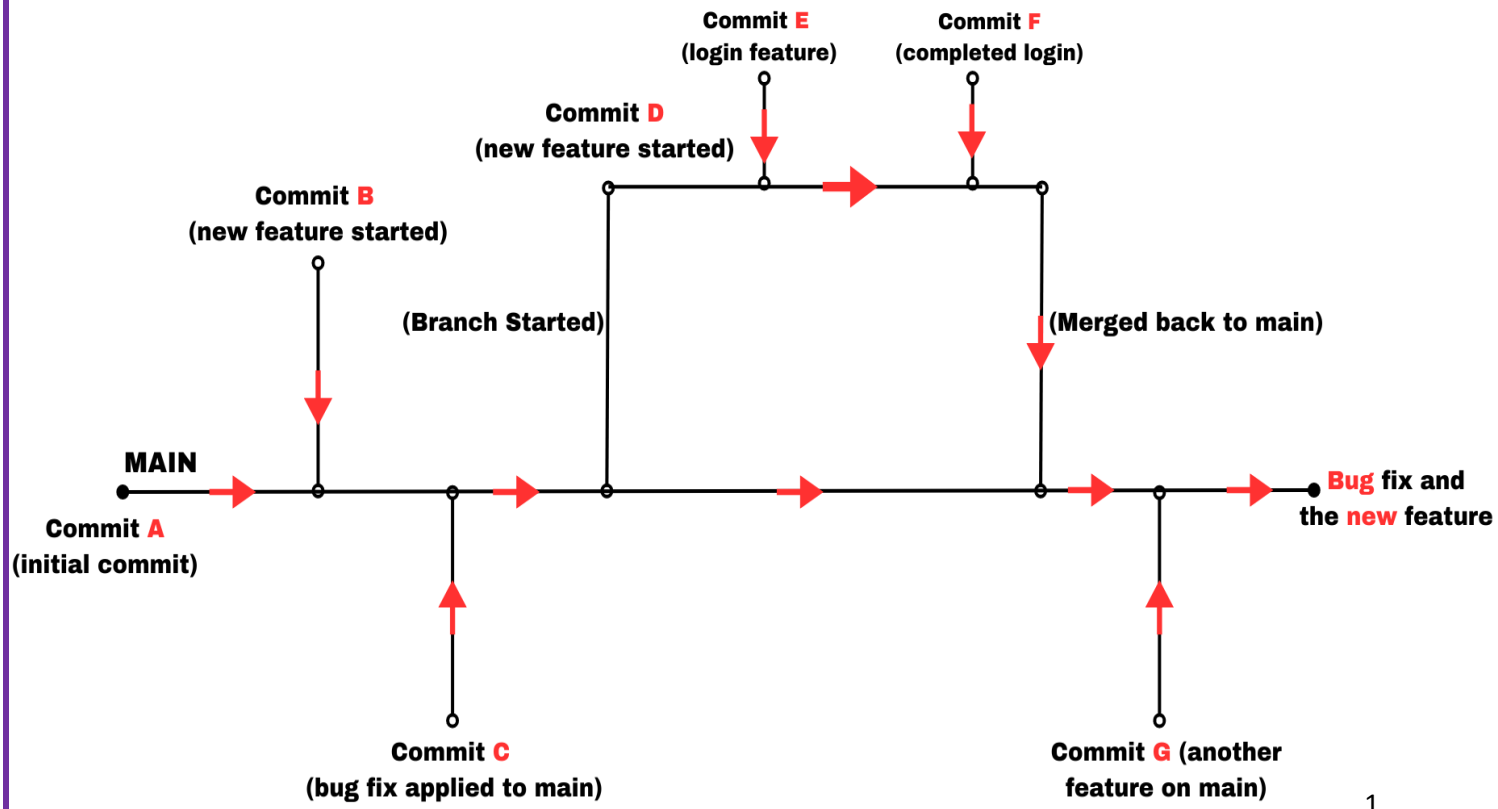
DevOps Shack

Scenario-Based Git Solutions: From Basics to Advanced

Git is one of the most widely used version control systems in software development. It allows developers to track changes, collaborate efficiently, and manage codebases of any size. Whether you're working on a solo project or contributing to a large-scale enterprise application, Git provides the tools necessary to ensure a smooth and organized development process.



1. Managing Branches for Feature Development

Scenario: You're developing multiple features in parallel and want to keep each feature isolated until it's ready to be merged into the main project.

