DevOps Shack Git Assignment | Task:3

Task 3: Merging and Rebasing Workflows

3.1 Introduction to Merging and Rebasing

In **collaborative environments**, multiple developers work on **feature branches** independently. Eventually, these branches need to be **integrated** back into a **main branch**. Two powerful Git tools for this are:

- **Merge**: Combines histories while **preserving their separate timelines**.
- **Rebase**: **Rewrites history** to create a **linear sequence** of commits.

Both serve **different purposes** and understanding **when and how to use them** is essential for maintaining a **clean, understandable Git history**.

3.2 Why Merging and Rebasing Matter in Real-World Projects

Corporate Example:

At **DevOps Shack**, multiple teams work on different features simultaneously:

- **Team A** works on the **frontend**.
- **Team B** works on the **backend**.

Both teams work on **feature branches**:

- feature-frontend
- feature-backend

At some point:

- **Frontend** is ready to integrate directly into main.
- **Backend** wants to **reapply its commits** on top of the updated main (after frontend is merged) to keep history **linear**.

This is where **merge** and **rebase** come into play.

3.3 Conceptual Difference: Merge vs Rebase

Aspect	Merge	Rebase
What it does	Combines branches, preserving **branching history**	Reapplies commits on top of another branch, **rewriting history**
Commit History	Maintains **divergent branches**	Creates a **linear sequence of commits**
Use Case	**Team collaboration**—preserve context	**Solo feature development**—keep history clean
Merge Commit	Creates a **merge commit**	No merge commit; **rebases commits individually**
Conflict Resolution	Conflicts resolved **once at merge**	Conflicts may be resolved at **each commit during rebase**

Visualizing Merge:

```
      D---E feature-frontend
     /
A---B---C main
```

After merging feature-frontend into main:

```
      D---E feature-frontend
     /     \
A---B---C-----F main (merge commit)
```

Visualizing Rebase:

```
      D---E feature-backend
     /
A---B---C main
```

After rebasing feature-backend onto main:

A---B---C---D'---E' feature-backend

Commits D and E are **reapplied** as D' and E'.

3.4 When to Merge and When to Rebase (Real-World Scenarios)

When to Use Merge:

- **Multiple developers** working on a **shared branch**.

- You want to **preserve the context** of the branch:
    - **What branch the work was done on**.
    - **When the integration happened**.

When to Use Rebase:

- You're working on a **feature branch alone**.
- Before merging your feature, you want to **linearize history** so it looks like your work was done **after the latest changes in main**.

Golden Rule: Never rebase public/shared branches!
Only rebase **local feature branches** that **haven't been pushed** or **shared** yet.

3.5 Step-by-Step Implementation of Merging and Rebasing

Scenario Setup:

You've completed:

- feature-frontend (ready to merge into main).
- feature-backend (will rebase onto updated main).

Part 1: Merging feature-frontend into main

Step 1: Ensure main is up-to-date

- Pull the latest changes from remote:

```
git checkout main

git pull origin main
```

Step 2: Merge feature-frontend

```
git merge feature-frontend
```

- Creates a **merge commit**.

Step 3: Visualize Merge History

```
git log --oneline --graph --all
```

Expected output:

```
*   f1c2d34 (HEAD -> main) Merge branch 'feature-frontend'
|\
| * e4d9a10 (feature-frontend) Finalize frontend logic
| * d34d123 Initial frontend setup
* c3b2e98 Previous main commit
```

- Shows **branch divergence and convergence**.

Step 4: Push Changes

```
git push origin main
```

Part 2: Rebasing feature-backend onto Updated main

Step 1: Checkout feature-backend

```
git checkout feature-backend
```

Step 2: Rebase onto main

```
git rebase main
```

- Git **reapplies commits** from feature-backend **one by one** on top of the latest main.

Step 3: Resolve Conflicts (if any)

If conflicts occur during rebase:

- Git pauses at the conflicting commit.
- You resolve the conflict manually.
- Continue the rebase process:

```
git add <resolved-files>
```

```
git rebase --continue
```

Step 4: Visualize Rebased History

```
git log --oneline --graph --all
```

Expected output:

```
* 3f4a678 (HEAD -> feature-backend) Finalize backend logic
* 2d3b456 Add backend setup
* f1c2d34 (main) Merge branch 'feature-frontend'
* c3b2e98 Previous main commit
```

- Shows a **clean, linear sequence**.

Step 5: Push Rebased Branch (Force Required)

```
git push -f origin feature-backend
```

- **Force push** is necessary because **rebase rewrites commit history**.

3.6 Visualizing the Workflow (Merge vs Rebase)

Merge Example (Branch Context Preserved):

```
*   f1c2d34 (main) Merge branch 'feature-frontend'
|\
| * e4d9a10 (feature-frontend) Finalize frontend logic
| * d34d123 Initial frontend setup
* c3b2e98 Previous main commit
```

Rebase Example (Linear History):

```
* 3f4a678 (feature-backend) Finalize backend logic
* 2d3b456 Add backend setup
* f1c2d34 (main) Merge branch 'feature-frontend'
* c3b2e98 Previous main commit
```

3.7 Best Practices for Merge and Rebase

For Merging:

- **Always merge main into feature branches** periodically to **reduce conflicts**.
- **Write clear merge commit messages**.
- **Squash merge** if desired (GitHub/GitLab supports this).

For Rebasing:

- **Only rebase local, unshared branches**.
- **Avoid rebasing shared/public branches**.
- **Resolve conflicts patiently**—rebase pauses at **each commit**.

3.8 Pitfalls to Avoid

Mistake	**How to Avoid**
Rebasing a shared/public branch	Only rebase **private branches**.
Force pushing without warning	Communicate **before force pushes**.
Forgetting to resolve conflicts	Always **resolve conflicts** and **continue rebase**.