Week 1 - Source Code Management & Networking Concepts

- Difference between CVCS and DVCS
- Importance of Git
- Installation of Git
- Git three-stage Architecture
- Detail explanation of Repository, Commit, Tags, Snapshots, Push-Pull Mechanism, and Branching Strategy
- Working with Git stash and Git pop
- Resolve Merge conflicts in Git
- Git Revert and Reset (Reset vs Revert)
- Git rebase
- Working with git Squash
- Git cherry-pick
- What is Git fork?
- Git Integration on VScode, Git Authentication with Github via SSH and HTTPS Protocol
- Github Introduction, Creating Repositories, PR's
- Networking Concepts in Detail

—----------

Git and GitHub

- Git is a distributed version control system that enables multiple developers to collaborate on projects efficiently. It tracks changes in source code during development, allowing users to:
    - *Clone*: Create a local copy of a Git repository.
    - *Commit*: Save changes to files with a meaningful message.
    - *Push*: Upload local changes to a shared repository.
    - *Pull*: Download changes from a shared repository to update the local copy.
    - *Branch*: Create isolated environments to work on features or fixes.
    - *Merge*: Combine changes from different branches.
    - *Conflict resolution*: Handle conflicting changes during merges.
    - *Log*: View a history of changes with commit messages.
    - *Tag*: Mark specific points in history for releases or reference.
    - *Remote*: Manage connections to remote repositories.

- Git is widely used in software development for its flexibility, speed, and ability to handle complex workflows. It helps teams collaborate seamlessly and track the evolution of code over time.

- **GitHub** is a web-based platform that uses Git for version control and provides additional collaboration and project management features. Here's a brief overview:
    - *Repository (Repo)*: A repository is a storage space where your project's files and revision history are stored. It can be public, accessible to everyone, or private, restricted to specified collaborators.

    - *Branches*: GitHub allows developers to create branches to work on different features or fixes independently. Branches can later be merged back into the main codebase.

    - *Pull Requests*: When a developer has finished working on a feature or fix in a branch, they can open a pull request to propose changes. This allows team members to review the code, discuss modifications, and eventually merge the changes into the main branch.

    - *Issues*: GitHub's issue tracker helps manage and discuss tasks, enhancements, bugs, and other project-related topics. Issues can be linked to pull requests, facilitating seamless communication.

    - *Collaboration*: GitHub supports collaboration through features like code review, comments, and discussions. Contributors can interact on specific lines of code, providing feedback or asking questions.

    - *Actions*: GitHub Actions allows you to automate workflows, such as building, testing, and deploying your code. This helps streamline development processes.

    - *Projects*: GitHub Projects enable teams to organize tasks using boards, which can be customized based on the project's needs. It provides a visual representation of the workflow.

    - *Security and Integrations*: GitHub provides security features like dependency scanning, code scanning, and alerts to help maintain the security of your projects. It also integrates with various third-party services and tools.

Git Commands

- *#git init*: Initializes git in a folder.
- *#git status*: Checks the status of git.
- *#git add:* Adds files to the staging area.
- *#git commit*: Commits changes.
- *#git log:* Displays commit history.
- *#git reset <commit-id>*: Resets the pointer to a previous commit.
- #git fetch: Retrieves changes from a remote repository without merging them into the local branch.
- *#git remote -v:* Lists all remote repositories along with their URLs.
- *# git branch -a:* Lists all local and remote branches.
- *#git checkout -b <branch-name>:* Creates a new branch and switches to it in one step.
- *#git merge --abort:* Aborts a merge operation and restores the state before the merge starts.
- *#git rm <file>:* Removes a file from the repository and stages the change.
- *#git mv <old-path> <new-path>*: Moves or renames a file and stages the change.
- *#git diff*: Shows differences between current and previous code.
- *#git checkout <branch-name>*: Switches between branches.
- *#git tag <tag-name>:* Creates a lightweight tag at the current commit.
- *#git tag -a <tag-name> -m "message"*: Creates an annotated tag at the current -commit with a message.
- *#git show <tag-name>*: Displays the details of a particular tag.
- *#git cherry-pick <commit-id>:* Applies the changes introduced by the specified commit onto the current branch.
- *#git clean -n:* Shows a list of untracked files that will be deleted by git clean -f.
- *#git branch <branch-name>*: Creates a new branch.
- *#git merge <branch-name>*: Merges a branch with the master branch.
- *#git clone <repo-url>*: Clones a repository from GitHub.
- *#git remote add origin <repo-url>:* Adds a remote repository.
- *#git push origin <branch-name>:* Pushes code to the remote repository.
- *#git pull origin <branch-name>:* Pulls code from the remote repository.

