DevOps Course Summary

1. Introduction to DevOps DevOps integrates infrastructure, cloud computing, and coding to streamline the software development lifecycle and improve collaboration between development and operations teams.

2. Key DevOps Tools

- **Ansible**: Configuration management.
- **Docker**: Containerization for application isolation.
- **Kubernetes**: Container orchestration for managing multiple containers.
- **Terraform**: Cloud automation and infrastructure as code.

3. Software Development Lifecycle (SDLC) in DevOps The SDLC in a DevOps environment involves multiple stages, each with specific goals and responsibilities:

1. **Requirement Analysis**:
   - Focus on gathering user requirements, understanding market needs, and identifying key product features.
2. **Planning**:
   - Determine what needs to be built, the associated costs, required resources, and risks involved.
3. **Design**:
   - Architects create a roadmap based on planning inputs and hand over design documents to developers.
4. **Development**:
   - Developers write code based on the design provided by the architects.
5. **Software Testing**:
   - Testing is performed to identify and fix issues before the software moves to production.
6. **Deployment**:
   - Operations teams deploy the software to a production environment, ensuring that it runs smoothly with minimal downtime.
7. **Maintenance**:
   - Monitor system health, performance, and uptime while managing regular changes and updates in the production environment.

4. Overview of the DevOps Approach to SDLC

- **Requirement Analysis**
- **Planning**
- **Design**
- **Development**
- **Testing**
- **Deployment and Maintenance**

This structured process ensures the continuous delivery of high-quality software with an emphasis on collaboration, automation, and efficiency.

DevOps SDLC Overview

1. Roles in the Software Development Lifecycle (SDLC)

- **Architects**: Design the software.
- **Developers**: Develop the software.
- **Testers**: Test the software.
- **Operations Team**: Deploy and maintain the software.

2. SDLC Models

- **Waterfall**: A linear, one-way model where no working software is available until the end. It takes a long time to produce a final product.
- **Spiral**: Combines iterative development with systematic aspects of the waterfall model.
- **Big Bang**: A development approach without clear planning or structure, often leading to unpredictable results.
- **Agile**: Breaks development into smaller, incremental sprints (2-4 weeks), delivering product features quickly with frequent client feedback.

3. Agile Model Advantages

- Frequent collaboration with clients ensures the product evolves based on customer needs.
- After each sprint, new features are tested and integrated into the application, allowing quick iterations and feature releases.

4. Operations and Deployment Challenges

- Developers regularly request operations teams to publish software on servers.
- Operations teams often face challenges due to unclear instructions from developers, leading to deployment issues and misalignment.
- Developers follow Agile methodologies, making frequent changes, while operations teams are often bound by ITIL processes (focused on stability), leading to conflicts.

5. Miscommunication Between Dev and Ops

- Developers work in agile, making rapid changes.
- Operations teams prioritize stability and follow waterfall-like approaches.
- Miscommunication between teams results in delayed deployments, poor product quality, and unsatisfied customers.

6. The Introduction of DevOps

- **DevOps** was introduced to bridge the gap between development (agile) and operations (waterfall).
- A DevOps engineer combines agile principles with operational stability.

7. Importance of Automation in DevOps

- Automation reduces manual intervention and improves efficiency in:
  - Code builds
  - Code and software testing
  - Infrastructure changes
  - Deployments
- Automated processes speed up the overall development lifecycle, facilitating faster product delivery.

8. Dev vs Ops Conflict

- Developers and operations teams often argue due to misaligned workflows (Agile vs Waterfall), leading to poor customer feedback.
- DevOps helps align these teams by promoting collaboration and automation.

9. DevOps Team Structure A successful DevOps team includes: 1. **Developers** 2. **Testers** 3. **System Administrators** 4. **Database Administrators** 5. **Build and Release Teams**

Each stage of the development process automates its tasks, integrating tools to streamline the workflow.

10. DevOps Lifecycle

- DevOps automates processes to reduce human intervention, enabling faster feedback loops between operations and development teams.
- This integration of automation tools accelerates the delivery of applications to customers, resulting in a faster and more efficient product lifecycle.

DevOps: Continuous Integration and Continuous Delivery

1. Continuous Integration (CI)

- **Definition**: CI is an automated DevOps process that enables developers to integrate code changes frequently, ensuring quick and efficient software generation.
- **Key Components**:
  - Developers write code and commit it to a version control system (e.g., GitHub).

  - The code is continuously built, tested, and stored as artifacts (e.g., WAR, JAR, ZIP).
  - Artifacts are deployed on servers, where they undergo additional testing.

2. Continuous Integration Process

1. **Code**: Developers write and commit code.
2. **Fetch**: The latest code is fetched from the repository.
3. **Build**: The code is compiled and built.
4. **Test**: The code is automatically tested.
5. **Notify**: The developer is notified of the results.
6. **Feedback**: Developers receive feedback on the build and test success.

- **Goal**: CI aims to detect errors early, during the development process, by automating build and testing for every code commit.

3. Version Control Systems (VCS) VCS stores and manages code versions, enabling collaboration: - **Examples**: Git, SVN, TFS, Perforce.

4. Build Tools Different tools are used based on the programming language: - **Examples**: Maven, Gradle, MSBuild, IBM UrbanCode.

5. Software Repository Tools Used to store and manage artifacts generated during the build process: - **Examples**: Sonatype Nexus, JFrog Artifactory, Archiva.

6. CI Tools Automation tools for the CI process: - **Examples**: Jenkins, CircleCI, TeamCity, Bamboo CI.

7. Continuous Delivery (CD)

- **Definition**: CD automates the process of delivering code changes to servers after CI has completed. It ensures efficient and error-free deployment of applications.
- **Process**: After successful builds and tests, the artifact is automatically deployed to the server with minimal manual intervention. This involves:
  - **Server provisioning**
  - **Dependency installation**
  - **Configuration changes**
  - **Network setup and artifact deployment**

8. Continuous Delivery Process Every step in the deployment process is automated: - **Automation Tools**: - **System Automation**: Ansible, Puppet, Chef. - **Cloud Infrastructure Automation**: Terraform, CloudFormation. - **CI/CD Automation**: Jenkins, Octopus Deploy. - **Helm Charts**: Used for

managing Kubernetes applications. - **Code Deploy**: Automates application deployment.

9. Automating Software Testing

- All types of testing (functional, load, database, security, performance) should be automated.
- **Ops team**: Automates deployment.
- **QA team**: Automates testing.
- **Developers**: Perform integrated CI (build, test).

10. Continuous Delivery Lifecycle

- **CI + Ops + QA = Continuous Delivery**
- Continuous delivery automates code integration, testing, and deployment, ensuring faster and more reliable releases.

DevOps Overview

What is DevOps? DevOps is a philosophy of merging development (Dev) and operations (Ops) teams to streamline collaboration, culture, practices, and tool integration.

Continuous Integration (CI)

Definition: Continuous Integration is a DevOps practice where developers frequently merge their code into a central repository. After each commit, automated builds and tests are triggered to catch errors early in the development cycle.

CI Process Steps: 1. Code: Developer writes the code. 2. Fetch: Fetch the latest code before deployment. 3. Build: Build the code. 4. Test: Test the new code for compatibility and functionality. 5. Notify: Notify developers about any errors or issues. 6. Feedback: Provide feedback if the build succeeds, allowing progression to production.

Version Control Tools: - Git - SVN - TFS - Perforce

CI Build Tools: - Maven, Ant, Gradle - MSBuild, Visual Build - IBM UrbanCode, Make

CI Repository Tools: - Sonatype Nexus - JFrog Artifactory - Archiva

CI Tools: - Jenkins - CircleCI - TeamCity - Bamboo

Continuous Delivery (CD)

Definition: Continuous Delivery automates the process of preparing code changes for release into production environments, ensuring fast and reliable deployments with minimal manual intervention.

CD Process: 1. The development team generates an artifact. 2. The artifact is built, tested, and sent to the operations team. 3. Automated steps handle server provisioning, configuration changes, dependency installation, and artifact deployment. 4. Continuous testing is performed automatically (functional, load, DB, security).

Automation Tools: - **System Automation:** Ansible, Puppet, Chef - **Cloud Infrastructure:** Terraform, CloudFormation - **CI/CD Automation:** Jenkins, Octopus Deploy - **Testing Automation:** Functional, Load, Security, Performance

---

Virtualization

What is Virtualization? Virtualization allows one physical machine to run multiple operating systems as isolated virtual machines (VMs), enabling efficient resource utilization.

Virtualization Components: - **Hypervisor:** A software tool that creates and manages VMs.
- **Type 1:** Bare-metal hypervisor (e.g., VMware ESXi, Xen). - **Type 2:** Software-based hypervisor for testing (e.g., Oracle VirtualBox, VMware Server).
- **Snapshot:** Backup method for VMs.

VM Setup: - Install OS manually or automate using Vagrant (VirtualBox as hypervisor).

Vagrant for Virtual Machine Setup

Vagrant Overview: Vagrant automates VM setup through predefined configurations called "Vagrant boxes." It manages virtual machines using a configuration file (Vagrantfile).

Tools Required: - Virtual Technology enabled in BIOS - Vagrant - Hypervisor (Oracle VirtualBox, etc.) - Command-line tools (Git Bash, Cygwin)

Common Vagrant Commands: - `vagrant up`: Start the VM. - `vagrant ssh`: SSH into the VM. - `vagrant halt`: Stop the VM. - `vagrant destroy`: Remove the VM. - `vagrant reload`: Reboot the VM.

Linux Basics for DevOps

Introduction to Linux: Linux is an open-source, Unix-like operating system kernel combined with GNU utilities. It is widely used in DevOps for server and infrastructure management.

Linux Architecture: - **Hardware:** Core component. - **Kernel:** Manages system resources and interacts with hardware. - **Shell:** Interface to execute Linux commands. - **Applications:** Layer around the shell for running software.

Popular Linux Distributions: - **RPM-based:** RHEL, CentOS, Oracle Linux - **Debian-based:** Ubuntu, Kali Linux

Linux File System: - `/root`: Admin home directory. - `/bin`, `/usr/bin`: User executables. - `/sbin`, `/usr/sbin`: System admin executables. - `/etc`: Configuration files. - `/var`: Server data (e.g., web server, MySQL data).

Common Linux Commands

Basic Commands: - `pwd`: Print current directory. - `ls`: List files in the directory. - `cd /`: Change to the root directory. - `uptime`: Show system uptime. - `free -m`: Show memory usage.

File Operations: - `mkdir <dir>`: Create a new directory. - `touch <file>`: Create a new file. - `cp <source> <destination>`: Copy files or directories. - `mv <source> <destination>`: Move or rename files. - `rm <file>`: Delete a file.

These summarized notes provide a structured overview of the core DevOps concepts, including CI/CD practices, virtualization, VM setup using Vagrant, and Linux basics.

Linux Commands and Utilities

Directory and File Operations

- `rm -r *`: Removes everything from the current directory.
- `rm -rf *`: Removes everything from the current directory forcefully.
- `history`: Displays all executed commands in the current session.

Vim Editor

- **Installation**: `sudo yum install vim -y`
- **File Creation/Editing**: `vim <filename>` creates or opens a file.

Vim Modes:

1. **Command Mode**: Default mode when a file is opened.
2. **Insert Mode**: Press `i` to enter, and insert text.
3. **Extended Command Mode**: Access using `:`

Vim Commands:

- `:wq`: Save and quit.
- `:q`: Quit without saving.
- `:q!`: Force quit without saving.
- `:se nu`: Display line numbers.
- `G`: Go to the last line.
- `gg`: Go to the first line.
- `yy`: Copy the current line.
- `p`: Paste below.
- `P`: Paste above.
- `u`: Undo.
- `5dd`: Delete the next 5 lines.
- `/word`: Search for a word.

File Types in Linux

- Use `file <filename>` to determine the file type.
- **File Type Indicators**:
  - `d`: Directory
  - `c`: Character file
  - `b`: Block file
  - `l`: Symbolic link

Commands:

- `mkdir -p <directory_path>`: Create a directory and any necessary parent directories.
- `ln -s <source_path> <destination_path>`: Create a symbolic link.

File Sorting and Hostname Configuration

- `ls -lt`: List files sorted by timestamp (latest first).
- `ls -lrt`: List files sorted by timestamp (latest last).
- Change hostname: Edit `/etc/hostname/` using vim.

File Search and Filters

Grep Command

- `grep <word> <file_path>`: Search for a word in a file.
- `grep -i <word> *`: Case-insensitive search in all files.

- `grep -v <word> <path>`: Exclude lines containing the word.

File Content Commands

- `less <file_name>`: View the file contents, better than `cat`.
- `head -n <num_lines> <file_name>`: Display the first n lines.
- `tail -f <file_name>`: Show dynamic content updates of a file.

Cut and Awk

- `cut -d<delimiter> -f<field> <file>`: Extract specific fields using a delimiter.
- `awk -F'<delimiter>' '{print $<field>}' <file>`: Filter and display specific fields.

Text Replacement

- Vim: `:%s/old/new/g` (global replacement).
- Sed: `sed -i 's/old/new/g' <file_name>` (global replacement with save).

Redirections

Output Redirection

- `> filename`: Redirect output to a file (overwrite).
- `>> filename`: Append output to a file.
- `/dev/null`: Discard output (e.g., `yum install vim -y > /dev/null`).

Memory and Disk Space

- `free -m`: Display memory usage.
- `df -h`: Display disk partition usage.

Error Redirection

- `command 2>> filename`: Capture standard error in a file.

Word Count and Pipes

- `wc -l <file_name>`: Count the number of lines in a file.
- `ls | wc -l`: Count the number of files in the current directory.
- `ls | grep <pattern>`: Search for files matching a pattern.

Find and Locate Commands

- `find /etc -name host*`: Search for files by name.
- `locate <file_name>`: Find files (requires `mlocate` installation).

Users and Groups

User Types in Linux

1. **Root User** (ID = 0)
2. **Regular User** (ID = 1000-60000)
3. **Service User** (ID = 1-999)

Password and Group Files

- `/etc/passwd`: Stores user information.
- `/etc/group`: Stores group information.

User Management

- `useradd <new_user>`: Add a new user.
- `groupadd <group_name>`: Add a new group.
- `usermod -aG <group_name> <user>`: Add user to a group (secondary group).
- `passwd <user>`: Set or reset the password.

Checking User and Group Information

- `id <user>`: Show user and group information.
- `tail -4 /etc/passwd`: Check the last 4 users added.
- `tail -4 /etc/group`: Check the last 4 groups added.

Track User Activity

- `last`: Show recent logins.
- `who`: Display currently logged-in users.
- `lsof -u <user>`: List open files by a user.

Deleting Users and Groups

- `userdel -r <user>`: Delete a user along with the home directory.
- `groupdel <group>`: Delete a group.

DevOps Notes

1. Users and Groups

- To reset the password for the current user:

  ```
  passwd
  ```

2. **File Permissions**

- Every file in Linux has its own permissions, which can be viewed using:

  ```
  ls -l <file_path>
  ```

  - The first character indicates the file type ('d' for directory, 'l' for link).
  - Next three characters (rwx) represent the permissions for the user, group, and others.
  - To check directory permissions:

    ```
    ls -ld <folder_name>
    ```

- **Changing ownership and permissions**:
  - Change ownership:

    ```
    chown user_name:group_name <folder_name>
    ```

  - Change permissions:

    ```
    chmod o-x <folder_name>  # Remove execute permission for others
    chmod o-r <folder_name>  # Remove read permission for others
    chmod g+x <folder_name>  # Add execute permission for group
    ```

- **Permission Methods**:
  - **Symbolic Method**: Modify permissions using letters (rwx).
  - **Numeric Method**: Represent permissions using numbers. For example:
    - 4 for read, 2 for write, 1 for execute.

3. **Sudo Privileges**

- **Sudo** allows a normal user to execute commands as the root user.
- Common commands:

  ```
  sudo yum install <package> -y
  sudo useradd <user_name>
  sudo -i  # Switch to root user
  ```

- **Adding a user to the sudoers file**:

  ```
  visudo
  ```

  Add the following line to give a user sudo privileges:

  ```
  ansible ALL=(ALL) ALL
  ```

- To allow a user to run sudo commands without a password:

  ```
  ansible ALL=(ALL) NOPASSWD: ALL
  ```

4. **Package Management**
   - Install packages using **yum**:

     ```
     yum install <package_name>
     yum remove <package_name>
     ```

   - **rpm** commands for installing, verifying, or deleting packages:

     ```
     rpm -i <package_file>    # Install package
     rpm -qa | grep <package>  # Check installed packages
     rpm -e <package_name>     # Remove package
     ```

5. **Services**
   - **httpd Service**: Install, check, and manage services like httpd (web server).
     - To install and check status:

       ```
       yum install httpd -y
       systemctl status httpd
       systemctl start httpd  # Start service
       ```

   - To enable a service at boot:

     ```
     systemctl enable <service_name>
     ```

6. **Processes**
   - **top**: Shows running processes dynamically.
   - **ps**: Shows static process info:

     ```
     ps aux
     ```

   - **Kill a process**:

     ```
     kill <process_id>
     kill -9 <process_id>  # Forcefully kill process
     ```

   - Use `awk` and `xargs` to filter and kill multiple processes:

     ```
     ps -ef | grep <process> | awk '{print $2}' | xargs kill -9
     ```

7. **Archiving**
   - **tar** for creating and extracting archives:
     - Create:

       ```
       tar -czvf <archive_name> <directory>
       ```

     - Extract:

```
tar -xzvf <archive_name>
```

- **zip** and **unzip**:

```
yum install zip unzip -y
zip -r <archive_name> <directory>
unzip <archive_name>
```

8. Ubuntu vs CentOS Commands

- **Package management in Ubuntu** uses `apt`:
  - Update repositories:
    ```
    apt update
    apt upgrade
    ```
  - Install package:
    ```
    apt install <package_name>
    ```
- **User creation in Ubuntu**:
  - Ubuntu:
    ```
    adduser <user_name>
    ```

9. Docker Introduction

- **Isolation**: Services are isolated using containers, which share the OS but run separately.
- Each service runs in a container with its own binaries, libraries, and dependencies.
- Containers are smaller than VMs and do not require a separate OS.

