DevOps Shack

Module-3 | GIT

[Click Here To Enrol To Batch-5 | DevOps & Cloud DevOps]

Introduction to Git: Git is a distributed version control system used for tracking changes in source code during software development. It enables multiple collaborators to work on the same codebase simultaneously while maintaining a history of changes, making it easier to manage, collaborate, and track progress.

Key Concepts:

1. **Repository:** A repository (repo) is a collection of files and directories along with their complete history of changes.
2. **Commit:** A commit is a snapshot of the repository at a specific point in time. It contains changes made to files, a timestamp, and a commit message explaining the changes.
3. **Branch:** A branch is a separate line of development that allows you to work on features, bug fixes, or experiments without affecting the main codebase. The default branch is usually called "master" or "main."
4. **Merge:** Merging combines changes from one branch into another. It's commonly used to integrate feature branches back into the main branch.
5. **Pull Request (PR):** In a collaborative setting, a pull request is a request to merge changes from one branch (often a feature branch) into another (typically the main branch). It allows for discussion, code review, and testing before changes are merged.

Basic Git Commands:

1. **git init:** Initialize a new Git repository.

   ```
   git init
   ```

2. **git clone:** Create a copy of a remote repository on your local machine.

   ```
   git clone <repository_url>
   ```

3. **git add:** Stage changes for commit.

   ```
   git add <filename>
   ```

4. **git commit:** Create a new commit with staged changes.

   ```
   git commit -m "Commit message"
   ```

5. **git push:** Upload local commits to a remote repository.

   ```
   git push origin <branch_name>
   ```

6. **git pull:** Fetch remote changes and integrate them into the current branch.

   ```
   git pull origin <branch_name>
   ```

7. **git branch:** List, create, or delete branches.

8. ```
   git branch
   ```
9. ```
   git branch <branch_name>
   git branch -d <branch_name>
   ```

10. **git checkout:** Switch to a different branch or commit.

    ```
    git checkout <branch_name>
    ```

11. **git merge:** Combine changes from one branch into another.

    ```
    git merge <source_branch>
    ```

12. **git pull request:** Create a pull request on platforms like GitHub or GitLab.

    This command doesn't exist directly in Git. You perform this action on the platform where your remote repository is hosted.

Example Workflow:

Let's walk through a simple workflow involving creating a new feature branch, making changes, and merging them back into the main branch.

1. **Create a New Feature Branch:**

   ```
   git checkout -b feature/my-feature
   ```

2. **Make Changes:** Edit files in your codebase.

3. **Stage and Commit Changes:**

4. ```
   git add <changed_files>
   git commit -m "Added new feature"
   ```

5. **Push Changes to Remote:**

   ```
   git push origin feature/my-feature
   ```

6. **Create a Pull Request:** On your remote repository's platform (e.g., GitHub), create a pull request from your feature branch to the main branch. Discuss, review, and test changes if necessary.

7. **Merge the Pull Request:** After approval, merge the pull request on the remote platform.

8. **Update Local Main Branch:**

9. ```
   git checkout main
   git pull origin main
   ```

Git is a powerful version control tool that facilitates collaboration and history tracking in software development projects. It provides a structured way to manage code changes, collaborate with teammates, and ensure a stable and organized codebase. With the basic commands and concepts outlined above, you can begin to effectively utilize Git in your development workflow.

Git Merge & Git Rebase

Both `git rebase` and `git merge` are used to integrate changes from one branch into another. However, they have different approaches and use cases. Let's explore both with an example:
Suppose you have two branches: `feature` and `main`, where `main` is your main development branch, and `feature` contains a new feature you've been working on.

Here's how you might use both `git rebase` and `git merge` in this scenario:

Using `git merge`:

1. Switch to the `main` branch:
   `git checkout main`
2. Merge the changes from the `feature` branch into `main`:
   `git merge feature`

In this case, the commit history would look something like this:

```
*   Merge branch 'feature' into main
|\
| * New feature commit 3
| * New feature commit 2
| * New feature commit 1
* | Another main branch commit
* | Main branch commit
|/
* Initial commit
```

Here, the merge commit records that you merged the `feature` branch into `main`.