Git Three-Stage Architecture

- *Working Area*:
  The working area is where you do all your work: you create, edit, and delete files here. When you make changes to files in your project, Git recognizes these modifications as "unstaged changes" in the working directory.
  These changes are not yet tracked by Git, meaning Git is unaware of them until you explicitly tell Git to do so by staging them.

- *Staging Area (Index):*
  The staging area is an intermediate area where you prepare changes before committing them to the repository.
  When you stage changes using git add, you are telling Git to include these changes in the next commit. Staging allows you to selectively choose which changes you want to commit.
  Files in the staging area are considered "staged changes." They are not yet part of the repository's history but are ready to be committed.

- *Commit Area:*
  The repository is where Git stores the history of commits for your project.
  When you commit changes using git commit, Git creates a new commit object that represents a snapshot of the project at that particular moment in time.
  Each commit contains a unique identifier (SHA-1 hash), metadata (author, timestamp, commit message), and a pointer to the previous commit(s).
  Once committed, changes become part of the project's history and are stored in the repository permanently, allowing you to track changes, revert to previous states, and collaborate with others.

Git Snapshots

A Git snapshot is a record of the state of a project at a specific point in time. Unlike other version control systems that store differences or changes between file versions, Git captures a complete snapshot of all the files in the repository each time you commit.

This snapshot includes:

The exact contents of each file: Git saves the entire contents of each file that has changed since the last commit.

References to unchanged files: For files that haven’t changed, Git simply references the previous version of the file.

How Snapshots Work?

When you make changes to your project and commit those changes, Git creates a new snapshot.

This snapshot includes:

- The current state of each file.

- A reference to the previous commit, forming a chain of commits.

Git Tags

Git, tags are used to mark specific points in the repository's history as important. Usually, these are used to denote release points (e.g., v1.0, v2.0, etc.). Tags can be considered as named pointers to specific commits.

Types of Tags

Git supports two types of tags:

Lightweight Tags: These are simply pointers to a specific commit. They are like branches that don’t change. Lightweight tags do not contain any extra metadata apart from the commit they point to.

Annotated Tags: These are full objects in the Git database. They store additional metadata such as the tagger's name, email, date, and a tagging message. Annotated tags are considered more permanent and are typically used for releases.

Creating Tags

1. Lightweight Tag

To create a lightweight tag, you use the git tag command followed by the tag name:

#git tag v1.0
This command creates a lightweight tag named v1.0 pointing to the latest commit.

2. Annotated Tag

To create an annotated tag, you use the -a flag and provide a message with the -m option:

#git tag -a v1.0 -m "Release version 1.0"
This creates an annotated tag named v1.0 with the message "Release version 1.0".

Viewing Tags
To see the tags in your repository, you can use:

#git tag
For more details about a specific tag, you can use:

#git show <tagname>

Practical Uses of Tags

- Releases: Tags are commonly used to mark release points (e.g., v1.0, v2.0).
- Milestones: Tags can be used to mark other significant milestones in the project's history.
- Versioning: Tags help in maintaining a versioning system where specific versions of the codebase are easily accessible.

Git Branches

Branching - Branching is a concept of git, Which provides us with a good way of management of our code and team, For example, we know that In a project, different developers are working on different functionalities/features, then making the changes directly in the master branch is the worst practice, good practice is to create respective feature branches and make changes there, And only after the permission of the reviewer merge it to the master branch by creating a PR(Pull Request)

Merge Strategies

In Git, merging is the process of combining/Integrating the changes from different branches. There are several merge strategies in Git, each with its own use cases. Here are some common merge strategies along with examples:

- **Fast-Forward Merge:**

  Description: This is the simplest and default merge strategy. If the branch being merged into has no new commits since the target branch was created, Git performs a "*fast-forward*" merge, which simply moves the pointer forward.

