GIT

➔ Git is a
- Distributed
- Compatible
- Non-Linear
- Branching
- Light Weight
- Speed
- Open Source
- Reliable
- Secret
- Economical

What is SCM?

➢ **SCM** stands for **Source Code Management** in the context of Git.
➢ SCM is a critical tool used to track modifications to a source code repository. It maintains a running history of changes to a code and helps resolve conflicts when merging updates from multiple contributors.

Key Principles of SCM?

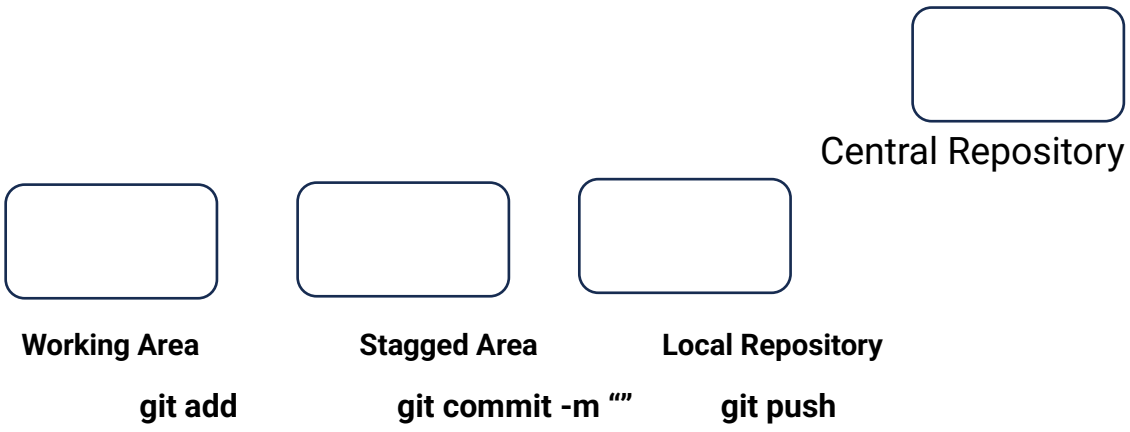
➢ It provide safeguards to prevent loss of work.

Benefits of SCM?

➢ Historical Record.

Git Config Commands

➢ git config --global user.name "RASHANTH793"
➢ git config --global user.email "jprashanth429@gmail.com"



Repository

- A central location where all version of source code are stored.

Branching

- Creating a separate line to development from the main codebase.

Merging

- Combing changes from different branches back into main code base.

Version History

- History of all the Changes made in the Repository.

How to add Repository from Local To Remote

- We will use git remote origin

GIT COMMANDS

- cd ~
- git status
- git add . OR git add FileName
- git log
- git push
- git diff
- git diff -w
- git diff --word-diff
- git diff –cached
- git branch branchName
- git checkout
- git checkout -b BranchName
- git log –oneline
- rm -rf FileName

Basics of Git

- Git is a distributed version control system that helps developers track changes to files, especially source code, during software development.
- It allows multiple developers to collaborate effectively by maintaining a history of changes and managing different versions of a project. Unlike centralized systems, Git stores a complete copy of the repository on every developer's machine, ensuring redundancy and fast performance.
- Some of Git's key features include its distributed architecture, ability to branch and merge easily, strong data integrity, and the fact that it is free and open-source. To get started, you need to install Git from its official website, and you can verify its installation by running the command git --version in your terminal.

Git File Systems

Git operates on a three-layered file system:

1. **The Working Directory**, where all the files of your project reside, allows you to make changes directly.
2. **The Staging Area (Index)**, which is an intermediate layer where changes are prepared before being committed. Files are moved to the staging area using the git add command.
3. **The Git Repository**, which is a hidden .git directory that contains all the metadata and history of the project. It includes objects like blobs (file contents), trees (directory structure), and commits (snapshots of changes). These layers together enable Git to track changes efficiently.

Creating a Local Repository

- To create a new local Git repository, navigate to your project directory and initialize Git by running the command git init.
- This command creates a .git folder that stores all the version control information. If you want to work on an existing repository, you can clone it using git clone <repository-url>.
- Cloning copies the entire repository, including its history, from a remote server to your local machine.