Docker Overview

Introduction to Containers

- Containers share the host machine's OS kernel, making them lightweight compared to virtual machines.
- Containers package software, including code and dependencies, into a standardized unit for development, shipment, and deployment.
- Unlike virtual machines, containers do not require a separate OS, as they use the host OS for compute resources.

Isolation vs Virtualization

- **Containers:** Provide OS-level isolation, utilizing the host machine's OS and sharing resources.
- **Virtual Machines (VMs):** Virtualize hardware, offering separate virtualized environments (e.g., virtual RAM, disk).
- Containers do not require a guest OS, whereas VMs need a fully independent OS.

Docker and Containers

- Docker is a platform that manages containers, acting as a container runtime environment.
- While containers can run without Docker, Docker simplifies container management by automating tasks like creating namespaces and c-groups.
- Docker was originally known as DotCloud and used Amazon EC2 to run applications using Linux Containers (LXC).
- Docker images, which are reusable and customizable, brought containerization into the spotlight, contributing to Docker's success.

Features of Docker Containers

- **Portability:** Docker images follow a standard, allowing containers to run anywhere with Docker installed.
- **Lightweight:** Containers don't need an OS, as they share the system's OS.
- **Security:** Docker provides robust isolation capabilities, ensuring applications inside containers remain secure.

Docker Setup

EC2 Instance Setup

1. **Launch EC2 Instance:** Choose an Amazon Machine Image (AMI), instance type (e.g., t2.micro for free-tier), and configure instance details.
2. **Add Storage:** Configure root volume and additional volumes as needed. Instance volumes are temporary and non-persistent.
3. **Add Tags:** Use key-value pairs to identify and manage instances.
4. **Configure Security Groups:** Set inbound and outbound rules for network traffic, typically allowing SSH access from your IP address.
5. **Review and Launch:** After configuring, download the key pair to securely access the instance via SSH.

Accessing EC2 Instances

- Use a key pair (private key) to securely SSH into an EC2 instance.
- Example SSH command: `bash ssh -i <path-to-.pem-file> ubuntu@<ip-address>`

Installing Docker on Ubuntu

1. Follow official Docker installation steps for Ubuntu Docker Install Guide.
2. Verify Docker installation:

```
systemctl status docker
docker images
```

3. Allow non-root users to access Docker daemon:

   ```
   sudo usermod -aG docker ubuntu
   ```

Docker Commands and Concepts

Working with Docker Containers

- **Run Docker Container:**

  `docker run hello-world`

  This creates and runs a container from the `hello-world` image.

- **List Containers:**
  - Running containers:

    `docker ps`

  - All containers (including stopped ones):

    `docker ps -a`

- **Starting and Stopping Containers:**
  - Start container:

    `docker start <container-id>`

  - Stop container:

    `docker stop <container-id>`

Docker Images

- Docker images consist of multiple layers, where each `RUN` command in a Dockerfile creates a new layer.
- Images can be stored in registries like Docker Hub, Google Container Registry (GCR), or Amazon Elastic Container Registry (ECR).
- Images are read-only, and containers are created from these images with an additional writable layer.

Docker Networking

- Containers can be run in detached mode (`-d`), allowing background execution, and port mapping can be specified using the `-p` option:

  `docker run --name myweb -p 7090:80 -d nginx`

  This maps port 7090 on the host to port 80 inside the container.

Executing Commands in Containers

- Run commands inside a container:

  ```
  docker exec <container-name> <command>
  ```

- Access the container's shell:

  ```
  docker exec -it <container-name> /bin/bash
  ```

Managing Docker Containers and Images

- **Remove container:**

  ```
  docker rm <container-id>
  ```

- **Remove image:**

  ```
  docker rmi <image-id>
  ```

  (Note: Containers using an image must be stopped before the image can be removed.)

Docker Commands and Concepts

Docker Logs

- Command: `docker logs <container-name/container-id>` retrieves container logs.
- For viewing image metadata: Use `docker inspect <image-name>`, which provides output in JSON format.
- In detached mode, logs can be viewed using the `docker logs` command since output isn't visible in the shell.
- Logs only show the output of executed commands inside the container, not application-specific logs.

Environmental Variables

- Variables can be exported as environmental variables during container runtime.
- Command: `docker run -d -P -e <VARIABLE_NAME>=<variable_value> <image-name>`.

Docker Volumes

- Docker provides two methods for persistent data storage:
  1. **Volumes**: Managed by Docker, typically located at `/var/lib/docker/volumes/`.
  2. **Bind Mounts**: Stored anywhere on the host system.

Docker Volume Commands

- Create a volume: `docker volume create mydbdata`.
- List volumes: `docker volume ls`.
- Mounting a volume: `-v hostlocation:remotedirlocation`.

Accessing Containers

- To enter a running container's shell: `docker exec -it <container-name> /bin/bash`.

Docker Volume vs. Bind Mounts

- **Bind Mounts**: Used for adding files during container execution.
- **Volumes**: Preferred for data persistence, allowing more control over where data is stored.

Container Removal

- Remove a container: `docker rm <container-name>`.
- Remove an image: `docker rmi <image_name>`.

Docker Image Building

Dockerfile Instructions

- **FROM**: Specifies the base image.
- **LABEL**: Adds metadata.
- **RUN**: Executes commands.
- **ADD/COPY**: Transfers files into the container.
- **CMD**: Specifies the default command to run.
- **ENTRYPOINT**: Defines an executable that will always run.
- **VOLUME**: Defines a mount point for external volumes.
- **EXPOSE**: Declares ports on which the container listens.
- **ENV**: Sets environment variables.
- **USER**: Defines the user.
- **WORKDIR**: Sets the working directory.
- **ARG**: Allows passing build-time variables.
- **ONBUILD**: Adds trigger instructions for later stages.

Optimizing Dockerfile

- Reduce layers by combining multiple `RUN` commands using `&` to minimize image size.

Docker Image Example

- Base Image: `ubuntu:latest`.
- Commands:
  1. Install updates and packages (e.g., Apache2).
  2. Set Apache to run in the foreground.
  3. Expose port 80 to the host.
  4. Define a working directory and volume for log data.
  5. Use `docker run -d --name nanowebsite -p 9080:80 nanoimg:V2` to run the image in detached mode, mapping port 9080 on the host to port 80 in the container.

Pushing Docker Images to Docker Hub

Steps to push an image: 1. Sign up on Docker Hub. 2. Create a repository. 3. Login to Docker: `docker login`. 4. Push image: `docker push <DockerHubUName>/<ImageName>:<VersionName>`.

Docker Compose

Overview

Docker Compose is used to run multi-container applications. It defines services in a YAML file and allows running all containers with a single command: `docker compose up`.

Steps to Run Docker Compose

1. Define environment in a `Dockerfile`.
2. Configure services in `docker-compose.yaml`.
3. Run `docker compose up`.

Common Commands

- `docker-compose ps`: View running containers.
- `docker compose up -d`: Run in detached mode.
- `docker compose down`: Stop and remove containers.

Multi-Stage Dockerfile

Purpose

- Reduces the image size by separating build and runtime stages. Dependencies and source code are built in one stage, while only necessary artifacts are copied to the final image.

Example

1. First stage downloads dependencies.
2. Second stage copies artifacts, discarding unnecessary data.

This summary covers the major Docker concepts, commands, and best practices as discussed in the provided transcript.

DevOps Notes: Containerization & Kubernetes

1. Containerization: Build & Run Microservice App

Amazon EC2 Instance Setup

- When using multiple containers dependent on each other, local machines may not suffice.
- A **t3-medium** instance (chargeable) is recommended for better performance; **t2-micro** is free but insufficient.
- Create a key pair for SSH access and allow HTTP for internet access (e.g., for Nginx).
- Modify storage to 20GB and add necessary Dockerfile code in the **User Data** section during instance creation.

Connecting to EC2 Instance

- Access the instance via SSH using the downloaded PEM file:

  ```
  ssh -i Downloads/dockerKey.pem ubuntu@<IP-Address>
  ```

- Pre-installed Docker and Docker Compose are ready based on the user data entered earlier.

Building Docker Images

- Navigate to the project directory and use:

  ```
  docker-compose build
  ```

  This command builds all images declared in the `docker-compose.yaml`.

Running Containers

- To run the built images, use:

  ```
  docker-compose up
  ```

- Monitor running containers with:

  ```
  docker ps
  ```

- View logs for individual containers:

```
docker logs <container-id>
```

Updating and Rebuilding Images

- After modifying the code, rebuild using:

  ```
  docker-compose build
  ```

  This ensures only changes are built, not the entire image.

Stopping Containers

- Stop and remove all running containers:

  ```
  docker-compose down
  ```

Cost Control

- Stop the instance when not in use to avoid charges.

---

2. Kubernetes: Introduction

What is Kubernetes?

- Kubernetes is a popular container orchestration tool.
- It manages clusters of Docker engines to ensure high availability.
- If one Docker engine fails, others take over the workload.

Kubernetes Architecture

- **Nodes**: Docker engines in a cluster.
- **Master Node**: Controls and monitors the worker nodes, redistributing workloads when necessary.
- **Worker Nodes**: Execute tasks as instructed by the master node.

Container Orchestration Tools

1. Docker Swarm
2. Kubernetes
3. Mesosphere Marathon
4. AWS ECS & EKS
5. Azure Container
6. Google Container Engine
7. CoreOS Fleet
8. OpenShift

Kubernetes Origins

- Initially created by Google (known as **Borg**).
- Partnered with the Linux Foundation to form the **Cloud Native Computing Foundation (CNCF)**.
- Other tools include **Minikube**, **Kubeadm**, **Kops**, and **Istio** (a load balancer).

---

3. Key Kubernetes Features

1. Service Discovery & Load Balancing

- Containers are called **pods** in Kubernetes.
- Kubernetes automatically balances the load across pods and replicates failed pods.

2. Storage Orchestration

- Supports storage solutions like **SAN**, **NetApp**, **AWS EBS**, **CephFS**, allowing easy rollback for stateful containers.

3. Automated Rollouts & Rollbacks

- Kubernetes automates application updates and rollbacks.

4. Automated Bin Packing

- Ensures containers are placed on the appropriate nodes with suitable resources.

5. Self-Healing

- If a node fails, Kubernetes automatically replaces it with a replica, maintaining service availability.

6. Managing Secrets & Configurations

- Kubernetes allows secrets and configurations to be managed as variables and volumes, using encoded values.

DevOps Notes: Kubernetes Architecture

1. Kubernetes Architecture Overview

- Kubernetes has two types of nodes:
  - **Master Node (Control Plane):** Manages the cluster and worker nodes.
  - **Worker Nodes:** Where Docker engines run the containers.

- Communication with worker nodes is done through the master node.
- The master node's services communicate with the worker nodes via **YAML files**.

2. Master Node Services

Kube API Server

- Handles communication across the Kubernetes cluster.
- Acts as the frontend for the Kubernetes cluster.
- Administrators use **kubectl** to interact with the API server.

ETCD Server

- A key-value store that holds configuration, state data, and service discovery details of the cluster.
- **Backup is essential** to avoid data loss in case of failure.

Kube Scheduler

- Assigns newly created pods to appropriate nodes based on factors like resource requirements, hardware/software constraints, and data locality.
- Supports **affinity/anti-affinity** rules to control pod placement.

Controller Manager

- Manages different controllers (e.g., Node, Replication, Endpoints) in a single process.
- Key controllers:
  - **Node Controller:** Detects and responds to node failures.
  - **Replication Controller:** Ensures the correct number of pods are running.
  - **Endpoints Controller:** Connects services and pods.
  - **Service Account & Token Controllers:** Manages authentication and authorization.

---

3. Worker Node Components

Kubelet

- Agent running on each worker node.
- Ensures containers within pods are running by fetching and executing container images as instructed by the scheduler.

Kube Proxy

- A network proxy running on each worker node.
- Allows custom network configurations such as IP allow/deny lists.

Container Runtime

- Kubernetes supports various container runtimes (e.g., **Docker**, **containerd**, **cri-o**).
- Unlike Docker Swarm, Kubernetes supports multiple container runtimes.

---

4. Additional Worker Node Components (Add-ons)

- **DNS**: Provides domain name resolution for containers.
- **WebUI**: User interface for cluster management.
- **Container Resource Monitoring**: Tracks resource usage of containers.
- **Cluster Level Logging**: Logs cluster activity, often sourced from third-party vendors.

---

5. Pod and Container Architecture

Pod Structure

- A **pod** contains one or more containers and provides resources for them (similar to how a virtual machine runs processes).
- The container runs inside the pod, and the pod provides isolation, not virtualization.

Pod Communication

- Each pod has an IP address, enabling communication via the pod's IP and the container's port number.
- Pods can share **volumes** with containers, providing persistent storage across containers within the same pod.

DevOps Notes: Kubernetes Tools and Objects

1. Pods and Containers in Kubernetes

- **Pod Structure**:
  - A pod can contain one main container and multiple helper or **init containers**.
  - **Main containers** cannot run together in the same pod, e.g., MySQL and Tomcat can't be in one pod.
  - Helper containers assist the main container but have limited roles.

2. Pod Networking and Communication

- **Worker Node Networking**:
  - Pods across multiple worker nodes can interact using a **subnet** or private network.
  - **Bridge zero** acts like a switch for communication within the same node.
  - **wg0 (router)** forwards requests to the correct node based on IP addresses, allowing pods from different nodes to communicate.

3. Kubernetes Cluster Tools

Minikube:

- Used for **testing and learning** purposes with a single-node Kubernetes cluster.
- Runs Kubernetes on a local VM (using VirtualBox).
- Ideal for setting up a local Kubernetes cluster but not suited for production.

Kubeadm:

- Popular for setting up **production Kubernetes clusters**.
- Supports multiple nodes (both master and worker) on various platforms (VMs, EC2, physical machines).

Kops:

- Best for **multi-node Kubernetes clusters** on AWS and other cloud platforms like Google Cloud, Digital Ocean, etc.
- Stable for production use.

4. Minikube Setup Process

- **Installing Minikube**:
  1. Open **PowerShell as Administrator**.
  2. Install Chocolaty, then use it to install Minikube (`choco install minikube kubernetes-cli -y`).
  3. Start Minikube (`minikube start`), which creates a local Kubernetes cluster inside a VM.
- **Basic Kubernetes Commands**:
  - `kubectl get nodes`: Check the nodes running in the cluster.
  - `kubectl get pod`: List all available pods.
  - `kubectl expose deployment <deployment-name>`: Expose a deployment.
  - `minikube service <service-name> --url`: Get the URL to access the exposed deployment.

5. Minikube Cluster Maintenance

- **Stopping and Deleting**:
  - `minikube stop`: Stops the Minikube cluster.
  - `minikube delete`: Deletes the Minikube VM.

6. Kops Setup Process

- **Kops Setup for AWS**:
  1. Purchase a domain (e.g., **groophy.in**) for Kubernetes DNS records.
  2. Create a **Linux VM** and install **kops, kubectl, ssh keys, and awscli**.
  3. Set up an S3 bucket and configure **IAM user** for AWS CLI.

7. Kubernetes Objects

Key Kubernetes Objects:

- **Pod**: The smallest object in Kubernetes. Changes to containers are made via pods.
- **Service**: Provides a **static endpoint** (like a load balancer) for a pod.
- **Replica Set**: Ensures **replication** of pods for scalability.
- **Deployment**: Most commonly used object for rolling out updates or deploying new versions.
- **ConfigMap**: Stores **configuration variables** for applications.
- **Secret**: Used to store **sensitive information**, like credentials, without exposing them in plaintext.
- **Volumes**: Provides persistent storage to pods, similar to **EBS volumes** in AWS.

8. Managing Kubernetes Objects

- **Using YAML**:
  - Define multiple objects in a **YAML file** and apply them using `kubectl apply <url-for-yaml>`.
  - Example commands:
    - `kubectl get deployments`: Check if the deployment is created.
    - `kubectl delete deploy <deployment-name>`: Deletes a deployment and its associated pod.

DevOps Notes: Kubernetes Configuration and Namespace Management

1. Kube Config File

- **Purpose**: The kube config file helps `kubectl` connect to the Kubernetes cluster and access cluster resources.

