GIT Documentation

What is GIT?

Git is a distributed version control system used to manage and track changes in source code during software development. It allows multiple developers to work on a project simultaneously without overwriting each other's changes.

What is the Version Control System in GIT?

Version control in Git involves tracking and managing changes to files in a project, enabling multiple developers to collaborate efficiently.

Here's a detailed look at how version control works in Git:

1. **Repositories**:
    - **Local Repository**: Each developer has a complete copy of the project, including its entire history, on their local machine.
    - **Remote Repository**: A central repository, often hosted on platforms like GitHub, GitLab, or Bitbucket, which serves as the main hub for collaboration.
2. **Commits**:
    - Changes are saved in the repository as commits, which are snapshots of the project at a specific point in time.
    - Each commit has a unique identifier (hash) and a commit message describing the changes.
3. **Branching**:
    - Git allows developers to create branches, which are separate lines of development. This enables working on features, bug fixes, or experiments in isolation from the main codebase (usually the `main` or `master` branch).
    - Branches can be merged back into the main branch once the work is complete and tested.
4. **Merging**:
    - The process of integrating changes from one branch into another. This can be done using merge commits, which combine the histories of the branches.
    - Git also supports rebasing, which replays changes from one branch onto another, creating a linear history.

5. **Pull Requests (or Merge Requests)**:
    - A feature provided by platforms like GitHub and GitLab to propose changes, review code, discuss modifications, and merge branches collaboratively.
6. **Staging Area (Index)**:
    - Before committing changes, files are placed in the staging area. This allows developers to prepare and review changes before creating a commit.
7. **History and Logging**:
    - Git keeps a detailed history of all commits, allowing developers to view the evolution of the project, revert to previous states, and understand who made specific changes and why.
8. **Distributed Nature**:
    - Since each developer has a full copy of the repository, they can work offline and have access to the entire history. This also provides redundancy and resilience.
9. **Conflict Resolution**:
    - When multiple changes conflict, Git provides tools to resolve these conflicts manually, ensuring a coherent codebase.
10. **Tags**:
    - Tags are used to mark specific points in the repository's history, often used to denote releases (e.g., v1.0, v2.0).

These features make Git a powerful and flexible tool for version control, supporting various workflows and collaboration strategies in software development.

GIT basic Commands:

1) Setup and Configuration:

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

Sets the username for commits.

```
git config --global user.email "your.email@example.com"
```

Sets the email for commits.

```
git config --list
```

Lists all the Git configurations.

2) **Repository Management:**

```
git init
```

Initializes a new Git repository in the current directory.

```
git clone <repository_url>
```

Clones an existing repository from a remote server to your local machine.

3) **Basic Operations:**

```
git status
```

Shows the current status of the working directory and staging area, indicating any changes.

```
git add <file>
```

Stages a specific file for the next commit.

```
git add .
```

Stages all changes in the working directory for the next commit.

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

Commits the staged changes with a descriptive message.

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

Stages and commits all changes to tracked files with a message.

4) **Viewing History:**

   `git log`**: Shows the commit history.**

   `git log --oneline`**: Shows a compact version of the commit history.**

5) **Branching and Merging:**

   `git branch`**: Lists all local branches in the repository.**

   `git branch <branch_name>`**: Creates a new branch.**

   `git checkout <branch_name>`**: Switches to the specified branch.**

   `git checkout -b <branch_name>`**: Creates and switches to a new branch.**

   `git merge <branch_name>`**: Merges the specified branch into the current branch.**

6) **Undoing Changes:**

   `git reset <file>`**: Unstages a file, keeping the changes in the working directory.**

   `git reset --hard`**: Resets the working directory and staging area to match the latest commit, discarding all changes.**

   `git revert <commit>`**: Creates a new commit that undoes the changes made by the specific commit.**

7) **Tags:**

   `git tag <tag_name>`: **Creates a new tag at the current commit.**

   `git tag`: **Lists all tags in the repository.**

8) **Advanced Branching and Merging:**

   `git branch -d <branch_name>`

   Deletes a local branch.

   `git branch -D <branch_name>`

   Forcefully delete a local branch (useful if the branch has unmerged changes).

   `git merge --no-ff <branch_name>`

   Merges the specified branch into the current branch without fast-forwarding, creating a merge commit.

   `git rebase <branch_name>`

   Replays commits from the current branch onto the specified branch, creating a linear history.

   `git cherry-pick <commit>`

   Applies the changes from a specific commit onto the current branch.

General Best Practices

1. **Commit Often, Commit Early:**
    - Make frequent commits with small, manageable changes. This makes it easier to track progress, find bugs, and understand the history of the project.

2. **Write Meaningful Commit Messages:**
    - Use clear and descriptive commit messages that explain the why, not just the what. This helps others (and your future self) understand the purpose of changes.

3. **Use Branches Effectively:**
    - Create branches for new features, bug fixes, or experiments. Keep the main branch (e.g., `main` or `master`) stable and deployable.
    - Use descriptive names for branches (e.g., `feature/add-login`, `bugfix/fix-typo`).

4. **Keep Your Branches Up-to-Date:**
    - Regularly pull changes from the main branch into your feature branches to avoid large merge conflicts later.

Handling Merges and Conflicts

5. **Merge Frequently:**
    - Merge your feature branch into the main branch frequently to minimize conflicts and ensure integration with other changes.

6. **Resolve Conflicts Carefully:**
    - Take your time to resolve conflicts manually. Understand both sets of changes before deciding how to integrate them.

Code Reviews and Collaboration

7. **Use Pull Requests (PRs):**
    - Use PRs for code reviews and discussions before merging changes into the main branch. This ensures code quality and collective code ownership.

8. **Review Code Thoroughly:**
    - Provide constructive feedback during code reviews. Check for functionality, readability, and adherence to coding standards.

Keeping the Repository Clean

9. **Remove Unused Branches:**
    - Regularly delete branches that have been merged to keep the repository clean and manageable.

10. **Ignore Unnecessary Files:**
    - Use a `.gitignore` file to exclude files and directories that shouldn't be tracked (e.g., build artifacts, temporary files).

Using Git Tools and Features

11. **Use Stashing for Temporary Changes:**
    - Use `git stash` to save uncommitted changes temporarily if you need to switch branches or pull updates.

12. **Rebase with Caution:**
    - While `git rebase` can create a cleaner history, it can also rewrite commit history. Avoid rebasing public branches that others are using.

13. **Tag Important Commits:**
    - Use tags to mark important commits, such as releases or milestones, for easy reference.

Handling Mistakes

14. **Undo Changes Safely:**
    - Use `git revert` to undo changes in a safe manner, creating new commits that reverse the changes. Use `git reset` for local changes that haven't been pushed yet.
15. **Check the Commit History Before Resetting:**
    - Before using `git reset`, use `git log` to understand the impact on the commit history.

Understanding Git Internals

16. **Learn Git Internals:**
    - Understanding how Git works under the hood (e.g., commit objects, trees, blobs) can help you troubleshoot issues and use advanced features more effectively.

Performance and Optimization

17. **Optimize Large Repositories:**
    - For large repositories, consider using tools and techniques like Git LFS (Large File Storage) to manage large files efficiently.

Documentation and Training

18. **Document Your Workflow:**
    - Maintain documentation for your team's Git workflow, including branching strategies, commit message conventions, and code review processes.

19. **Continuous Learning:**
    - Git is a powerful tool with many features. Invest time in learning advanced Git techniques and best practices.

By following these notes and best practices, you can make the most out of Git, ensuring efficient version control, collaboration, and project management.