Version Control System

* Need For Version Control System

Here are some of the key reasons why VCS is essential:

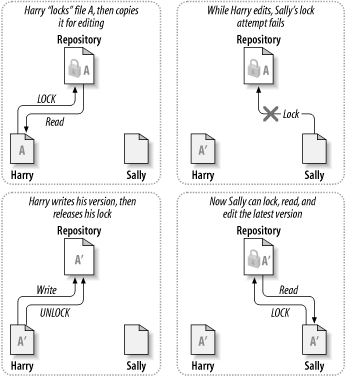
* History and Track Changes:
  1. Records and tracks modifications made to the project.
  2. Provides information on who, when, and what changes were made.
  3. Enables reverting to previous versions and comparing changes.
* Collaboration and Teamwork:
  1. Allows multiple developers to work on the same project simultaneously.
  2. Facilitates merging changes from different team members.
  3. Assists in resolving conflicts that may occur during collaboration.
* Backup and Recovery:
  1. Acts as a backup system by creating multiple copies of the project.
  2. Protects against accidental deletions or hardware failures.
  3. Enables restoration of previous versions from the VCS repository.
* Experimentation and Branching:
  1. Creates independent branches for experimentation and feature development.
  2. Isolates experimental changes from the main codebase.
  3. Allows easy discarding of branches that don't work out.
* Code Reviews and Quality Control:
  1. Facilitates code reviews and feedback from team members.
  2. Helps maintain code quality and adherence to coding standards.
  3. Encourages collaboration and knowledge sharing within the team.
* Deployment and Release Management:
  1. Integrates with deployment and release processes.
  2. Enables tagging specific versions as releases.
  3. Simplifies tracking and deploying specific versions to production.
* Traceability and Auditing:
  1. Captures detailed information about changes made to the codebase.
  2. Provides traceability for auditing purposes.
  3. Helps understand who made changes, why, and when.
* Version Control System Goals  
   VCS Or Version Control Systems (VCS) are crucial tools in software development and collaborative projects. It Satisfy Following Goals i.e. A version Control System Should Have Following Characteristics

1. Backup & Restore
2. Synchronization
3. Undo
4. Track Changes And Ownership
5. Sandboxing
6. Branching

* Version Control System Category
* Centralized VCS

1. In a CVCS, there is a central server that stores the entire repository.
2. Developers have working copies on their local machines.
3. Changes made by developers are directly committed to the central server.
4. Examples: CVS (Concurrent Versions System), Subversion (SVN).

* Lock Modify Unlock

 "Lock Modify Unlock" is a concept or workflow commonly associated with Centralized Version Control Systems (CVCS) that allows exclusive access to a file or set of files.  
1. Lock:

* + Before making any modifications to a file, a developer requests a lock on the file they intend to work on.
  + The lock ensures that other developers cannot make concurrent changes to the same file, preventing conflicts.

1. Modify:
   * Once the lock is obtained, the developer has exclusive write access to the file.
   * They can freely make modifications, such as adding, modifying, or deleting content within the file.
2. Unlock:
   * After completing the modifications, the developer releases the lock on the file.
   * Releasing the lock allows other developers to obtain the lock and make their own changes.

The purpose of the "Lock Modify Unlock" workflow is to ensure that only one developer can modify a file at a given time, reducing the likelihood of conflicts that may arise when multiple developers make simultaneous changes to the same file. This approach is more commonly used in older CVCS systems, where central servers maintain a single repository and control access to files.

* Decentralized VCS

1. In a DVCS, each developer has a complete copy of the repository, including the entire history.
2. Developers can work offline and have the full history available locally.
3. Changes are committed to the local repository first, and then they can be synchronized with other repositories.
4. Examples: Git, Mercurial, Bazaar.

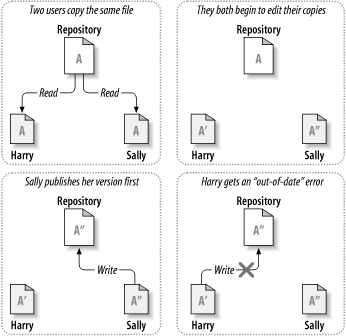
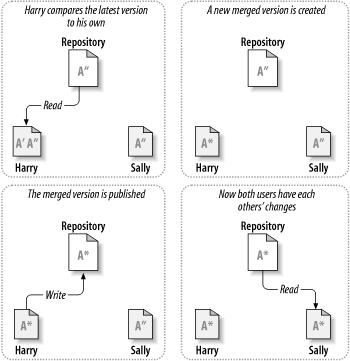
* Copy Modify Merge

Fig 1 Fig 2  
Copy Modify Merge (CMM) is a version control system (VCS) strategy that allows multiple developers to work concurrently on the same set of files without locking the files exclusively. In the CMM approach, each developer creates a personal copy of the files they are working on, makes modifications independently, and then merges their changes back into the main repository.

Here's how the Copy Modify Merge strategy works:

1. Copy: Each developer creates their own personal copy (also known as a working copy or private workspace) of the files they intend to modify. This copy is separate from the main repository.
2. Modify: Developers can freely make changes to their personal copies of the files without any restrictions or exclusive locks. They can add, modify, or delete code as needed. The modifications are made in isolation from other developers.
3. Merge: Once developers have completed their modifications, they merge their changes back into the main repository. The merge process compares the changes made in the personal copies with the latest version of the files in the main repository and applies the modifications while trying to preserve as much of the original content as possible.

The merge process in CMM is responsible for combining the changes made by different developers into a cohesive version. It examines the modifications made to each file, resolves any conflicts that arise when two or more developers have made conflicting changes to the same portion of a file, and produces a merged version that includes all the changes.

The Copy Modify Merge strategy allows for parallel development, as developers can work independently and make modifications concurrently. It avoids the need for exclusive locks that would prevent others from accessing or modifying the files. However, it relies on effective merge tools and practices to ensure that conflicts are resolved correctly and that the merged code remains coherent and functional.

* Git

Git is a distributed version control system (DVCS) widely used for managing source code and tracking changes in software development projects. It was created by Linus Torvalds, the creator of the Linux operating system, in 2005. Git is designed to be fast, efficient, and capable of handling both small and large-scale projects with ease.

