DevOps Zero to Hero

Fundamentals of DevOps

What is DevOps?
DevOps is a culture or practice adopted by organizations to improve their ability to deliver applications quickly and efficiently [02:46].

It goes beyond just delivery, encompassing automation, quality assurance, continuous monitoring, and continuous testing [06:28].

The ultimate goal is to eliminate manual processes and accelerate application delivery [10:58].

Why DevOps?
The video explains the limitations of older software development models (pre-DevOps) where slow, manual processes were common due to the involvement of various roles like system administrators and build engineers [14:11].

DevOps was created to automate these processes, reduce manual effort, and improve communication and efficiency through a collaborative culture [19:53].

How to Introduce Yourself as a DevOps Engineer
When introducing yourself, be truthful about your experience. If you have a background in related fields, be sure to mention it to provide context [23:36].

Focus on your responsibilities in automation, quality assurance, monitoring, and testing [26:19].

While it's good to mention the tools you know, the main focus should be on your role and the value you bring through DevOps practices [26:41].

SDLC with DevOps

Introduction to SDLC
What it is: SDLC is a standard process or culture followed in the software industry to design, develop, and test high-quality products [02:22].

Importance: It's essential knowledge for anyone in the software industry, including developers, testers, and DevOps engineers [00:57].

Phases of SDLC
The video explains the phases using an example of an e-commerce website:

Planning & Requirements Gathering: The initial phase where requirements are gathered, often from customer feedback. For example, deciding to add a new "kids" clothing section to a website [05:43].

Defining & Designing:

Defining: The requirements are documented in a Software Requirement Specification (SRS) document [11:43].

Designing: This phase includes High-Level Design (HLD), which focuses on overall system architecture, and Low-Level Design (LLD), which details specific functions and modules [12:29].

Building (Developing): Developers write the application code based on the designs, often using version control systems like Git [16:10].

Testing: Quality Assurance (QA) engineers test the software to ensure it meets quality standards [18:48].

Deploying: The tested application is released to the production environment for customers to use [19:29].

Role of DevOps: DevOps engineers primarily focus on automating and improving the efficiency of the building, testing, and deployment phases [14:27].

Absolute Prerequisite for Learning DevOps

Detailed Note from YouTube Video

1. Understanding Organizational Roles and Requirement Flow

The video explains the organizational structure and the flow of requirements using an example of a DevOps engineer working at "Amazon Fresh."

- Customers: The primary source of feedback and requirements.
- Business Analyst (BA): Gathers requirements from customers and the business and prepares a Business Requirement Document (BRD).
- Product Manager (PM): Prioritizes requirements based on product vision and market analysis.
- Product Owner (PO): Breaks down prioritized requirements into actionable items ("epics" or "features").
- Solutions Architect (SA) / Software Architect: Provides technical deep-dives and defines High-Level Design (HLD) and Low-Level Design (LLD).
- Development Teams (Scrum Team): Includes Developers, DevOps Engineers, QA Engineers, Database Administrators (DBA), and Technical Writers.
- SRE Engineers: Focus on reliability and availability of the product once it's live.

2. Software Development Life Cycle (SDLC)

The video explains that the organizational role flow outlines the SDLC, which includes the following phases:

- Planning
- Analysis
- Design
- Implementation
- Testing and Integration
- Maintenance

3. Role of DevOps Engineers Beyond Requirements
DevOps engineers also proactively:

• Identify gaps and improve efficiency.
• Automate processes.
• Integrate security.

4. Project Management with Jira
Jira is a project management tool used for tracking work, accountability, and communication.

• Key Features:
  • Tracks the status of projects.
  • Provides visibility into accountability and blockages.
• Practical Demonstration:
  • Account creation.
  • Organization and project setup.
  • Creating epics.
  • Scrum methodology and sprints.
  • Creating stories from epics.
  • Backlog refinement.
  • Updating task status.

