Hands-On Lab (HOL): Git Branching, Switching, and Merging (with Internal Mapping)

Author

Dr. Sandeep Kumar Sharma

Learning Objective

The objective of this Hands-On Lab is to help learners understand **Git branching from both a command and internal architecture perspective**. This lab explains how branches work, how `HEAD` moves, how parallel development happens, and how feature branches are finally merged into the main branch.

Learning Outcome

After completing this lab, learners will be able to:

- Explain what a Git branch actually is (pointer concept)
- Create and manage multiple branches
- Switch between branches using `git switch`
- Perform parallel development safely
- Merge feature branches into the main branch
- Read and explain `git log` output across branches
- Answer interview questions related to branching and merging

---

Pre-requisite Concept (Very Important)

Before starting, remember:

- A **branch is NOT a copy of code**
- A branch is simply a **pointer to a commit**
- `HEAD` points to the current branch

```
HEAD → master → Commit-A → Commit-B
```

When a branch moves, only the **pointer moves**, not the entire codebase.

Initial Lab Assumption

- Git repository already initialized
- At least one commit exists on `master`
- Working tree is clean

Step-by-Step Hands-On Lab: Git Branching

Step 1: List Existing Branches

```
git branch
```

Explanation:

- Displays all local branches
- `*` indicates the current branch

Step 2: Check Repository Status

```
git status
```

Explanation:

- Confirms clean working tree
- Shows current branch name

Step 3: View Current Files

```
ls
```

Step 4: View Commit History

```
git log
```

Internal Mapping:

- Shows commit history of the **current branch only**

Creating Feature Branches

Step 5: Create First Feature Branch

```
git branch feature-1
```

Internal Mapping:

- New pointer `feature-1` created
- Points to the same commit as `master`

Step 6: Verify Branch Creation

```
git branch
```

Step 7: Create Second Feature Branch

```
git branch feature-2
```

Step 8: Verify All Branches

```
git branch
```

Working on Feature-1 Branch

Step 9: Switch to Feature-1 Branch

```
git switch feature-1
```

Internal Mapping:

- `HEAD` now points to `feature-1`
- Files update to match branch snapshot

Step 10: Verify Current Branch

```
git branch
```

Step 11: Create Feature-1 File

```
vi feature-1-file
```

- Add content
- Save and exit

Step 12: Verify File Creation

```
ls
```

Step 13: Stage and Commit Feature-1 Work

```
git add .
git commit -m "this is feature-1 commit" feature-1-file
```

Internal Mapping:

- New commit created
- `feature-1` pointer moves forward
- `master` remains unchanged

Working on Feature-2 Branch

Step 14: Switch to Feature-2 Branch

```
git switch feature-2
```

Step 15: Verify Branch

```
git branch
```

Step 16: Verify Files (Isolation Proof)

```
ls
```

Explanation:

- `feature-1-file` is NOT visible
- Confirms branch isolation

Step 17: Create Feature-2 File

```
vi feature-2-file
```

Step 18: Stage and Commit Feature-2 Work

```
git add .
git commit -m "this is feature-2 commit" feature-2-file
```

Internal Mapping:

- `feature-2` pointer moves forward
- Work remains isolated

Merging Feature Branches into Master

Step 19: Switch Back to Master Branch

```
git switch master
```

Step 20: Verify Files on Master

```
ls
```

Step 21: View Master Branch History

```
git log
```

Step 22: Merge Feature-1 into Master

```
git merge feature-1
```

Internal Mapping:

- Fast-forward merge (if no divergence)
- `master` pointer moves to feature-1 commit

Step 23: Verify Files After Merge

```
ls
```

Step 24: View Updated History

```
git log
```

Step 25: Merge Feature-2 into Master

```
git merge feature-2
```

Step 26: Final Verification

```
ls
git log
```

Explanation:

- Both feature files exist
- History includes both feature commits

Common Interview Questions

Q1: Does branch create a copy of code?
➡️No, it creates a pointer

Q2: What moves during commit?
➡️Branch pointer

Q3: Why is branching cheap in Git?
➡️Because only pointers are created

Conclusion

Git branching allows **parallel development without fear**. Once learners understand that branches are just pointers and `HEAD` decides the active timeline, Git branching becomes simple, logical, and extremely powerful.