* Key Features of **Git**:
  + Distributed Model:
    - Developers have their own local copies of the entire repository.
    - Work offline and synchronize changes later.
  + Commit-Based System:
    - Tracks changes using commits representing snapshots of files.
    - Each commit has a unique identifier, author, timestamp, and message.
  + Branching and Merging:
    - Supports creating multiple branches for isolated development.
    - Branches can be merged back into the main branch later.
  + Lightweight and Fast:
    - Designed to be efficient and perform well, even for large projects.
  + Collaboration and Remote Repositories:
    - Supports collaboration through remote repositories.
    - Platforms like GitHub, GitLab, or Bitbucket host remote repositories.
  + Branch Protection and Access Control:
    - Administrators can define rules and restrictions for branches.
    - Access control ensures code review and security.
  + Integration with CI/CD and DevOps:
    - Integrates with CI/CD pipelines and DevOps practices.
    - Triggers automated build, test, and deployment processes.
  + Extensibility and Community Support:
    - Large community with extensions, plugins, and integrations.
* Benefits of Git:
  + Enables efficient collaboration and code sharing.
  + Provides powerful branching and merging capabilities.
  + Lightweight and fast performance.
  + Integrates well with CI/CD and DevOps.
  + Extensible and supported by a vibrant community.
* Platforms and Tools:
  + Popular platforms: GitHub, GitLab, Bitbucket.
  + Command-line interface (CLI) and graphical user interface (GUI) tools available.
  + Integrated with numerous software development tools and workflows.
* Installation of Git
* Installation
  + Windows:
    - Visit the official Git website: <https://git-scm.com/>
    - Download the latest Git installer for Windows.
    - Run the installer and follow the prompts.
    - Choose the desired installation options, such as the installation location and components to install.
    - Select appropriate settings (e.g., adjusting the PATH environment variable).
    - Complete the installation process.
  + macOS:
    - Git is typically pre-installed on macOS. Open Terminal (Command Line).
    - Check if Git is installed by running the command: git --version.
    - If Git is not installed, you will be prompted to install the Xcode Command Line Tools. Follow the instructions to install Git.
  + Linux (Ubuntu):
    - Open the Terminal.
    - Run the command: sudo apt update to update the package lists.
    - Run the command: sudo apt install git to install Git.
    - Verify the installation by running: git --version.
* Configuration:
  + Open a command line interface (Terminal, Command Prompt, or Git Bash).
  + Configure your identity:
    - Set your name: git config --global user.name "Your Name"
    - Set your email: git config --global user.email "youremail@example.com"
  + (Optional) Configure default text editor:
    - Set a default text editor for Git commits (e.g., for Windows, you can use Notepad++): git config --global core.editor "notepad++"
  + (Optional) Set default branch name:
    - Git 2.28 and later versions allow setting the default branch name to main instead of master: git config --global init.defaultBranch main
  + (Optional) Set other preferences:
    - Explore other Git configuration options based on your preferences or project requirements, such as aliases, colors, and merge strategies. You can use the git config command with various options to set these configurations.
* Verification:
  + To verify your Git installation and configuration, run the following commands in the command line interface:
    - git --version (should display the installed Git version)
    - git config --list (should display your configured Git settings)
* What Git Installation Does To Windows Os?
* Git installer provides an option to choose the default shell for Git commands.
* By default, it offers the option to use Git Bash, which is a customized version of the Bash shell specifically designed for Git on Windows.
* Choosing Git Bash ensures a more seamless and Unix-like experience when working with Git commands.
* Need Of SSH Key In Git.

SSH keys are used in Git for secure authentication and communication between your local machine and remote Git repositories. They serve the following needs:

1. Secure Authentication:
   * When you interact with a remote Git repository, such as pushing or pulling changes, Git uses SSH (Secure Shell) protocol for authentication.
   * SSH keys provide a secure and passwordless authentication method, ensuring that only authorized users can access the repository.
2. Enhanced Security:
   * SSH keys use public-key cryptography, where a pair of keys is generated: a public key and a private key.
   * The public key is shared with the Git server, while the private key remains securely stored on your local machine.
   * The private key acts as a digital signature, proving that you are the authorized user without transmitting sensitive information like passwords.
   * The keys are mathematically related, making it computationally infeasible for someone to derive the private key from the public key.
3. Convenience and Efficiency:
   * SSH keys eliminate the need for entering a password each time you interact with a Git repository.
   * Once the SSH key pair is set up, your local machine can automatically authenticate with the remote repository, making the workflow smoother and more efficient.
4. Multiple Remote Repositories:
   * SSH keys allow you to work with multiple remote repositories, as you can configure different SSH keys for different Git services or accounts.
   * This flexibility enables you to manage multiple projects or collaborate with different teams, each with their own Git repositories.
5. Integration with Git Hosting Platforms:
   * Many Git hosting platforms, such as GitHub, GitLab, and Bitbucket, support SSH key-based authentication.
   * By adding your SSH public key to your account on these platforms, you gain secure access to your repositories, whether for cloning, pushing, or pulling changes.

* Difference Between Git & GitHUB

| **Git** | **GitHub** |
| --- | --- |
| - Distributed version control system | - Web-based hosting service for Git repositories |
| - Local repository maintenance | - Remote repository hosting and collaboration |
| - Command-line interface | - Web-based interface with additional features |
| - No built-in access control | - Provides access control and permission management |
| - No issue tracking or pull requests | - Integrated issue tracking and pull request system |
| - Manages source code and tracks changes | - Stores and hosts Git repositories in the cloud |
| - Provides branching and merging | - Offers advanced branch management capabilities |
| - Works locally on developer's machine | - Enables collaboration and code review |
| - Does not have a graphical interface | - Provides web-based graphical interface |
| - Can integrate with various tools | - Integrates with a wide range of third-party tools |
| - Can be used with any remote repository | - Hosts repositories on GitHub's servers |
| - No distinction between public/private | - Offers options for public and private repositories |
| - Open-source and free | - Offers free and paid plans for different features |

* Git Terminologies
  + Repository:
    - Collection of files and their complete history.
  + Local Repository:
    - Copy of a Git repository on your local machine.
  + Remote Repository:
    - Repository hosted on a remote server, often used for collaboration.
  + Clone:
    - Creating a local copy of a remote repository on your machine.
  + Pull:
    - Updating your local repository with the latest changes from a remote repository.
    - Fetching new commits and merging them into your current branch.
  + Push:
    - Sending your local commits to a remote repository.
    - Updating the remote repository with your changes.
  + Branch:
    - Parallel line of development in Git.
    - Used to work on different features or bug fixes independently.
  + Commit:
    - Snapshot of changes made to the repository.
    - Recording a specific set of modifications with a unique identifier and commit message.
  + Merge:
    - Combining changes from one branch into another.
    - Integrating changes to create a new commit.
  + Pull Request:
    - Proposing changes from your branch to be merged into another branch.
    - Facilitating code review, discussion, and collaboration before merging.
  + Fork:
    - Creating a personal copy of a repository under your account.
    - Allowing experimentation, changes, and contributions without affecting the original project.
  + Staging:
    - Preparing changes for a commit.
    - Selecting modified files or specific changes to include in the next commit.
  + Pull Conflict (Merge Conflict):
    - Conflicting modifications in the same file or lines of code.
    - Manual intervention required to resolve conflicts before merging.
* HEAD:
  + Pointer to the currently checked out commit or branch.
* Tag:
  + Named reference to a specific commit.
  + Used to mark important points in the commit history (e.g., releases).
* Remote:
  + A remote repository that is linked to the local repository.
  + Allows fetching and pushing changes to/from a different repository.
* Fetch:
  + Retrieving the latest changes from a remote repository.
  + Updating the local repository's remote-tracking branches.
* Reset:
  + Moving the current branch pointer to a specific commit.
  + Discarding or unstaging changes.
* Rebase:
  + Incorporating changes from one branch onto another.
  + Moving or replaying commits onto a different base commit.
* Cherry-pick:
  + Applying a specific commit to the current branch.
  + Selectively picking a commit from one branch and applying it elsewhere.
* Remote Tracking Branch:
  + A local copy of a remote branch.
  + Reflects the state of the branch in the remote repository.
* Diff:
  + Showing the differences between two commits, branches, or files.
* Git ignore:
  + A file that specifies intentionally untracked files and patterns to be ignored by Git.
* Submodule:
  + A separate Git repository embedded within a parent repository.
  + Allows including external code as a component of the main project.
* Blame/Annotate:
  + Showing who last modified each line of a file and in which commit.
* Fetch vs. Pull:
  + Fetch retrieves changes from a remote repository without merging.
  + Pull retrieves changes and merges them into the current branch.
* Commit Message:
  + Description or summary of the changes made in a commit.
  + Helps provide context and understanding of the modifications.
* Git GUI:
  + Graphical User Interface (GUI) tools for interacting with Git repositories.
  + Provides a visual representation of Git operations and repository history.
* Revert:
  + Creating a new commit that undoes the changes made in a previous commit.
* Bisect:
  + A binary search method to locate a specific commit where a bug was introduced.
* Stash:
  + Temporarily saving changes that are not ready to be committed.
  + Allows switching branches without committing unfinished work.
* Repository Forking:
  + Creating a copy of a repository to your own account or organization.
  + Enables independent development and experimentation.
* Pull Remote Branch:
  + Fetching changes from a specific branch in a remote repository and merging them locally.
* Upstream:
  + Refers to the original repository from which a fork was created.
* Squash:
  + Combining multiple commits into a single commit.
  + Helps keep the commit history clean and organized.
* Remote Branch:
  + A branch in the remote repository.
  + Represents a specific line of development in the remote repository.
* Fast-forward:
  + A type of merge where the branch being merged has no new commits since the divergence.
  + The branch pointer is simply moved forward to the latest commit.
* Detached HEAD:
  + A state where the HEAD points directly to a commit instead of a branch.
  + Changes made in this state can be lost if not properly handled.
* Git LFS (Large File Storage):
  + An extension for Git that allows managing large files efficiently.
  + Large files are stored outside the Git repository, reducing its size.
* Git Hooks:
  + Custom scripts that can be executed at specific Git events.
  + Examples include pre-commit hooks for code linting or post-receive hooks for notifications.
* Interactive Rebase:
  + Rewriting commit history by combining, modifying, or deleting commits interactively.
* Reflog (Reference Log):
  + A log that records all changes to Git references (branches, HEAD, etc.).
  + Helps recover lost commits or branches.
* Git Bisect:
  + A command that automatically performs a binary search to find the commit that introduced a bug.
* Git Commands