Virtual Machines Part-1

Introduction to Virtual Machines
• This video is part of the "DevOps Zero to Hero" course, focusing on virtual machines (VMs).
• The main goal is to explain VMs with real-world examples to show their importance in technology.

Servers and Inefficiency
• Server: A place where applications are deployed to be accessible to users.
• Inefficiency Problem (Real-World Analogy):
  • Owning a one-acre plot for a single-family home leads to unused land, representing inefficient resource use.
  • Building another property on the unused land to rent out improves efficiency.
• Inefficiency Problem (Software Industry):
  • Deploying a small application on a powerful server with excess resources (e.g., 4GB RAM app on a 100GB RAM server) leads to waste.
  • Allocating entire physical servers to different teams often results in underutilization.

Virtualization and Hypervisors
• Virtualization: A solution to the inefficiency of physical servers.
• Hypervisor: Software (like VMware or Xen) that creates and manages VMs on a physical server.
• Logical Partitioning: The hypervisor logically divides the server's resources to create multiple VMs.
• Efficiency with VMs: A single physical server can host multiple VMs, allowing different teams to use separate virtual machines efficiently, which significantly increases resource utilization.
• Virtual Machine (VM): A virtual environment that functions as a virtual computer system with its own CPU, memory, and hardware. VMs are logically isolated and independent.

Virtual Machines in Cloud Computing
• Cloud providers like AWS, Google, and Azure use virtualization in their data centers.
• Cloud Data Centers: These centers house a vast number of physical servers.
• User Request for a VM:
  • When a user requests a VM with specific resources, the request is sent to a data center.

• A hypervisor on a physical server creates a VM that matches the user's requirements.
• The user gets an IP address and other information to access their VM.
• Logical Access: Users have logical access to their VMs but do not physically own them.
• Exponential Efficiency: This model allows cloud providers to serve millions of users with a finite number of physical servers.

Conclusion
• Virtualization, through hypervisors, has revolutionized resource management by improving efficiency in both personal and enterprise computing.

AWS & Azure - How to Create Virtual Machines

Key Learning Points for Day 4
• **Advanced Virtual Machine Concepts:** The video explains how to create virtual machines using top cloud providers (AWS and Azure) and in an on-premise scenario.
• **Cloud Providers Covered:**
  • **AWS (Amazon Web Services):** Demonstrates creating an EC2 instance through the AWS console UI.
  • **Microsoft Azure:** Shows how to create a VM through the Azure portal UI.

The Importance of Automation in DevOps
• Manual VM creation is inefficient for multiple requests.
• DevOps engineers prioritize automation to avoid manual errors and save time.

Methods for Automating VM Creation (via APIs)

- Cloud providers offer APIs for their services.
- Automation can be achieved using:
  - **AWS CLI (Command Line Interface)**
  - **Direct AWS API calls** (e.g., with Python's `boto3`)
  - **AWS CFT (CloudFormation Templates)**
  - **AWS CDK (Cloud Development Kit)**
  - **Terraform** (a popular, cloud-agnostic tool)

Practical Demonstration

- **Creating an AWS Account:** The video guides viewers on how to sign up for a new AWS account, including the card verification process.
- **Creating an EC2 Instance on AWS (UI Method):**
  1. Navigate to the EC2 service in the AWS console.
  2. Click "Launch instance."
  3. Provide an instance name.
  4. Choose an Operating System (Ubuntu is recommended).
  5. Select "Free Tier Eligible."
  6. Create and save a new Key Pair.
  7. Click "Launch instance."
- **Creating a Virtual Machine on Azure (UI Method):**
  1. Go to `portal.azure.com`.
  2. Sign up.
  3. Click "Create resource" or go to "Virtual machines."
  4. Follow similar steps as AWS.

Next Steps

The next video will cover logging into these virtual machines and automating the creation process using CLI or CFT.