- **Contents**:
  1. **Cluster Information**: Details about the Kubernetes cluster.
  2. **User Information**: Specifies users accessing the cluster.
  3. **Namespaces**: Grouping of resources in the cluster.
  4. **Authentication Mechanisms**: Similar to SSH, including IP address, username, password, or key-based authentication.
- **File Location**: `~/.kube/config` holds details like cluster information, server certificates, and API server details.
- **How `kubectl` Connects**:
  - The kube config file provides the cluster's **API server** address, which connects to the master node.
  - IP addresses of the master node are stored in **Route53** (for AWS setups via Kops) and can be verified in **EC2 Instances**.
- **Contexts**:
  - A kube config file may have **multiple contexts** (combinations of clusters and users).
  - **Current-context** defines the active cluster and user configuration that `kubectl` uses when commands are executed.
- **Automation Integration**:
  - Kube config is required for CI/CD tools like **Jenkins** or **Ansible**.

2. Namespaces in Kubernetes

- **Purpose**: Namespaces allow for resource grouping and isolation within the Kubernetes cluster.
- **Default Namespaces**:
  - **Default**: Standard namespace.
  - **kube-system**: Contains control plane resources (e.g., ETCD manager, KOPS-controller, Kube-Proxy).
  - **kube-public**: Public resources.
- **Namespace Usage**:
  - Useful for dividing resources between **projects or environments** (e.g., development, production).
  - Resources need to be unique within a namespace but not across namespaces.
  - Namespace-based scoping applies to namespaced objects (e.g., Deployments, Services), not cluster-wide objects (e.g., Storage classes, Nodes).
- **Namespace Commands**:

  - List all namespaces: `kubectl get ns`
  - List objects in default namespace: `kubectl get all`
  - List objects across all namespaces: `kubectl get all --all-namespaces`
  - Get service object from another namespace: `kubectl get svc -n <namespace-name>`

- **Namespace Creation and Deletion**:
  - Create a namespace: `kubectl create ns <namespace-name>`
  - Create a pod in a specific namespace: `kubectl run <pod-name> --image=<image-name> -n <namespace-name>`
  - Apply objects using a YAML file: `kubectl apply -f <yaml-file>`
  - Delete a namespace and its contents: `kubectl delete ns <namespace-name>`

DevOps Notes: Kubernetes Pods and Services

1. Pods in Kubernetes

- **Definition**: A pod is the smallest and most basic unit in Kubernetes, encapsulating containers. It represents a process running in the cluster.
- **Structure**:
  - Typically, one container per pod is the common setup, but **multi-container pods** can be created where additional containers serve as helper or init containers.
  - **Pod Abstraction**: Kubernetes manages pods, not containers directly. Commands are executed at the pod level.
- **Resource Sharing**: Containers in a multi-container pod share the same resources provided by the pod.
- **Pod Scaling**:
  - **Horizontal Scaling**: Adding more nodes to the cluster.
  - **Vertical Scaling**: Increasing the resources (CPU, memory) of the current node.
- **Pod Definition File**:
  - Pods are defined using YAML files, which typically include:
    * **apiVersion**: Defines the version of the API (e.g., `v1` for Pod, `apps/v1` for Deployment).
    * **kind**: Type of object (Pod, Service, Deployment).
    * **metadata**: Contains information like pod name, labels (key-value pairs).
    * **spec**: Includes container configurations (image, ports, etc.).
- **Important Commands**:
  - Create an object: `kubectl create -f <yaml-file>`

  - Get pod details: `kubectl get pod`
  - Describe pod: `kubectl describe pod <pod-name>`
  - Get pod details in YAML format: `kubectl get pod <pod-name> -o yaml`
  - Delete a pod: `kubectl delete pod <pod-name>`

2. **Pod Logging and Troubleshooting**

- **Common Pod States**:
  - **Running**: Pod is working as expected.
  - **ImagePullBackOff**: Indicates an issue with pulling the container image.
  - **CrashLoopBackOff**: Occurs when a container is failing to restart repeatedly.
- **Troubleshooting Commands**:
  - View pod logs: `kubectl logs <pod-name>`
  - Check for errors: `kubectl describe pod <pod-name>`
  - Get pod status with detailed logging: `kubectl get pod -o wide`

3. **Services in Kubernetes**

- **Purpose**: A service connects pods and exposes applications running in pods to external or internal networks.
- **Why Use Services?**
  - Pods have **dynamic IPs**, which change upon pod replacement. Services provide a **static endpoint** to ensure consistent communication.
- **Types of Services**:
  - **NodePort**: Exposes the pod on a specific port on the node. Typically used for development or non-production purposes.
  - **ClusterIP**: Provides internal network connectivity within the cluster. Ideal for internal services like communication between Tomcat and MySQL.
  - **LoadBalancer**: Used for production use cases where the service needs to be exposed to the internet. Automatically integrates with cloud providers' load balancers.
- **Service Structure**:
  - Services connect to pods using **label selectors**, which match the labels of the target pods.
  - Each service has its own **IP address**, independent of the pod IP address.
- **Service Commands**:
  - Create a service: `kubectl create -f <yaml-file>`

- Get service details: `kubectl get svc`
- Describe service: `kubectl describe svc <service-name>`

Kubernetes Notes

1. Kubernetes Pods

- **Pods**: The most basic unit in Kubernetes. A pod runs applications in isolated environments with containers inside them.
  - Pods are wrappers around containers, and Kubernetes manages pods instead of containers directly.
  - A pod represents a process running in the cluster, usually running one container per pod.
  - In multi-container pods, there's usually one main container, with additional helper containers.
- **Multi-container Pods**: When running an application with components like Tomcat, MySQL, and RabbitMQ, each component should be in its own pod. It is better to avoid putting multiple containers in one pod for these types of applications.
- **Horizontal and Vertical Scaling**:
  - **Horizontal Scaling**: Adding more nodes to the cluster.
  - **Vertical Scaling**: Increasing the capacity (CPU, memory) of existing nodes.

2. Pod YAML Structure

- **YAML Components**:
  - `apiVersion`: Defines the version of the Kubernetes API.
  - `kind`: Describes the object type (e.g., Pod, Service, Deployment).
  - `metadata`: Includes information like the pod's name and labels.
  - `spec`: Specifies the containers and their configurations.
  - Common `apiVersion` values:
    - Pod: `v1`
    - Service: `v1`
    - Deployment: `apps/v1`
    - Ingress: `networking.k8s.io/v1beta1`

3. Kubectl Commands

- `kubectl create -f <yaml-file>`: Creates a Kubernetes object from a YAML file.
- `kubectl get pod`: Retrieves details about running pods.
- `kubectl describe pod <pod-name>`: Gets detailed information about a specific pod (similar to `docker inspect`).

- `kubectl edit pod <pod-name>`: Allows partial editing of a pod's configuration.

4. **KOPS Cluster**
   - **KOPS Commands**:
     - `kubectl validate`: Validates the cluster, showing the number of master and worker nodes.
     - `kubectl get nodes`: Retrieves information about the cluster nodes.
     - `kubectl describe node <node-name>`: Provides detailed information about a specific node.

5. **Pod Logging and Debugging**
   - **Common Issues**:
     - `ImagePullBackOff`: Indicates an issue pulling the image due to a wrong name.
     - `CrashLoopBackOff`: Occurs when a container repeatedly fails to start.
   - **Key Commands**:
     - `kubectl logs <pod-name>`: Retrieves logs from a container within the pod.
     - `kubectl get pod -o yaml`: Provides the YAML configuration of the pod for debugging.

6. **Kubernetes Services**
   - **Service Purpose**:
     - Connects external requests to the internal pod network.
     - Exposes applications running inside pods to external users.
     - Provides dynamic IP addresses to avoid issues when replacing pods.
   - **Service Types**:
     - **NodePort**: Exposes the service on a fixed port number, allowing access from outside the cluster.
     - **ClusterIP**: Internal communication within the cluster, often used for backend services like MySQL.
     - **LoadBalancer**: Exposes services to the internet, typically used for production.
   - **NodePort Example**:
     - A service object has frontend and backend ports that map the requests to the correct pods using labels.
     - External port (e.g., `30001`) maps to the internal pod's container port (e.g., `8080`).
   - **LoadBalancer Example**:
     - Automatically creates an Elastic Load Balancer (ELB) in AWS, mapping the service to a pod.

- Service rules route requests to the correct pods and return responses via the load balancer.

7. **ClusterIP Service**

- **Usage**: For internal communication between pods. For example, a pod wanting to access a Tomcat service will interact with the ClusterIP service on port `8080`.

8. **General Commands**

- `kubectl get svc`: Retrieves all services running in the cluster.
- `kubectl describe svc <service-name>`: Provides detailed information about a specific service.
- `kubectl get all`: Displays all running objects in the current namespace.

This summary provides an overview of Kubernetes components, including pods, services, scaling, and common commands.

Kubernetes Notes

1. **ReplicaSet**

- **Purpose**:
  - Ensures the availability of a specified number of identical pods.
  - Automatically replaces a pod if it crashes or goes down, eliminating manual intervention.
  - Distributes pods across worker nodes to balance load and availability.
- **ReplicaSet YAML Structure**:
  - **apiVersion**: Typically `apps/v1`.
  - **spec**: Contains the number of replicas and `matchLabels` to select pods.
  - **matchLabels**: Ensures that new replicas created match the specified pod labels.
  - Example: If `replicas` is set to 3 and `matchLabels` is set to `frontend`, the ReplicaSet will ensure there are always 3 pods running with the `frontend` label.
- **Key Commands**:
  - `kubectl get rs`: Check if the ReplicaSet is created and running.
  - `kubectl get pod`: View all pods created by the ReplicaSet.
  - **Scaling**:
    - `kubectl scale --replicas=<num> <replica-set-name>`: Scale the ReplicaSet up or down.
    - `kubectl edit rs <replica-set-name>`: Modify the ReplicaSet configuration directly.
- **Best Practices**:

- Avoid direct scaling using commands in production; always modify and apply the YAML file to scale ReplicaSets.
- Delete a ReplicaSet with `kubectl delete rs <replica-set-name>` when no longer needed.

---

2. Deployment

- **Purpose**:
  - Automates rolling updates and rollback of applications.
  - Ensures that any changes to the pod (e.g., container image updates) are applied in a controlled manner.
  - Manages and updates ReplicaSets to maintain the desired state of the application.
- **Deployment Workflow**:
  - **Creates ReplicaSet**: A Deployment defines a ReplicaSet that manages the pods.
  - **Desired State**: Deployment defines the desired state (e.g., number of replicas, container image version).
  - **Rolling Updates**: When changes are applied (e.g., updating image versions), pods are updated one by one without downtime.
- **Use Case**:
  - If you're running a pod with version `v1` of an image and want to update to version `v2`, the Deployment will update each pod one at a time, ensuring the application remains available.
- **Key Commands**:
  - **Creating a Deployment**:
    - `kubectl apply -f <deployment-file>`: Creates or updates a Deployment.
  - **Scaling**:
    - `kubectl scale deployment/<deployment-name> --replicas=<num>`: Scale the number of pods in a Deployment.
  - **Updating a Deployment**:
    - Modify the deployment YAML file (e.g., change the container image version) and apply the changes.
    - New pods will be created by the new ReplicaSet, while the old ReplicaSet will be scaled down.
  - **Rolling Back Changes**:
    - `kubectl rollout undo deployment/<deployment-name>`: Roll back to the previous version of the Deployment.
    - `kubectl rollout undo deployment/<deployment-name> --to-revision=<revision_number>`: Roll back to a specific revision.
- **Rollout Management**:
  - `kubectl rollout history deployment/<deployment-name>`:

View the history of changes applied to a Deployment.
  - **Rollbacks**:
    * Revert to a previous Deployment version if issues arise.
    * Old ReplicaSets are reinstated, and new ReplicaSets are scaled down.
- **Best Practices**:
  - For updates in production, always define changes in the YAML file and apply them, allowing for controlled rollouts and potential rollbacks.

Kubernetes Notes

1. Commands and Arguments

- **Overview**: In Kubernetes, commands and arguments for a container are defined at the pod level. Containers within a pod execute the commands, not the pod itself.
- **CMD and ENTRYPOINT**:
  - **CMD**: Provides default values for the container's execution, with only the last CMD in a Dockerfile being executed.
  - **ENTRYPOINT**: Has higher priority than CMD and can work with CMD to define executables and default parameters.
  - **Kubernetes Execution**: In a pod definition file, commands and arguments are passed using the `command` and `args` fields under the `spec.containers` section. This allows overriding image parameters.

---

2. Volumes

- **Overview**: Volumes in Kubernetes allow persistent storage for pods. This is essential as pods are temporary and data can be lost when they are deleted or terminated.
- **Types of Volumes**:
  - **HostPath**: Maps a directory from the host node to the pod. Changes in the pod persist in the host.
  - **Persistent Volumes**: Separate storage that persists beyond the lifecycle of the pod, often backed by cloud storage like Azure or AWS.
- **HostPath Configuration**:
  - **volumeMounts**: Defines where the volume will be mounted in the container.
  - **volumes**: Defines the source directory for the volume in the worker node, typically using `DirectoryOrCreate` to ensure the directory exists.

---

3. **ConfigMap**

- **Overview**: ConfigMaps allow you to inject configuration data into pods without hardcoding it into the container images. It is useful for handling environment variables or configuration files that change frequently.
- **Creating ConfigMaps**:
  - **Imperative Command**:
    - `kubectl create configmap <name> --from-literal=<key>=<value>`.
    - `kubectl get cm`: View all ConfigMaps.
    - `kubectl describe cm <config-map-name>`: Get detailed information about a ConfigMap.
  - **Declarative Format**: Define the ConfigMap in a YAML file under `metadata` and `data`, then apply it using `kubectl`.
- **Using ConfigMaps in Pods**:
  - **Environment Variables**: Use `spec.containers.envFrom.configMapRef` or `spec.containers.env.valueFrom.configMapKeyRef` to map the ConfigMap to a pod.
  - **Volume Mounting**: Use `volumeMounts` and `volumes` sections to mount the ConfigMap as files in a pod. The keys in the ConfigMap can be referenced and stored at specific paths in the container.
- **Limitations**:
  - **Non-confidential Data**: ConfigMaps are not designed for sensitive information. For confidential data, use **Secrets** instead.
  - **Final Output**: After mounting, environment variables or files from the ConfigMap can be accessed within the container.

Kubernetes Notes

1. **Secrets in Kubernetes**

- **Overview**: Secrets in Kubernetes store and manage sensitive data such as passwords, API keys, and tokens. Unlike ConfigMaps, which handle non-sensitive data, Secrets are used to ensure security by encoding or encrypting sensitive information.

---

2. **Creating Secrets**

- **Imperative Way**:
  - **Command**: `kubectl create secret generic <secret-name> --from-literal=<key>=<value>`
    - Example: `kubectl create secret generic db-secret --from-literal=MYSQL_ROOT_PASSWORD=somecomplexpassword`

    - Note: The values are **base64-encoded**, not encrypted.
- **For Files**:
  - **Command**: `kubectl create secret generic <secret-name> --from-file=<path-to-file>`
    - Example: `kubectl create secret generic db-user-pass --from-file=./username.txt --from-file=./password.txt`

---

3. Base64 Encoding and Decoding

- **Encoding**: `echo -n 'secretpass' | base64` (Encodes the value to base64)
- **Decoding**: `echo -n 'encodedvalue' | base64 --decode` (Decodes base64 back to its original value)
- Although base64 encoding adds a layer of security, it's not encryption. The encoded data is visible to Kubernetes control plane, but generally secure unless access to the control plane is compromised.