1. **git init:**  Initializes a new Git repository in the current directory.

Example: git init

1. **git clone:** Clones a remote repository and creates a local copy.

Example: git clone https://github.com/example/repo.git

1. **git add**: Adds files to the staging area for the next commit.

Example: git add file.txt

1. **git commit**: Commits changes to the repository with a descriptive message.

Example: git commit -m "Add new feature"

1. **git status**: Shows the current state of the repository, including tracked/untracked files.

Example: git status

1. **git diff**: Shows the differences between the working directory and the staging area.

Example: git diff

1. **git branch**: Lists existing branches or creates a new branch.

Example: git branch (to list branches)

git branch new-feature (to create a new branch)

1. **git checkout:** Switches to a different branch or restores files from a specific commit.

Example: git checkout branch-name (to switch branches),

git checkout commit-hash file.txt (to restore a file from a specific commit)

1. **git merge**: Merges changes from one branch into the current branch.

Example: git merge feature-branch

1. **git pull:** Fetches changes from a remote repository and merges them into the current branch.

Example: git pull origin master

1. **git push:** Pushes local commits to a remote repository.

Example: git push origin branch-name

1. **git remote:** Shows the remote repositories associated with the current repository.

Example: git remote -v

1. **git log:** Displays the commit history, including commit messages and authors.

Example: git log

1. **git reset:** Resets the current branch to a specific commit, discarding subsequent commits.

Example: git reset commit-hash

1. **git stash:** Temporarily saves changes that are not ready to be committed.

Example: git stash save "Work in progress"

1. **git fetch:** Fetches changes from a remote repository without merging them.

Example: git fetch origin

1. **git remote add:** Adds a new remote repository.

Example: git remote add origin https://github.com/example/repo.git

1. **git remote remove:** Removes a remote repository.

Example: git remote remove origin

1. **git tag:** Creates a lightweight tag or an annotated tag for a specific commit.

Example: git tag v1.0.0 (lightweight tag)

git tag -a v1.0.0 -m "Version 1.0.0" (annotated tag)

1. **git blame:** Shows who last modified each line of a file and in which commit.

Example: git blame file.txt

* Creating And Cloning a Repository

Creating a Repository:

1. Open a terminal or Git Bash on your local machine.
2. Navigate to the directory where you want to create the repository.
3. Use the following commands to initialize a new Git repository:

* git init

1. Optionally, create a README file and other files in the repository directory.
2. Use the following commands to add and commit the initial files:

* git add .
* git commit -m "Initial commit"

1. Go to the repository hosting platform (e.g., GitHub) and create a new repository.
2. Copy the repository URL (HTTPS or SSH) provided by the hosting platform.

Cloning a Repository:

1. Open a terminal or Git Bash on your local machine.
2. Navigate to the directory where you want to clone the repository.
3. Use the following command to clone the repository:
   * git clone <repository-url>
4. Replace <repository-url> with the URL of the repository you want to clone.
5. Press Enter to execute the git clone command.
6. Git will create a new directory with the repository name and clone the repository into it.

After cloning the repository, you can make changes, create branches, commit your changes, and perform other Git operations as needed. Remember to use git add, git commit, and git push to save and push your changes to the remote repository.

* Pull From Remote

When you run **git pull origin branch-name** (assuming "origin" is the remote repository and "branch-name" is the branch you want to pull from), the following actions take place:

1. Fetching: The first step of **git pull** is to fetch the latest changes from the remote repository. Git contacts the remote repository specified in the configuration (usually "origin") and retrieves the latest commit history, branches, and tags. It does not automatically merge these changes into your local branch.
2. Comparing and Updating: After fetching the changes, Git compares the fetched commits to your local branch's commit history. It determines the commits that are different or new in the remote repository compared to your local branch.
3. Merging or Rebasing: Once Git identifies the diverging commits, it determines how to incorporate them into your local branch. There are two possible scenarios:

a. Merge: If your local branch has its own commits that are not in the remote repository, Git performs a merge operation. It creates a new commit that combines the changes from both the remote and local branches. If there are no conflicts, the merge is completed automatically.

b. Rebase: If your local branch does not have any commits that are not in the remote repository, Git performs a rebase operation. It applies the new commits from the remote repository on top of your local branch, replaying your local commits on the updated base. This results in a linear commit history.

1. Conflict Resolution (if applicable): In case there are conflicting changes between the remote and local branches (i.e., changes to the same lines of code), Git halts the process and notifies you about the conflicts. You need to manually resolve these conflicts by editing the conflicting files and selecting the desired changes.
2. Committing and Updating: After resolving conflicts (if any), Git creates a new merge commit (in the case of a merge) or updates the commit references (in the case of a rebase) to finalize the integration of the remote changes into your local branch.
3. Fast-Forward (if applicable): If the remote changes can be applied directly on top of your local branch (i.e., there are no diverging commits), Git performs a fast-forward merge. In this case, the branch pointer is simply moved forward to the latest commit from the remote branch.
4. Pushing (optional): Once the **git pull** operation is complete, you may need to push the merged or rebased commits from your local branch to the remote repository using git push if you want to update the remote repository with your local changes.

