* Need For Version Control System

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

* History and Track Changes:
* Collaboration and Teamwork:
* Backup and Recovery:
* Experimentation and Branching:
* Code Reviews and Quality Control:
* Deployment and Release Management:
* Traceability and Auditing:
* Benefits of Git:
  1. Enables efficient collaboration and code sharing.
  2. Provides powerful branching and merging capabilities.
  3. Lightweight and fast performance.
  4. Integrates well with CI/CD and DevOps.
  5. Extensible and supported by a vibrant community.

<https://git-scm.com/downloads>

* 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"
* 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:
5. Integration with Git Hosting Platforms:

* 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>

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:

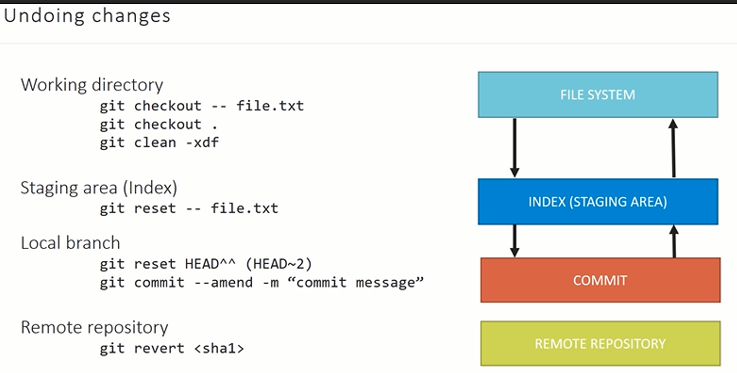
There are two possible scenarios:

a. Merge: If your local branch has its own commits that are not in the remote repository

b. Rebase: If your local branch does not have any commits that are not in the remote repository,

* 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**

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.

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.

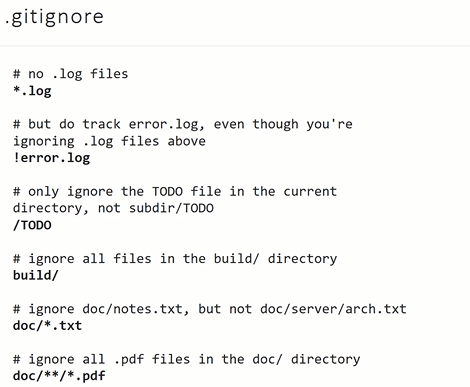
* Alternatively, if you have the necessary permissions and want to forcefully remove the last commit from the remote repository, you can use the command:

**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 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.
* 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.
* 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.