---

4. Declarative Format for Secrets

- **Creating Secrets in YAML**:
  - The secret values should be encoded in base64 before being placed in the YAML file under the `data` section.
  - Example:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
data:
  MYSQL_ROOT_PASSWORD: encoded_value_here
```

- **Using Secrets in Pods**:
  - Use `spec.containers.envFrom.secretRef` to refer to the secret in the pod's specification.

---

5. Docker Registry Secrets

- When pulling images from a private repository, Docker credentials are required. These credentials can be stored as a secret of type **docker-registry**.
  - **Command**:

```
kubectl create secret docker-registry <secret-name> \
--docker-email=<email> \
--docker-username=<username> \
```

```
--docker-password=<password> \
--docker-server=<registry-url>
```

- **Using Docker Registry Secrets in Pods**:
  - Specify the secret in `imagePullSecrets` under the pod specification, allowing the pod to pull the image from a private registry without providing credentials each time.

---

6. Retrieving Secrets

- To view the contents of a secret, use the following command and decode the output:
  - **Command**: `kubectl get secret <secret-name> -o jsonpath='{.data.*}' | base64 -d`

---

7. Running Commands in a Pod with Secrets

- **Command**: `kubectl exec --stdin --tty <pod-name> -- /bin/bash`
  - This command opens a bash terminal inside a running pod that uses secrets.

Kubernetes Ingress

Definition and Purpose

- **Ingress** is an API object that manages external access to services in a Kubernetes cluster, primarily through HTTP/HTTPS.
- It provides key features such as:
  - Load balancing
  - SSL termination
  - Name-based virtual hosting

Functionality

- Ingress exposes HTTP/HTTPS routes from outside the cluster to services within.
- The user interacts with the cluster, and the Ingress routes the request to the correct service based on predefined rules.
- Ingress acts as a **load balancer** and manages external access while providing SSL termination.

Ingress Controller

- To use Ingress, an **Ingress Controller** is needed, such as **NGINX**.
- The **NGINX Ingress Controller** works with the NGINX web server and acts as a proxy.

- The Ingress Controller can route requests from users to services within the cluster.

NGINX Ingress in AWS

- In AWS, a **Network Load Balancer (NLB)** is used to expose the NGINX Ingress Controller.
- Example commands to interact with namespaces and Ingress:
  - Check if a namespace exists: `kubectl get ns`
  - Check all objects in a namespace: `kubectl get all -n ingress-nginx`
- NGINX Controller pods route requests from the user to the appropriate service.

Creating and Configuring Ingress

1. **Create Deployment, Service, and Ingress:**
   - Use YAML files to define deployment, service, and ingress.
   - Ingress connects external users, load balances, and routes to the appropriate service, which accesses the necessary pod.
2. **Load Balancer and DNS Configuration:**
   - Retrieve DNS name from the Load Balancer section in AWS.
   - Create a CNAME record for this DNS name in a domain management tool (e.g., GoDaddy).
   - CNAME settings: `TYPE = CNAME`, `Name = Random`, `Value = DNS Name from AWS`.

Important Considerations

- Ensure the service is active and running before creating Ingress.
- Use `kubectl describe svc <service-name>` to find service endpoints and bind them to the Ingress.

Sample Ingress Setup Process

1. Set up **Ingress Controller**.
2. Deploy the **Service** (must be working and running).
3. Create a **DNS CNAME** record for the Load Balancer.
4. Create and apply the **Ingress**.
5. Verify Ingress with: `kubectl get ingress`.

Routing Types in Ingress

- **Port-based routing**
- **Host-based routing**
- **Path-based routing**

Deleting Ingress and Namespaces

- To delete a namespace and its objects: `kubectl delete ns <namespace-name>`.
- Alternatively, use a manifest file: `kubectl delete -f <manifest-link>.yaml`.

Documentation Reference

- Always refer to the **Ingress documentation** for more details on routing and deleting objects.

Kubernetes Kubectl CLI & Cheatsheet

Dry Run Command

- The **dry run** is used to generate a template for an object (like a pod) without actually creating it.
- Command example:

```
kubectl run <pod-name> --image=<image-name> --dry-run=client -o yaml > ngpod.yaml
```

  - This generates a YAML file for a pod with the specified name and image.

Workload Balancing Commands

- **kubectl cordon** : Marks a node as unschedulable, preventing new pods from being scheduled on it.
- **kubectl drain** : Removes all pods from the node and transfers them to healthy nodes.
- **kubectl uncordon** : Makes the node schedulable again after maintenance.

Taints and Tolerations

- **Taints**: Applied to nodes to prevent pods from being scheduled unless they have a matching **toleration**.
- **Toleration**: Declared for pods, allowing them to run on nodes with matching taints.

Resource Requests and Limits

- **resources.requests**: Minimum amount of resources (CPU/memory) reserved for a container. The pod will only be created if this minimum can be allocated.
- **resources.limits**: Maximum resources a container can consume. The container cannot use more than the defined limits.

Jobs and CronJobs

Jobs

- A **Job** creates one or more pods and retries their execution until a specified number of successful completions is reached.
- Deleting a job cleans up the pods it created.
- Jobs can run multiple pods in parallel or a single pod to completion.

CronJobs

- A **CronJob** runs jobs at scheduled times based on a cron pattern (minute, hour, day, month, day of the week).
- Difference between Job and CronJob:
  - **Job**: Runs immediately and completes.
  - **CronJob**: Runs at specific times according to the schedule.

DaemonSets

Definition and Use Cases

- **DaemonSets** ensure that all nodes in a cluster run a copy of a specific pod.
- Use cases include:
  - Running cluster-wide storage, log collection, or monitoring daemons.

DaemonSet Behavior

- As nodes are added to or removed from the cluster, pods are automatically created or garbage collected.
- Deleting a DaemonSet removes the pods it created.

Tolerations in DaemonSets

- Example of a **DaemonSet** that runs on master nodes using tolerations:
  - Tolerations allow pods to run on nodes that are otherwise unschedulable, such as master nodes.

Commands Related to DaemonSets

- **kubectl get ds -A**: Lists all DaemonSets across namespaces.
- **kubectl get pod -n** : Lists pods created by the DaemonSet in the specified namespace (e.g., `kube-system`).

DaemonSets and ReplicaSets

- DaemonSets create pods across nodes, ensuring one pod per node.

- Both DaemonSets and ReplicaSets ensure new pods are created if existing pods are deleted. However, DaemonSets are designed to maintain one pod per node while ReplicaSets maintain a specific number of replicas.

Kubernetes with Lens

Introduction to Lens

- **Lens** is an Integrated Development Environment (IDE) designed for managing Kubernetes clusters across platforms like Mac, Windows, and Linux.
- It offers a graphical interface for deploying and managing clusters directly from the console.

Kubernetes Configuration in Lens

- To transfer a **Docker image** between servers without using a repository, **Docker config files** are used.
- For Kubernetes clusters, **kubeconfig files** can be utilized to transfer clusters between servers without a repository.
- Lens allows users to add the kubeconfig and visualize the cluster's usage. It also offers metrics management to monitor cluster performance.

Networking Essentials for DevOps

Core Responsibilities of DevOps in Networking

1. Automating the management of cloud computing environments.
2. Establishing connections between multiple systems.

Networking Components Overview

- **Key Topics**:
  1. Components responsible for networking.
  2. OSI Model and its layers.
  3. Classification of networks based on geography.
  4. Networking devices (e.g., routers, switches).
  5. Home networks.
  6. IP addresses and their ranges.
  7. Protocols, DNS, and DHCP services.
  8. Key networking commands.

What is a Computer Network?

- A **computer network** is a communication system between two or more devices.
  - Every device on the network has an **IP address**.
  - Network-enabled devices (like IoT devices) communicate through these interfaces.

  - Examples include laptops with ethernet/wireless adapters, smartphones, and IoT devices.

Components of a Computer Network

1. **Devices**: Computers, smartphones, IoT devices.
2. **Connection Medium**: Cables or wireless networks linking the devices to network interfaces.
3. **Switches**: Connect multiple network interfaces.
4. **Routers**: Connect multiple networks.
5. **Operating System**: Software that analyzes data received on the network.

OSI Model

Importance of OSI Model

- The **Open Systems Interconnection (OSI)** model provides a standard for communication between all network devices.
- It ensures worldwide compatibility, enabling seamless communication across devices, apps, and operating systems.

OSI Model Overview

- Developed by the **International Organization for Standardization (ISO)** in 1984, it is a 7-layer architecture designed for data communication.
- Devices and networks across the world follow this model to ensure standardized communication.

OSI Model's Layered Architecture

- **Services**: Actions a layer provides to higher layers.
- **Protocols**: Set of rules for exchanging information between layers.
- **Interfaces**: Communication pathways between layers.

Example of OSI Model in Action

- When posting a picture on **Instagram**, the user acts as the sender, and the Instagram server acts as the receiver. The OSI model governs the communication between the two.

Summary

- Lens provides a convenient way to manage Kubernetes clusters with visual tools.

- Understanding the components of a network and OSI model is essential for a DevOps engineer to manage and automate cloud and network tasks effectively.

OSI Model Overview

The **OSI Model** (Open Systems Interconnection) defines a framework for standardizing communication between computers over a network. It consists of seven layers, each responsible for specific functions. Below is a detailed breakdown of each layer.

---

1. Physical Layer

- **Responsibilities**: Establishes the physical connection between two devices (e.g., computer A and computer B).
- **Data Format**: Data is in bits (1s and 0s).
- **Transmission**: Manages media, signals, and binary transmission.
- **Devices**: Uses cables, hubs, and physical media like Ethernet cables, fiber optics, or wireless networks.

---

2. Data Link Layer

- **Function**: Ensures error-free transfer from one node's physical layer to another's.
- **Data Format**: Data is framed for transmission.
- **Addressing**: Uses **MAC addresses** to identify devices.
- **Key Concepts**:
  - Every device has a **Network Interface Card (NIC)**.
  - The MAC address is assigned by NIC for physical addressing.
  - Data is transferred to the **Network Layer** after framing.
- **Devices**: Switches and bridges operate at this layer.

---

3. Network Layer

- **Function**: Manages the transmission of data between nodes in different networks using **IP addresses**.
- **Data Format**: Data is structured into **packets**.
- **Key Concepts**:
  - Assigns IP addresses to packets and determines the best route for data.
  - Involves protocols like **IP**, **ICMP**, and **IGMP**.
  - Routers and Layer 3 switches manage routing between networks.

- **Devices**: Routers, firewalls.

---

4. Transport Layer

- **Function**: Responsible for end-to-end data delivery, ensuring that data is reliably sent between hosts.
- **Key Concepts**:
  - Protocols: Uses **TCP** and **UDP** for data transfer.
  - Manages port numbers for both the sender and receiver.
  - Ensures data is retransmitted in case of failure.
- **Devices**: Gateways handle transport layer operations.

---

5. Session Layer

- **Function**: Manages the establishment, maintenance, and termination of communication sessions.
- **Key Concepts**: Involved in security, encryption, and maintaining active connections.
- **Devices/Applications**: Authentication and login management for applications.

---

6. Presentation Layer

- **Function**: Ensures that data is in a usable format and may handle encryption/decryption.
- **Key Concepts**: Translates data for the application layer from network formatting.
- **Applications**: Responsible for encoding, compressing, and encrypting data for the application layer.

---

7. Application Layer

- **Function**: Provides network services directly to the end-user applications.
- **Key Concepts**:
  - Protocols: **DNS**, **DHCP**, **HTTP**, **HTTPS**, **FTP**, **SSH**, etc.
  - Manages interaction between user applications and lower layers.
- **Devices/Applications**: Web servers, browsers, email clients, etc.

OSI Model Data Flow

- **Sender Side**: Data originates from the **Application Layer**, passes through the lower layers (Transport, Network, Data Link, and Physical) before being sent over the network.
- **Network Layer Routing**:
  - **Routing Protocol**: Finds the shortest path (e.g., **RIP**, **OSPF**).
  - **Routed Protocol**: Uses IP addresses to send data, guided by routers.
- **Receiver Side**: The process is reversed as the data moves back up the layers, from the Physical to the Application layer.

Key Devices and Protocols by Layer

1. **Physical Layer**: Uses Ethernet cables, hubs, and wireless media.
2. **Data Link Layer**: Uses MAC (ARP, RARP) addresses; devices include switches and bridges.
3. **Network Layer**: Uses IP-based protocols; devices include routers and firewalls.
4. **Transport Layer**: Uses TCP/UDP protocols; gateways manage communication.
5. **Session, Presentation, Application Layers**: Manage user-level communication through protocols like DNS, DHCP, HTTP, FTP, etc.

Error Management in OSI Layers

- **Data Link Layer**: Ensures error-free transmission through **Logical Link Control** and **Media Access Control**.
- **Physical Layer**: Manages the actual transmission media (e.g., cables, wireless).

This model standardizes how different types of hardware and software communicate, making global data exchange possible across varied platforms.

OSI Model Layers and Devices

Layer 1: Physical Layer

- **Purpose:** Network access connection
- **Protocols:** Ethernet, Token Ring
- **Devices:** Hub, cables, fiber optics, wireless media
- **Note:** This layer involves actual hardware transmission mediums.

Layer 2: Data Link Layer

- **Purpose:** Network access, ensures error-free delivery
- **Protocols:** MAC (ARP, RARP)
- **Devices:** Bridge, Layer 2 switch
- **Sub-layers:**
  - **Logical Link Control (LLC):** Ensures error-free transmission.
  - **Media Access Control (MAC):** Manages transmission access, adds source and destination MAC addresses.

Layer 3: Network Layer

- **Purpose:** Internet access, routing, and packet delivery
- **Protocols:** IP, ICMP, IGMP
- **Devices:** Router, Firewall, Layer 3 switch
- **Function:** Determines the path for data packets using routing protocols.

Layer 4: Transport Layer

- **Purpose:** Host-to-host communication
- **Protocols:** TCP, UDP
- **Devices:** Gateway
- **Function:** Stores port numbers of sender and receiver, responsible for application-to-application delivery.

Layer 5: Session, Presentation, and Application Layers

- **Purpose:** Application-level data handling (combined in some models)
- **Protocols:** DNS, DHCP, NTP, SNMP, HTTPS, FTP, SSH, Telnet, HTTP, POP3
- **Devices:** Web server, Mail server, Browser, Mail client
- **Function:** Manages application-level protocols and authentication.

Additional Concepts

Network Layer Protocols

- **Routing Protocol:** Determines the shortest path for packet transfer (e.g., RIP, OSPF).
- **Routed Protocol:** Uses IP addresses to transfer data, assisted by routers.

Data Link Layer Responsibilities

- Adds source and destination MAC addresses for packet transmission.
- Ensures the next hop delivery is error-free.

Physical Layer Details

- Involves transmission media such as Ethernet cables, fiber optics, or wireless connections.

For further learning on the OSI model, refer to this video.

Network Classifications Based on Geography

Local Area Network (LAN)

- **Description:** Devices are close to each other (e.g., in a room or building).
- **Example:** Connecting computers and servers in a small area using switches.

Wide Area Network (WAN)

- **Description:** Devices are far apart, like across cities or countries.
- **Example:** Accessing European data centers from a smartphone over the internet.

Metropolitan Area Network (MAN)

- **Description:** Network spread across a metropolitan area, like a city's network system.
- **Example:** City-wide computer networks or metro train network systems.

Campus Area Network (CAN)

- **Description:** Network restricted to a campus (e.g., offices, universities).
- **Example:** Office or university campus network, also called an intranet.

Personal Area Network (PAN)

- **Description:** Small personal network with limited range.
- **Example:** Bluetooth connections or personal Wi-Fi hotspot.

Network Devices

Switch

- **Function:** Connects multiple computers within the same network.
- **Example:** A Wi-Fi router internally contains a switch to connect devices within a LAN.

Router

- **Function:** Connects multiple networks together.
- **Example:** Connects two buildings' networks or routes data from one subnet to another.

Modem

- **Function:** Provides internet access by connecting the router to the internet service provider.

Firewall

- **Function:** Ensures data protection and security during internet data transmission.

IP Addressing

Subnets

- **Description:** Group of devices in the same network, connected by switches.
- **Example:** Corporate datacenters have multiple subnets, each with its own IP address.

IPv4 Address

- **Structure:** Composed of four octets, each of 8 bits, totaling 32 bits.
- **Range:** 0.0.0.0 to 255.255.255.255 (binary representation).

IP Address Types

- **Public IP Address:** Identifies a device on the wider internet.
- **Private IP Address:** Used within a local network, not exposed to the wider internet.

Private IP Ranges:

- **Class A:** 10.0.0.0 - 10.255.255.255
- **Class B:** 172.16.0.0 - 172.31.255.255
- **Class C:** 192.168.0.0 - 192.168.255.255
- **Class D & E:** Reserved for research and multicasting (not commonly used).

Network Flow and Communication

- Devices under the same Wi-Fi network communicate through switches, while routers connect different networks to form the global internet.
- Each device, including switches and routers, has its own unique IP address for identification and communication.

1. Protocols and Ports in Networking

- **Protocol**: A formal specification that defines procedures for transmitting/receiving data. It includes format, timing, sequence, and error-checking mechanisms.
- **TCP (Transmission Control Protocol)**:
  - Reliable, connection-oriented protocol.
  - Ensures error-free data transmission using a three-way handshake.
  - Used for applications requiring reliability.
  - Examples: FTP, HTTP, HTTPS.
- **UDP (User Datagram Protocol)**:
  - Unreliable, connectionless protocol.
  - Faster than TCP but without error-checking or feedback mechanisms.
  - Suitable for speed-critical applications.
  - Examples: DNS, DHCP, TFTP, ARP, RARP.

2. OSI and TCP/IP Layer Mapping

- **TCP/IP Model Layers**:
  1. **Application Layer**: Telnet, FTP, DHCP, HTTP, SMTP, DNS, SNMP.
  2. **Transport Layer**: TCP, UDP.
  3. **Internet Layer**: ICMP, ARP, RARP, IP.
  4. **Network Interface Layer**.

3. Networking Concepts

- **Server Communication**: To connect applications like Tomcat and MySQL, you need to know their IP addresses and port numbers. For example, Tomcat uses port 8080, MySQL uses port 3306.
- **Firewall Permissions**: Services must have appropriate firewall permissions to bypass security and run applications.

4. Networking Commands

- **ifconfig**: Displays active network interfaces, loopback addresses (e.g., 127.0.0.1), and IP addresses.
- **ip addr show**: Similar to `ifconfig`, shows network interfaces.
- **ping [IP address]**: Sends ICMP packets to check if the target IP address is reachable and shows packet transmission statistics.

- **tracert [website/IP address]**: Traces the path to a target server, displaying the hops (routers) the data passes through.
- **netstat -antp**: Shows all open TCP ports and the associated process IDs and service names.
- **ps -ef | grep [service]**: Lists processes and allows filtering for specific services like Apache.
- **ss -tunlp**: Displays all information about TCP/UDP sockets, including processes and services.

5. Advanced Network Commands

- **nmap**:
  - Scans networks for hosts and services.
  - Provides detailed information about open ports, services, and vulnerabilities.
  - Example: `nmap localhost` or `nmap [IP address]`.
- **dig [domain]**: Fetches the IP address of a domain via DNS.
- **nslookup [domain]**: Provides DNS details of a domain, including IP addresses.
- **route**: Manages IP/kernel routing tables. Use `route -n` for numeric IPs and `sudo route add default gw [gateway]` to assign a gateway.
- **arp**: Manages ARP (Address Resolution Protocol) tables, mapping IP addresses to MAC addresses.
- **mtr**: Combines ping and traceroute, showing real-time packet exchange and network issues.

6. Telnet and Connectivity

- **telnet [IP address]**: Connects to a remote machine via IP and checks if the machine and its port are accessible.

These notes summarize key concepts, protocols, and commands used in DevOps networking, focusing on essential tools and protocols for network management and troubleshooting.

Introduction to Cloud Computing and AWS

What is Cloud Computing?

- **Definition**: Cloud computing is the on-demand delivery of IT resources over the Internet with pay-as-you-go pricing.
- **Traditional vs. Cloud**:
  - **Traditional Setup**: Organizations maintain their own data centers with physical servers and virtualization teams. Employees request resources from the virtualization team, which uses hypervisors to create virtual machines.

- **Cloud Computing**: Users can self-provision virtual machines and other resources via web portals or command-line interfaces, eliminating the need to contact a virtualization team.

Benefits of Cloud Computing

1. **Agility**
   - Quickly procure resources without the overhead of building infrastructure.
   - Easily release resources when they're no longer needed.
   - Saves on maintenance and establishment costs.
2. **Global Access**
   - Access virtualized resources over the network from anywhere.
   - Manage resources through APIs or portals.
3. **Cost Efficiency**
   - Pay-as-you-go pricing model reduces upfront costs.
   - No need to invest in physical data centers and servers.

Types of Cloud Computing Services

1. Infrastructure as a Service (IaaS)

- **Description**: Provides virtualized computing resources over the internet.
- **User Responsibility**: Manage the operating system and applications on the virtual machines.
- **Example**: Amazon EC2 (Elastic Compute Cloud)
  - Users are given infrastructure components like memory, RAM, and processors.
  - Ideal for users who need control over their computing environment.

2. Platform as a Service (PaaS)

- **Description**: Offers hardware and software tools over the internet, allowing users to develop and manage applications without dealing with the underlying infrastructure.
- **User Responsibility**: Focus on deploying and managing applications.
- **Example**: AWS RDS (Relational Database Service) for Oracle Database
  - AWS handles the provisioning and management of the database platform.
  - Users can directly use the database without managing virtual machines.

3. Software as a Service (SaaS)

- **Description**: Delivers software applications over the internet on a subscription basis.

- **User Responsibility**: Just use the software; no need to manage infrastructure or platforms.
- **Example**: Web-based email services, CRM applications.
  - Simply subscribe and start using the software.

Cloud Deployment Models

- **Private Cloud**: Cloud infrastructure dedicated to a single organization.
  - Offers greater control and security.
  - Accessible over a private network.
- **Public Cloud**: Services offered over the public internet and available to anyone.
  - Examples include AWS, Microsoft Azure, and Google Cloud Platform.
  - Users sign up and subscribe to services as needed.

Key Takeaways

- **Shift from Virtualization to Cloud Computing**: Organizations are moving from traditional virtualization setups to cloud computing for greater flexibility and efficiency.
- **Self-Service Portals and APIs**: Users can create, manage, and maintain virtual resources without relying on a dedicated team.
- **Widespread Adoption**: Approximately 90% of organizations are utilizing cloud computing services.
- **Cost and Resource Efficiency**: Cloud computing reduces the need for physical infrastructure and allows for scalable resource management.

By understanding the fundamentals of cloud computing and the various services offered by providers like AWS, DevOps professionals can better manage and deploy applications in modern, scalable environments.

Introduction to AWS

Setting Up AWS Free Tier Account

- **Requirements**:
  - Set up an AWS Free Tier account.
  - Create billing alarms to monitor costs.
  - Configure IAM (Identity and Access Management) users for access control.

AWS Global Infrastructure

- **AWS Overview**: AWS is the largest public cloud provider, operating across many countries.
- **Regions and Availability Zones (AZs)**:
  - AWS is divided into **regions** (physical locations worldwide).
  - Each region consists of **multiple availability zones** (AZs), which are clusters of data centers.
  - A region can have a minimum of **two AZs** and a maximum of six. Each AZ contains multiple data centers.
  - **As of 2020**, AWS has 77 availability zones across 24 geographic regions.

Types of AWS Data Centers

- **Availability Zones**:
  - Each zone is a set of clustered data centers with independent power, cooling, and security systems.
  - AZs are interconnected via low-latency, high-bandwidth networks, ensuring fault tolerance and high availability.
  - Applications spread across multiple AZs are protected from local disasters like power outages or natural disasters.
- **Local Zones**:
  - These bring AWS services like compute, storage, and databases closer to large populations, reducing latency.
  - Ideal for use cases like media content creation, gaming, and machine learning that require low-latency access.

High Availability and Disaster Recovery

- **High Availability**:
  - Deploy resources in multiple AZs to ensure continuous operation, even if one AZ goes down.
- **Global Reach**:
  - Use multiple regions to reduce latency and provide global disaster recovery. If one region fails, data can be backed up or redirected to another region.

AWS Services

- **Core Services**:
  - Compute: **Amazon EC2**, **Lambda**, **Elastic Load Balancing**
  - Storage: **Amazon S3**, **Elastic Block Store (EBS)**
  - Databases: **RDS**, **DynamoDB**
  - Networking: **VPC**, **CloudFront**, **Route 53**
  - Developer Tools: **CodeCommit**, **CodeBuild**, **CodeDeploy**

  - Monitoring: **CloudWatch**
  - Security: **IAM**, **Key Management Service (KMS)**, **CloudTrail**
- **Services in Region Launches**:
  - AWS launches core services like **EC2**, **S3**, **RDS**, and **CloudWatch** in every new region.
  - Some services, like **Athena**, **CloudFront**, **EKS**, and **GuardDuty**, are introduced within 12 months of a new region launch.

AWS Edge Computing Services

- **AWS Wavelength**:
  - Brings AWS services to the edge of the 5G network, minimizing latency for mobile devices.
  - Useful for applications requiring low-latency performance like gaming, video streaming, AR/VR, and machine learning at the edge.
- **AWS Outposts**:
  - Extends AWS services to on-premises locations, providing consistent hybrid cloud experience for workloads requiring low latency or local data processing.

AWS Management Console

- **Console Overview**:
  - Central hub to access all AWS services.
  - Popular services for DevOps include **EC2** for virtual machines, **S3** for storage, **RDS** for databases, and **VPC** for networking.
  - Tools for monitoring and automation include **CloudWatch**, **CloudTrail**, and **CodePipeline**.
  - DevOps can leverage services such as **Elastic Beanstalk** for app deployment and **ECS** for containerized applications.

AWS Free Tier Limitations

- **Region Restrictions**:
  - The AWS Free Tier allows access to certain regions (primarily US-based).
  - To use services in other regions or access additional availability zones, payment is required.

By understanding the global infrastructure and key services offered by AWS, DevOps engineers can effectively design and manage cloud environments for high availability, scalability, and fault tolerance.

AWS EC2 Overview

Elastic Compute Cloud (EC2):
EC2 is a widely used AWS service that provides virtual machines and related services. It enables users to manage and provision virtual machines in Amazon's cloud infrastructure. EC2 offers scalability, allowing users to adjust resources like CPU, RAM, and storage as needed.

- **Billing**: Users only pay for what they use, either by the hour or by seconds, and can reserve capacity for long-term usage (1-3 years) to get discounts.
- **Integration**: EC2 integrates seamlessly with other AWS services like S3, EFS, RDS, DynamoDB, and Lambda.

EC2 Instance Types and Pricing

1. **On-Demand Instances**: Pay on an hourly or seconds basis for virtual machines.
2. **Reserved Instances**: Discounts are available for committing to 1 to 3 years of usage.
3. **Spot Instances**: Purchase unused instances at a lower price. These instances are terminated when the original owner needs them back.
4. **Dedicated Servers**: Expensive, dedicated physical servers for exclusive use.

Key EC2 Components

1. **Amazon Machine Image (AMI)**:
   - Ready-made virtual machines, similar to Docker images.
   - Acts as a base image for launching EC2 instances.
2. **Instance Types**:
   - Determines CPU, RAM, network, and storage specifications for the instance.
3. **Elastic Block Store (EBS)**:
   - A virtual hard disk used for storing operating systems and data.
   - Default storage is 8GB for Linux and 30GB for Windows, but this can be scaled as needed.
4. **Tags**:
   - Key-value pairs that help manage, search, and filter resources.
   - Useful for tracking EC2 instances based on projects, environments, or ownership.
5. **Security Groups**:
   - Virtual firewalls that control inbound and outbound traffic to the instances.

Accessing EC2 Instances

- **Key Pairs**:

- To log in, SSH keys (public and private) are used.
- It's essential to create and download the key pair when creating an EC2 instance.
- The same key can be used to access Windows instances, though the key is used to generate a password.

Steps to Create an EC2 Instance

1. **Choose AMI**: Select the base Amazon Machine Image.
2. **Choose Instance Type**: Define the size and computing resources.
3. **Configure Instance**: Set up network configurations, permissions, and scripts.
4. **Add Storage**: Assign storage volumes using EBS.
5. **Tag Instance**: Apply key-value pairs for easy identification and billing.
6. **Configure Security Group**: Create a security group for managing traffic.
7. **Review & Create Instance**: Finalize settings and launch the instance.

By following these steps, users can easily create and manage EC2 instances with the required configurations and security.

Bash Scripting Overview

Bash scripting is a powerful way to automate tasks on Unix-based systems. However, errors in scripts are often due to typographical mistakes, such as missing or extra spaces. When errors occur, it is essential to carefully review the script.

Basic Shell Commands in Bash

- **System Uptime**: `uptime`
- **Memory Utilization**: `free -m`
- **Disk Utilization**: `df -h`

Redirecting Output

To suppress the output from a command, redirect it to `/dev/null`:

```
command > /dev/null
```

Variable Declaration in Bash

- Declare a variable using `=` without spaces:

```
SKILL="DevOps"
echo $SKILL
```

- Example for package installation:

```
PACKAGE="httpd wget unzip"
yum install $PACKAGE -y
```

File Operations

- **Rename a script**:

  ```
  mv old_script.sh new_script.sh
  ```

- **Copy a script**:

  ```
  cp script.sh copy_script.sh
  ```

Command Line Arguments

In Bash scripts, you can access command-line arguments as follows:

```
./script.sh arg1 arg2 arg3
```

- `$0`: The script name.
- `$1`, `$2`, etc.: Arguments passed to the script.
- `$#`: The number of arguments.
- `$@`: All the arguments.
- `$?`: The exit status of the last process.
- `$$`: The process ID of the script.

System Variables

Some useful system variables include:

- `$USER`: Current username.
- `$HOSTNAME`: Hostname of the system.
- `$SECONDS`: Time since the script started.
- `$RANDOM`: Generates a random number.
- `$LINENO`: Current line number in the script.

Quotes in Bash

- **Single Quotes**: Treat everything literally. Variables inside single quotes are not expanded.

  ```
  SKILL='DevOps'
  echo 'I have $SKILL skill'  # Output: I have $SKILL skill
  ```

- **Double Quotes**: Variables and special characters inside double quotes are expanded.

  ```
  SKILL="DevOps"
  echo "I have $SKILL skill"  # Output: I have DevOps skill
  ```

- To print special characters like `$`, use a backslash `\` to escape them:

```
echo "Cost is \$9 million"  # Output: Cost is $9 million
```

By understanding these basic concepts, you can effectively write and debug Bash scripts for various DevOps tasks.

Bash Scripting: Key Concepts

1. Command Substitution Command substitution allows you to store the output of a command in a variable. This is done using backticks (`) or the more modern `$()`.

- **Example**:

```
UP=$(uptime)
echo $UP
```

- **Alternative**:

```
CURRENT_USER=$(who)
echo $CURRENT_USER
```

2. Example with Command Chaining Using `free`, `grep`, and `awk` to capture specific data from memory usage:

- **Command**:

```
free -m | grep Mem | awk '{print $4}'
```

- **Storing Output in a Variable**:

```
FREE_RAM=$(free -m | grep Mem | awk '{print $4}')
echo $FREE_RAM
```

3. Exporting Variables To make variables available to child processes (shells, scripts), use `export`.

- **Export Example**:

```
export VARNAME="VARVALUE"
```

Variables can also be made globally persistent by adding them to `.bashrc` or `/etc/profile`.

- **Edit `.bashrc` for user-specific variables**: Variables declared here will persist across user sessions.
- **Edit `/etc/profile` for system-wide variables**: This makes variables available to all users.

4. User Input in Scripts In bash scripting, user inputs can be taken using the `read` command.

- **Basic User Input**:

```
read SKILL
echo "Your $SKILL skill is in high demand."
```

- **Prompting for Input with `-p`**:

```
read -p 'Username: ' USR
```

- **Password Input (Hidden Characters) with `-sp`**:

```
read -sp 'Password: ' pass
echo "Login Successful"
```

5. Avoiding User Interaction in DevOps In DevOps, avoid manual user interaction in scripts to reduce errors. Automate processes to ensure that applications run seamlessly in the background without needing input during execution.

Conclusion

These key concepts in bash scripting, from command substitution to exporting variables and handling user input, are essential for automating and managing processes efficiently, especially in DevOps environments.

Bash Scripting: Decision Making and Loops

1. Decision Making in Bash

Conditional Statements (If-Else) Conditional statements allow decision making in scripts based on user input or system states. Here's a basic example of an `if-else` condition:

- **Example**:

```
#!/bin/bash
read -p "Enter a number: " NUM
if [ $NUM -gt 100 ]
then
  echo "We have entered in IF block."
  sleep 3
  echo "Your number is greater than 100."
  date
else
  echo "You have entered a number less than 100."
fi
echo "Script execution completed successfully."
```

If-Else-If Statements The `if-else-if` construct checks multiple conditions sequentially.

- **Example**:

```
#!/bin/bash
value=$(ip addr show | grep -v LOOPBACK | grep -ic mtu)
if [ $value -eq 1 ]
then
  echo "1 Active Network Interface found."
elif [ $value -gt 1 ]
then
  echo "Found Multiple active Interfaces."
else
  echo "No Active interface found."
fi
```

Grep Commands Used in Scripts

- **Command Examples**:
  - `grep -v <Text>`: Excludes lines containing `<Text>`.
  - `grep -ic <Word>`: Counts occurrences of `<Word>`, case-insensitive.

2. Loops in Bash

For Loop The `for` loop iterates over a set of values and performs actions in each iteration.

- **Simple For Loop**:

```
#!/bin/bash
for VARI in java .net python ruby php
do
  echo "Looping..."
  sleep 1
  echo "########################################"
  echo "Value of VARI is $VARI."
  echo "########################################"
  date
done
```

- **For Loop to Add Users**:

```
#!/bin/bash
MYUSERS="alpha beta gamma"
for usr in $MYUSERS
do
  echo "Adding user $usr."
  useradd $usr
```

```
  id $usr
  echo "####################################"
done
```

- **For Loop with C-Style Syntax**:

```
#!/bin/bash
for (( c=1; c<=5; c++ ))
do
  echo "Welcome $c times"
done
```

- **Infinite For Loop**:

```
#!/bin/bash
for (( ;; ))
do
  echo "Infinite Loop"
done
```

While Loop The `while` loop executes as long as the condition remains true.

- **Basic While Loop**:

```
#!/bin/bash
counter=0
while [ $counter -lt 5 ]
do
  echo "Looping..."
  echo "Value of counter is $counter."
  counter=$(( counter + 1 ))
  sleep 1
done
echo "Out of the loop"
```

- **Infinite While Loop**:

```
#!/bin/bash
counter=2
while true
do
  echo "Looping..."
  echo "Value of counter is $counter."
  counter=$(( counter * 2 ))
  sleep 1
done
```

Conclusion

This guide covers conditional statements (`if-else`, `if-else-if`) and loop structures (`for` and `while`) in Bash scripting. These concepts are essential for automating tasks, making decisions, and performing repeated actions in scripts.

Bash Scripting: SSH Key Exchange and Redirection

1. SSH Key Exchange SSH key exchange is used to log in to a remote server without needing to enter a password every time. This process allows for more secure and efficient server management.

Steps for SSH Key Exchange:

1. **Generate SSH Keys**:
   - Run `ssh-keygen` to create two files:
     - `id_rsa` (private key)
     - `id_rsa.pub` (public key)
   - Leave the directory and passphrase unchanged (default settings).
2. **Copy Public Key to Remote Server**:
   - Copy `id_rsa.pub` to the remote server using the following command:
     `ssh-copy-id remote-username@remote-servername`
3. **Passwordless Login**:
   - After copying the public key, you can log in to the remote server without a password using:
     `ssh remote-username@remote-servername`
4. **Using SSH Key for Remote Access**:
   - For more specific cases, you may need to specify the private key file:
     `ssh -i ~/.ssh/id_rsa remote-username@remote-servername`

Key Mechanism:

- **id_rsa**: The private key used on the local machine.
- **id_rsa.pub**: The public key stored on the remote machine.
- SSH login happens when the private key matches the public key, allowing passwordless access.

2. Redirecting Output to `/dev/null` The `/dev/null` device discards any data sent to it, which is useful for suppressing command outputs.

Redirect Standard Output:

```
command > /dev/null
```

This redirects the command's standard output to the null device, effectively discarding it.

Redirect Standard Output and Error:

```
command &> /dev/null
```

This redirects both the standard output and the standard error to the null device.

- In shells that don't support `&>`, use the following:

  ```
  command > /dev/null 2>&1
  ```

  Here, `2>&1` redirects standard error (2) to standard output (1), which is then sent to `/dev/null`.

3. Using SCP for File Transfer The `scp` command is used to securely copy files between local and remote systems over SSH.

SCP Syntax:

- Basic usage:

  ```
  scp <filename> remote-username@remote-server:/path-to-directory
  ```

- With SSH key authentication:

  ```
  scp -i ~/.ssh/id_rsa <filename> remote-username@remote-server:/path-to-directory
  ```

When SSH key exchange is in place, no password is required during the file transfer.

4. Automating Script Execution on Multiple Servers A simple script can automate copying and executing scripts on multiple remote servers using `scp` and `ssh`.

Example Script:

```
#!/bin/bash
USR='devops'
for host in $(cat remhosts)
do
  echo "#############################################################"
  echo "Connecting to $host"
  echo "Pushing Script to $host"
  scp multios_websetup.sh $USR@$host:/tmp/
  echo "Executing Script on $host"
  ssh $USR@$host sudo /tmp/multios_websetup.sh
  ssh $USR@$host sudo rm -rf /tmp/multios_websetup.sh
  echo "#############################################################"
done
```

- `scp` copies the script to the remote server.

- `ssh` logs in and executes the script.
- After execution, the script is removed from the remote server.

Conclusion

This section of the notes covers essential techniques in Bash scripting, including SSH key exchange for passwordless server access, redirecting command output to `/dev/null` to suppress output, securely copying files with `scp`, and automating tasks across multiple servers.

GIT: Introduction to Version Control

1. What is Git?

- Git is a version control system (VCS) and a crucial tool for developers.
- As a DevOps engineer, understanding Git is essential for collaborating with developers, managing code changes, and deploying applications.
- Git helps in writing scripts, configuration files, and automation code.

2. Importance of Version Control

- DevOps engineers often modify multiple files simultaneously, making it easy to lose track of changes.
- Version control solutions provide:
  - **Local backups** of files.
  - An updated version in a Git repository for easy rollback if necessary.
- The primary purpose of Git is to manage and maintain code efficiently.

3. Features of Version Control Systems (VCS)

- VCS tracks the history of files, allowing for:
  - Easy access to previous versions.
  - Simplified collaboration among team members.
  - Efficient merging of changes.
- It enables users to keep track of modifications and collaborate effectively.

4. Types of Version Control Systems

- **Localized Version Control Systems**:
  - Keep local copies of files.
  - Useful for rollback and local backups.
- **Centralized Version Control Systems**:
  - Utilize a central server that holds the master repository.
  - Developers have local copies and can make changes before pushing to the central repository.
  - Always recommended to pull changes before pushing to ensure the repository is up-to-date.

5. Centralized vs. Localized VCS

- **Centralized VCS**:
  - A single central repository serves as the official code source.
  - Developers check out files, make changes, and check them back in.
  - Changes are synchronized with the central repository, maintaining a single source of truth.
- **Localized VCS**:
  - Developers work with their own copies of the repository.
  - Changes are made and committed locally before being shared with others through push/pull operations.
  - This model supports parallel development on different features, enhancing collaboration.

6. Limitations of VCS

- Both systems have drawbacks:
  - In centralized systems, issues with the server can halt development.
  - In localized systems, problems with individual machines can disrupt work.
- It's important to manage these risks to maintain workflow efficiency.

Conclusion Git is an essential tool for version control in the DevOps landscape, allowing for effective collaboration, code management, and deployment. Understanding both centralized and localized systems, along with their advantages and limitations, is crucial for any DevOps professional.

GIT: Understanding Version Control Systems

1. Centralized vs. Localized Version Control

- **Conflict Resolution**:
  - In centralized VCS, conflicts occur when multiple users attempt to check in changes to the same file simultaneously. Developers must resolve these conflicts before proceeding.
  - In localized VCS, conflicts can happen anytime two developers modify the same file. Developers can choose to resolve conflicts immediately or defer resolution.
- **Structure**:
  - **Centralized VCS**: A single central repository exists where all users edit and check in changes. Users maintain copies of files they wish to edit.
  - **Localized VCS**: Users have a local copy of the repository, allowing them to make and test changes before pushing to the central repository. Users must pull changes from the central repo to stay updated.

2. Distributed Version Control Systems (DVCS)

- **Multiple Repositories**:
  - DVCS includes a master repository and multiple child or feature branches. This structure allows for flexibility in developing and integrating code.
  - Developers can work on multiple features simultaneously without affecting the main repository.
- **Resilience to Server Issues**:
  - In DVCS, each developer has a complete copy of the repository. Server issues do not disrupt development; changes can be made locally and synced with the central repository when available.

3. Git Overview

- Git is a distributed version control system enabling developers to track code changes and collaborate on projects.
- Each developer maintains a local copy of the entire repository, allowing for local commits before sharing changes.

4. Git vs. GitHub

- **Git**: A version control system for tracking changes in code.
- **GitHub**: A web-based platform that hosts Git repositories, providing tools for collaboration like issue tracking, pull requests, and project management.

Summary

Understanding the differences between centralized, localized, and distributed version control systems is crucial for effective code management in a collaborative environment. Git serves as a powerful tool for tracking changes, while GitHub enhances collaboration and project management capabilities.

Git Versioning Notes

Introduction

Git is a version control system that tracks changes in files. It does not track empty directories. The key purpose of Git is to manage versions of files, particularly in collaborative environments. Below are important Git commands and concepts.

Basic Git Commands

Check Git Installation

- **Command:** `git`

- Checks if Git is installed on the system.

Check Git Version

- **Command:** `git --version`
- Displays the installed version of Git.

Initialize a Git Repository

- **Command:** `git init`
- Initializes the current directory as a Git repository by adding a `.git` folder. This folder tracks changes and stores the history of the project.

Git Status & File Tracking

Git Status

- **Command:** `git status`
- Shows the current state of the working directory and staging area. It lists untracked files, staged files, and informs whether the branch is behind in versions.

Staging Files

- To stage files before committing, use:
  - **Command:** `git add directory_name/` – stages all files in a directory.
  - **Command:** `git add <file_name>` – stages specific files.

Committing Files

- **Command:** `git commit -m "commit message"`
- Commits the staged files to the local repository with a descriptive message.

Global Configuration

Configure Global Username and Email

- Set these details once to help identify the user who makes changes:
  - **Command:** `git config --global user.email "email@example.com"`
  - **Command:** `git config --global user.name "username"`

Working with Remote Repositories

Cloning a Remote Repository

- **Command:** `git clone <URL>`
- Clones a remote repository (from GitHub, Bitbucket, etc.) to the local machine.

Adding Remote Repository to Local

- **Commands:**
  1. `git remote add origin <https-link-of-git-repository>`
  2. `git push` – Pushes local changes to the remote repository.
  3. `git pull` – Pulls the latest changes from the remote repository.

Pushing to a Remote Repository

- **Command:** `git push origin <branch-name>`
- Pushes committed changes from the local branch to the specified branch in the remote repository.

Pulling from a Remote Repository

- **Command:** `git pull`
- Fetches and integrates changes from the remote repository to the local repository.

Viewing Commit History

Viewing Git Logs

- **Command:** `git log`
- Displays the commit history with details of each commit.

Shortened Commit Logs

- **Command:** `git log --oneline`
- Shows a more concise view of the commit history with shortened commit IDs.

Viewing Changes in Specific Commits

- **Command:** `git show <commit-id>`
- Displays changes made in a specific commit using its ID.

Summary

Git is essential for managing file versions in both local and remote repositories. Understanding how to stage, commit, push, and pull changes ensures smooth collaboration in a team environment. Git's flexibility in tracking changes makes it an invaluable tool for developers.

Git Branches, Rollback, and SSH Login Notes

Branching in Git

Sub-repositories (Branches)

In Git, branches allow you to create sub-repositories that facilitate parallel development. For instance, feature branches can be created from the main branch, and after changes, they can be merged back into the main branch. This method is useful for team collaboration, especially in Agile development, where work is done in small increments.

Creating a Branch

1. **Switch to the main branch:**
   - **Command:** `git checkout main`
2. **Create a new branch:**
   - **Command:** `git branch <branch-name>`
3. **List all branches:**
   - **Command:** `git branch -a` – Displays all local and remote branches.
4. **Switch to a branch:**
   - **Command:** `git checkout <branch-name>`

Moving and Removing Files

- **Remove a file:**
  - **Command:** `git rm <file-names>` – Removes a file from the repository and stages the change.
- **Move/rename a file:**
  - **Command:** `git mv <src-filename> <dst-filename>` – Moves or renames a file and stages the change.

Pushing Changes to a Branch

To push changes to a specific branch: - **Command:** `git push origin <branch-name>`

Switching Between Branches

- **Command:** `git checkout <branch-name>` or
- **Command:** `git switch <branch-name>` – Switches between branches.

Merging Branches

To merge one branch into another: 1. Switch to the target branch:
Command: `git checkout <target-branch>` 2. Merge the source branch:
Command: `git merge <source-branch>`

Rollback and File Management

Rolling Back Changes

Git allows you to rollback changes in two scenarios: 1. **Before staging:**
- To rollback a modified file to the previous committed state:
Command: `git checkout <filename>`

2. **After staging:**
   - To un-stage a file that has been staged:
     Command: `git restore --staged <filename>`

Checking Differences

- **Show unstaged differences:**
  Command: `git diff` – Displays file changes that haven't been committed.
- **Show staged differences:**
  Command: `git diff --cached` – Shows the difference between the staged files and the last commit.

Comparing Commits

- **Compare two commits:**
  Command: `git diff <prev-commit-id> <curr-commit-id>`

Reverting Commits

- **Revert the most recent commit:**
  Command: `git revert HEAD`
- **Revert a specific commit:**
  Command: `git revert <commit-id>` – Creates a new commit that undoes the changes.

Hard Reset

To permanently remove a commit from history (caution: cannot be undone): - **Command:** `git reset --hard <commit-id>`

SSH Login for Git

Why Use SSH?

SSH keys provide a secure way to interact with Git repositories without exposing your username or password, especially useful when working with private repositories.

Setting Up SSH Keys

1. **Generate SSH keys:**
   Command: `ssh-keygen.exe`
   This generates a pair of keys: `id_rsa` (private) and `id_rsa.pub` (public).
2. **Add the public key to GitHub:**
   - Go to: `Settings > SSH and GPG keys > New SSH key`
   - Paste the contents of `id_rsa.pub` and give it a title.

Using SSH for Git

1. After adding your SSH key to GitHub, you can clone repositories without a password:
   - **Command:** `git clone <ssh-link>`
2. **Fingerprint verification:**
   When cloning for the first time, Git may ask for fingerprint verification. Confirm by entering "yes."

Configuring SSH Keys

- View Git SSH config:
  Command: `cat .git/config`
- Remove SSH keys (if needed):
  Command: `rm -rf .ssh/*`

Summary

Branches in Git enable collaborative development, and rollback options help manage mistakes efficiently. Using SSH keys offers secure, password-free access to repositories, making it a preferred method for managing Git in a team setting.

Continuous Integration with Jenkins

Introduction to Continuous Integration (CI)

What is Continuous Integration?

- Continuous Integration (CI) refers to the practice where developers frequently push code to a centralized version control system after local testing.
- Multiple developers work in parallel, pushing code multiple times a day. Despite testing locally, issues and conflicts can arise when the code is merged.

Why Continuous Integration?

- **Code Conflicts:** When code is pushed but not properly integrated, it can lead to bugs and issues, even after local testing.
- CI integrates multiple commits from different developers and ensures the application works correctly at both the module and application levels.
- CI helps identify compatibility issues between new code and the existing application, allowing developers to focus on fixing integration problems early.

Automation of Continuous Integration

- Since CI checks code workability for every commit, the process should be automated. This automation is where tools like Jenkins come into play.
- Jenkins automates the process by fetching the code, building, evaluating, and notifying developers of any issues.

Jenkins as a Continuous Integration Tool

- **Open-source CI Tool:** Jenkins started as a CI tool but now offers much more functionality.
- It supports plugins for version control systems (e.g., Git), build tools (Java, Node.js, etc.), cloud integration, testing, and other DevOps tools.
- Jenkins can be used as a Continuous Delivery tool, for running scripts, cloud automation, or integrating with other development and testing tools.

Installing Jenkins

Pre-requisites for Jenkins Installation

- **Java:** Jenkins requires Java (JDK or JRE). The recommended version is JDK 11.
- **Additional Tools:** Tools like Maven are often required to run builds.

Installing Jenkins on Linux

To install Jenkins on a Linux system, follow these steps:

```
#!/bin/bash
sudo apt update
sudo apt install openjdk-11-jdk -y
curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo tee \
/usr/share/keyrings/jenkins-keyring.asc > /dev/null
echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \
https://pkg.jenkins.io/debian-stable binary/ | sudo tee \
/etc/apt/sources.list.d/jenkins.list > /dev/null
sudo apt-get update
sudo apt-get install jenkins -y
```

This script updates the system, installs JDK 11, adds the Jenkins repository, and installs Jenkins.

Jenkins Installation on Other Platforms

- Jenkins can also be set up on **Docker**, **Kubernetes**, and other platforms.
- For Fedora OS, Jenkins requires minimal resources (1 GB RAM and disk space), making it lightweight compared to other systems.

Jenkins Installation on Cloud (AWS Example)

Setting Up Jenkins on AWS

1. **Launch an EC2 Instance:**
   - Choose **Ubuntu Server 20.04 LTS** as the AMI (Amazon Machine Image).
   - Instance type: **t2.micro** (sufficient for basic use but may be slow), or **t2.small** for better performance.
2. **Create a Key Pair:**
   - Name the key pair and choose **RSA** as the key pair type for passwordless login.
   - The private key file should be saved in **.pem** format for OpenSSH or **.ppk** format for PuTTY.
3. **Connect to the Cloud Server:**
   - Use SSH to connect to the instance using the private key file.

By following these steps, you can easily set up Jenkins on a cloud server and begin automating CI tasks.

Summary

Jenkins is a powerful tool for automating Continuous Integration. It integrates multiple developers' code, ensuring that all commits are continuously checked for compatibility. With its vast plugin ecosystem and ease of installation on various platforms (Linux, Docker, Cloud), Jenkins is a popular choice for DevOps automation.

Jenkins Installation for Continuous Integration

Setting Up Security Group for the Jenkins Instance

To configure the security settings of your AWS instance, you need to create a **Security Group** to control inbound and outbound traffic. Here's how you do it:

Inbound Security Group Rules:

1. **SSH Rule:**
   - **Type:** SSH
   - **Protocol:** TCP
   - **Port Range:** 22
   - **Source Type:** My IP
   - This allows passwordless SSH login to the instance from your specific IP address.
2. **Custom TCP Rule:**
   - **Type:** Custom TCP
   - **Protocol:** TCP
   - **Port Range:** 8080
   - **Source Type:** My IP
   - This allows you to connect to Jenkins through a web browser using port 8080.

Storage Configuration:

You can configure the storage of your instance in the **Configure Storage** section during instance setup.

Advanced Details:

In the **User Data** section, you can include initialization scripts to install necessary software upon instance creation, using tools like `yum` or `apt-get`.

Launching the Jenkins Instance

Steps After Launching:

1. After launching the instance, you will get details like security, storage, networking, and the public IPv4 address.
2. **Accessing Jenkins GUI:** Use the public IPv4 address of the instance followed by port 8080 to access Jenkins through the browser.

Unlocking Jenkins

1. After accessing Jenkins through the browser, you will be prompted to unlock Jenkins.
   - The initial admin password is located at:

     `/var/lib/jenkins/secrets/initialAdminPassword`
2. **Login via SSH:**
   - SSH into the instance to retrieve the password using the command:

     `ssh -i Downloads/jenkins-key.pem ubuntu@<instance-IP>`
3. **View User Data Logs:**
   If the instance has any missing software (not installed via the shell script), you can check the script by running:

   `curl http://169.254.169.254/latest/user-data`

Jenkins Service Status:

- You can check the status of Jenkins using:

  `systemctl status jenkins`
- If Jenkins is not running, there may be an issue with the user-data or bash script.

Jenkins Configuration and Plugins

Jenkins Files:

- All Jenkins files are stored under `/var/lib/jenkins/`. The initial admin password is located at:

  `/var/lib/jenkins/secrets/initialAdminPassword`

Plugins:

Jenkins' power comes from its extensive collection of plugins. Some key categories include: 1. **Organization and Administration** 2. **Build Features** 3. **Build Tools** 4. **Build Analysis and Reporting** 5. **Pipelines and Continuous Delivery** 6. **Source Code Management** 7. **Distributed Builds** 8. **User Management and Security** 9. **Notifications and Publishing** 10. **Languages**

Accessing Jenkins After Reboot

- The public IP address of the instance is dynamic and will change after every reboot. However, the Jenkins URL remains tied to the initial IP address when the Jenkins setup was first created.
- Save the Jenkins URL for continued access as the admin, even if the instance IP changes.

Conclusion

After completing these steps and configurations, you'll be greeted with the **"Welcome to Jenkins"** screen, and Jenkins will be ready for use in automating your Continuous Integration tasks.

Continuous Integration with Jenkins: Freestyle vs Pipeline as Code

Jobs in Jenkins

In Jenkins, a **Job** refers to the workload or task that Jenkins runs. There are two primary types of jobs: 1. **Freestyle Jobs** 2. **Pipeline as a Code**

Freestyle Jobs

- Freestyle jobs are graphical and simple, created through Jenkins' **GUI**.
- You create a job by filling out a form, specifying the job's details, running it, and viewing the output.
- **Best for small projects**: Freestyle jobs are easy to set up but are mainly used for learning or simple use cases.
- **Downside**:
  - Manually linking multiple jobs to form a pipeline can be cumbersome.
  - Replicating an existing pipeline across projects requires repeating the manual process, which is inefficient.

Pipeline as a Code

- With **Pipeline as a Code**, the entire job (pipeline) is scripted using **Groovy**.
- Instead of using the GUI, the pipeline code is stored as text, which allows it to be **version-controlled**.
- **Advantages**:
  - It's scalable and easy to replicate across multiple projects.
  - The pipeline becomes part of the project's codebase, supporting **DevOps principles** like Infrastructure as Code (IaC).
- **Recommended for real-world applications**: Modern DevOps practices emphasize using code for everything, including pipelines.

Installing Tools in Jenkins

Setting Up JDK and Maven in Jenkins

For Jenkins to execute tasks, it requires **JDK** and **Maven** to be properly configured.

Steps to Configure JDK:

1. On the Jenkins Dashboard, navigate to:

   `Jenkins Dashboard > Manage Jenkins > Global Tool Configuration`

2. Add the required JDK by specifying:

   - **JDK Name**: (e.g., JDK 1.8)
   - **Path** to the JDK inside the server's `JAVA_HOME`, not your local machine.

3. To find the correct JDK path inside the Jenkins instance:

   - SSH into your Jenkins server:

     ```
     ssh -i Downloads/jenkins.pem ubuntu@<instance-IP>
     ```

   - Update the server and install JDK:

     ```
     sudo apt update
     sudo apt install openjdk-8-jdk -y
     ```

   - By default, JDK is installed at `/usr/lib/jvm/java-1.8.0-openjdk-amd64`. Copy this path and paste it in **JAVA_HOME**.

Setting Up Maven:

- Similar to JDK, Maven also needs to be configured in the **Global Tool Configuration**.
- Provide the Maven **name** and **version**.
- These tools will be triggered and installed during the Jenkins build process.

By ensuring that both JDK and Maven are set up, Jenkins can successfully compile and build projects as part of your Continuous Integration pipeline.

Continuous Integration with Jenkins: First Job

Creating a Job in Jenkins

To create a job in Jenkins: 1. Go to the Jenkins Dashboard and click on **"Create a Job"**. 2. Name your build and select the type of job, such as **Freestyle mode**. 3. You will be directed to the configuration page where the following sections are available: - **General**: General configuration for the job. - **Source Code Management**: To pull code from a repository. - **Build Triggers**: Conditions that trigger the build. - **Build Environment**: Define the environment in which the build will run. - **Build Steps**: Commands or scripts to execute during the build. - **Post-build Actions**: Actions to take after the build is completed.

Configuring Source Code Management

In the **Source Code Management** section: - Select **Git** as the source control tool. - Enter the **Repository URL** where the code is hosted. - For public repositories, no credentials are required. For private repositories, credentials need to be provided. - Specify the **branch** to be used from the repository.

Build Steps

- The **Build Step** is the entry point for a Jenkins job, and this is where you specify the tasks to be executed.
- You can write the script for the build step using the command-line options such as **Windows terminal**, **Linux terminal**, or others.
- Plugins, such as **Maven plugin**, can also be installed to run Maven commands directly, instead of writing them as general commands. This provides more customization for executing tasks.

Execution Process

- The typical process for a Jenkins job involves pulling the source code, running the build steps, and then producing an output.
- The **Jenkins Dashboard** offers options such as:
  - **Status**: View the status of the job.

  - **Changes**: Track any changes made.
  - **Workspace**: Store data and dependencies related to the job.
  - **Build Now**: Trigger a new build for the job.
  - **Configure**: Adjust job configurations.
  - **Delete Project**: Remove the project.

Workspace and Build Data

- **Workspace**: This is where Jenkins stores the directory structure and data related to the repository. Any files downloaded or created during the build process are stored in the workspace.
- **Multiple Builds**: Jenkins allows you to build a job multiple times, and each time the data and build files are stored in the workspace.

Archiving and Decluttering the Workspace

- To declutter the workspace, you can **archive** specific files after the build is complete.
- You can define patterns for the files to be archived, allowing Jenkins to store only the necessary files and clean up the workspace afterward.

Continuous Integration with Jenkins: Plugins, Versioning & More

Workspace and Versioning

- **Workspace**: Jenkins uses the workspace to store all files downloaded during the build process. It acts like a local environment where files and binaries are kept. The workspace is located under:
  - **Dashboard > Build > Workspace**
- **Binary Updates**: For each build, binaries get updated, but if you need to retrieve a previous version, it's important to maintain versioning.
- **Versioning Artifacts**: To track different versions of artifacts, you can create a versioning system using Jenkins build numbers.

Creating a Versioning Job

- **Job Creation**: You can create a new job (e.g., **Versioning-Builds**) by copying the configuration from an existing job (e.g., **Build** job). The configuration sections include:
  - **General**
  - **Source Code Management**
  - **Build Triggers**
  - **Build Environment**
  - **Build**

– **Post-build Actions**

- **Shell Commands for Versioning**: In the **Build** section, you can add shell commands using the **Add Build Step** option. An example of creating a versioned artifact:

  ```
  mkdir -p versions
  cp target/vprofile-v2.war versions/vprofile-V$BUILD_ID.war
  ```

  This creates a new versioned artifact for each build, stored in the `versions` folder with the build number.

Jenkins Variables

- You can use predefined variables in Jenkins builds. A list of such variables can be found in the Jenkins variable list.

Building with Parameters

- **Parameterized Builds**: You can add parameters under **General** by checking **"This project is parameterized"**. This allows you to pass values to the build process, such as version numbers or branch names.
- After defining parameters, use the **Build with Parameters** option on the Job Dashboard to supply values and execute the build.

  Note: Avoid using **Build with Parameters** for regular builds as it can slow down the CI/CD process due to waiting for input.

Versioning with Plugins

- **Plugins for Versioning**: A more efficient way to handle versioning is by using plugins like the **Zentimestamp** plugin, which provides variables such as **BUILD_TIMESTAMP**. For example:

  ```
  cp target/vprofile-v2.war versions/vprofile-V$BUILD_TIMESTAMP-$BUILD_ID.war
  ```

  This helps in automatically tagging builds with a timestamp for better version control.

Managing Plugins in Jenkins

- **Manage Plugins**: Plugins are a crucial aspect of Jenkins. You can install or uninstall plugins via **Dashboard > Manage Jenkins > Manage Plugins**.
- Plugins add additional functionality, such as predefined variables and tools that help enhance the CI/CD pipeline.

Storing Artifacts Outside Jenkins

- Even though Jenkins stores artifacts in its workspace, it's not intended to be a storage space. Therefore, you should shift your artifacts to external storage solutions during the build process. This will be discussed further in later steps.

Continuous Integration Pipeline with Jenkins

Overview of the CI Pipeline

In this section, we walk through the flow of a typical **Continuous Integration (CI) pipeline** using Jenkins, integrating various tools such as Git, Maven, SonarQube, and Nexus Sonatype. These tools help automate code building, testing, code analysis, and artifact storage, all while ensuring quality control.

Flow of the CI Pipeline

1. **Code Development and Local Testing**:
   - **Developer's Role**: Developers write and test code locally. Once satisfied with the changes, they push the code to a centralized repository such as GitHub.
   - **Git Integration**: Developers use Git to push their code, which then integrates with a GitHub repository.
2. **Jenkins Fetches Code**:
   - **Automated Detection**: Jenkins detects changes in the GitHub repository.
   - **Fetching the Code**: Using the Git plugin, Jenkins fetches the latest changes and stores them in the Jenkins workspace.
3. **Code Build Using Maven**:
   - **Build Process**: Jenkins uses **Maven** to build the code. The specific build tool might vary based on the codebase (e.g., Gradle for other languages), but for this example, we use Maven due to the Java code.
   - **Artifact Generation**: After the build, Maven generates artifacts (e.g., WAR or JAR files) for further steps.
4. **Unit Testing**:
   - **Maven Testing Framework**: Unit tests, included in the source code, are executed using Maven. These tests ensure that individual components of the application work as expected.
   - **Report Generation**: Test results are typically stored in XML format.
5. **Code Analysis Using SonarQube**:
   - **SonarQube Scanner**: Jenkins uses the SonarQube scanner to analyze the code for vulnerabilities, bugs, and adherence to best practices. SonarQube generates detailed reports on code quality.
   - **Setting Quality Gates**: SonarQube can be configured with quality

gates to halt the pipeline if the code fails to meet defined standards. The reports are uploaded to the SonarQube server for further analysis.

6. **Artifact Verification and Versioning**:
   - **Verified Artifacts**: Once the code passes both unit tests and code analysis, the artifacts are deemed verified.
   - **Artifact Storage in Nexus**: Before deployment, these verified artifacts are versioned and stored in a repository such as **Nexus Sonatype** for easy retrieval during deployment.

Integration of CI Tools

- **CI Tool**: Jenkins serves as the CI tool in this pipeline, but other CI tools like GitLab CI, CircleCI, or Bamboo could be used with similar steps.
- **Tool Integration**: Regardless of the CI tool used, integration with GitHub (for version control), SonarQube (for code analysis), and Nexus (for artifact storage) is essential.

Key Stages in CI Pipeline:

1. **Build the Code**: Compile the source code using Maven.
2. **Test the Code**: Run unit tests as part of the build process.
3. **Analyze the Code**: Perform static code analysis to check for vulnerabilities and code quality.
4. **Get Verified Artifacts**: If the code passes all stages, version and store artifacts in Nexus for deployment.

This process ensures continuous feedback on code quality and automation of repetitive tasks, leading to efficient and reliable software delivery.

Continuous Integration Pipeline Setup with Jenkins, Nexus, and SonarQube

This guide outlines the steps to create a **Continuous Integration (CI) pipeline** using Jenkins, Nexus, and SonarQube. We'll also cover the required plugins, EC2 instance configurations, and security group settings to ensure proper integration and communication between the tools.

Step 1: Setting Up Jenkins, Nexus, and SonarQube on EC2

Tools and Instances:

- **Jenkins**: The CI tool for managing the pipeline.
- **Nexus**: Artifact repository for storing built artifacts.
- **SonarQube**: Code analysis tool for ensuring code quality.

EC2 Instances:

- **Jenkins**:
  - **OS**: Ubuntu 18
  - **Instance Type**: t2.small
  - **Security Group**:
    - SSH (Port 22) – My IP
    - Custom TCP (Port 8080) – Anywhere
  - **User Data Script**: Jenkins setup script to automate installation.
- **Nexus**:
  - **OS**: CentOS 7
  - **Instance Type**: t2.medium
  - **Security Group**:
    - SSH (Port 22) – My IP
    - Custom TCP (Port 8081) – Anywhere
  - **User Data Script**: Nexus setup script for automated installation.
  - **Jenkins Integration**: Jenkins accesses Nexus on Port 8081.
- **SonarQube**:
  - **OS**: Ubuntu 18
  - **Instance Type**: t2.medium
  - **Security Group**:
    - SSH (Port 22) – My IP
    - Custom TCP (Ports 80 and 9000) – Anywhere
  - **Ports**: SonarQube generally runs on Port 9000, but Port 80 can be used for integration through Nginx (which forwards requests from Port 80 to 9000).

Security Group Configuration:

For all instances, create appropriate security groups with rules such as: - **SSH (Port 22)**: For secure access to the instances. - **Custom TCP (Ports 80, 8080, 8081, 9000)**: For Jenkins, Nexus, and SonarQube communication.

Step 2: Integrating Jenkins, Nexus, and SonarQube

Plugin Installation:

To enable integration between Jenkins, Nexus, and SonarQube, install the necessary plugins on Jenkins: - **Git Plugin**: For fetching code from version control systems like GitHub. - **Nexus Plugin**: To store artifacts in Nexus. - **SonarQube Plugin**: For running code quality checks using SonarQube. - **Maven Plugin**: To build Java projects using Maven. - **BuildTimestamp Plugin**: For managing timestamps during builds. - **Pipeline Utility Steps**: For utilities in Jenkins pipelines.

How to Install Plugins:

1. Go to **Jenkins Dashboard** > **Manage Jenkins**.
2. Click on **Manage Plugins**.
3. Navigate to the **Available** tab.
4. Search and install the required plugins.

Step 3: Writing the CI Pipeline Script

Pipeline Script Overview:

Once the necessary tools and plugins are installed, write a pipeline script in Jenkins to automate the process: 1. **Source Code Fetching**: Use the Git plugin to pull the code from the repository (e.g., GitHub). 2. **Build Process**: Build the project using Maven, which will generate the artifacts. 3. **Code Quality Checks**: Integrate SonarQube to analyze the code for vulnerabilities and best practices. 4. **Artifact Storage**: Store the generated artifacts in Nexus after they pass all tests.

Notifications:

Set up notifications within the pipeline to alert users if any steps fail during the build process.

Step 4: Creating User Data Scripts

To automate the setup of Jenkins, Nexus, and SonarQube on EC2 instances, use user data scripts provided in the following GitHub repository: - GitHub Repo for Setup Scripts

This repository contains scripts for configuring Jenkins, Nexus, and SonarQube on their respective EC2 instances.

Conclusion

By setting up Jenkins, Nexus, and SonarQube with the required plugins and using EC2 instances, we can create a fully automated **Continuous Integration Pipeline**. This pipeline enables seamless code integration, building, testing, and storing of artifacts while ensuring code quality.

Pipeline as Code with Jenkins

In this section, we will discuss how to implement **Pipeline as Code** using a `Jenkinsfile`, which defines the stages of a CI/CD pipeline in a text file format. This approach allows for automation and easier version control of the pipeline configuration.

Introduction to Pipeline as Code

What is Pipeline as Code?

- **Automates pipeline setup** by defining the pipeline in a `Jenkinsfile`.
- A `Jenkinsfile` contains the pipeline's stages and steps using **Pipeline DSL Syntax**, which is based on **Groovy**.
- There are two types of pipelines:
  - **Scripted Pipeline**: More flexible but harder to manage.
  - **Declarative Pipeline**: More structured and easier to use.

In most cases, we use **Declarative Pipeline** to define the CI/CD process.

Pipeline Concepts

Key Components:

1. **Pipeline Block**: The core block where the entire Jenkins pipeline code resides.
2. **Node/Agent**: Specifies where the pipeline should execute (on which node or agent).
3. **Stages**: Represents the phases in the pipeline (e.g., Build, Test, Deploy).
4. **Steps**: Commands or actions to be executed in each stage (e.g., Maven install, Git pull, upload to Nexus).

Example Pipeline Structure

```
pipeline {
    agent { label "master" }
    tools { maven "Maven" }
    environment {
        NEXUS_VERSION = "nexus3"
        NEXUS_PROTOCOL = "http"
        NEXUS_URL = "your-ip-address-here:8081"
        NEXUS_REPOSITORY = "maven-nexus-repo"
        NEXUS_CREDENTIAL_ID = "nexus-user-credentials"
        ARTVERSION = "${env.BUILD_ID}"
    }
    stages {
        stage('Build') {
            steps {
                // Build commands go here
            }
        }
        stage('Test') {
            steps {
                // Testing steps
            }
```

```
        }
        stage('Deploy') {
            steps {
                // Deployment commands
            }
        }
    }
}
```

Pipeline Breakdown

1. Pipeline Block

The main block of the pipeline where all configurations and stages are defined.

```
pipeline {
    // Configuration and stages go here
}
```

2. Agent Block

The `agent` block specifies where the pipeline will run. You can choose to run the job on the master node, a specific agent, or any available node.

```
agent {
    label "master"
}
```

3. Tools Block

The `tools` block defines tools (like Maven, JDK, etc.) to be used, which are pre-configured in Jenkins' global tool configuration.

```
tools {
    maven "Maven"
}
```

4. Environment Variables

Environment variables are declared to store configuration values such as Nexus server details, repository information, and build numbers. These variables can be reused throughout the pipeline.

```
environment {
    NEXUS_VERSION = "nexus3"
    NEXUS_PROTOCOL = "http"
    NEXUS_URL = "your-ip-address-here:8081"
    NEXUS_REPOSITORY = "maven-nexus-repo"
    NEXUS_CREDENTIAL_ID = "nexus-user-credentials"
```

```
    ARTVERSION = "${env.BUILD_ID}"
}
```

5. Stages

Stages represent the different phases of the CI/CD process, such as **Build**, **Test**, and **Deploy**. Each stage contains multiple steps to execute specific tasks.

```
stages {
    stage('Build') {
        steps {
            // Steps for building the project
        }
    }
    stage('Test') {
        steps {
            // Steps for running tests
        }
    }
    stage('Deploy') {
        steps {
            // Steps for deployment
        }
    }
}
```

6. Steps

Each stage can have multiple steps. Steps represent the individual tasks to be executed in that stage, such as running commands or scripts.