Overall, the git pull command combines the **git fetch** and **git merge** (or **git rebase**) operations to fetch the latest changes from the remote repository and integrate them into your local branch.

* Git GUI And gitk Commands

 Git GUI:

1. Git GUI is a cross-platform graphical tool that provides a user-friendly interface for performing various Git operations.
2. It allows users to perform Git actions such as initializing a repository, making commits, creating branches, merging branches, and pushing changes.
3. Git GUI provides a visual representation of the repository's commit history, branches, and tags.
4. Users can view file differences, stage changes, and resolve conflicts through the GUI.
5. It also offers additional features like browsing and searching commit messages, creating and applying patches, and managing remote repositories.

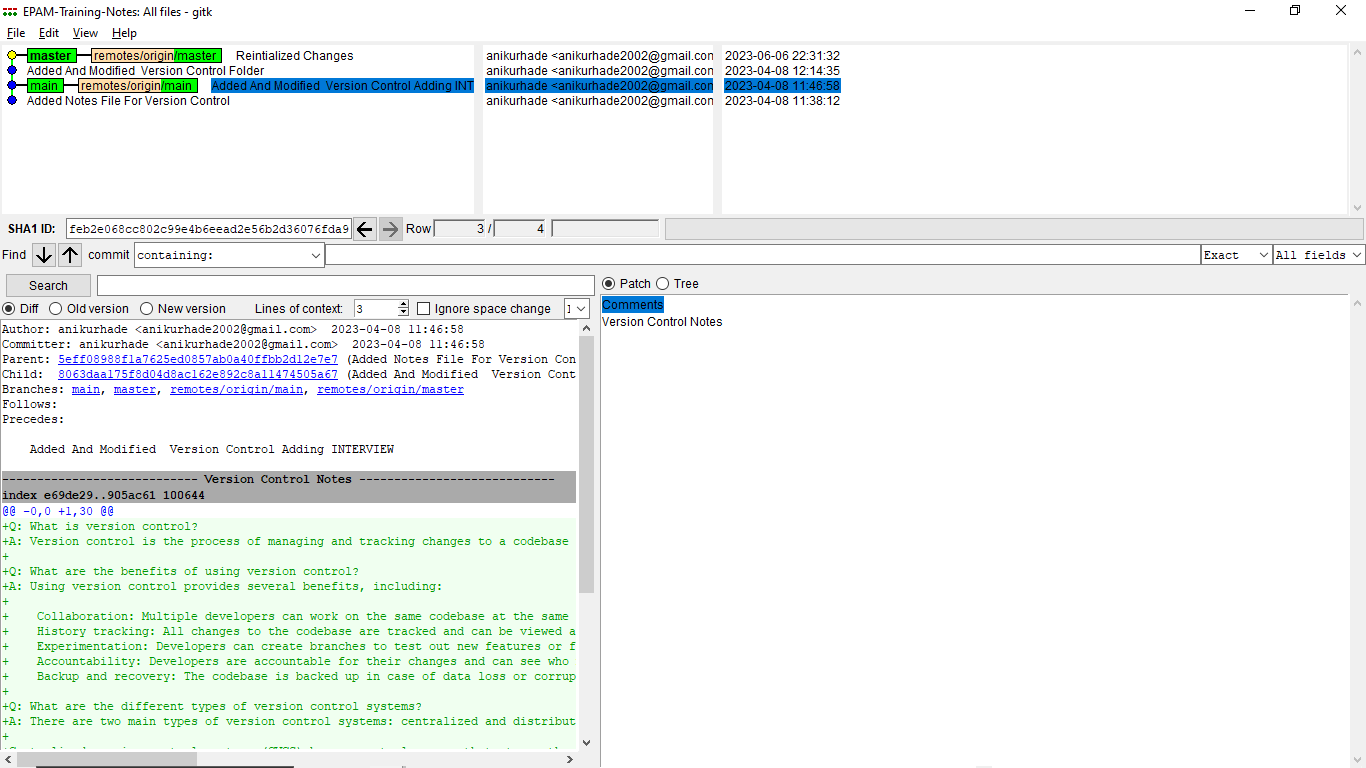
A screenshot of a computer

Description automatically generated

git gui& 👆👆

 gitk:

1. gitk is a lightweight graphical viewer for Git repositories.
2. It provides a simple and intuitive interface for visualizing the commit history of a repository.
3. Users can view the commit graph, branches, tags, and individual commits in a graphical tree format.
4. gitk allows users to browse commit details, including author, timestamp, and commit message.
5. It supports filtering and searching for specific commits based on different criteria.
6. Users can compare file changes between different commits and view the differences using a built-in diff viewer.
7. gitk does not provide all the Git functionality like making commits or pushing changes. It is primarily focused on visualizing the commit history.

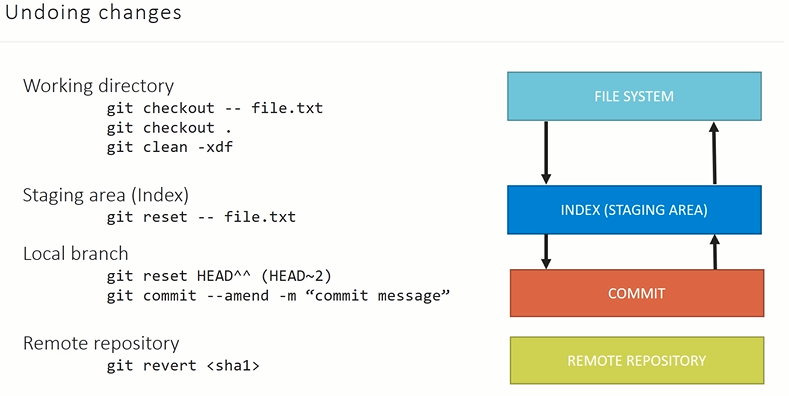


gitk👆👆

* Inside .git Folder

The **.git** folder is a hidden directory at the root of a Git repository that contains all the necessary information for version control. It is responsible for tracking changes, maintaining commit history, and managing various configurations. Here is an overview of the folders and subfolders typically found inside the **.git** directory:

1. .**git/branches**: Contains files related to branch references, storing pointers to specific commits.
2. **.git/hooks:** Contains customizable scripts that can be executed in response to certain actions, such as pre-commit or post-receive.
3. **.git/info**: Contains project-specific Git configuration that should not be shared with others, such as exclude patterns for ignoring files.
4. **.git/logs:** Stores logs related to various operations, including commit history and ref updates.
5. **.git/objects:** Stores the actual content of the repository, including blobs (file content) and trees (directory structure).
   * **.git/objects/info:** Contains additional information related to the objects, such as alternate object stores.
   * **.git/objects/pack:** Contains compressed packs of objects for efficient storage and transfer.
6. **.git/refs:** Stores references to commits, branches, tags, and other objects.
   * **.git/refs/heads:** Contains files representing branch references.
   * **.git/refs/tags:** Contains files representing tag references.
   * **.git/refs/remotes**: Contains files representing remote branches and references.
7. **.git/refs/remotes:** Stores references to remote repositories, including branches and other objects.
8. **.git/refs/remotes/{remote}:** Contains files representing references to remote branches for a specific remote.
9. **.git/refs/remotes/{remote}/HEAD:** Points to the default branch for a specific remote repository.
10. **.git/refs/stash:** Stores references to stash commits created by the git stash command.
11. **.git/branches:** Deprecated in favor of **.git/refs/heads**.
12. **.git/config**: Contains the repository-specific Git configuration, including remote URLs, branch settings, and user information.
13. **.git/description:** Contains a brief description of the repository.
14. **.git/index:** Stores the staging area (also known as the index), which tracks changes before committing them.
15. **.git/packed-refs:** Contains a list of all references, including tags, stored in a packed format.
16. **.git/HEAD:** Points to the currently checked out branch or commit.
17. **.git/COMMIT\_EDITMSG**: Temporary file used during the commit process to store the commit message.