AWS CLI Full Guide | How to connect to EC2 Instance from UI & Terminal

Detailed Note from "Day-5 | AWS CLI Full Guide" YouTube Video
Efficient VM Creation and Management
• Introduction: The video emphasizes creating virtual machines (VMs) efficiently, building on previous lessons.
• Automation Methods: It covers various ways to automate VM creation on AWS:
  • AWS API
  • AWS CDK
  • AWS CLI
  • AWS CloudFormation templates
  • Terraform

Logging into an AWS EC2 Instance
• AWS Console (UI): A straightforward way to connect, but not efficient for multiple machines and sessions can time out.
• Terminal: The recommended method for better efficiency.
  • Recommended Terminals: iTerm (Mac), PuTTY, MobaXterm (Windows) , Nomachine.
  • Connection Steps:
    1. Get the public IP of the EC2 instance.
    2. Use the `SSH` command with the correct user and IP.
    3. Provide the path to your PEM key file using the `-i` flag.
    4. Important: Set the PEM file permissions to `600` using `chmod` to avoid security errors.

Managing EC2 Instances
• Deleting an Instance: To avoid charges, stop the instance first, then terminate it.

Billing is based on running time.

Automating with AWS CLI
• What it is: A command-line tool to interact with AWS services.
• Setup:
  1. Install the AWS CLI from the official AWS documentation.
  2. Create an Access Key ID and Secret Access Key in your AWS security credentials.
  3. Configure the CLI by running `aws configure` and entering your credentials and default region.
• Usage: Once configured, you can run commands like `aws s3 ls` to list S3 buckets or use `aws ec2 run-instances` to create EC2 instances.

Automating with Infrastructure as Code (IaC)
• AWS CloudFormation: Use YAML or JSON templates to define and create AWS resources. You can find examples on GitHub.
• AWS API (Python/boto3): Use the `boto3` Python library to write scripts that interact with AWS services.

Assignment for Viewers
• Install AWS CLI.
• Create and configure security credentials.
• Practice using AWS CLI commands to manage services like S3 and EC2 by referring to the official documentation.

Linux & Shell Scripting

Detailed Note from "Day-6 | Linux & Shell Scripting" YouTube Video
Understanding Operating Systems (OS)
• An OS acts as a bridge between software and hardware, managing communication between them.
• Examples: Windows, macOS, and Linux.

Why Linux is Popular in DevOps
• Free and Open Source: No licensing costs and can be customized.
• Security: Inherently secure, often not needing extra antivirus software.
• Distributions: Offers a wide variety of distributions like Ubuntu, CentOS, and Red Hat.
• Performance: Known for its speed and stability, which is crucial for production environments.

Linux Operating System Architecture
• Kernel: The core of the OS, responsible for:
  • Device Management
  • Memory Management
  • Process Management
  • Handling System Calls
• System Libraries: Perform tasks initiated by the user and communicate with the kernel.
• Compilers, User Processes, and System Software: The outermost layer that includes applications and system utilities.

Basics of Shell Scripting
• Shell: A command-line interface for interacting with the OS.
• Essential Commands:
  • `ls`: List files and directories.
  • `pwd`: Show the present working directory.

- `cd`: Change directory.
- `ls -ltr`: Show a detailed list of files.
- `touch [filename]`: Create an empty file.
- `vi [filename]`: Create and edit a file.
- `cat [filename]`: Display the content of a file.
- `mkdir [directoryname]`: Create a new directory.
- `rm [filename]`: Remove a file.
- `rm -r [directoryname]`: Remove a directory.
- `free -m`: Show memory usage.
- `nproc`: Show the number of CPUs.
- `df -h`: Show disk space usage.
- `top`: Display real-time system performance.

Further Learning
• The video recommends exploring other videos on the channel for more advanced shell commands and interview questions.
• The next class will focus on a real-time DevOps project using shell scripting.

