Version Control Systems

Version Control Systems are software tools that help manage changes to source code over time. They keep track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Why VCS

1. Collaboration: Allows multiple people to work on the same codebase simultaneously without overwriting each other's work.
2. Track Changes: Keeps a comprehensive history of project changes, including who made what changes and when, providing a clear audit trail.
3. Revert and Compare: Enables the ability to revert to previous versions of a project, which is crucial for troubleshooting and understanding project evolution.
4. Branching and Merging: Facilitates concurrent development by allowing branching (creating another line of development) and merging (bringing different lines of development together).

Types of Version Control Systems

Centralized Version Control Systems (CVCS)

In CVCS, all files and historical data are stored on a central server. Developers check out files from that central place. Examples: Subversion (SVN) and Perforce.

Pros

- Simplicity: Easier to understand for beginners.
- Centralized Control: Administrators have fine-grained control over the repository.

Cons

- Single Point of Failure: If the central server goes down, no one can collaborate or save versioned changes.
- Limited Offline Capabilities: Requires connection to the central server for most operations.

Distributed Version Control Systems (DVCS)

In DVCS, each contributor has a complete copy of the entire repository, including its history, on their local machine. Examples: Git and Mercurial.

Pros

- Full Repository Backup: Every checkout is a full backup of the repository.
- Enhanced Collaboration: Enables more complex workflows and offline work.

Cons

- Complexity: Can be more complex to understand and manage, especially for beginners.

- Larger Repository Size: As each clone is a full copy of the repository, it can take up more space.

Git

Git is a version control system that allows developers to track changes made to their codebase over time. It was created in 2005 by Linus Torvalds, the founder of Linux, as a more powerful and efficient alternative to existing version control systems.

Git Init

The git init command is a fundamental tool in Git, used to initialize a new Git repository or reinitialize an existing one.

Purpose and Functionality

- Initialization of New Repositories: git init is used to start a new repository. When you run this command in a new directory, it creates a new Git repository in that directory.
- Reinitializing Existing Repositories: It can also be used to reinitialize an existing repository with additional options or to repair a broken repository.

Usage

Syntax:

```
git init [repository name]
```

This creates a new subdirectory named .git that houses all of your necessary repository files. This .git directory is what makes your directory a Git repository and tracks the history and configuration.

Inside an Existing Directory:
If you have a project directory already, navigate into it and run git init. This transforms the current directory into a Git repository.

Creating a New Directory for Repository:
You can also create a new directory and initialize it as a Git repository in one step by specifying the repository name: git init new-repo, where new-repo is the name of your new directory.

Staging Area in Git

The staging area is a layer between the working directory and the repository. It's a place where Git stores changes that you indicate (via git add) are ready to be committed to the repository.

Purpose

1. Organizing Commits: It allows you to craft your commits with precision. You can choose exactly which changes make up a commit.
2. Partial Commits: This area facilitates partial commits, enabling you to stage only certain parts of the changes made in the working directory.

How It Works
When you make changes in your working directory, Git sees them as "modified." Using `git add`, these changes move to the staging area.

Once staged, changes are marked as "staged" and are ready to be committed.
The `git status` command is useful to see which files are in which state (modified, staged, etc.).

Git Add Command

Git add adds changes in the working directory to the staging area, preparing them to be included in the next commit. This allows developers to choose which changes to commit.

Syntax:

```
git add [file or directory]
```

Adding a Single File: To stage a specific file, use `git add filename`.
Adding Multiple Files: You can add multiple files by listing them: `git add file1 file2`.
Adding All Changes: To add all new and modified files, use `git add .` or `git add -A`.
This stages all changes in the current directory and its subdirectories.

Git Commit

When you run git commit, Git takes the changes that are in the staging area and saves them into the repository. When you make a commit, it is added to the end of the current latest commit.

Syntax:

```
git commit -m "Your message here"
```

Key Features
1. Immutability: Each commit is a permanent snapshot of the repository at a given point in time. This snapshot includes references to all the files as they existed at that moment.
2. Unique Identifier: Each commit is identified by a unique SHA-1 hash. This allows tracking and referencing specific commits in the repository history.

Best Practices
1. Logical Changes: Commit related changes together. Avoid including unrelated changes in the same commit.

2. Frequent Commits: Commit often to capture stages of development. Smaller, more frequent commits make it easier to identify where and when changes were made.
3. Testing Before Committing: Ensure your code works as expected before committing. This helps in maintaining a stable project history.