*  Undoing Changes In Git

In Git, you can undo changes in each phase of the four phases of the file lifecycle: working directory, staging area (index), local repository, and remote repository. Here's how you can undo changes in each phase:

1. Undoing Changes in the Working Directory:

* To discard changes in the working directory and revert back to the last committed state, you can use the command:

**git checkout -- <file>**

* This command discards the local modifications and replaces the file with the last committed version.
* Additionally, if you want to remove untracked files and directories from the working directory, you can use the command:

**git clean -xdf**

* This command cleans the working directory by removing all untracked files and directories. The **-x** option ensures that ignored potential files rules are also removed, and the **-d** option for deleting including directories. **-f** for forced cleaning
* Note: Be cautious when using **git clean -xdf** as it permanently deletes untracked files and directories. Make sure to double-check before running this command.

1. Undoing Changes in the Staging Area (Index):

* To unstage changes that have been added to the staging area but not yet committed, you can use the command:

**git reset HEAD <file>**

1. Undoing Changes in the Local Repository (Commit):

* To undo the last commit and remove it from the commit history, you can use the command:

**git reset - - soft HEAD^**

- - soft: This command moves the branch pointer back to the previous commit, keeping the changes from the undone commit in the staging area.

* To completely undo the last commit and discard the changes, you can use the command:

**git reset - -hard HEAD^**

- - hard: This command moves the branch pointer back to the previous commit and discards all changes associated with the undone commit.

* The **^** represents Following :
  1. **HEAD^**: Refers to the parent commit of the current commit.
  2. **HEAD^^:** Refers to the grandparent commit of the current commit.
  3. **HEAD~n:** Refers to the nth commit ancestor of the current commit.
* In addition to resetting the commit, if you want to modify the content of the last commit without creating a new commit, you can use the command:

**git commit - -amend -m “commit Message”**

* This command allows you to modify the commit message or add changes to the previous commit. It opens the default text editor for you to make the necessary changes. Once you save and exit the editor, Git will update the commit with the amended content.
* Note: Be cautious when using git commit --amend as it modifies the commit history. It should only be used for local commits that have not been pushed to a remote repository. If the commit has already been pushed, it is generally not recommended to amend it, as it can cause issues for other collaborators.

1. Undoing Changes in the Remote Repository:

* If you have already pushed changes to the remote repository and want to undo them, you can use the command:

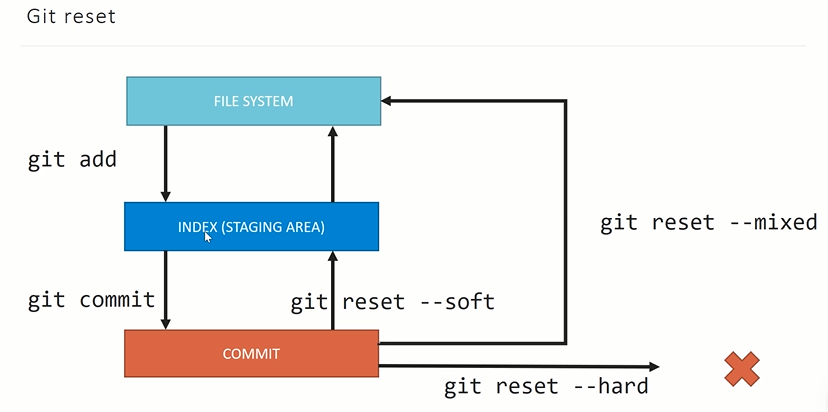
**git revert <commit>**

This command creates a new commit that undoes the changes made in the specified commit. It is a safe way to undo changes in a shared repository without rewriting history.

* Alternatively, if you have the necessary permissions and want to forcefully remove the last commit from the remote repository, you can use the command:

**git push origin +<branch>**

This command forcibly pushes the local branch, overwriting the remote branch's history. However, it is generally not recommended to force-push in a collaborative environment.

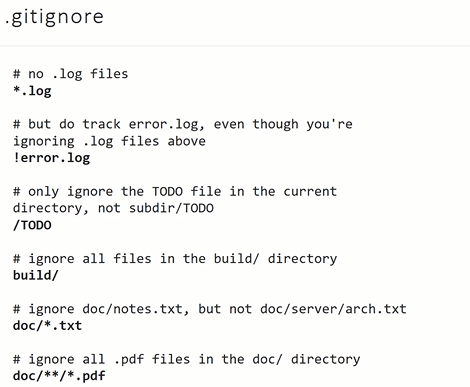
* Git Reset

Git Reset is a powerful command in Git that allows you to manipulate the commit history by moving the branch pointer to a specified commit. It provides different modes, or types, to control how Git handles the changes associated with the reset. Here are the different types of Git Reset, their use cases, and explanations:

1. Soft Reset:
   * Use Case: Soft reset is commonly used when you want to move the branch pointer to a previous commit while preserving the changes from the undone commit in the staging area.
   * Command: **git reset --soft <commit>**
   * Explanation: Soft reset moves the branch pointer to the specified commit and keeps the changes from the undone commit in the staging area. It allows you to review, modify, and commit the changes again.
2. Mixed Reset (Default):
   * Use Case: Mixed reset is useful when you want to move the branch pointer to a previous commit and remove the changes from the undone commit from the staging area, but keep them in the working directory.
   * Command: **git reset <commit> (or git reset --mixed <commit>)**
   * Explanation: Mixed reset moves the branch pointer to the specified commit and removes the changes from the undone commit from the staging area. The changes are kept in the working directory as modified, untracked files.
3. Hard Reset:
   * Use Case: Hard reset is used when you want to completely discard the changes from the undone commit and move the branch pointer to a previous commit.
   * Command: **git reset --hard <commit>**
   * Explanation: Hard reset moves the branch pointer to the specified commit and discards all changes associated with the undone commit. It resets both the staging area and the working directory to match the specified commit.

* Git Revert
* **git revert** is a command used to undo specific commits by creating new commits that reverse the changes made in the original commits.
* It is a safe way to undo changes because it does not modify the commit history. Instead, it adds new commits that revert the changes, keeping the commit history intact.
* Use Case:
  + git revert is useful when you want to undo the changes introduced by one or more commits while preserving the commit history and avoiding conflicts with other collaborators.
  + It is commonly used in scenarios where you want to revert specific changes without rewriting history, such as reverting a faulty commit or undoing changes made by a particular commit.