Shell Scripting for DevOps

Detailed Note from Shell Scripting YouTube Video
Introduction to Shell Scripting
• Automation: Reducing manual tasks in a Linux environment.
• Why Shell Scripting?: Essential for automating repetitive and large-scale tasks.
• Role in DevOps: Crucial for infrastructure automation, configuration management, code management, and monitoring.

Basic Linux and Shell Scripting Commands
• `touch`: Creates new, empty files.
• `ls`: Lists files and folders.
• `man`: Displays the manual for any command.

• `vim` / `vi`: Text editors for creating and editing files.
• `cat`: Displays the content of a file.
• `chmod`: Changes file permissions.
• `history`: Shows a list of previously executed commands.
• `pwd`: Prints the present working directory.
• `mkdir`: Creates a new directory.
• `cd`: Changes the current directory.
• `rm -rf`: Removes directories and their contents recursively and forcefully.

Writing Your First Shell Script
1. Shebang Line: Starts with `#!/bin/bash` or `#!/bin/sh` to specify the interpreter.
2. Comments: Use `#` to add comments.
3. `echo`: Prints text to the terminal.
4. Execution: Make the script executable with `chmod +x` and run it with `./your_script.sh`.

Advanced Concepts and Monitoring
• Node Health Monitoring:
  • `nproc`: Displays the number of CPU cores.
  • `free`: Shows memory usage.
  • `top`: Provides a real-time summary of system performance.

Detailed Note from "Shell Scripting for DevOps | Zero 2 Hero Part-2" YouTube Video
I. Review and Node Health Commands
• Recap: The video starts with a review of basic shell scripting and revisits essential node health commands:
  • `df -H`: Disk space
  • `free -G`: Memory
  • `nproc`: Number of CPUs
  • `top`: Real-time process view

II. Building a Node Health Script (`nodehealth.sh`)
• Structure and Metadata:
  • Shebang: `#!/bin/bash` is crucial for specifying the interpreter.
  • Comments: Use `#` for metadata like author, date, purpose, and version to improve readability.
• Output Readability:
  • `echo`: Use descriptive messages to clarify the output.
  • `set -x`: Run the script in debug mode to print each command before execution.

III. Process Management
• `ps -ef`: Lists all running processes.
• `grep`: Filters output to find specific information.
• `|` (Pipe): Chains commands together, sending the output of one to the input of another.
• `awk`: Extracts specific columns from the output, making it more powerful than `grep` for data extraction.

IV. Best Practices for Script Robustness
• `set -e`: Exits the script immediately if any command fails.
• `set -o pipefail`: Ensures that errors within a piped command sequence are caught.

V. Retrieving Information from External Sources
• `curl`: Transfers data from URLs, useful for making API calls and retrieving files without saving them locally.
• `wget`: Downloads files from web servers.

VI. File System Navigation and Search
• `find`: Searches for files and directories based on various criteria.

VII. User Management
• `sudo`: Executes a command as another user (usually root).
• `su`: Switches the current user.

VIII. Conditional Statements and Loops
• `if-else`: Executes different blocks of code based on a condition.
• `for` Loops: Repeats a set of actions for a specified number of iterations.

NOTE : Use comparison operators, such as -eq, -ne, -lt, -gt, -le, -ge, for numeric values, or ==, !=, <, >, <=, >=, for string values.

IX. Signal Handling with `trap`
• `trap`: Intercepts and handles signals (like Ctrl+C) to perform custom actions, such as cleanup, before a script exits.

Detailed Note from "Shell Scripting Interview Questions" YouTube Video
1. Commonly Used Shell Commands
• Be honest and mention commands you use daily, such as `ls`, `cp`, `mv`, `mkdir`, `touch`, `vim`, `grep`, `find`, `top`, and `df`.
• Avoid listing debugging-specific commands as daily tools.

---

2. Script to List All Processes
• Use `ps -ef` to list all processes.
• Extension: To print only the Process ID (PID), use `ps -ef | awk '{print $2}'`.

