Git Scenarios Lab Guide

This document contains **all Git scenario-based exercises** you can perform in your lab, including: - Scenario description - Step-by-step actions (both terminal and GitHub UI) - Commands to run - Files to edit/add - Expected output - Explanation to say in viva

---

Scenario 1: "Rejected - non-fast-forward" while pushing

Scenario: Remote repository has commits that your local branch does not have.

Steps:

1. Edit a file directly on GitHub (e.g., README.md) and commit.
2. On your local system, try to push local changes:

   ```
   git push origin main
   ```

   You will see the error.
3. Pull the remote changes and rebase:

   ```
   git pull --rebase origin main
   ```

4. Resolve any conflicts if prompted (edit files, then `git add .`).
5. Push the updated local branch:

   ```
   git push origin main
   ```

Files to add/edit: README.md or any file you changed on GitHub.

Expected output: Push succeeds with remote and local history aligned.

Explanation to say: Remote branch had new commits. I updated local branch first using rebase, then pushed successfully.

Scenario 2: Push a feature branch without affecting main

Scenario: You are developing a new feature and want to keep main untouched.

Steps:

1. Create a feature branch locally:

```
git checkout -b feature/login
```

2. Edit or create files for the feature.
3. Stage and commit changes:

```
git add .
git commit -m "Add login feature"
```

4. Push the feature branch to remote:

```
git push origin feature/login
```

Files to add/edit: Any files for the feature.

Expected output: Remote now has a branch `feature/login` independent of main.

Explanation to say: Feature branch allows isolated development without affecting main.

Scenario 3: Fetch new branches created on remote

Scenario: New branches are added to remote after cloning.

Steps:

1. Fetch all updates from remote:

```
git fetch origin
```

2. List all branches:

```
git branch -a
```

3. Checkout a new branch:

```
git checkout <new-branch-name>
```

Files to add/edit: None required.

Expected output: Local repo now knows about all remote branches.

Explanation to say: `git fetch` updates local metadata without merging changes automatically.

Scenario 4: Pull updates from main without merge conflicts

Scenario: You have local changes and want to pull remote changes safely.

Steps:

1. Stash local changes:

```
git stash
```

2. Rebase onto latest remote main:

```
git pull --rebase origin main
```

3. Apply stashed changes back:

```
git stash pop
```

Files to add/edit: Any local files that were changed and stashed.

Expected output: Local changes are applied on top of updated main branch without conflicts.

Explanation to say: Stashing and rebasing keeps history clean and avoids conflicts.

Scenario 5: Accidentally pushed sensitive file

Scenario: Sensitive file (e.g., API keys) was committed.

Steps:

1. Remove the file from Git tracking:

```
git rm --cached secret.txt
```

2. Commit the removal:

```
git commit -m "Remove sensitive file"
```

3. Push the changes:

```
git push origin main
```

Files to add/edit: Remove or edit `secret.txt` as needed.

Expected output: File removed from GitHub, local copy remains.

Explanation to say: Sensitive data must be removed from history to secure repository.

Scenario 6: Update feature branch with latest main

Scenario: Manager requests feature branch to be updated with main changes.

Steps:

1. Checkout your feature branch:

```
git checkout feature/login
```

2. Fetch latest main:

```
git fetch origin
```

3. Rebase feature branch onto main:

```
git rebase origin/main
```

Files to add/edit: Feature files as needed if conflicts occur.

Expected output: Feature branch now has all latest main commits applied first.

Explanation to say: Keeps feature branch up to date without merge commits.

Scenario 7: Change remote URL (push to new repository)

Scenario: Need to push to a different remote repo.

Steps:

1. Change the remote URL:

```
git remote set-url origin <new-remote-url>
```

2. Push current branch:

```
git push -u origin main
```

Files to add/edit: None.

Expected output: Local branch now pushes to new remote.

Explanation to say: Updating remote allows switching repositories without reinitializing Git.

Scenario 8: Bring branch up to date without losing local changes

Scenario: Local branch behind remote but has uncommitted changes.

Steps:

1. Stash local changes:

```
git stash
```

2. Pull updates with rebase:

```
git pull --rebase origin main
```

3. Apply stashed changes:

```
git stash pop
```

Files to add/edit: Any local files you had edited.

Expected output: Local changes applied cleanly on top of updated remote main.

Explanation to say: Stash + rebase ensures clean history without losing local changes.

Scenario 9: Resolve conflicts after pull

Scenario: Local changes conflict with remote commits.

Steps:

1. Pull remote changes:

```
git pull origin main
```

2. Git will mark conflicts with <<<<<<<, =======, >>>>>>>.
3. Edit conflicting files to resolve.
4. Stage resolved files:

```
git add .
```

5. Commit the merge:

```
git commit -m "Resolve merge conflicts"
```

Files to add/edit: All files with conflicts.

Expected output: Conflicts resolved, merge completed.

Explanation to say: Manual conflict resolution ensures code integrity before pushing.

Scenario 10: Delete branch from remote

Scenario: Feature branch merged and no longer needed on remote.

Steps:

1. Delete remote branch:

```
git push origin --delete feature/login
```

Files to add/edit: None.

Expected output: Remote branch deleted.

Explanation to say: Clean up remote branches after merging to keep repository organized.

✅Notes

- Always use `git status` to check current branch and uncommitted changes.
- `git log --oneline` helps verify commit history.
- Use GitHub UI for creating branches, editing files, or committing directly as needed.