**git revert <commit>**

* git revert creates new commits that revert the changes of the specified commits, effectively undoing them while preserving the commit history.
* The new revert commits are applied on top of the existing branch, allowing you to keep track of the reversion in the commit history.
* Each revert commit is accompanied by a revert message indicating the commit being reverted, allowing others to understand the reason for the reversion.
* The revert process does not remove any commits; it adds new commits that counteract the changes made by the original commits.
* It's important to note that git revert can introduce conflicts if the changes being reverted conflict with subsequent commits. In such cases, manual conflict resolution is required.
* Git Ignore
* The .gitignore file is a configuration file used by Git to specify which files and directories should be ignored and not tracked by the version control system. It allows you to exclude certain files and patterns from being committed to the repository.
* Purpose:
  + The .gitignore file is used to specify files, directories, and patterns that Git should ignore when tracking changes in a repository.
  + It is particularly useful for excluding files that are generated during the build process, log files, temporary files, and other files that are not essential for the project's source code.
* File Location:
  + The .gitignore file is typically placed at the root directory of the Git repository.
  + It can also be placed in specific subdirectories to define rules specific to those directories.
* Syntax:
  + Each line in the .gitignore file represents a pattern or rule to ignore files or directories.
  + Blank lines and lines starting with a # are considered comments and are ignored.
  + Patterns can include wildcards and regular expressions to match files or directories based on their names or paths.
  + The patterns can be exact matches, directory matches, or glob patterns.
* Examples:
  + Ignore a specific file: filename.ext
  + Ignore all files in a directory: directory/
  + Ignore files with a specific extension: \*.ext
  + Ignore all files and directories with a specific name: name/
  + Ignore files in nested directories: nested\_directory/\*
  + Ignore all files and directories matching a pattern: pattern\*
* Creating and Modifying .gitignore:
  + You can create a .gitignore file manually using a text editor and save it with the name .gitignore.
  + Alternatively, you can use command-line tools or IDEs that provide built-in .gitignore file generation.
  + The .gitignore file can be modified at any time to add or remove rules as needed.
* Applying .gitignore:
  + Once the .gitignore file is in place, Git will automatically ignore the files and directories specified in the file.
  + Ignored files will not appear in the list of untracked files when running git status.
  + However, note that if a file is already tracked by Git before it is added to the .gitignore file, it will continue to be tracked.
* Branching And Merge
* Branching
* Branching allows for parallel development by creating separate branches for different tasks or features.
* Each branch represents an independent line of development, enabling multiple team members to work on different aspects of a project simultaneously.
* Branches provide isolation, allowing you to experiment, make changes, and commit them without affecting the main codebase until you're ready to merge the changes. A picture containing text, screenshot, diagram, design

  Description automatically generated
* Creating a Branch:
* To create a new branch, you can use the following command:

**git branch <branch-name>**

* This command creates a new branch with the specified name, pointing to the same commit as the current branch.
* Switching to a Branch:
* To switch to a different branch, you can use the following command:

**git checkout <branch-name>**

* This command moves the HEAD pointer to the specified branch, allowing you to start working on that branch.
* Creating and Switching to a Branch Simultaneously:
* To create and switch to a new branch in a single command, you can use the following command:

**git checkout -b <branch-name>**

* This command creates a new branch with the specified name and immediately switches to it.
* Listing Branches:
* To view the list of branches in a repository, you can use the following command:

**git branch**

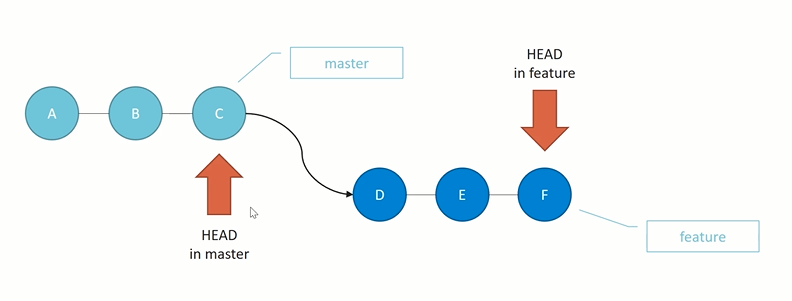
* This command displays all branches, with an asterisk (\*) indicating the current branch.
* Merging Branches:
* Once you have completed work on a branch and want to incorporate the changes into another branch (often the main branch), you can merge the branches.
* To merge a branch into the current branch, use the following command:

**git merge <branch-name>**

* This command incorporates the changes from the specified branch into the current branch.
* Deleting a Branch:
* To delete a branch that is no longer needed, use the following command:

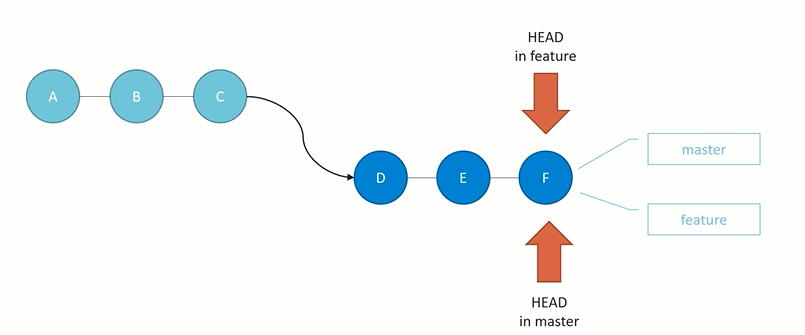
**git branch -d <branch-name>**

* This command deletes the specified branch. However, it can only be deleted if the changes in the branch have been merged into another branch.
* Merging:
  + Merging is used to integrate changes made in one branch into another branch, usually to bring the changes made in a feature branch into the main branch.
  + It combines the commit history, file changes, and any other modifications from the source branch into the target branch.
* Merging Process:
* Identify the Target Branch:
  + The target branch is the branch where you want to incorporate the changes. It is typically the branch where the changes will be merged into, such as the main branch.
* Switch to the Target Branch:
  + Use the command git checkout <target-branch> to switch to the target branch where you want to merge the changes.
* Initiate the Merge:
  + Run the command git merge <source-branch> to initiate the merge process.
  + The source branch is the branch that contains the changes you want to merge into the target branch.
* Resolve Conflicts:
  + Git will attempt to automatically merge the changes if there are no conflicts between the branches.
  + If conflicts arise, Git will indicate the conflicted files and you will need to manually resolve the conflicts.
  + Open the conflicted files, identify the conflicting sections marked by Git, and make the necessary modifications to resolve the conflicts.
  + Once the conflicts are resolved, mark them as resolved using git add <file>.
* Complete the Merge:
  + After resolving conflicts, run git commit to create a new commit that finalizes the merge.
  + Git will automatically generate a merge commit message summarizing the changes that were merged.
* Fast Forward Merging
* A fast-forward merge occurs when the target branch has not diverged from the source branch since the branch was created.
* In this case, Git simply moves the branch pointer of the target branch forward to the commit of the source branch, incorporating the changes.
* This type of merge is common when merging a feature branch into the main branch that hasn't received any new commits since the feature branch was created.
* Example:

Let's say you have the following commit history:

Before Merge👆

In this scenario, the main branch has not received any new commits since the feature branch was created. To merge the feature branch into the main branch, a fast-forward merge is performed.

After the fast-forward merge, the branch history would look like this:

After Merge 👆

* Initially, the main branch consists of commits A, B, and C, while the feature branch has commits D, E, and F.
* To merge the feature branch into the main branch, a fast-forward merge is performed.
* Since the main branch has not received any new commits, the branch pointer is moved forward to commit F, effectively incorporating the changes from the feature branch into the main branch.
* After the fast-forward merge, the branch history becomes a simple linear progression, where the main branch includes the commits from both branches (A-B-C-D-E-F).
* Non-Fast-Forward Merge:
* A non-fast-forward merge occurs when the target branch has received new commits since the branch was created, resulting in a divergent commit history.
* In this case, Git creates a new merge commit that combines the changes from the source branch into the target branch.

