two branches modify the same part of the same file** and Git **can't automatically decide which change to keep**.

When merging, Git tries to **combine histories**:

- If **no overlapping changes**, Git **auto-merges**.
- If **overlapping changes exist**, Git stops and asks **you to resolve**.

Analogy:
Think of **merge conflicts** like **two people editing the same sentence in a document**—someone needs to **decide how to combine their edits**.

8.2 Why Merge Conflicts Matter in Real-World Projects

In **corporate workflows**, multiple developers often work on:

- **Different branches** (feature, bugfix, release).
- **The same files**.

When these changes **overlap**, merge conflicts occur. Understanding and resolving them is **critical for collaboration**.

Typical Scenarios:

1. **Two developers modify the same function**.
2. **One developer deletes a file** that another modifies.
3. **Concurrent hotfixes** on **different branches**.

Example:
Developer A changes the API function signature.
Developer B changes the API logic.
Merging **both changes** triggers a conflict in the **same file and lines**.

8.3 How Git Detects Conflicts Internally

1. Git looks for a **common ancestor commit** (three-way merge).
2. Compares changes from:
    - **HEAD** (current branch).
    - **MERGE_HEAD** (branch being merged).
3. If both branches modify the **same lines**:
    - Git **can't resolve automatically**.
    - Marks those sections as **conflicted**.

Conflict Markers:

<<<<<<< HEAD

Changes from current branch (main)

=======

Changes from merging branch (feature-frontend)

>>>>>>> feature-frontend

- **HEAD**: Your branch.
- **MERGE_HEAD**: Incoming branch.

8.4 Step-by-Step Implementation: Simulating Merge Conflicts

Scenario Setup:

- Two branches:
    - feature-frontend.
    - feature-backend.

Both **modify the same line** in README.md.

Step 1: Prepare the Main Branch

```
git checkout main
echo "Welcome to DevOps Shack Project" > README.md
git add README.md
git commit -m "Initial README"
```

Step 2: Create Two Feature Branches

git checkout -b feature-frontend

On **feature-frontend**:

echo "Frontend logic implemented." >> README.md

git commit -am "Add frontend section"

Switch back:

git checkout main

git checkout -b feature-backend

On **feature-backend**:

echo "Backend logic implemented." >> README.md

git commit -am "Add backend section"

Step 3: Merge One Branch

Merge **feature-frontend** into **main**:

git checkout main

git merge feature-frontend

Step 4: Trigger the Conflict

Now, merge **feature-backend**:

git merge feature-backend

Conflict occurs in README.md:

Auto-merging README.md

CONFLICT (content): Merge conflict in README.md

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

8.5 Resolving Merge Conflicts (Step-by-Step)

Step 1: Identify the Conflict

git status

- Shows **README.md** as **unmerged**.

Step 2: Open the Conflicted File

<<<<<<< HEAD

Frontend logic implemented.

=======

Backend logic implemented.

>>>>>>> feature-backend

- Git marks the **conflicted lines**.

Step 3: Resolve Manually

Choose:

- **Keep both changes**:

Frontend logic implemented.

Backend logic implemented.

Or:

- **Favor one side**.

Step 4: Mark Conflict as Resolved

git add README.md

Step 5: Finalize the Merge

git commit -m "Merge feature-backend into main with conflict resolution"

8.6 Visualizing Conflict History

Before Conflict:

main: A---B (Add frontend)

\

feature-backend C (Add backend)

During Conflict:

- Merge blocked by **README.md conflict**.

After Resolution:

main: A---B---D (merged with conflict resolved)

8.7 Best Practices for Resolving Conflicts

1. **Communicate early**:
    - If working on **shared files**, inform teammates.
2. **Merge frequently**:
    - Avoid **large divergence**.
3. **Use merge tools**:
    - GUI tools:
        - **VSCode**, **Sourcetree**, **KDiff3**, **Meld**.
    - Git's built-in:

git mergetool

4. **Understand the context**:
    - Don't blindly resolve—**understand both sides**.

8.8 Advanced Conflict Management: Git Rerere

What is Git Rerere?

- **Rerere** = **Reuse Recorded Resolution**.
- Automatically remembers how you resolved a conflict.

Enable Rerere:

git config --global rerere.enabled true

- Git records conflict resolutions.
- When the **same conflict happens again**, Git **auto-applies your last resolution**.

Rerere Workflow:

1. Conflict occurs → resolve → commit.
2. Same conflict happens → Git suggests **auto-resolution**.

Why useful?

- Speeds up **repetitive conflict resolution** in **repeated merges or rebases**.

8.9 Common Mistakes & Pitfalls

Mistake	How to Avoid
Blindly choosing one side	Understand both changes and **merge meaningfully**.
Ignoring conflict markers	Never commit files with <<<<<<<, =======, >>>>>>>.
Overwriting uncommitted changes	Always **stash or commit changes** before merging.

8.10 Merge Conflicts in CI/CD Pipelines

1. **Avoid conflicts in pipelines**:
    - Pipelines **shouldn't merge branches** automatically.
2. **Pre-merge checks**:
    - Use **protected branches** to **require merges via pull requests**.

3. **Conflict detection**:
    - Use **status checks** to block merges until conflicts are resolved.