  Example:
    - Switch to the branch you want to merge into
      #git checkout main

    - Merge feature branch (assuming there are no new commits in main)
      #git merge --ff-only feature_branch

- **Recursive Merge:**

Description: This is the default non-fast-forward merge strategy. If the branches being merged have diverged, Git creates a new merge commit that combines the changes from both branches.
Example:

- Switch to the branch you want to merge into
  #git checkout main

- Merge feature branch with a new merge commit
  #git merge -s recursive feature_branch

- **Octopus Merge**:

  Description: This is used when merging more than two branches simultaneously. It creates a merge commit with more than two parent commits.
  Example:

  - Assuming you have branches feature1, feature2, and feature3
    #git checkout main

  - Merge multiple branches into main using octopus merge
    #git merge -s octopus feature1 feature2 feature3

- **Squash Merge:**

  Description: This is not a traditional merge strategy but involves combining all the changes from a branch into a single commit before merging.
  Example:

  - Switch to the branch you want to merge into
    #git checkout main

  - Merge feature branch with squashing
    #git merge --squash feature_branch

The choice of strategy depends on the specific use case and the desired outcome. Always consider the implications of each strategy, especially in a collaborative development environment.

Git Rebase

git rebase is a Git command used to integrate changes from one branch onto another by reapplying commits on top of the target branch. It effectively moves the entire branch to a new base commit, altering the commit history. The primary purpose of rebasing is to maintain a linear, cleaner commit history by incorporating changes from one branch onto another without creating merge commits.

Comparison between Rebase and Merge:

Rebase:

- Linear History: Rebase results in a linear commit history by incorporating changes from one branch onto another without creating merge commits. This helps maintain a cleaner and more readable history.

- Interactive Rebase: Git offers interactive rebasing, allowing users to selectively choose which commits to apply, reorder commits, or squash multiple commits into a single commit.

- Potential for History Rewrite: Since rebasing changes the commit history, it has the potential to rewrite history, which can lead to conflicts if the changes being rebased conflict with existing commits.

- Use Cases: Rebase is suitable for feature branches with relatively few commits, maintaining a clean and linear history, and integrating changes from a long-lived branch like master or main.

Merge:

- Preserves History: Merge creates a merge commit that preserves the commit history of both branches. This results in a more comprehensive view of how the branches evolved over time.
- Non-Destructive: Unlike rebase, merge is non-destructive and does not alter the commit history of the branches involved. It preserves the original commits and their relationships.

- Easier Collaboration: Merge simplifies collaboration by explicitly showing when and where branches were merged, providing clear points of integration in the commit history.

- Use Cases: Merge is suitable for integrating feature branches with complex histories, incorporating changes from multiple contributors, and when preserving the chronological order of commits is important.

When to Use What:

- Use git rebase when you want to maintain a clean and linear commit history, especially for feature branches, and when integrating changes from a long-lived branch.

- Use git merge when you want to preserve the original commit history, collaborate on changes with multiple contributors, or merge branches with complex histories.

Git Reset

- *#git reset --soft <commit>*

  Description - This form of reset moves the HEAD pointer to the specified commit but leaves the changes staged. It effectively "undoes" the commits after the specified commit, while keeping the changes in the staging area. It's commonly used when you want to undo a commit but keep the changes ready for a new commit.

- *#git reset --mixed <commit>*

  Description- This is the default behavior if no reset type is specified. It moves the HEAD pointer to the specified commit and unstages the changes. The changes are still preserved in the working directory.
  It's useful when you want to undo a commit and unstage its changes, but still keep them in your working directory for modifications or further staging.

- *#git reset --hard <commit>*

  Description - This form of reset moves the HEAD pointer to the specified commit and discards all changes since then. It resets both the staging area and the working directory to match the state of the specified commit.
  It's essentially a more aggressive form of reset, as it permanently discards any changes made after the specified commit. Use with caution, as it can result in the loss of unsaved work.