Basic Git Configurations

- Before using Git, you need to set your identity by running

  git config --global user.name "Your Name" and

  git config --global user.email "you@example.com".

- These settings ensure that your commits are associated with the correct identity. You can view all your configurations by running

  git config --list.

- Additionally, you can set a default editor, such as Vim or VSCode, using the

  git config --global core.editor command.

Adding Files to a Project

- When you create or modify files in your working directory, they must be added to the staging area before being committed.
- Use git add <filename> to stage a specific file, or git add . or git add -a to stage all changes. Once staged, these files are ready to be committed to the repository.

Status of the Project

- The git status command allows you to check the current status of your project. It shows you which files have been modified, staged, or remain untracked.
- This command is crucial for understanding the current state of your working directory and staging area before committing changes.
- Three Different status Tracked file, Untracked file, Updated file, Deleted file.

Adding, Checking In Files, and Committing to Git

- After staging files, you can commit them to the repository using

  git commit -m "Your commit message".

- A commit represents a snapshot of your project at a specific point in time. You can use git diff to see the unstaged changes and git diff --cached to view the staged changes before committing.

Ignoring Specific File Types

- To avoid tracking unnecessary files, create a .gitignore file in your repository.
- This file contains patterns for file types or directories you want Git to ignore, such as temporary files or logs. For example, a .gitignore file might include

  “*.log” , “ *.tmp”

  Run git status --ignored to verify which files are being ignored.

Tags

- Tags are used in Git to mark specific points in your project’s history, often for releases.
- You can create a lightweight tag with git tag <tagname> or an annotated tag with git tag -a <tagname> -m "Tag message".
- Annotated tags include additional metadata like the author and date.
- Use git tag to list all tags in the repository, and push tags to a remote repository using git push origin <tagname> or git push --tags to push all tags.

Branching

- Branches in Git allow you to create independent lines of development.
- To create a new branch, use git branch <branchname>, and switch to it with git switch <branchname> or the older command git checkout <branchname>. To see all branches, use git branch.
- By default, Git creates a main or master branch, depending on your configuration

Merging

- Merging is the process of integrating changes from one branch into another.
- To merge a branch into the current branch, use git merge <branchname>.
- If there are conflicts, Git will prompt you to resolve them manually. Once resolved, stage the changes with git add and commit them to complete the merge.

Rebasing

- Rebasing rewrites the new commit history of a branch to apply its changes on top of another branch.
- To rebase, use git rebase <base-branch>.
- For example, rebasing feature onto main moves the feature branch commits to the top of the main branch. Use git rebase -i for interactive rebasing, which lets you edit commit history. If something goes wrong, you can abort the rebase with git rebase --abort.

Reverting a Commit

- To undo a specific commit, use the git revert <commit-hash> command.
- This creates a new commit that undoes the changes introduced by the specified commit.
- For more drastic undo actions, such as resetting the branch to a previous commit, you can use git reset with the --soft or --hard options.

Git Reset

What is git reset?

- git reset is a command that moves the current branch's HEAD pointer and optionally modifies the staging area and working directory to align with a specific commit.
- It is typically used to undo commits or unstage changes.

Modes of git reset

a. Soft Reset (git reset --soft <commit>**)**

- Moves the HEAD pointer to the specified commit, but the changes in the commits remain in the staging area.
- It is useful when you want to modify or redo the last few commits without losing the changes.

Example:

git reset --soft HEAD~1

- This moves HEAD to the previous commit (HEAD~1) but keeps the changes staged.

b. Mixed Reset (Default: git reset --mixed <commit>**)**

- Moves the HEAD pointer to the specified commit and unstages changes from the staging area, but the changes remain in the working directory.
- This is the most commonly used reset mode.

Example:

git reset --mixed HEAD~1

- The last commit is undone, and its changes are moved to the working directory.

c. Hard Reset (git reset --hard <commit>**)**

- Moves the HEAD pointer to the specified commit and **discards all changes** in the staging area and working directory.
- It is destructive as it removes all changes after the reset point.

Example:

git reset --hard HEAD~1

- Completely erases the last commit and associated changes.

Note: Be cautious when using --hard as it permanently deletes uncommitted changes.