Understanding Commits

1. Commits in History: Commits are a part of the branch history. When you change branches, the commits that are part of that branch's history become the current state of the project.
2. Relationship with Branches: When you make a commit, it is added to the end of the current branch. This commit then becomes the latest snapshot on that branch.
3. Immutability of a Git Commit

Immutable Nature:

Once a commit is created in Git, it becomes an immutable snapshot of the repository at a certain point in time. This means that the commit itself cannot be altered or changed without generating a new commit.

How Immutability is Achieved: Each commit is identified by a unique SHA-1 hash, which is generated based on the content of the commit (including changes, author information, date, and commit message). Changing any aspect of the commit would result in a different hash, thus creating a new commit altogether.

Implications of Immutability

1. Data Integrity: The immutability of commits ensures data integrity and traceability within the repository. It allows Git to detect changes or corruption in the repository history.
2. Traceability and Accountability: The immutable nature of commits allows for a reliable and traceable history of changes, which is crucial for collaboration and version control.
3. Amending Commits: While you can't change a commit once it's made, Git allows you to "amend" a commit. However, amending actually creates a new commit with a new SHA-1 hash, replacing the old commit in the project history.

Contents of a Git Commit

1. Snapshot of Changes: Contrary to a common misconception, Git does not store the complete snapshot of the entire repository in every commit. Instead, it stores a snapshot of the changes (delta) made since the last commit. This approach is more space-efficient.
2. Parent Commit(s): Each commit stores references to its parent commit(s). In the case of a merge commit, there will be multiple parents.
3. Author Information: Commits include the name and email address of the author who made the changes.
4. Timestamp: The date and time at which the commit was made.
5. Commit Message: A description provided by the author, explaining the changes made in the commit.

Git Branching

Multiple developers might be working on independent features at the same time. The primary purpose of branching is to diverge from the main line of development and continue to work independently without affecting the main line. Once the independent work is finished, you can converge the branches (merging back into the main branch). Branches serve as pointers to the latest commit. When you make a new commit, the branch pointer moves forward automatically. Creating a new branch in Git is a very quick and simple process as it is just creating a variable name.

Creating a New Branch: Use `git branch [branch-name]` to create a new branch.
Switching Branches: Use `git checkout [branch-name]` to switch to an existing branch.
Create and switch to a New Branch: `git checkout -b [branch-name]`

Branching Strategies

1. Feature Branching: Creating a branch for each new feature or bug fix keeps the work organized and separated until it's ready to be merged.
2. Release Branching: This involves creating a branch for release preparation, allowing for bug fixes and preparation without disrupting the main development work.
3. Long-Running Branches: Typically, a project has long-running branches like master or develop, which are stable versions of the project at different stages.

Best Practices

1. Regular Commits and Mergers: Regularly commit your changes within your branch and merge frequently from the main branch to minimize merge conflicts.
2. Naming Conventions: Use clear, descriptive names for branches to easily identify their purpose.
3. Cleanup: Delete branches after their changes have been merged to keep the repository clean and manageable.

Advanced Branching Commands

1. Branch Deletion: `git branch -d [branch-name]` deletes a branch.
2. Renaming Branches: `git branch -m [old-name] [new-name]` renames a branch.
3. Viewing Branches: `git branch` lists all branches.
4. Git Log: `git log --graph` provides a visual representation of the branching and merging history.

Git Merge

Merging in Git is a way to combine changes from different branches. It's like taking the work from one branch and adding it to another. For example, you might develop a new feature in a separate branch and then merge it into the main branch once it's ready.

How to Merge

1. Choose the Target Branch: First, switch to the branch where you want the changes to go. For example, `git checkout main`.

2. Merge Another Branch: Then, use `git merge [branch-name]` to merge changes from [branch-name] into your current branch.

Dealing with Merge Conflicts

Sometimes, Git can't automatically combine changes from different branches. This happens if the same part of a file was changed in both branches. This is called a conflict. Git will mark the conflicting areas in your files. You'll need to open these files and decide how to combine the changes. After editing, save the file. Once you've resolved the conflicts, use git add [file-name] to mark them as resolved, then make a commit to complete the merge.

Fast-Forward Merge and a Three-Way Merge

Imagine you're working on a project with a main branch, and you decide to work on a new feature. You create a new branch where you can make changes without affecting the main road.