Example: Consider the same commit history as before:

Main Branch: A -- B -- C -- G -- H

\

Feature Branch: D -- E -- F

In this scenario, the main branch has received new commits (G and H) since the feature branch was created. To merge the feature branch into the main branch, a non-fast-forward merge is performed.

After the non-fast-forward merge, Git creates a new merge commit that combines the changes from the feature branch into the main branch. The branch history would look like this:

Main Branch: A -- B -- C -- G -- H -- M

\ /

Feature Branch: D -- E -- F ------

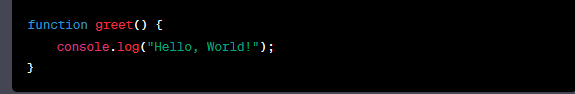
1. Here, the commit M represents the merge commit, which contains the combined changes from both branches.

Non-fast-forward merges are common when merging long-lived feature branches, release branches, or when multiple developers have made commits to the target branch since the source branch was created. They help maintain a clear and separate history of changes while incorporating the changes from one branch into another.

* Conflict In Git

Let's consider a scenario where two developers, Alice and Bob, are working on the same file in separate branches and attempt to merge their changes into a common branch. However, their changes conflict, resulting in a merge conflict.

1. Initial Commit:

 The main branch contains an initial commit with a file named "script.js" that includes the following code:

1. Alice's Branch:

A black screen with white text

Description automatically generated with low confidence Alice creates a new branch called "alice-feature" from the main branch and modifies the "script.js" file by adding a new function:

1. Bob's Branch:

A picture containing font, text, screenshot, graphics

Description automatically generatedMeanwhile, Bob also creates a new branch called "bob-feature" from the main branch and modifies the "script.js" file by changing the existing function:

1. Merge Conflict:
   * Both Alice and Bob attempt to merge their branches into the main branch, resulting in a merge conflict.
   * Git is unable to automatically merge the changes because both Alice and Bob modified the same section of code in the "script.js" file.

A picture containing text, screenshot, font

Description automatically generatedWhen Git encounters the conflicting changes, it marks the file with conflict markers

The conflict markers indicate the conflicting changes made by Alice ("Hello, World!") and Bob ("Hola, Mundo!").

* A screenshot of a computer

  Description automatically generated with medium confidenceConflict Resolution

Merge conflicts occur when Git is unable to automatically merge the changes from different branches due to conflicting modifications in the same file or section of code. Resolving merge conflicts requires manual intervention to determine the desired outcome. Here's a step-by-step guide on how to solve merge conflicts in Git:

1. Identify the Merge Conflict:
   * When you attempt to merge branches and conflicts arise, Git will notify you about the conflicting files.
   * Use the command **git status** to see which files have conflicts.
2. Open the Conflict Files:
   * Open the conflicting files in a text editor or an integrated development environment (IDE).
   * Git will mark the conflicting sections with special markers, such as **<<<<<<<, =======, and >>>>>>>,** to indicate the conflicting changes.
3. Understand the Conflict:
   * Read and understand the conflicting changes in the file.
   * Identify the conflicting sections and the changes made in both branches.
4. Resolve the Conflict:
   * Edit the conflicting file to manually resolve the conflict.
   * Decide on the desired outcome by choosing one version of the code or creating a new version that combines the changes.
5. Remove Conflict Markers:
   * Remove the conflict markers **(<<<<<<<, =======, and >>>>>>>)** from the file, as they are no longer needed once the conflict is resolved.
6. Save the Changes:
   * Save the resolved file with the changes you made to resolve the conflict.
7. Add the Resolved File:
   * Use the command **git add <file>** to stage the resolved file for the next commit.
   * Replace **<file>** with the path to the resolved file.
8. Commit the Merge:
   * Once all conflicts are resolved and the resolved files are staged, use the command git commit to create a new commit that finalizes the merge.
9. Repeat for All Conflicting Files:
   * If there are multiple conflicting files, repeat the above steps for each file until all conflicts are resolved.
10. Continue the Merge Process:
    * After resolving all conflicts, the merge process can continue with the remaining changes.
    * Use the appropriate Git command (e.g., **git merge --continue or git rebase --continue**) to proceed with the merge process.

* Rebase In Git

Git rebase is a command used to integrate changes from one branch into another by applying commits from one branch onto another branch. It essentially allows you to move, combine, or modify commits in a branch's commit history.

The need for using git rebase arises when you want to incorporate changes from one branch onto another while maintaining a linear commit history. It is commonly used in the following scenarios:

1. Keeping the Commit History Clean:
   * Rebasing allows you to combine multiple small, related commits into a single commit or a smaller set of cohesive commits.
   * This helps keep the commit history clean and concise, making it easier to understand and review.
2. Integrating Feature Branches:
   * When working on a feature branch, rebasing onto the main branch before merging can result in a more straightforward merge.
   * It allows you to incorporate the latest changes from the main branch into your feature branch, reducing the likelihood of conflicts during the final merge.
3. Maintaining a Linear Commit History:
   * Some teams prefer a linear commit history, where commits are applied one after another without any branching or merging.
   * Rebasing allows you to incorporate changes from different branches into a single branch while maintaining a linear commit history.

Usage of **git rebase**: To perform a rebase, follow these steps:

1. Checkout the branch onto which you want to apply the changes (the target branch):

**git checkout target-branch**

1. Execute the rebase command, specifying the branch from which you want to apply the changes (the source branch):

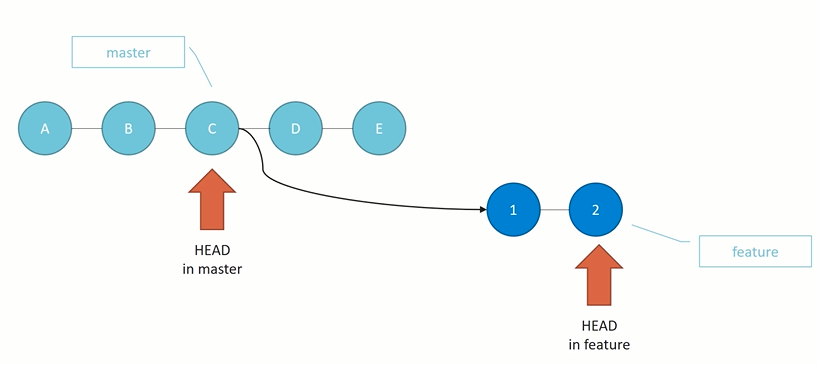
**git rebase source-branch**

1. Git will try to apply the commits from the source branch onto the target branch. If conflicts arise, Git will pause the rebase process and prompt you to resolve the conflicts manually.
2. Resolve any conflicts in the files that Git highlights as having conflicts.
3. After resolving the conflicts, mark the files as resolved by using git add on each conflicted file.
4. Continue the rebase process by running the command:

**git rebase - -continue**

1. If there are any additional conflicts, repeat steps 4-6 until the rebase process is completed.
2. Once the rebase is successfully completed, the changes from the source branch will be incorporated into the target branch, creating a linear commit history.