Use Cases for git reset

1. Undoing commits while keeping changes:
    - Use --soft to retain changes in the staging area.
2. Reverting commits for rework:
    - Use --mixed to keep changes in the working directory.
3. Cleaning up commits entirely:
    - Use --hard to remove changes permanently.

Git Restore

What is git restore?

- git restore is a command introduced in Git 2.23 to make the functionality of undoing changes more explicit and user-friendly.
- It is designed to restore files in the working directory or staging area to a specific state.
- When we use cmd { git restore –staged } this will restore the file from Head (Local Repo) to Staged
- Git restore –worktree & git restore –staged.
- Default restore is from Staged to Workspace Tree try with cmd { git restore Filename }.

Modes of git restore

a. Restoring Files to the Last Commit

Restores changes in the working directory to match the state of the last commit. This effectively discards modifications made to the file.

Example:

git restore <file>

- Discards uncommitted changes in the specified file.

b. Restoring Files to a Specific Commit

Restores a file to match its state in a specific commit, leaving the working directory unchanged.

Example:

git restore --source=<commit> <file>

- Retrieves the version of the file from the specified commit.

c. Removing Files from the Staging Area

Unstages a file while keeping the changes in the working directory.

Example:

git restore --staged <file>

- Removes the file from the staging area but keeps the local changes intact.

Comparison: git restore vs. git reset

Aspect	git reset	git restore
Scope	Works at the commit level and staging area.	Works primarily at the file level.
Purpose	Used for undoing commits or resetting the branch pointer.	Used for restoring file states.
Modes	--soft, --mixed, --hard.	--source, --staged.
Risk Level	Can be destructive (--hard).	Less destructive; works on specific files.

Aspect	git reset	git restore
User-Friendliness	Legacy command; can be complex for beginners.	Newer and more intuitive for file-level changes.

Common Scenarios for Reset and Restore

1. **Undoing a Commit and Retaining Changes:**

   git reset --mixed HEAD~1

   - Unstages changes from the last commit but retains them in the working directory.
2. **Removing Unstaged Changes in a File:**

   git restore <file>

   - Reverts the file to its last committed state.
3. **Unstaging a File:**

   git restore --staged <file>

   - Removes a file from the staging area while keeping local changes.
4. **Discarding All Changes Permanently:**

   git reset --hard HEAD

   - Resets the working directory and staging area to the last commit.
5. **Restoring a File from a Previous Commit:**

   git restore --source=<commit> <file>

   - Replaces the current file with its version from the specified commit.

Important Notes

- Use git restore for safer and more precise operations on specific files.
- Reserve git reset --hard for situations where you are certain about discarding all changes.
- Always make a backup or commit your work before performing potentially destructive actions.

Using the diff Command

- The git diff command compares changes in your working directory with the staging area or repository.
- For example, git diff shows unstaged changes, while git diff --cached displays staged changes.
- To compare two commits, use git diff <commit1> <commit2>, which highlights the differences between their snapshots.

Garbage Collection

- Git uses garbage collection to clean up unnecessary files and optimize repository storage.
- Run git gc to trigger this process manually. It compresses loose objects, removes unreachable objects, and repacks the repository to save space.
- To remove only unreachable objects, use git prune.

Git Logging and Auditing

- Git provides powerful logging tools to track commit history.
- Use git log to view a detailed history of commits. To see a concise version, use git log --oneline. For specific details, filter logs by author (git log --author="Name") or date (git log --since="YYYY-MM-DD").
- To analyze changes line by line, use git blame <filename>, which shows the commit and author responsible for each line. For a graphical view, git log --graph represents branching and merging visually.

Cloning Repositories

- Cloning is the process of copying an entire Git repository, including its history, branches, tags, and files, to your local machine.
- This creates a fully functional Git repository that is connected to the original (remote) repository.

Steps to Clone a Repository:

1. Open your terminal or Git Bash.
2. Use the command.

   git clone <repository-URL>

Cloning a Local Repository

- Cloning a repository from your local file system allows you to duplicate an existing repository for experimentation or backup purposes.