Using `git rebase`:

1. Switch to the `feature` branch:
   `git checkout feature`
2. Rebase the `feature` branch onto `main`:
   `git rebase main`

In this case, the commit history would look something like this:

```
* New feature commit 3
* New feature commit 2
* New feature commit 1
* Another main branch commit
* Main branch commit
* Initial commit
```

Here, the `feature` branch's commits are applied on top of the latest `main` branch commit. The history appears linear and cleaner compared to a merge commit.

Key Differences:

- `git merge` creates a new merge commit that combines changes from different branches. This can make the history more complex.
- `git rebase` moves the entire history of one branch onto another branch. It creates a linear history, avoiding additional merge commits.

Which to Choose?

Use `git merge` when you want to maintain a clear record of branch integration and want to capture the fact that a particular feature branch was merged into the main branch. Use `git rebase` when you want a cleaner and more linear history, often for feature branches or topic branches that you're not sharing with others.
Note: The choice between `git merge` and `git rebase` depends on your team's workflow and the desired history structure. Always communicate with your team to decide which approach to use.

Git Stash and Git Pop: Explained with Examples

Git Stash: In Git, the `git stash` command is used to temporarily save changes that you're not ready to commit yet, so you can switch to a different branch or perform other operations without committing incomplete work.

Git Pop: The `git stash pop` command is used to apply the most recent stash and remove it from the stash list. It's like a combination of `git stash apply` and `git stash drop`.

Example Scenario: Imagine you're working on a feature branch and need to switch to another branch to fix a bug. However, you don't want to commit the incomplete changes on the feature branch. This is where `git stash` comes in handy.

Step-by-Step Example:

1. **Create a New Feature Branch:**

   ```
   git checkout -b feature/my-feature
   ```

2. **Make Changes:** Edit files in your codebase.

3. **Stash Changes:** Stash the changes you've made but aren't ready to commit.

   ```
   git stash
   ```

4. **Switch to a Different Branch:**

   ```
   git checkout main
   ```

5. **Fix a Bug on Main Branch:** Make necessary changes on the `main` branch to fix a bug.

6. **Commit Bug Fix:**

```
7. git add <changed_files>
   git commit -m "Fixed bug"
```

8. **Switch Back to Feature Branch:**

```
   git checkout feature/my-feature
```

9. **Apply Stashed Changes:** Apply the stashed changes from the feature branch.

```
   git stash pop
```

   This will apply the stashed changes and remove the stash from the stash list.

10. **Continue Working:** Now you can continue working on your feature branch, which now includes the stashed changes.

11. **Commit Stashed Changes:** If needed, commit the stashed changes.

```
12.  git add <changed_files>
   git commit -m "Added stashed changes"
```

Additional Stash Commands:

- `git stash list`: Lists all stashes.
- `git stash apply stash@{n}`: Applies a specific stash without removing it from the stash list.
- `git stash drop stash@{n}`: Removes a specific stash from the stash list.
- `git stash clear`: Removes all stashes.

Note: It's important to understand that using `git stash` is a temporary solution. It's generally recommended to commit your changes properly before switching branches. Stashing is more suitable for quick switches or cases where you're not ready to commit yet.
`git stash` and `git stash pop` are valuable commands in Git when you need to temporarily save your changes, switch branches, and then reintegrate your changes. This allows you to maintain a clean and organized development workflow while still preserving your work in progress.

Git Revert and Git Reset: Explained with Examples

Git Revert: `git revert` is used to create a new commit that undoes the changes introduced by a previous commit. It's a safe way to undo changes while preserving the commit history.

Git Reset: `git reset` is used to move the current branch pointer to a different commit, effectively resetting the state of the branch. It can be used to discard commits or move branches to a previous state. Be cautious as it can rewrite history.

Example Scenario: Suppose you have a repository with the following commit history:

```
A --- B --- C --- D (main)
```

- Commit A: Initial state
- Commit B: Added new feature
- Commit C: Made some changes
- Commit D: Introduced a bug

You want to undo the changes introduced by commit D and go back to the state after commit C.