---

3. Script to Print Errors from a Remote Log
• Use `curl` to fetch the log from a URL and pipe it to `grep` to filter for errors.

• Example: `curl <URL_of_log_file> | grep "error"`

---

4. Script to Print Numbers Divisible by 3 and 5, but Not 15
• Use a `for` loop and `if` conditions with the modulo operator (`%`).
• Logic: `if (( i % 3 == 0 || i % 5 == 0 )) && (( i % 15 != 0 ))`

---

5. Script to Count a Specific Character in a String
• Use `grep -o` to print each match on a new line and `wc -l` to count the lines.
• Example: `grep -o "s" <<< "Mississippi" | wc -l`

---

6. How to Debug a Shell Script
• Add `set -x` at the beginning of the script to run it in debug mode, which prints each command before it's executed.

---

7. What is Crontab?
• A utility for scheduling scripts to run automatically at specified intervals.
• Use Case: Automating daily reports on node health.

---

8. How to Open a File in Read-Only Mode
• Use `vim -R <filename>` or `view <filename>`.

---

9. Difference Between Soft and Hard Links
• Hard Link: A mirror copy of a file. The link persists even if the original is deleted.
• Soft Link (Symbolic Link): A pointer or alias to a file. The link breaks if the original is deleted.

---

10. Difference Between `break` and `continue`
• `break`: Terminates the entire loop.
• `continue`: Skips the current iteration and proceeds to the next one.

---

11. Disadvantages of Shell Scripting
• It is dynamically typed, which can lead to bugs from undeclared variables if not handled carefully (e.g., using `set -u`).

---

12. Different Kinds of Loops
• Common loops include `for`, `while`, and `until` loops, each with different use cases.

---

13. Is Bash Dynamically or Statically Typed?
• Bash is dynamically typed. You can assign values of different types to the same variable without explicit declaration.

---

14. Networking Troubleshooting Tools
• `traceroute` and `tracepath` are used to trace the network path to a destination.

---

15. How to Sort a List of Names
• Use the native Linux `sort` command.

---

16. How to Manage Huge Log Files
• Use `logrotate`, a utility to automate the rotation, compression, and removal of log files.

Of course! Here are the answers to your questions about Linux and shell scripting.

1. What is the sticky bit in Linux?

The **sticky bit** is a special permission setting for directories.[1] When the sticky bit is set on a directory, a user can only delete or rename files within that directory if they are the **owner of the file**, the **owner of the directory**, or the **root user**.[2]

This is useful for shared directories like , where many users need to create files but shouldn't be allowed to delete each other's files.[3]

You can identify a sticky bit by the at the end of the permission string in the output.[4]

Bash

```
$ ls -ld /tmp
drwxrwxrwt 15 root root 12288 Aug 30 14:55 /tmp
```

2. How do we verify if our shell script is executed successfully?

You can verify a script's execution by checking its **exit status**.[5] By convention, a script that runs successfully exits with a status of . [6]A non-zero exit status (from 1 to 255) indicates an error.[7]

The special shell variable holds the exit status of the last command that was executed.[8]

Example:

Bash

```
./my_script.sh
```

```
$?
```

3. What is the flag to check if a file is empty or not?

The `-s` flag is used within a test condition (`[` or `test`) to check if a file exists **and** has a size greater than zero.
Example:
Bash

```
[ -s  ];
```

4. What is a positional parameter?

A **positional parameter** is a variable in a shell script that stores an argument passed to the script from the command line.[9] They are accessed by their numerical position.[10]

• : The name of the script itself.[11]

• : The first argument.[12]

• : The second argument.[13]

• `$3`: The third argument, and so on.

• `$#`: The total number of arguments.

• : All arguments, treated as separate strings (this is generally the safest way to refer to all arguments).[14]

Example:
If you have a script user.sh and run it like this:
Bash

```
./user.sh abhishek devops
```

Inside the script:
◇ `$1` would be "abhishek".

◇ `$2` would be "devops".