Steps to Clone a Local Repository:

1. Navigate to the directory containing the original repository.
2. Run the command:
   git clone /path/to/local/repo /path/to/destination
3. Replace /path/to/local/repo with the path to the source repository and /path/to/destination with the directory where you want the clone to reside.

Cloning a Remote Repository over HTTPS

- When cloning a repository from a remote server like GitHub using HTTPS, you authenticate with your username and password or a personal access token.

Steps to Clone a Remote Repository Over HTTPS:

1. Go to the repository on GitHub (or another hosting platform).
2. Click the green **Code** button and copy the HTTPS URL.

   git clone https://github.com/<username>/<repository-name>.git
3. Enter your GitHub username and password (or token) when prompted

Forking a Repository

Forking is creating a copy of someone else's repository in your own GitHub account. This allows you to work independently while keeping the original project intact.

Steps to Fork a Repository:

1. Navigate to the repository you want to fork on GitHub.
2. Click the **Fork** button in the upper-right corner.
3. Once the fork is complete, clone your forked repository using the git clone command.

Example Workflow After Forking:

1. Clone the forked repository to your local machine

   git clone https://github.com/<your-username>/<repository-name>.git

2. Add the original repository (upstream) as a remote:
   git remote add upstream https://github.com/<original-owner>/<repository-name>.git
3. Sync your fork with the upstream repository
   git fetch upstream
   git merge upstream/main

Webhooks

- Webhooks are automated messages sent from a source repository to a specified endpoint when certain events occur, such as pushing commits or creating pull requests.
- They enable integration with external tools or services like CI/CD pipelines, Slack notifications, or custom applications.

Steps to Configure Webhooks on GitHub:

1. Go to your GitHub repository.
2. Navigate to **Settings > Webhooks**.
3. Click **Add webhook**.
4. Enter the payload URL where the webhook data should be sent. This is typically the endpoint of the service you're integrating with.
5. Choose the content type (usually application/json).
6. Select the events that trigger the webhook, such as:
   - Push events.
   - Pull request events.
   - Release events.

7. Add a secret key for security (optional but recommended).
8. Click **Add webhook** to save.

Push, Pull, and Tracking Remote Repositories

Tracking Remote Repositories

- When you clone a repository, Git automatically tracks the remote repository, typically named origin. You can manually add a remote using:

  git remote add origin https://github.com/username/repo.git

- To view the list of remotes, run:

  git remote -v

- Git fetches updates from the tracked remote with git fetch, but it does not merge changes into your local branch until you explicitly pull.

Pushing to Remote Repositories

- Pushing sends your committed changes to the remote repository.
  Command: git push origin main
- If the branch does not exist in the remote, use:

  git push --set-upstream origin <branch-name>

- This command tracks the local branch with the remote branch, allowing you to use just git push.

Pull Requests

- A pull request is a feature in GitHub and GitLab that facilitates code review and collaboration.
- It allows you to propose changes to a repository.
  Steps to create a pull request in GitHub:

1. Push your changes to a new branch in your forked repository.
2. Go to the original repository and click **Pull Requests**.
3. Click **New Pull Request** and choose your branch and the base branch.
4. Add a title, description, and reviewers before submitting.
   Pull requests allow maintainers to review and merge your changes into the main branch.

Managing Conflicts

Conflicts occur when changes in your branch and the branch you're merging or pulling from overlap.
Steps to resolve conflicts:

1. Identify conflicts using git status or by opening the conflicted files.
2. Manually edit the conflicted sections marked with.

   <<<<<<< HEAD

   Your changes

   =======

   Incoming changes

   >>>>>>>

4. Once resolved, stage the file
   git add <filename>
5. Commit the changes:
   git commit

Adding Users and Groups to GitLab

1. **Adding Users**:
   - Log in to GitLab as an administrator and go to **Admin Area > Users > New User**.
   - Enter the user's details and assign them roles.
2. **Creating Groups**:
   - Go to **Groups > Create Group**, name the group, and set visibility.
   - Add members to the group by navigating to **Group > Members > Invite Members**.

Creating and Managing Projects

1. Go to **Projects > Create New Project**.
2. Choose a blank project, template, or import an existing repository.
3. Assign permissions and visibility (public, internal, or private).