Git Revert:

1. **Reverting a Commit:**

   ```
   git revert D
   ```

   This creates a new commit that undoes the changes from commit D, resulting in:

```
A --- B --- C --- D --- E (main)
```

- Commit E: Revert of commit D

Git Reset:

1. **Soft Reset:**

   `git reset --soft C`
   This moves the `main` branch pointer back to commit C, leaving the changes from commit D in the staging area. Your working directory will have the changes from commit D.
   ```
   A --- B --- C (main)
            \
             D
   ```

2. **Mixed Reset:**

   `git reset --mixed C`
   This is the default mode. It moves the `main` branch pointer to commit C and removes the changes from commit D from the staging area. Your working directory will have the changes from commit D as uncommitted changes.
   ```
   A --- B --- C (main)
            \
             D
   ```

3. **Hard Reset:**

   `git reset --hard C`
   This moves the `main` branch pointer to commit C and discards all changes introduced by commit D. Be cautious with this option as it permanently removes changes.
   ```
   A --- B --- C (main)
            \
             D (unreferenced)
   ```

Diagrams:

Here's a visual representation of the commit history and the effects of using `git revert` and different modes of `git reset`:
Original commit history:

```
A --- B --- C --- D (main)
```

After using `git revert D`:
```
A --- B --- C --- D --- E (main)
```

After using `git reset --soft C`:
```
A --- B --- C (main)
         \
          D
```
After using `git reset --mixed C` (default behavior):
```
A --- B --- C (main)
         \
          D
```

After using `git reset --hard C`:

```
A --- B --- C (main)
          \
           D (unreferenced)
```

Both `git revert` and `git reset` are powerful tools for undoing changes in a Git repository. `git revert` creates a new commit to undo changes while preserving history, while `git reset` moves the branch pointer to a different commit, affecting the branch's history. Be cautious when using `git reset`, especially the `--hard` option, as it can result in permanent data loss. Always make sure to have backups or understand the implications before using these commands.

Git Cherry-Pick: Explained with Examples

`git cherry-pick` is a command used to apply a specific commit from one branch to another. It allows you to pick and apply a single commit's changes onto another branch without having to merge the entire branch.
Example Scenario: Suppose you have the following commit history:

```
          E --- F (feature)
         /
A --- B --- C --- D (main)
```

- Commit A: Initial state
- Commit B: Added initial feature
- Commit C: Bug fix on main
- Commit D: Another feature on main
- Commit E: New feature on feature branch
- Commit F: Bug fix on feature branch

You want to apply the bug fix introduced in commit C onto the feature branch.

Using Git Cherry-Pick:

1. **Identify the Commit to Cherry-Pick:**

   First, identify the commit you want to cherry-pick. In this case, it's commit C.

2. **Checkout the Target Branch:**

   ```
   git checkout feature
   ```

   Switch to the branch where you want to apply the cherry-picked commit.

3. **Cherry-Pick the Commit:**

```
git cherry-pick C
```

This applies the changes introduced in commit C onto the `feature` branch. Resulting commit history:

```
             E --- F --- C' (feature)
            /
A --- B --- C --- D (main)
```

- Commit C': Cherry-picked bug fix from commit C

Explanation:

By using `git cherry-pick C`, you applied the changes from commit C to the `feature` branch, resulting in a new commit C' on the feature branch. The commit C' contains the same changes as commit C but has a different commit hash because it's a separate commit.

Diagrams:

Original commit history:

```
          E --- F (feature)
         /
A --- B --- C --- D (main)
```

After using `git cherry-pick C`:

```
             E --- F --- C' (feature)
            /
A --- B --- C --- D (main)
```

`git cherry-pick` is a useful command for selectively applying changes from one commit to another branch. It's particularly handy when you want to bring specific changes from one branch into another without merging the entire branch. Keep in mind that the cherry-picked commit will have a new commit hash, and you should ensure that the changes are still valid in the new context.

Branching Strategy

Establishing a robust branching strategy is essential to ensure smooth development, testing, and deployment processes across different environments. Below is a common branching strategy that you can consider for a typical software development lifecycle, including development, testing (QA), pre-production (PPD), production, and disaster recovery (DR) environments.

