DevOps Shack Git Assignment | Task:7

Task 7: Squashing Commits for Clean History

7.1 Introduction to Git Commit Squashing

In **Git**, **squashing commits** means **combining multiple commits into one**. This process helps you **clean up your commit history**, especially before **merging feature branches** into **main**.

Imagine working on a feature and making several **incremental commits**:

- "Fix typo".
- "Update formatting".
- "Add backend logic".
- "Refactor backend logic".

If you merge this directly into **main**, these **small, fragmented commits** clutter the history.

Squashing allows you to **condense these commits** into **one logical unit**, making the history **easier to understand**.

7.2 Why Squashing Commits Matters in Real-World Projects

In **corporate DevOps workflows**, clean commit history is vital for:

1. **Readability and clarity**:
    - Easier for new developers to **trace changes**.
    - Important for **code reviews**.
2. **Releases and audits**:
    - Clear, **concise history** helps **track what went into each release**.
3. **Debugging**:
    - A **clean history** makes **bisecting** (finding problematic commits) **faster**.
4. **Maintaining professionalism**:
    - Avoids **"work in progress" (WIP)**, **typos**, or **fixes** cluttering mainline history.

Analogy:
Think of **squashing commits** like **compiling multiple drafts** of a document into **one polished version** before publishing.

7.3 Conceptual Understanding of Squashing

When you **squash commits**, you:

1. **Select a range of commits**.
2. **Combine them into one commit**.
3. Replace the **original commits** with **the new combined commit**.

Visualizing Commit History:

Before Squashing:

A---B---C---D---E (feature-backend)

- **Commits C, D, E** are small, fragmented commits.

After Squashing (C, D, E):

A---B---F (feature-backend)

- **F** represents the **squashed commit**.

7.4 When to Squash Commits

Ideal Scenarios:

1. **Before merging feature branches**:
    - Combine **incremental commits** into **one feature commit**.
2. **During code review**:
    - Clean up history **before pushing to remote**.
3. **Correcting messy local history**:
    - Condense **WIP commits** into **meaningful units**.

When Not to Squash:

1. **Shared commits** on a **public branch**:
    - Avoid **rewriting shared history**.
2. **Commits that logically need separation**:
    - Example: Commit for **database migration** and another for **API changes**.

7.5 Step-by-Step Implementation of Squashing Commits

Scenario Setup:

You're on **feature-backend**, with **3 commits**:

1. "Add backend logic".
2. "Refactor backend logic".
3. "Fix minor bug in backend".

You want to **squash these commits into one** before merging into **main**.

Step 1: View Commit History

git log --oneline

Example output:

e7c4b2d Fix minor bug in backend
b2f9c8e Refactor backend logic
d4e2f1a Add backend logic
c4d1a3b Merge feature-frontend into main

- The **top 3 commits** will be **squashed**.

Step 2: Start Interactive Rebase

git rebase -i HEAD~3

- **HEAD~3** means rebase the **last 3 commits**.

Step 3: Interactive Rebase Editor Opens

Example:

pick d4e2f1a Add backend logic
pick b2f9c8e Refactor backend logic
pick e7c4b2d Fix minor bug in backend

Step 4: Change Picks to Squash

Modify to:

pick d4e2f1a Add backend logic
squash b2f9c8e Refactor backend logic
squash e7c4b2d Fix minor bug in backend

- The **first commit** remains as **pick** (the base commit).

- The others are **squashed into the first**.

Step 5: Combine Commit Messages

Git opens another editor to **combine commit messages**:

This is a combination of 3 commits.

The first commit's message is:

Add backend logic

The following commit messages will be merged:

Refactor backend logic

Fix minor bug in backend

- Edit to a **single meaningful message**:

Add backend logic with refactoring and minor bug fixes.

Step 6: Complete the Rebase

- After saving and exiting:
    - Git rewrites the **3 commits** as **one squashed commit**.

Step 7: Verify New Commit History

git log --oneline

Expected output:

a9f3c7e Add backend logic with refactoring and minor bug fixes

c4d1a3b Merge feature-frontend into main

- The **3 commits** are now **combined** into **one**.

Step 8: Push Changes (Force Push Required)

git push -f origin feature-backend

- **Force push** is required because **rebasing rewrites commit history**.

7.6 Visualizing Squashing Workflows

Before Squashing:

A---B---C---D---E (feature-backend)

After Squashing (C, D, E):

A---B---F (feature-backend)

- **F** represents the **squashed commit**.

7.7 Best Practices for Squashing

1. **Squash before merging into main**:
    - Keeps **mainline history clean**.
2. **Use interactive rebase for precise control**:
    - Allows you to **pick and squash specific commits**.
3. **Write meaningful commit messages**:
    - Summarize **what was squashed**.
4. **Communicate with your team**:
    - Before **force pushing** squashed commits.

7.8 Squashing in CI/CD Workflows

1. **Squash merges**:
    - GitHub/GitLab support **automatic squash merges**:
        - Combine **all commits** in a pull request into **one commit**.
2. **Release pipelines**:
    - Squashing ensures **concise history** for **release tracking**.

7.9 Advanced Squashing Techniques

Squash During Merge (GitHub/GitLab):

- When merging a **pull request**:
    - Select **“Squash and Merge”**.

Squash All Commits into One (root commit):

git rebase -i --root

- **Squashes entire branch history** into **one commit**.

7.10 Common Mistakes & Pitfalls

Mistake	**How to Avoid**
Squashing **shared commits**	Only squash **private/unshared branches**.
Forgetting **force push after rebase**	Always **force push** (git push -f).
Losing commit messages during squash	Always **edit messages** clearly.