A picture containing diagram, screenshot, design

Description automatically generatedIt's important to note that rebasing rewrites the commit history, so it should be used with caution and primarily for local branches. Rebasing shared or public branches can cause issues for other team members who have already based their work on those branches.

Before Rebase 👆 After Rebase 👆

Consider Above Scenario :

* 1. Initial Commits :-

Here The Master Branch has 5 initial commits named A,B,C,D,E And The Feature Branch Has 2 Commits 1 and 2 Which Has C As Its Master Head.

* 1. Now, let's say there have been some updates on the Master branch by other team members, and you want to incorporate those changes into your feature branch before merging.
  2. At Such Time You can Perform Rebase
  3. Switch to the feature branch and execute the following command:

**git rebase master**

* 1. This will take the commits from the feature branch and replay them on top of the latest commit on the main branch.
  2. Git will rewind the feature branch to the commit where it diverged from the main branch (commit 1 in this case), apply the changes from the main branch, and then replay the commits from the feature branch one by one.
* Where To Use Rebase And Where Not
* A diagram of a diagram

  Description automatically generated with low confidenceA picture containing diagram, screenshot, design

  Description automatically generatedCherry-Pick In Git

Before Cherry-Pick 👆 After Cherry-Pick 👆

The head of the master branch is at commit E, and the head of the feature branch is at commit 2. Now, let's cherry-pick commit D from the master branch into the feature branch.

1. Switch to the feature branch:

**git checkout feature**

1. Cherry-pick commit D:

**git cherry-pick D**

Git will create a new commit on the feature branch, applying the changes from commit D. Let's call this commit 3.

1. The commit history after cherry-picking(fig 2):

Commit D is a new commit that contains the changes introduced by commit D. It is applied on top of the existing commits on the feature branch.

Now, the feature branch includes the changes introduced by commit D from the master branch. You can continue working on the feature branch and incorporate any further changes or improvements.

Cherry-picking is particularly useful when you want to bring specific changes from one branch into another branch without merging the entire branch. It allows you to select and apply individual commits selectively.

It's important to note that cherry-picking creates new commits with different commit hashes from the original commits. Although the changes may be the same, they are treated as separate commits in the commit history.

Remember to consider any potential conflicts that may arise during the cherry-pick process. If there are conflicts between the changes in the cherry-picked commit and the existing code in the target branch, you'll need to resolve those conflicts manually.

* Tags in Git

In Git, tags are used to mark specific points in the commit history of a repository. They serve as references to easily identify and access important commits, such as version releases or significant milestones. Here's an explanation of tags in Git:

1. Lightweight Tags:

* A lightweight tag is simply a reference to a specific commit. It is created using the git tag command followed by the tag name and the commit hash.

**git tag <tag-name> <commit-hash>**

* Lightweight tags are easy to create and don't contain any additional metadata.

1. Annotated Tags:

* An annotated tag is similar to a lightweight tag but includes additional metadata such as the tagger's name, email, date, and an optional message. Annotated tags are recommended for creating more descriptive and informative tags.

**git tag -a <tag-name> -m "<message>" <commit-hash>**

* Annotated tags are created using the -a flag with the git tag command and can include a message using the -m flag.

1. Viewing Tags:

* To view the list of tags in a repository, you can use the git tag command.

**git tag**

* This will display a list of all tags in alphabetical order.

1. Pushing Tags:

* By default, when you push changes to a remote repository, tags are not automatically pushed. To push tags, you need to use the --tags flag with the git push command.

**git push - -tags**

* This will push all local tags to the remote repository.

1. Checking Out Tags:

* You can switch to a specific tag by using the git checkout command followed by the tag name.

**git checkout <tag-name>**

* This will place your repository in a "detached HEAD" state, where you can inspect the code at the specific tagged commit but cannot make new commits.

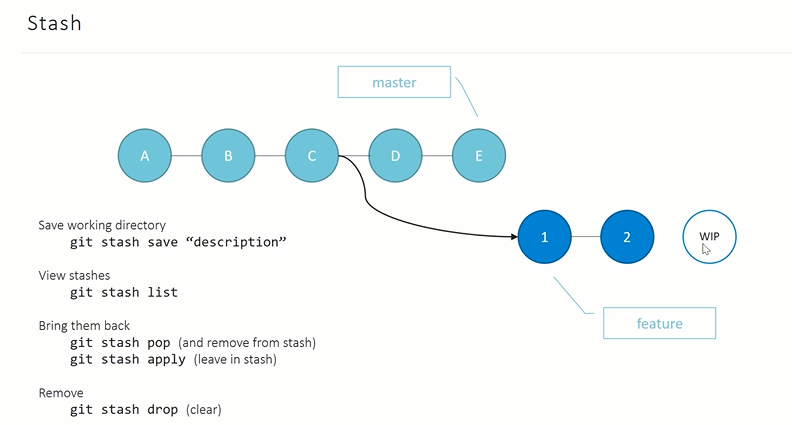
1. Deleting Tags:

* To delete a tag, you can use the git tag -d command followed by the tag name.

**git tag -d <tag-name>**

* This will delete the specified tag from your local repository. To delete a tag from a remote repository, you need to use the git push --delete origin <tag-name> command.

Tags provide a convenient way to mark important points in your Git history. They are commonly used to label releases, milestones, or specific versions of your project. By using lightweight or annotated tags, you can easily reference and share these significant points with others.

Top of Form

Bottom of Form

* Stashing In Git

In Git, the git stash command allows you to temporarily save your local changes in a "stash" without committing them. This is useful when you need to switch to a different branch or work on a different task without committing your current changes. Here's an explanation of Git stash and its usage:

1. Stash Changes:

* To stash your local changes, use the git stash command.

**git stash**

* Git will save your changes and revert your working directory to the state of the last commit, allowing you to switch branches or perform other operations.

1. List Stashes:

* You can view the list of stashes in your repository using the git stash list command.

**git stash list**

* This will display a list of all stashes, showing their stash ID, branch, and a message.

1. Apply Stash:

* To apply the most recent stash and restore the changes to your working directory, use the git stash apply command.

**git stash apply**

* If you have multiple stashes, you can apply a specific stash by providing its stash ID.

**git stash apply <stash-ID>**

* The stash changes will be applied, and you can continue working on the branch with the applied changes.

1. Pop Stash:

* If you want to apply the most recent stash and remove it from the stash list in a single operation, use the git stash pop command.

**git stash pop**

* Similar to applying a specific stash, you can pop a specific stash by providing its stash ID.

**git stash pop <stash-ID>**

1. Create Stash with a Message:

* You can add a descriptive message to your stash to provide more context for the changes. Use the -m flag followed by the message when creating a stash.

**git stash save -m "<message>"**

1. Stash Untracked Files:

* By default, Git does not include untracked files in the stash. However, you can include untracked files using the **--include-untracked** or **-u** flag.

**git stash save --include-untracked**

1. Clear Stash:

* To remove all stashes from your stash list, you can use the git stash clear command.

**git stash clear**

Git stash provides a convenient way to temporarily store your changes without committing them. It allows you to switch branches, perform operations, and later apply or pop the stashed changes when you need them. This can be helpful when you're in the middle of working on a feature or fixing a bug and need to switch to a different task quickly.

* Remotes
* Branching Strategies