1. **Main/Branch:**
    - Name: `main` or `master`
    - Purpose: This is the main branch that holds production-ready code. It's always stable and should ideally reflect the code running in the production environment.
2. **Development Branch:**
    - Name: `develop` or `dev`
    - Purpose: All ongoing development work takes place in this branch. New features and bug fixes are merged into this branch. It should be relatively stable but not necessarily production-ready at all times.
3. **Feature Branches:**
    - Name: `feature/<feature-name>`
    - Purpose: Each new feature or task gets its own branch, created from the `develop` branch. Developers work on these branches and merge them back into the `develop` branch when the feature is complete.
4. **QA Branch:**
    - Name: `qa` or `testing`
    - Purpose: Once features are considered complete in the `develop` branch, they are merged into the `qa` branch for testing. This branch should reflect a stable state for testing purposes.
5. **Pre-Production Branch:**
    - Name: `ppd` or `staging`
    - Purpose: This branch is used to simulate the production environment closely. After successful QA testing, code is merged from the `qa` branch to the `ppd` branch for final validation before deployment.
6. **Production Branch:**
    - Name: `prod` or `release`

    - Purpose: Once the code is thoroughly tested in the `ppd` environment and ready for deployment, it's merged into the `prod` branch and deployed to the production environment.

7. **Disaster Recovery Branch:**
    - Name: `dr` or `backup`
    - Purpose: This branch holds code that is identical to the currently deployed production code. It's useful for disaster recovery scenarios, allowing rapid deployment of the latest stable code in case of critical issues.

Workflow:

1. Developers work on feature branches derived from `develop`.
2. Once a feature is complete, it's merged into `develop`.
3. Regular integration and automated tests take place in `develop`.
4. Periodic merges from `develop` to `qa` for testing.
5. After successful QA, merge to `ppd` for final validation.
6. After final validation in `ppd`, merge to `prod` for deployment.
7. Maintain a mirror of the production code in the `dr` branch.

Benefits:

- Clear separation of environments and responsibilities.
- Code stability is maintained in each environment.
- Isolates ongoing development from testing and production environments.
- Facilitates parallel development of multiple features.

Considerations:

- Use automation for testing and deployment processes.
- Implement code reviews and pull request approvals.
- Communicate and document the branching strategy for the team.
- Tailor the strategy to your team's workflow and project requirements.

Git Troubleshooting

Git is a powerful version control system, but like any software, it can encounter issues from time to time. Here are ten common issues that Git users might face, along with troubleshooting steps and example scenarios for each.

Issue 1: Merge Conflicts Merge conflicts occur when Git can't automatically merge changes from different branches. This typically happens when changes to the same part of a file conflict.

Troubleshooting Steps:

1. Use `git status` to identify conflicted files.
2. Open the conflicted file(s) in a text editor and look for conflict markers (`<<<<<<<`, `=======`, and `>>>>>>>`).
3. Manually resolve the conflicts.
4. Use `git add <conflicted_file>` to mark the file as resolved.
5. Commit the changes using `git commit`.

Example Scenario: Suppose you're merging the "feature" branch into the "master" branch. A merge conflict occurs in "file.txt".

```
$ git merge feature
# Conflict in file.txt
$ git status
# Resolve conflicts in file.txt
$ git add file.txt
$ git commit -m "Resolve merge conflict"
```

Issue 2: Detached HEAD State A detached HEAD state occurs when you're not on a branch, usually after checking out a specific commit.

Troubleshooting Steps:

1. Use `git branch` to see which commit you're on.
2. Create a new branch at the current commit using `git checkout -b <new_branch_name>`.

Example Scenario: You accidentally check out a commit directly instead of a branch.

```
$ git checkout abc123
# Detached HEAD state at commit abc123
$ git checkout -b new-branch
# Create a new branch "new-branch" at commit abc123
```

Issue 3: Untracked Files Untracked files are files that Git doesn't recognize or track.

Troubleshooting Steps:

1. Use `git status` to see untracked files.
2. Add untracked files to the staging area using `git add <file>`.

Example Scenario: You create a new file "new_file.txt" but Git doesn't recognize it.

```
$ git status
# Untracked file: new_file.txt
$ git add new_file.txt
# Add new_file.txt to the staging area
```

Issue 4: Undoing Mistakes (git reset) You made a commit and need to undo it.