◇ `$#` would be 2.

5. What is command substitution?

Command substitution allows you to use the output of a command as the value of a variable or as an argument for another command.[15] It essentially replaces the command with its output.[16]

The modern and preferred syntax is `$(command)`.
Example:
Bash

```
CURRENT_DATE=$(date)
```

6. How do you set crontab?

You set up a cron job using the `crontab` command. To add or edit jobs, you use the `-e` flag, which opens your personal crontab file in a text editor.

```
crontab -e
```

Each line in the file represents a scheduled job with the following format:
minute hour day-of-month month day-of-week command
Example:
To run a script located at /home/user/backup.sh every night at 2:30 AM, you would add this line:

```
30 2 * * * /home/user/backup.sh
```

7. How will your .sh script configured in CRONTAB run when the system is restarted?

A standard time-based cron job (e.g., `30 2 * * *`) will **not** run if the system was offline during its scheduled time. Cron does not catch up on missed jobs after a reboot.
To run a script specifically at startup, you should use the special string in your crontab.[17]

Example:
To run your script every time the system boots up, add this line to your crontab using crontab -e:

```
@reboot /path/to/your/script.sh
```

The cron daemon will execute this command once when it starts up during the system boot process.

Live AWS Project using SHELL SCRIPTING for DevOps

Detailed Note from "Real-time Shell Script Project for DevOps" YouTube Video
1. Why Use Cloud Infrastructure?
• Manageability: Reduces the need for physical server maintenance.

• Cost-Effectiveness: Pay-as-you-go model avoids costs of unused infrastructure.

---

2. The Problem: Untracked Resource Usage
• Unchecked resource creation by developers leads to unnecessary costs from unused resources (e.g., EC2 instances, EBS volumes).
• Tracking resource usage is a key responsibility for DevOps engineers to maintain cost-effectiveness.

---

3. Project Goal: Daily AWS Resource Usage Report
• Generate a daily report on AWS resource usage for:
  • EC2 instances
  • S3 buckets
  • Lambda functions
  • IAM users

---

4. Tools and Technologies Used
• Shell Scripting (Bash): For writing the automation script.
• AWS CLI: To interact with AWS services.
• Cron Job: To schedule the script to run daily.
• JQ: A JSON parser to filter and extract data from the AWS CLI output.

---

5. Prerequisites
• AWS CLI installed.
• AWS credentials configured using `aws configure`.

---

6. Script Development Walkthrough
• Script Initialization:
  • Start with `#!/bin/bash`.
  • Include comments for metadata (author, date, version, description).
• Retrieving Resource Information:
  • Use AWS CLI commands like `aws s3 ls`, `aws ec2 describe-instances`, etc.
• Improving Readability:
  • Use `echo` statements to label the output.
  • Use `set -x` for debugging.
• Simplifying Output with JQ:
  • Parse the JSON output of AWS CLI commands to extract specific information (e.g., instance IDs).
  • Example: `aws ec2 describe-instances | jq '.Reservations[].Instances[].InstanceId'`
• Redirecting Output to a File:
  • Redirect the script's output to a file (e.g., `resource_tracker.txt`) to create a report.

---

7. Scheduling with Cron Job
• The final step is to integrate the script with a Cron job to automate the daily report generation.

Detailed Note from "Day-8 | DevOps Zero to Hero | Shell Scripting Project" YouTube Video

Purpose of the Video

- This video is a re-recording of a previous "Day 8" session to provide a clearer explanation and a complete code script on GitHub.
- It focuses on a real-world DevOps project using shell scripting and the GitHub API to automate the process of listing users with access to a repository.

Why This Project is Important for DevOps

- Offboarding: Easily identify and revoke access for employees who have left the company.
- Access Management: Regularly monitor who has access to sensitive repositories.
- Efficiency: Avoid the time-consuming manual process of checking collaborators in the GitHub UI.