Fast-Forward Merge

A fast-forward merge in Git is like having a straight road from your feature branch back to the main road. This occurs when there have been no new changes on the main branch since you started your feature branch. If nothing new has been added there, git can simply move the end of the main branch up to the end of your feature branch. Your changes are now part of the main branch. There won't be merge conflicts in this case.

Three-Way Merge

A three-way merge happens when there have been other changes on the main branch while you were working on your feature. In this case, your side branch doesn't directly fit onto the end of the main branch anymore. To merge your changes with the main branch, Git uses a special process. It looks at three points: the end of your feature branch, the end of the main branch, and where both branches were last the same. Git then figures out how to combine these changes. In this case, there can be merge conflicts that you will have to resolve.

Squash Merge

Squash merging is a way to combine all the changes from a feature branch into a single commit when you merge it into another branch. It's useful for keeping your history clean and organized. Instead of having many small commits from a feature branch, you have one concise commit.

Syntax: `Use git merge --squash [feature-branch]`. This combines all changes into a single commit. After this, you'll need to commit these changes manually with git commit.

Best Practices for Merge and Conflict Resolution

1. Regular Merges: Frequently merge changes from the main branch into your feature branch. This reduces the chances of conflicts.
2. Clear Commit Messages: When you complete a merge or resolve a conflict, write a clear commit message explaining what you did.
3. Test Before Merging: Always test your code to make sure it works correctly before and after merging.

Git Rebase

In Git, rebase is a command that helps to move or combine a sequence of commits to a new base commit. It's a way to rearrange the history of your commits for a cleaner, more linear progression.

<u>Why Rebase</u>

1. Clean History: Rebase is often used to tidy up a series of commits before merging them into a main branch. It can make the history more readable and easier to understand.
2. Integrating Changes: Rebasing can also be used to integrate changes from one branch into another, similar to merging but with a different approach.

<u>How Rebase Works</u>

1. Switch to Your Feature Branch: Start by being in the branch with your new work, for example, `git checkout feature-branch`.
2. Start the Rebase: Use `git rebase [base-branch]`, where [base-branch] is the branch you want to rebase onto (often main or master). Git Re-applies Commits: Git takes the commits from your feature branch and re-applies them on top of the latest commit in the base branch. This makes it look like your changes were made on top of the current state of the base branch.

<u>Conflict Resolution in Rebase</u>

1. Handling Conflicts: If Git encounters conflicts during rebase, it will pause and allow you to resolve them. This is similar to resolving conflicts in a merge.
2. Resolving and Continuing: After fixing the conflicts in a file, use `git add [file-name]` to mark it as resolved. Then, continue the rebase with `git rebase --continue`.

<u>Benefits of Rebase</u>

1. Linear History: Rebasing creates a linear project history, which can be easier to follow than the branched history created by merging.
2. Avoiding Extra Merge Commits: It eliminates the need for an extra merge commit that you get when using git merge.

<u>Best Practices for Rebase</u>

1. Not on Public Branches: Avoid rebasing commits that are already on public branches. Rewriting history in this way can cause problems for others working on the same repository.
2. Use in Local Development: Rebase is best used for cleaning up your local commits before integrating them into a shared branch.

Difference Between Merge and Rebase

<u>Merge:</u> Keeps the original history and adds a new commit that merges two branches.

Rebase: Moves the entire branch to start from a different point, creating a straight, linear history.

HEAD in Git

In Git, HEAD is a reference to the current or last commit in the currently checked-out branch. It can be thought of as a pointer to the current branch or current state of your repository. As you commit, the HEAD moves forward with each commit. It always represents the latest commit in the branch.

Detached HEAD State: When you check out a specific commit instead of a branch, Git enters a 'detached HEAD' state. In this state, HEAD points directly to a commit, rather than a branch. Changes made in this state aren't attached to a branch, so they can be difficult to find after switching back to a branch. It's useful only for temporary exploration or read-only purposes in a repository. It's generally a good idea to avoid making permanent changes while in a detached HEAD state, as these changes can be lost when switching back to a branch.

HEAD in Branches: When you switch branches with git checkout [branch-name], HEAD updates to point to the tip of the new branch. If you create a new commit on this branch, HEAD moves forward with the new commit.