Troubleshooting Steps:

1. Use `git log` to find the commit hash you want to reset to.
2. Use `git reset --hard <commit_hash>` to move the current branch and working directory to that commit.

Example Scenario: You accidentally committed a wrong change and want to remove the commit.

```
$ git log
# Find the commit hash you want to reset to
$ git reset --hard abc123
# Reset to commit abc123, discarding changes after it
```

Issue 5: Reverting Commits You want to undo a specific commit without discarding the subsequent changes.

Troubleshooting Steps:

1. Use `git log` to find the commit hash you want to revert.
2. Use `git revert <commit_hash>` to create a new commit that undoes the changes from the specified commit.

Example Scenario: You want to undo changes from a specific commit.

```
$ git log
# Find the commit hash you want to revert
$ git revert abc123
# Create a new commit that undoes changes from commit abc123
```

Issue 6: Deleted Branches You accidentally deleted a branch and want to recover it.

Troubleshooting Steps:

1. Use `git reflog` to find the commit hash of the deleted branch.
2. Create a new branch at that commit using `git branch <new_branch_name> <commit_hash>`.

Example Scenario: You deleted the "feature" branch and want to recover it.

```
$ git reflog
# Find the commit hash of the deleted "feature" branch
$ git branch feature abc123
# Create a new branch "feature" at commit abc123
```

Issue 7: Incorrect Commit Message You made a commit with a wrong or incomplete message.

Troubleshooting Steps:

1. Use `git commit --amend` to edit the most recent commit message.

Example Scenario: You committed with a typo in the message and want to fix it.

```
$ git commit --amend
# Opens a text editor to modify the commit message
```

Issue 8: Stash Changes You're in the middle of working on something, but you need to switch to a different branch.

Troubleshooting Steps:

1. Use `git stash` to save your changes.
2. Switch to the other branch.
3. Use `git stash pop` to apply the stashed changes back.

Example Scenario: You're working on a feature but need to switch to the "master" branch for a quick fix.

```
$ git stash
# Stash your changes
$ git checkout master
# Switch to the "master" branch
$ git stash pop
# Apply your stashed changes back
```

Issue 9: Renaming/Moving Files You renamed or moved a file outside of Git, and Git doesn't recognize the change.

Troubleshooting Steps:

1. Use `git status` to see the changes as untracked or deleted.
2. Use `git add <new_file>` to stage the renamed/moved file.

Example Scenario: You renamed "old_file.txt" to "new_file.txt" outside of Git.

```
$ git status
# Shows "old_file.txt" as deleted and "new_file.txt" as untracked
$ git add new_file.txt
# Stage the renamed file
```

Issue 10: Remote Repository Not Found (git remote) You want to connect your local repository to a remote, but it's not working.

Troubleshooting Steps:

1. Use `git remote -v` to check the configured remotes.
2. Add a remote repository using `git remote add <name> <url>`.

Example Scenario: You want to add a remote repository named "origin" with a URL.

```
$ git remote -v
# Check existing remotes
$ git remote add origin <repository_url>
# Add a new remote named "origin"
```

50 Git commands

1. **`git init`** Initializes a new Git repository.
```
$ git init
Initialized empty Git repository in /path/to/repository/
```

2. **`git clone`** Clones a remote repository to your local machine.
```
$ git clone https://github.com/username/repository.git
Cloning into 'repository'...
```

3. **`git add`** Stages changes for commit.
```
$ git add file.txt
```

4. **`git status`** Shows the status of the working directory and staged changes.
```
$ git status
```

5. **`git commit`** Commits staged changes.
```
$ git commit -m "Added new feature"
```

6. **`git log`** Displays commit history.
```
$ git log
```

7. **`git diff`** Shows differences between working directory and staged changes.
```
$ git diff
```

8. **`git branch`** Lists branches.
```
$ git branch
```

9. **`git checkout`** Switches branches or restores files.
```
$ git checkout branch_name
```

10. **`git merge`** Merges changes from one branch into another.
```
$ git merge feature_branch
```

11. **`git pull`** Fetches and integrates changes from a remote repository.
```
$ git pull origin master
```

12. **`git push`** Pushes changes to a remote repository.
```
$ git push origin master
```