Key Concepts: API vs. CLI

- API (Application Programming Interface): A way to interact with applications programmatically. DevOps engineers typically consume APIs, not write them. Tools like curl (shell) or requests (Python) are used.
- CLI (Command Line Interface): A text-based interface for interacting with applications (e.g., kubectl, aws cli).

Practical Demonstration: Listing Repository Collaborators

- Prerequisites:
  - An EC2 instance (Ubuntu is used in the video).
  - JQ (a command-line JSON processor) installed: sudo apt install jq -y.
- Steps to Execute the Script:
  1. Clone the project repository from GitHub.
  2. Generate a GitHub Personal Access Token (PAT) with appropriate permissions.
  3. Export your GitHub username and the PAT as environment variables:
     export USERNAME="your_github_username"
     export TOKEN="your_github_personal_access_token"
  4. Make the script executable: chmod 777 list_users.sh.

5. Run the script with the organization and repository names as arguments:
   ./list_users.sh <organization_name> <repository_name>

Understanding the Script (list_users.sh)

- Shebang: Starts with #!/bin/bash.
- API URL: Uses [suspicious link removed] for programmatic access.
- Environment Variables: Reads USERNAME and TOKEN to avoid hardcoding sensitive data.
- Command Line Arguments: Uses $1 and $2 to get the organization and repository names.
- curl Command: Makes an authenticated request to the GitHub API using the PAT.
- jq for JSON Parsing: Parses the JSON response from the API to extract the login (username) of collaborators with read access (permissions.pull is true).
- Conditional Output: Checks if the collaborator list is empty and provides a user-friendly message.

Assignments for Improvement

- Add a Comment Section: Improve the script by adding detailed comments explaining its purpose and required inputs.
- Implement a Helper Function: Create a function to provide usage instructions if the user runs the script with incorrect arguments.

Networking Concepts(Fundamentals)

Detailed Note from "Networking Concepts" YouTube Video
IP Address
• A unique identifier for a device on a network.
• Used for tracking, monitoring, and access control.
• IPv4 Standard: 32-bit address divided into four bytes (e.g., `172.16.3.4`). Each number ranges from 0 to 255.

Subnet
• A smaller, isolated network within a larger one, used for security, privacy, and isolation.
• Private Subnet: No internet access.
• Public Subnet: Has internet access, typically via a route table and internet gateway.

CIDR (Classless Inter-Domain Routing)
• Defines the range of IP addresses in a subnet.
• Represented as an IP address with a slash and a number (e.g., `172.16.3.0/24`).
• The number after the slash indicates the number of fixed bits.
• To calculate available IPs: `2^(32 - CIDR_number)`.
  • `/24` = 2^8 = 256 IPs.
  • `/27` = 2^5 = 32 IPs.

Ports
• A unique number to identify a specific application on a server.
• Directs incoming network traffic to the correct application.
• Accessed using the public IP address followed by the port number (e.g., `3.4.5.8:9191`).

Git and GitHub

Detailed Note from "Git and GitHub" YouTube Video

1. What is a Version Control System (VCS)?

- A system that helps manage and track changes in code over time.
- Solves two main problems:
  - Sharing of Code: Allows multiple developers to work on the same project and seamlessly integrate their changes.
  - Versioning: Keeps a history of all changes, allowing developers to revert to previous states of the code if needed.

2. Why Git is Popular: Centralized vs. Distributed VCS

- Centralized VCS (e.g., SVN): Uses a single central server. If the server goes down, work is halted.
- Distributed VCS (e.g., Git): Every developer has a full local copy (a "fork") of the repository. Work can continue even if the main server is offline.

3. Git vs. GitHub

- Git: The underlying open-source distributed version control technology that you install and run locally.
- GitHub: A platform built on top of Git. It provides a user-friendly interface and additional features like issue tracking, code reviews, and project management.