Git Revert

The git revert command can be considered an 'undo' type command, however, it is not a traditional undo operation. Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content. This prevents Git from

losing history, which is important for the integrity of your revision history and for reliable collaboration.

- Functionality: git revert is used to undo a commit by creating a new commit that reverses the changes made by the specified commit.
- Commit History: When you use git revert, it creates a new commit in the commit history that undoes the changes introduced by the specified commit, effectively keeping a record of the reversal.
- Collaboration: Since git revert creates a new commit, it's a safer option for undoing changes that have already been pushed to a shared repository, as it doesn't alter the commit history for others.

Git Revert Vs Reset

Git revert is used to create a new commit that undoes the changes introduced by a specified commit, while *git reset* is used to reset the branch's state to a previous commit, potentially removing commits from the branch's history. git revert is safer for undoing changes that have been shared with others, while git reset is more appropriate for local changes or when rewriting local history.

Pull Request

Pull requests are a way to propose changes to a project. They let you discuss and review code changes with collaborators before merging them into the main codebase.

Follow the below process to raise a PR

- Fork the Repository (if you're contributing to someone else's project):
  Go to the repository you want to contribute to.
  Click the "Fork" button in the upper right corner. This creates a copy of the repository under your GitHub account.

- Clone the Repository:
  #git clone <repository-url>

- Create a New Branch:
  #git checkout -b <branch-name>

- Make Changes:
  Make your changes to the codebase. This could involve adding new features, fixing bugs, or updating documentation.

- Commit Your Changes:
  Stage your changes: #git add .
  #git commit -m "Description of the changes made"

- Push Your Changes:
  Push your changes to the branch on your GitHub fork:
  #git push origin <branch-name>

- Create a Pull Request:
  Go to the original repository on GitHub.
  Click the "Compare & pull request" button that appears next to your recently pushed branch.
  Provide a title and description for your pull request and Click the "Create pull request" button.

Git Stash

- *#git stash*: Saves changes in the stash, leaving the working directory clean.
- *#git stash list*: Lists all stashes.
- *#git stash save "message"*: Saves changes in the stash with a message.
- *#git stash show stash@{0}*: Shows changes in the stash with stash id=0.
- *#git stash pop stash@{0}*: Adds stash changes to the working directory, deleting the stash.
- *#git stash apply stash@{0}*: Adds stash changes to the working directory, preserving the stash.
- *#git stash drop stash@{0}*: Deletes the stash with stash id=0.

Authentication in GitHub

1. Use HTTPS with a username and password/Github token.

Generate a Personal Access Token

- Log in to your GitHub account.
- Go to Settings > Developer settings > Personal access tokens.
- Click Generate new token.
- Select the scopes or permissions you need (for example, repo for full control of private repositories).
- Click Generate token and copy the token.
- Clone a Repository Using HTTPS (git clone https://github.com/username/repository.git)

- When prompted for a password, use your personal access token instead of your GitHub password.

2. For SSH, generate and add an SSH key with `ssh-keygen` command.

- Generate a New SSH Key -
    - #ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
- When prompted to "Enter a file in which to save the key," press Enter to accept the default file location.
- Copy the SSH Key to Your Clipboard - Copy the public key
- Add the Key to GitHub
- Log in to your GitHub account.
- Go to Settings > SSH and GPG keys.
- Click the New SSH key.
- Give your key a descriptive title.
- Paste the SSH key into the "Key" field.
- Click Add SSH key
- Clone a Repository Using SSH
    - git clone git@github.com:username/repository.git

GitKraken

GitKraken is a GUI tool for Git. It visualizes changes, branches, and commits
Install Git Kraken - https://www.gitkraken.com/

Networking

What is Networking?

Networking forms the backbone of any DevOps infrastructure, facilitating communication between various components of a system. Understanding key networking concepts such as IP addresses, netmasks, subnets, and routers is essential for DevOps professionals to effectively manage and troubleshoot modern IT environments.