Push Changes and Merge with GitLab

1. Clone the repository: git clone https://gitlab.com/username/repo.git
2. Make changes locally and commit them.
3. Push changes:git push origin <branch-name>
4. Create a merge request in GitLab under **Merge Requests** to merge changes into the main branch.

GitHub Signed Commits

Signed commits ensure that commits are verified as coming from a trusted source.
Steps to create signed commits:

1. Generate a GPG key: gpg --full-generate-key
2. Add the GPG key to your GitHub account in **Settings > SSH and GPG keys**.
3. Configure Git to sign commits: git config --global user.signingkey <key-ID>

git config --global commit.gpgsign true

4. Create a signed commit: git commit -S -m "Signed commit message"

GitHub LFS (Large File Storage)

What is Git LFS?

- Git Large File Storage (LFS) is an extension for handling large files like multimedia or datasets in Git repositories.
- Instead of storing the large file contents in the repository.Git LFS replaces them with pointers and stores the files on a separate server.

Installation

1. Install Git LFS: git lfs install
2. Enable Git LFS in your repository: git lfs track "*.filetype"

   Example:

   git lfs track "*.jpg"

Including/Excluding Git LFS Files

- **Include Specific Files**:
  Use the git lfs track command to include specific file types or directories in .gitattributes.
- **Exclude Files**:
  Update .gitignore to exclude files or directories from being tracked by Git entirely.

Locking Git LFS Files

Git LFS allows file locking to prevent conflicts in binary files.

1. Lock a file: git lfs lock <filename>
2. Unlock a file:git lfs unlock <filename>
3. View locked files:git lfs locks

GitHub Administration

What is GitHub Administration?

- GitHub Administration involves managing repositories, teams, permissions, and workflows for efficient collaboration and security in GitHub.
- Administrators ensure that repositories are organized, access is controlled, and security practices are implemented.

Best Practices for Team-Level Administration

1. **Define Clear Roles**: Assign appropriate roles like maintainers, contributors, and reviewers to team members.
2. **Set Up Teams**: Use GitHub Teams to group users and assign repository permissions consistently.
3. **Enforce Branch Protection**: Require pull requests, code reviews, and status checks before merging.
4. **Use Labels and Milestones**: Organize issues and pull requests with labels, and track progress with milestones.
5. **Monitor Activity**: Regularly review activity logs to ensure proper usage.

Administration at Organization Level

1. **Create an Organization**: Organizations in GitHub allow multiple repositories under a shared structure with centralized access management.
2. **Centralize Policies**: Use organization-wide settings for authentication, permissions, and repository defaults.
3. **Enable SSO**: Single Sign-On (SSO) integrates with enterprise authentication systems.
4. **Automate Workflows**: Use GitHub Actions to automate tasks like testing and deployment.
5. **Audit Logs**: Regularly monitor organization activity through audit logs.

GitHub's Authentication Options

1. **Username and Password**: Basic authentication (deprecated for API access).
2. **Personal Access Tokens**: Replaces passwords for API and command-line authentication.
3. **SSH Keys**: Secure authentication for cloning and pushing via SSH.
4. **Single Sign-On (SSO)**: Centralized access using your organization's identity provider.
5. **OAuth Apps**: Third-party integration with GitHub for specific features.

Repository Permission Levels

GitHub offers fine-grained control over repository access:

- **Read**: Users can view code, issues, and pull requests.
- **Triage**: Users can manage issues and pull requests without write access.
- **Write**: Users can push to branches and create pull requests.
- **Maintain**: Users can manage repository settings, branches, and permissions.
- **Admin**: Full control over repository settings, including access and permissions.

Team Permission Levels

Team permissions apply to groups of users within an organization:

- **No Access**: No interaction with the repository.
- **Read Access**: View repository content.
- **Write Access**: Push code, manage issues, and submit pull requests.
- **Admin Access**: Manage repository settings, teams, and permissions.

Organization Permission Levels

Organization permissions define access across all repositories:

- **Owners**: Full control over the organization and its repositories.
- **Members**: Default access level to repositories, limited to team assignments.