13. **`git remote`** Manages remote repositories.
```
$ git remote add origin https://github.com/username/repository.git
```

14. **`git fetch`** Downloads objects and refs from a remote repository.
```
$ git fetch origin
```

15. **`git stash`** Temporarily stores changes to work on something else.
```
$ git stash
```

16. **`git tag`** Creates and manages tags for specific commits.
```
$ git tag v1.0.0
```

17. **`git reset`** Unstages changes or moves the HEAD to a specific commit.
```
$ git reset HEAD file.txt
```

18. **`git rebase`** Reapplies commits on top of another base.
```
$ git rebase master
```

19. **`git config`** Sets configuration options.
```
$ git config --global user.name "Your Name"
$ git config --global user.email your.email@example.com
```

20. **`git log --oneline`** Displays compact commit history.
```
$ git log –oneline
```

21. **`git show`** Shows information about a commit.
```
$ git show commit_hash
```

22. **`git cherry-pick`** Applies a commit from one branch to another.
```
$ git cherry-pick commit_hash
```

23. **`git rm`** Removes files from the working directory and stages the removal.
```
$ git rm file.txt
```

24. **`git revert`** Creates a new commit that undoes changes from a previous commit.
```
$ git revert commit_hash
```

25. **`git reflog`** Displays the history of HEAD positions.
```
$ git reflog
```

26. **`git clean`** Removes untracked files and directories from the working directory.
```
$ git clean -n  # Dry-run
$ git clean -f  # Force removal
```

27. **`git tag -a`** Creates an annotated tag with a message.
```
$ git tag -a v1.0.0 -m "Version 1.0.0"
```

28. **`git log --graph`** Displays commit history as a graph.
```
$ git log --graph –oneline
```

29. **`git config --list`** Lists all Git configuration settings.
```
$ git config –list
```

30. **`git log --since`** / **`git log --until`** Displays commit history within a time range.
```
$ git log --since="2 weeks ago"
$ git log --until="2023-07-01"
```

31. **`git cherry`** Shows commits that have not been merged.
```
$ git cherry master feature_branch
```

32. **`git revert --no-commit`** Reverts changes interactively without committing.
```
$ git revert --no-commit commit_range
```

33. **`git log --author`** Filters commit history by author.
```
$ git log --author="John Doe"
```

34. **`git log --stat`** Displays file statistics with commit history.
```
$ git log –stat
```

35. **`git blame`** Shows who last modified each line in a file.
```
$ git blame file.txt
```

36. **`git tag -d`** Deletes a tag.
```
$ git tag -d v1.0.0
```

37. **`git log -p`** Displays commit history with patch diffs.
```
$ git log -p
```

38. **`git rev-parse`** Converts a revision string into a SHA-1 hash.
```
$ git rev-parse HEAD
```

39. **`git remote -v`** Lists remote repositories and their URLs.
```
$ git remote -v
```

40. **`git log --decorate`** Displays references (branches, tags) in commit history.
```
$ git log –decorate
```

41. **`git bisect`** Performs a binary search to find a faulty commit.
```
$ git bisect start
$ git bisect good <commit>
$ git bisect bad <commit>
$ git bisect reset
```

42. **`git log --grep`** Searches commit messages for a specific keyword.
```
$ git log --grep="bug fix"
```

43. **`git log --name-only`** Displays only file names in commit history.
```
$ git log --name-only
```

44. **`git rebase -i`** Interactively rewrites commit history.
```
$ git rebase -i HEAD~3
```

45. **`git log --before`** **/** **`git log --after`** Displays commit history before/after a specific date.
```
$ git log --before="2023-01-01"
$ git log --after="2022-01-01"
```

46. **`git checkout -b`** Creates a new branch and switches to it.
```
$ git checkout -b new_feature
```

47. **`git log --cherry-pick`** Shows commits that have been cherry-picked.
```
$ git log --cherry-pick master..feature_branch
```

48. **`git log -S`** Searches for changes that added or removed a specific string.
```
$ git log -S "function_name"
```

49. **`git reflog expire`** Expires old reflog entries.
```
$ git reflog expire --expire=30.days refs/heads/master
```

50. **`git commit --amend`** Modifies the most recent commit.

```
$ git commit --amend
```