Three Requirements for Devices to Communicate, If two systems A and B want to communicate/Share data then they should follow the below three basic rules for networking

1. Both A and B should have a valid IP Address
2. A and B should be physically connected wireless or Wired Connection

3. System A and B should be on the Same Network

Meaning of Same Network:

What is an IP Address?

An IP (Internet Protocol) address is a unique numerical label assigned to each device connected to a network. It serves as an identifier, allowing devices to communicate with each other across the internet or a local network. IP addresses come in two main versions: IPv4 and IPv6.

The basic definition of an IP address, Is that it is a numerical number of 32 bits in size, It means (e.g., 172.16.0.0, 13332433) are valid IP address. Normally we denote an IP address in the form of 4 octets like 192.168.3.4, But an IP address can also be denoted as in one octet like 13332433, So 13332433 is also an valid ip address, as it is an 32 bit number.

IPv4: The most commonly used version, IPv4 addresses are represented as four sets of numbers separated by dots (e.g., 192.168.1.1).
IPv6: With the depletion of IPv4 addresses, IPv6 was introduced to accommodate the growing number of devices. IPv6 addresses are longer and expressed in hexadecimal notation (e.g., 2001:0db8:85a3:0000:0000:8a2e:0370:7334).

Netmask

A netmask is used to divide an IP address into network and host portions. It determines which part of the IP address identifies the network and which part identifies the specific device within that network. Netmasks are typically represented in a format like "255.255.255.0" for IPv4 addresses.

For example, in the IP address 192.168.1.1 with a netmask of 255.255.255.0, the first three sets of numbers (192.168.1) represent the network, while the last set (1) identifies the host within that network.

Subnets

Subnetting involves dividing a single, larger network into smaller, more manageable sub-networks. This is useful for improving network performance, security, and organization. Subnets are created by borrowing bits from the host portion of an IP address to create additional network addresses.

For instance, by subnetting the IP address 192.168.1.0 with a netmask of 255.255.255.0, you can create multiple subnets such as 192.168.1.0/24, 192.168.2.0/24, and so on, each with its own range of assignable IP addresses.

Routers

Routers are networking devices that forward data packets between different computer networks. They serve as gateways, directing traffic between devices on the same network and those on different networks. Routers use routing tables to determine the most efficient path for data to travel based on destination IP addresses.

OSI Model

The Open Systems Interconnection (OSI) model is a conceptual framework used to understand and implement network protocols in seven distinct layers. Each layer has specific functions and communicates with the layers directly above and below it.

- Physical Layer (Layer 1)
  Function: The Physical Layer is responsible for the physical connection between devices. It deals with the hardware aspects of transmitting raw data bits over a physical medium such as cables, fiber optics, or radio waves.

- Data Link Layer (Layer 2)
  Function: The Data Link Layer is responsible for node-to-node data transfer and error detection and correction. It ensures that data sent from the Physical Layer is error-free and properly formatted for the Network Layer.

- Network Layer (Layer 3)
  Function: The Network Layer manages device addressing, tracks the location of devices on the network, and determines the best way to move data. It is responsible for packet forwarding, including routing through intermediate routers.

- Transport Layer (Layer 4)
  Function: The Transport Layer ensures the reliable transmission of data between devices. It provides error checking and recovery as well as flow control.

- Session Layer (Layer 5)

Function: The Session Layer manages sessions or connections between applications. It establishes, maintains, and terminates connections between two communicating devices.

- Presentation Layer (Layer 6)
  Function: The Presentation Layer translates data between the application layer and the network. It is responsible for data encryption, decryption, compression, and decompression.

- Application Layer (Layer 7)
  Function: The Application Layer provides network services directly to the end user or application processes. It facilitates communication between software applications and lower layers of the OSI model.

Physical Layer: Transmits raw bits over a physical medium.
Data Link Layer: Manages node-to-node data transfer and error detection/correction.
Network Layer: Handles logical addressing and routing.
Transport Layer: Ensures reliable data transfer with error detection and flow control.
Session Layer: Manages sessions between applications.
Presentation Layer: Translates, encrypts, and compresses data.
Application Layer: Provides network services directly to users and applications.