4. Git Life Cycle and Essential Commands

- git init: Initializes a new Git repository in your project folder.
- git status: Checks the current state of your repository (untracked, modified, staged files).
- git add <filename>: Adds a file to the staging area, marking it for the next commit.
- git diff: Shows the exact changes you've made to a file.
- git commit -m "message": Saves the staged changes as a new version (a "commit") in your local repository.
- git log: Displays the history of all commits.
- git reset --hard <commit-ID>: Reverts the repository to a previous state.

5. Sharing Code with GitHub

- To collaborate, you connect your local Git repository to a remote repository on GitHub.
- You can then git push your local commits to the remote repository, making your changes available to others.

Git Branching Strategy

Detailed Note from "Day-10 | Git Branching Strategy" YouTube Video
Understanding Branches
• Definition: A branch is a separation of your codebase, allowing new features to be developed without affecting the stable, main code.
• Purpose: To work on significant or breaking changes in isolation. Once the changes are tested and stable, they are merged back into the main branch.

Types of Branching Strategies
1. Master/Main Branch:
  • The primary branch for active development.
  • Should always be up-to-date with the latest code.
  • All other branches are eventually merged back into this one.
2. Feature Branches:
  • Created for new features or significant changes.
  • Developers collaborate on these branches.
  • Merged into the master branch after the feature is complete and tested.
3. Release Branches:
  • Created when a set of features is ready for a customer release.
  • Applications are built and shipped from this branch, not the master branch, to ensure stability.
  • No new development changes are introduced to a release branch during its

testing phase.
4. Hotfix Branches (or Bug Fix Branches):
  • Used for immediate fixes to critical issues found in production.
  • These are short-lived branches.
  • Changes must be merged into both the master and any active release branches to maintain consistency.

Practical Example: Kubernetes Branching Strategy
• Kubernetes, a large open-source project, uses these strategies effectively:
  • A `master` branch for active development.
  • Numerous `feature` branches for new functionalities.
  • `release` branches for each new version (e.g., `release-1.26`).

Key Takeaways and Interview Questions
• Most Common Branches: Master/Main, Feature, Release, Hotfix.
• Branch for Releases: Release branches.
• Purpose of Feature Branches: To introduce new, potentially breaking changes.
• Always Updated Branch: Master/Main branch.

Git Interview Q&A and Commands for DevOps

Detailed Note from "Day-11 | DevOps Course | Git Commands" YouTube Video

1. Initializing a Git Repository

  - git init: Initializes a local Git repository, creating a hidden .git folder to track changes, logs, and configurations.

2. Git Workflow: Add, Commit, Push

- git status: Shows the current state of the repository (e.g., untracked files).
- git add <filename>: Stages a file for the next commit, telling Git to start tracking it.
- git diff: Shows the changes made to a file since it was last staged.
- git commit -m "message": Records the staged changes to the repository with a descriptive message.
- git log: Displays the commit history.
- git push: Uploads local commits to a remote repository.

3. Remote Repositories and Cloning

- git clone <URL>: Downloads a remote repository to your local machine.
  - HTTPS: Requires a password for authentication.
  - SSH: Uses public/private keys for authentication.
- git remote -v: Shows the configured remote repositories.
- git remote add <name> <URL>: Adds a new remote repository.

4. Git Fork vs. Git Clone

- git clone: Downloads an existing repository.
- git fork: Creates a personal copy of a repository in your own GitHub account, allowing for independent development.

5. Branching in Git

- Purpose: To isolate development work and prevent breaking the main branch.
- git branch: Lists all branches.
- git checkout -b <new-branch>: Creates and switches to a new branch.
- git checkout <branch>: Switches between existing branches.

6. Merging Branches

- git cherry-pick <commit-ID>: Applies a single, specific commit from one

branch to another.
  - git merge <branch>: Integrates changes from another branch, creating a non-linear history with a "merge commit."
  - git rebase <branch>: Re-applies commits from one branch on top of another, creating a linear and cleaner commit history.

Deploy and expose your First App to AWS