Steps:

1. **Creating a Branch:**
    - To start working on a new feature, create a new branch:

```
git checkout -b feature/your-feature-name
```

    - This keeps your changes isolated from the main branch.
2. **Switching Between Branches:**
    - To switch back to the main branch or another branch:

```
git checkout main
```

3. **Merging Branches:**
    - Once the feature is completed, merge the branch back into main:

```
git checkout main
git merge feature/your-feature-name
```

4. **Best Practices:**
    - Use descriptive branch names (feature/login-page, bugfix/authentication).
    - Regularly merge main into your feature branch to keep it up to date.

2. Reverting Changes

Scenario: You accidentally introduced a bug and need to undo recent changes.

Steps:

1. **Undoing the Last Commit:**
    - If you want to revert the most recent commit but keep your changes:

```
git reset --soft HEAD~1
```

    - This moves your commit to the staging area, where you can edit and recommit.
2. **Reverting a Commit:**
    - If you want to undo a commit and create a new commit that undoes the changes:

```
git revert <commit-hash>
```

3. **Hard Reset:**
    - If you want to completely discard recent changes:

```
git reset --hard HEAD~1
```

4. **Recovering Lost Commits:**
    - If you accidentally lose commits, use the reflog to recover:

```
git reflog
```

```
git reset --hard <commit-hash>
```

3. Handling Merge Conflicts

Scenario: Two developers modified the same file, resulting in merge conflicts.

Steps:

1. **Identifying Conflicts:**
    - When attempting a merge, Git will automatically detect conflicts and mark them in the affected files:

```
git merge feature/your-feature-name
```

2. **Resolving Conflicts:**
    - Open the conflicting file and manually resolve the differences by choosing between the conflicting changes or combining them.
    - The conflict markers look like this:

```
<<<<<<< HEAD
(current changes)
=======
(incoming changes)
>>>>>>> feature/your-feature-name
```

3. **Marking as Resolved:**
    - Once the conflict is resolved, add the file to staging and continue the

```
git add <file-name>
git commit
```

4. **Preventing Conflicts:**
    - Pull changes from main often and make small, frequent commits to avoid large conflicts.

4. Tagging and Releases

Scenario: Your software is ready for production, and you need to create a release tag.

Steps:

1. **Creating a Tag:**
    - Tags help mark a specific point in Git history, usually for releases:

```
git tag -a v1.0.0 -m "Release version 1.0.0"
```

2. **Listing and Deleting Tags:**

- To list tags:

```
git tag
```

- To delete a tag:

```
git tag -d v1.0.0
```

3. **Pushing Tags to Remote:**
   - Tags aren't automatically pushed to a remote repository, so push them manually:

```
git push origin v1.0.0
```

4. **Best Practices:**
   - Use semantic versioning (v1.0.0, v2.1.3).
   - Create release branches (e.g., release/1.0) for stable releases.

5. Git Hooks for Automation

Scenario: You want to automate tasks like running tests or code formatting before committing changes.

Steps:

1. **Understanding Git Hooks:**
   - Git hooks are scripts that run automatically at specific stages of the Git workflow (e.g., pre-commit, pre-push).
   - Hooks are stored in the .git/hooks directory of your repository.
2. **Creating a Pre-Commit Hook:**
   - Write a pre-commit hook to automate tasks like linting or running tests before a commit:

```
#!/bin/sh
# Run tests before committing
npm test
```

```
if [ $? -ne 0 ]; then
 echo "Tests failed. Commit aborted."
 exit 1
fi
```

3. **Making the Hook Executable:**
    - Ensure your hook script has execution permissions:

```
chmod +x .git/hooks/pre-commit
```

4. **Common Use Cases for Git Hooks:**
    - **Pre-commit:** Running linters, code formatters, or security checks before allowing a commit.
    - **Pre-push:** Running tests or ensuring code quality checks before pushing to remote repositories.
    - **Post-merge:** Automatically installing dependencies or building the project after a merge.
5. **Best Practices:**
    - Keep hooks lightweight to avoid slowdowns in the Git workflow.
    - Share common hooks with your team via the repository or CI/CD configuration.

Conclusion

Each of these Git scenarios demonstrates how to apply Git's tools to real-world challenges, from managing code for multiple features, reverting changes, resolving merge conflicts, and tagging stable releases, to automating tasks with hooks. Following these practices can help streamline your development workflow and ensure better collaboration within your team.