Git Remotes

Git remotes are like offsite backups or mirrors of your local Git repository. They allow you to collaborate with others, share your code, and work on different versions of your project simultaneously. Think of them as remote locations where your Git repository lives. It could be:

1. A remote Git server: like GitHub, GitLab, or Bitbucket.
2. Another developer's local machine.

Setting Up a Remote Repository

1. Copy the URL of the remote repository. It's often in the format https://github.com/username/repository.git.
2. Clone the remote repository to your local machine, use: `git clone <remote-repository-URL>`

This creates a local copy of the repository on your computer, including all files and history.

Linking a Local Repository to a Remote

Initialize Local Repository: If you already have a local repository, initialize it with:

```
git remote add <remote-name> <remote-repository-URL>
```

List all remotes

```
git remote -v
```

Fetch Command

To fetch changes from the remote repository without merging them:

```
git fetch <remote-name>
```

Pull Command

To update your local repository with changes from the remote:

```
git pull origin master
```

Git attempts to automatically merge changes. If there's a conflict, you'll need to resolve it manually.

Internally, git pull starts with git fetch. This command retrieves commits, files, and refs from a remote repository into your local repo.Following the fetch, git pull executes git merge to merge the retrieved branch heads into the current local branch. As it executes git merge, pull can create a new merge commit as well. Optionally, git pull can rebase instead of merging. This is done using `git pull --rebase` This is often a recommended approach.

Push Command

Push your changes to the remote repository:

```
git push origin master
```

Delete Remote Branch

To delete a branch on the remote:

```
git push origin --delete <branch-name>
```

Tips and Best Practices

1. Regular Pulls: Regularly pull from the remote to stay updated.
2. Push Small Changes: Push small, incremental changes rather than large, infrequent updates.

Working on Github

Forking Repository

Forking is creating your own copy of a repository. This allows you to freely experiment with changes without affecting the original project. On the GitHub page of the repository, click the "Fork" button. This creates a copy under your GitHub account. Now clone this repository into your local. Use the command `git clone [URL of your fork]`. This URL is found on your fork's GitHub page.

Syncing with Original Repository

To ensure you have the latest changes from the original repository, add the original repository as a remote source: `git remote add upstream [URL of original repository]`. Whenever you want to fetch the changes, use the command: `git fetch upstream` and then merge the changes into your local branch: `git merge upstream/main`.

Creating a New Branch

It's a best practice to create a new branch for your changes. This keeps your contributions organized and separated from the main project. Use `git checkout -b [new-branch-name]` to create a new branch. Make your intended changes in the code.

Pushing Changes to GitHub

Use `git push origin [your-branch-name]` to push your changes to your fork on GitHub.

Creating a Pull Request (PR)

A PR is a way to propose your changes to the original repository. On your fork's GitHub page, select your branch and click "New pull request". Add details about your changes and submit the PR.

Making Further Changes

Maintainers might ask for changes. Make these changes in your local branch. After committing these changes, push them to GitHub. The PR will automatically update. Ensure you keep rebasing your repository to upstream.

Final Merging

Once your PR is approved, the maintainers of the original repository will merge your changes. Congratulations, your contributions are now part of the project!

Git Cherry-Pick

Git cherry-pick is a powerful tool that allows you to select and apply individual commits from one branch into another. Unlike merging or rebasing, which typically apply a series of commits, cherry-picking applies only the commits you specify.

When to Use Cherry-Pick?

1. Selective Changes: When you want to apply specific changes from one branch to another without bringing in the entire branch history.
2. Bug Fixes and Feature Backporting: For applying bug fixes or features from the main branch to a release or maintenance branch.

How to Cherry-Pick?

1. Identify the Commit Hash: First, you need the hash of the commit you want to cherry-pick.

2. Switch to the Target Branch: Use `git checkout [target-branch]` to switch to the branch where you want to apply the commit.
3. Cherry-Pick the Commit: Use git cherry-pick [commit-hash]. If the commit applies cleanly, it will be added to your branch. If there’s a conflict, Git will pause the cherry-pick operation. Manually resolve the conflicts in your code editor. After resolving, add the files with `git add [file-name]`. Complete the cherry-pick with `git cherry-pick --continue`.
4. Push the Changes: Use `git push origin [target-branch]` to push the changes to the remote repository.

Best Practices and Tips

1. Commit Isolation: Cherry-pick works best with commits that are isolated in their changes (i.e., the commit doesn’t contain a mix of unrelated changes).
2. Testing: Always test your changes after a cherry-pick to ensure functionality hasn’t been compromised.
3. Avoid Cherry-Picking Over Regular Merges: Cherry-pick is a useful tool, but it's not a substitute for regular merging or rebasing.