Repository Security and Management

1. **Use Branch Protection Rules**: Require reviews and enforce status checks before merging.
2. **Enable Dependabot Alerts**: Automatically detect and notify about vulnerable dependencies.
3. **Control Access**: Use least privilege access principles for team members.
4. **Audit Logs**: Regularly monitor repository activity to detect anomalies.
5. **Secure Tokens**: Use GitHub Secrets to manage sensitive data like API keys and tokens.

Git Hooks

What Are Git Hooks?

- Git hooks are scripts that run automatically at specific points in the Git lifecycle.
- They can enforce rules, automate tasks, and trigger external workflows.
- Hooks are stored in the .git/hooks directory of a repository.

Examples of Git Hooks

1. **pre-commit:** Check the Commit Message for Spelling Errors
    - This hook runs before the commit message is finalized.
2. **pre-receive:** Enforce Project Coding Standards
    - This hook runs on the server before accepting a push.
3. **post-commit**: Notify Team Members of a New Commit
    - This hook runs after a commit is made.
4. **post-receive**: Push the Code to Production
    - This hook automates deployment after receiving code.

How to Use Git Hooks

1. Navigate to the .git/hooks directory in your repository.
2. Find the sample hook scripts (e.g., pre-commit.sample).
3. Rename the desired hook (e.g., pre-commit.sample to pre-commit).
4. Edit the file and add your custom script.
5. Test the hook by triggering the associated action (e.g., committing code).

Git Config: Command Introduction

- The git config command is used to configure Git settings and preferences.

- It controls everything from user information and editor configuration to enabling colored outputs and merge tools.

Syntax:

```
git config [options] [<key> [<value>]]
```

To read a value: `git config <key>`

To set a value: `git config <key> <value>`

Git Config Levels and Files

Git configuration is divided into three levels:

1. **System Level**: Applies to all users on the system.
   - File: /etc/gitconfig
   - Command: `git config --system`
2. **Global Level**: Applies to a specific user on the system.
   - File: ~/.gitconfig or ~/.config/git/config
   - Command: `git config --global`
3. **Local Level**: Applies only to a specific repository.
   - File: <repository>/.git/config
   - Command: `git config --local`

The **order of precedence** is **local > global > system**. For example, if a setting is configured at both the local and global levels, Git uses the local setting.

Writing a Value

You can write or update a Git configuration setting using: `git config --global <key> <value>`

Examples:

1. Set a user name: `git config --global user.name "Your Name"`
2. Set an email address: `git config --global user.email "your.email@example.com"`
3. Enable colored outputs: `git config --global color.ui true`.

Git Config Editor - core.editor

The core.editor configuration specifies the default text editor Git uses for writing commit messages and other operations.

Setting the Core Editor

1. **For Vim**: `git config --global core.editor "vim"`
2. **For Nano**: `git config --global core.editor "nano"`
3. **For VS Code**: `git config --global core.editor "code --wait"`.

Merge Tools and Configuring Them

Git supports several merge tools for resolving conflicts during merges.

Common Tools:

- kdiff3
- meld
- vimdiff
- tortoisemerge

Setting a Merge Tool

1. Install a merge tool (e.g., meld).
2. Configure it as the default merge tool: git config --global merge.tool meld

Configuring Diff and Merge Behavior

- Set the path to the merge tool:git config --global mergetool.meld.path "/usr/bin/meld"
- To use the merge tool for conflicts: git mergetool

Colored Outputs

Git can display colored outputs for improved readability.

Enabling Colored Outputs : git config --global color.ui auto
Customizing Colors:

You can customize the colors for specific Git commands.

1. Set the color for the diff command: git config --global color.diff.meta "blue bold"
2. Set the color for branch output: git config --global color.branch.current "yellow reverse"
3. Check the current configuration for color: git config --get color.ui.

Formatting and Whitespace

Control Formatting in Commit Messages

You can set guidelines for commit message length and style.

- Set the recommended line length to 72 characters:

  git config --global format.pretty "%h %s (%an)"

Handling Trailing Whitespace

Git can automatically highlight or remove trailing whitespace in code.

1. Highlight trailing whitespace in diffs: git config --global core.whitespace trailing-space
2. Remove trailing whitespace on commit: git config --global apply.whitespace fix

Tabs and Spaces

Set a tab width for code consistency: git config --global core.editor.tabwidth 4