```
steps {
    sh 'mvn clean install'  // Example of a step to build a Maven project
}
```

Summary

Using **Pipeline as Code** with a `Jenkinsfile` enables automation and efficient version control of your CI/CD pipelines. The pipeline can be configured declaratively to define: - **Agents** (where the job runs) - **Tools** (like Maven, JDK) - **Environment variables** for reusable configuration - **Stages** and **steps** for build, testing, and deployment

This approach simplifies CI/CD management and improves maintainability by allowing you to store pipeline configurations in your version control system.

Continuous Integration with Jenkins: Pipeline as Code

Introduction to Pipeline as Code

Pipeline as Code allows us to automate the setup and execution of CI/CD pipelines by defining them in a `Jenkinsfile`. This file includes all the necessary stages and steps, which are executed automatically by Jenkins.

Key Features: - Automates pipeline configuration through a `Jenkinsfile`. - Supports defining multiple **stages** and **steps** for tasks like cloning code, building projects, publishing artifacts, and performing tests. - Each pipeline stage can include commands and post-build actions such as archiving artifacts or sending notifications.

Basic Pipeline Structure

The pipeline begins with a **pipeline block** that contains various stages. Each stage represents a significant phase in the CI/CD process, such as: - **Cloning code from VCS (Version Control System) - Running Maven build - Publishing artifacts to Nexus Repository - Code analysis or deploying artifacts**

Example:

```
pipeline {
    stages {
        stage("Clone code from VCS") {
            steps {
                // Commands to clone code
            }
        }
        stage("Maven Build") {
            steps {
                sh 'mvn clean install'
            }
        }
        stage("Publish to Nexus Repository Manager") {
            steps {
                // Commands to publish to Nexus
            }
        }
    }
}
```

Stages and Steps

- **Stages**: The main execution phases where specific tasks are performed. You can define multiple stages in a pipeline.
- **Steps**: Within each stage, steps are defined to execute specific tasks. For instance, in the build stage, Maven commands can be executed.

Each stage can include additional actions: - **Post-build steps**: These are actions that occur after a stage is completed. For example, archiving artifacts or sending email notifications upon successful builds.

Example of a pipeline with post-build steps:

```
pipeline {
    stage('BuildAndTest') {
        steps {
            sh 'mvn install'
        }
        post {
            success {
                echo 'Now Archiving...'
                archiveArtifacts artifacts: 'target/*.war'
            }
        }
    }
}
```

Automation in DevOps Pipelines

In DevOps, automation is key. The entire pipeline is defined in the `Jenkinsfile`, enabling consistent and repeatable processes. Automation extends not only to the build process but also to testing, code analysis, artifact publishing, and even notifications.

Tool Configuration in Jenkins

When specifying tools in the Jenkins pipeline, ensure that they are consistent with those configured in the **Global Tool Configuration** of Jenkins. This ensures proper execution of tasks such as building projects with Maven or deploying to Nexus.

Summary

- **Pipeline as Code** automates the CI/CD pipeline configuration.
- Stages such as cloning code, building with Maven, and publishing artifacts can be defined in a `Jenkinsfile`.
- Each stage contains steps and can include post-build actions like archiving and notifications.

- Proper tool configuration in Jenkins is essential for successful pipeline execution.

By defining the pipeline in code, DevOps teams can ensure that CI/CD processes are automated, repeatable, and version-controlled.

DevOps Notes: Continuous Integration with Jenkins

Pipeline as Code

- **Jenkinsfile**: Automates pipeline setup with stages defined for CI/CD processes.
  - Written in **Pipeline DSL Syntax**, similar to Groovy language.
  - Supports two types of pipelines:
    - **Scripted Pipeline**
    - **Declarative Pipeline** (used here).

Pipeline Concepts

- **Pipeline**: The main block where all tasks are executed.
- **Node/Agent**: Specifies where the pipeline should run (master, slave, etc.).
- **Stages**: Sections of the pipeline where tasks are performed (e.g., build, test, deploy).
- **Steps**: Commands executed within a stage (e.g., `mvn install`, `git pull`).

Pipeline Structure Example

```
pipeline {
    agent {
        label "master"
    }
    tools {
        maven "Maven"
    }
    environment {
        NEXUS_VERSION= "nexus3"
        NEXUS_PROTOCOL = "http"
        NEXUS_URL = "your-ip-address:8081"
        NEXUS_REPOSITORY = "maven-nexus-repo"
        NEXUS_CREDENTIAL_ID = "nexus-user-credentials"
        ARTVERSION = "${env.BUILD_ID}"
    }
    stages {
```

```
        stage('Clone code from VCS') {
            steps {
                git branch: 'master', url: 'https://github.com/repo.git'
            }
        }
        stage('Maven Build') {
            steps {
                sh 'mvn install -DskipTests'
            }
        }
        stage('Publish to Nexus Repository Manager') {
            steps {
                // Add your publishing steps here
            }
        }
    }
}
```

Detailed Explanation of the Pipeline

- **Agent Block**: Defines the node where the pipeline will run (e.g., `master`).
- **Tools Block**: Defines the tools (e.g., Maven, JDK) from global tool configuration.
- **Environment Block**: Sets environment variables (e.g., Nexus configuration).
- **Stages**: Consists of different phases like:
  - Cloning code from VCS.
  - Building with Maven.
  - Publishing artifacts to Nexus.

Example: Build and Test Stage

```
pipeline {
    stage('BuildAndTest') {
        steps {
            sh 'mvn install'
        }
        post {
            success {
                echo 'Now Archiving...'
                archiveArtifacts artifacts: '**/target/*.war'
            }
        }
    }
}
```

- **Post Section**: Defines actions after a successful build (e.g., archiving artifacts).

Example: Unit Testing and Code Analysis

```
pipeline {
    stages {
        stage('UNIT TEST') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Checkstyle Analysis') {
            steps {
                sh 'mvn checkstyle:checkstyle'
            }
        }
    }
}
```

- **Unit Test Stage**: Executes Maven tests.
- **Checkstyle Analysis Stage**: Runs a code quality check using Checkstyle.

Jenkins Pipeline Setup Options

1. **Pipeline Script**: Directly paste the pipeline code.
2. **Pipeline Script from SCM**: Link to a repository containing the `Jenkinsfile`.

Running the Pipeline

1. Navigate to the created pipeline.
2. Click **Build Now** to execute the pipeline.
3. Check build history and workspace under specific build IDs.

Additional Links

- Jenkins Pipeline Documentation

These notes cover essential Jenkins pipeline concepts, including pipeline structure, stages, and example code snippets for build, test, and deployment stages.

DevOps Notes: Continuous Integration with Jenkins - Code Analysis

Introduction

In a continuous integration (CI) pipeline, the goal is to: 1. Fetch code from the repository. 2. Build the code. 3. Run unit tests. 4. Perform code analysis. 5. Upload the final artifact.

Code analysis helps detect vulnerabilities and functional errors early, improving code quality before deployment.

Code Analysis Overview

- **Purpose**: To check the code against best practices in the programming language, and detect vulnerabilities like memory leaks or other weak spots that could expose the software to risks.
- **Functional Errors**: Detected before deployment, ensuring that these don't weaken the software even if they don't cause immediate issues.
- **Tools**: There are several tools available for code analysis. Popular ones include:
  - Checkstyle
  - Cobertura
  - MSTest
  - OWASP
  - **SonarQube** (used for this demonstration)

Integrating SonarQube with Jenkins

Steps:

1. **Install SonarQube Scanner**: Ensure SonarQube is installed and integrated with Jenkins.
2. **Set Up SonarQube in Jenkins**:
   - Navigate to Jenkins **Global Tool Configuration**.
   - Add SonarQube instance with the proper URL and port (from your SonarQube AWS instance).
   - Generate and add the SonarQube authentication token (Profile → Settings → Tokens → Generate Token).

Jenkins Global Tool Configuration

- After installing the SonarQube plugin, go to the **Global Tool Configuration** in Jenkins.

- Add SonarQube as a new tool and ensure it's accessible through the **SonarQube server URL**.

SonarQube Code Analysis in Jenkins Pipeline

Pipeline Stages for Code Analysis

```
pipeline {
    agent any

    stages {
        stage('Build & SonarQube Analysis') {
            steps {
                withSonarQubeEnv('My SonarQube Server') {
                    sh 'mvn clean package sonar:sonar'
                }
            }
        }

        stage('Quality Gate') {
            steps {
                timeout(time: 1, unit: 'HOURS') {
                    waitForQualityGate abortPipeline: true
                }
            }
        }
    }
}
```

- **SonarQube Analysis Stage**: Scans the code and generates a report on the SonarQube server.
- **Quality Gate**: A stage added to ensure the code meets certain quality metrics before proceeding. If the code fails the quality gate, the pipeline will be aborted.

Example: SonarQube Analysis with Environment Variables

```
pipeline {
    agent any

    environment {
        scannerHome = tool 'sonar4.7'
    }
```

```
    stages {
        stage('Sonar Analysis') {
            steps {
                withSonarQubeEnv('sonar') {
                    sh '''${scannerHome}/bin/sonar-scanner \
                    -Dsonar.projectKey=myproject \
                    -Dsonar.projectName=myproject \
                    -Dsonar.projectVersion=1.0 \
                    -Dsonar.sources=src/ \
                    -Dsonar.java.binaries=target/test-classes/ \
                    -Dsonar.junit.reportsPath=target/surefire-reports/ \
                    -Dsonar.jacoco.reportsPath=target/jacoco.exec \
                    -Dsonar.java.checkstyle.reportPaths=target/checkstyle-result.xml'''
                }
            }
        }
    }
}
```

- **Environment Variable**: `scannerHome` points to the SonarQube scanner's home directory.
- **SonarQube Scanner**: Scans the project code, uploads reports to the SonarQube server, and checks the following:
  - **sonar.projectKey**: The unique key for the project in SonarQube.
  - **sonar.sources**: The directory containing the source code.
  - **sonar.java.binaries**: The path to the Java binaries generated during the build.
  - **sonar.junit.reportsPath**: The location of unit test reports.
  - **sonar.jacoco.reportsPath**: The path to the JaCoCo code coverage report.
  - **sonar.checkstyle.reportPaths**: The location where Checkstyle results are stored.

Quality Gate in SonarQube

- **Quality Gate**: A set of conditions a project must meet to ensure code quality.
  - SonarQube provides default quality gates.
  - If the pipeline passes the quality gate, the build proceeds; otherwise, it fails.

Summary

- **Code Analysis** in Jenkins CI pipelines is crucial for detecting vulnerabilities and functional issues before deployment.
- Tools like **SonarQube** are integrated to provide insights into code quality and ensure compliance with coding standards.
- With **SonarQube**, the pipeline not only scans the code but also evaluates it against a **quality gate** to ensure it meets defined quality standards before proceeding.

Additional Reference:

- Jenkins SonarQube Documentation for further details on integrating SonarQube with Jenkins.

Continuous Integration with Jenkins

Automating Build and Notification with Jenkins

In our current CI pipeline, we aim to automate two essential tasks: 1. **Auto-building the project** without manual intervention. 2. **Sending build result notifications** automatically via Slack.

Jenkins Slack Notification Setup

1. **Slack Plugin Setup**: Jenkins has an extensive range of plugins. To send notifications after build completion, we use the **Slack Notification Plugin**. Slack is a widely-used collaboration tool for real-time communication.
2. **Slack Account Setup**:
   - Create a Slack account or log in.
   - Set up a dedicated Slack channel for build notifications.
   - Authenticate Jenkins with Slack by obtaining a **token**:
     - In Slack, go to **Add Apps for Slack**, search for **JenkinsCI**, and click **Add to Slack**.
     - Select a Slack channel to post build notifications and click **Add Jenkins CI Integration**.
     - **Save the token** for future use.
3. **Integrate Slack with Jenkins**:
   - In Jenkins, navigate to **Dashboard** > **Configuration** and locate the Slack section.
   - Enter the Slack workspace name, token, and Slack channel.
   - Click **Test Connection** to verify successful integration.
   - Once successful, save the configuration.

Slack Notification in Jenkins Pipeline To send notifications in Jenkins after the build, we need to add Slack notification steps to the Jenkinsfile.

Color Mapping for Slack Notification: Slack recognizes the status of the build as either **good** or **danger**. We map Jenkins' build results to these statuses:

```
def COLOR_MAP = [
    'SUCCESS':'good',
    'FAILURE':'danger',
]
```

Post Installation Step for Slack Notification: The notification is sent after the build completes by adding a **post** block in the Jenkinsfile:

```
post {
    always {
        echo 'Slack Notifications.'
        slackSend channel: '#jenkinscicd',
                  color: COLOR_MAP[currentBuild.currentResult],
                  message: "*${currentBuild.currentResult}:* Job ${env.JOB_NAME} build ${env
    }
}
```

This sends a notification to the Slack channel `#jenkinscicd`, highlighting the build result (success or failure) and providing the job name, build number, and build URL.

Continuous Integration for Docker with Jenkins

Pipeline Overview for Docker In the Docker integration, we automate the following process: 1. **Code Push to GitHub**: Developers push their code to GitHub after local testing. 2. **Jenkins Polls GitHub**: Jenkins continuously monitors the GitHub repository for changes. 3. **Build & Unit Testing**: - Jenkins fetches the updated code. - Unit tests are executed using **Maven**. 4. **Code Analysis**: - Checkstyle and SonarQube perform code quality checks. - If all quality gates are passed, the artifacts (WAR files and reports) are generated. 5. **Docker Build**: - The application is containerized using Docker. 6. **Artifact Push**: - Docker images are pushed to cloud registries such as **Amazon ECR**, **Google Cloud Registry**, or **Azure Cloud Registry**.

By integrating Docker into the pipeline, we ensure seamless deployment of applications, making the CI/CD process efficient and robust.

Python Notes for DevOps

1. Print Format in Python

The new and more efficient way to format strings using variables in Python is through **f-strings**:

```
name = "COVID-19"
disease = "Respiratory issues"
print(f"The name of the virus is {name} and it causes {disease}")
```

2. Built-in Functions or Methods

Python offers a wide range of built-in functions for various operations. Here are some helpful references for exploring these functions:

- Python Built-in Functions
- W3Schools Python Functions Reference
- W3Schools Python String Methods
- W3Schools Python List Methods
- W3Schools Python Dictionary Methods
- W3Schools Python Tuple Methods
- W3Schools Python Set Methods
- W3Schools Python File Methods

Immutability of Strings Strings in Python are **immutable**, meaning that any modification returns a new string rather than altering the original one.

```
original = "hello"
modified = original.upper()  # returns 'HELLO' but original remains 'hello'
```

3. `dir()` Function

The `dir()` function provides a list of attributes and methods available for an object.

```
dir(str)  # Lists all methods for a string object
```

4. Variable Arguments in Functions

Python allows for variable numbers of arguments in functions using `*args` for positional arguments and `**kwargs` for keyword arguments.

```
def time_activity(*args, **kwargs):
    print(args)  # Outputs tuple of arguments
    print(kwargs)  # Outputs dictionary of keyword arguments

time_activity(10, 20, sport="Boxing", work="DevOps")
```

5. Modules in Python

Modules are reusable pieces of code in Python that can be imported into other scripts.

```
# Import all functions from a module
from admin import *

# Import a specific function from a module
from admin import headmaster

# Import the entire module
import admin
```

6. OS Tasks in Python

Python's `os` module is useful for performing operating system-related tasks such as checking files or directories and managing system users and groups.

```
import os
path = '/tmp/testfile.txt'

if os.path.isdir(path):
    print("It is a directory")
elif os.path.isfile(path):
    print("It is a file.")
else:
    print("File or directory doesn't exist")
```

Example: Adding Users and Groups

```
#!/usr/bin/python3
import os

userlist = ["alpha", "beta", "gamma"]
print("Adding Users to system")
for user in userlist:
    exitcode = os.system(f"id {user}")
    if exitcode != 0:
        print(f"User {user} doesn't exist. Adding it.")
        os.system(f"useradd {user}")
    else:
        print("User already exists, skipping it.")

# Adding users to a group
exitcode = os.system("grep science /etc/group")
if exitcode != 0:
    print("Group 'science' doesn't exist, creating one.")
```

```
    os.system("groupadd science")
else:
    print("Group already exists, skipping it.")

for user in userlist:
    print(f"Adding user {user} to 'science' group")
    os.system(f"usermod -G science {user}")

# Managing directory
if not os.path.isdir("/opt/science_dir"):
    os.mkdir("/opt/science_dir")
    print("Created directory '/opt/science_dir'")
os.system("chown :science /opt/science_dir")
os.system("chmod 770 /opt/science_dir")
```

7. Python Fabric for Automation

Fabric is a Python library used for automating application development and system administration tasks over SSH. To install:

```
pip install fabric<2.0
```

Fabric simplifies repetitive tasks by allowing developers to script workflows.

8. Python-Jenkins Integration

The `python-jenkins` module allows for automating Jenkins server tasks through Python code. Installation is done via pip:

```
pip install python-jenkins -y
```

Common tasks that can be automated using `python-jenkins` include: - Creating, copying, deleting, and updating Jenkins jobs. - Retrieving job information, build status, and plugin details. - Managing Jenkins nodes and views.

For more details, refer to the Python Jenkins Documentation.

9. Boto3 for AWS

Boto3 is the AWS SDK for Python, which allows developers to interact with AWS services such as EC2, S3, and more.

```
pip install boto3
```

Documentation: Boto3 Documentation

Conclusion

This guide covers essential Python concepts, built-in functions, OS-level operations, module management, Jenkins automation with Python, and AWS inte-

gration with Boto3.

DevOps Notes: Python Fabric

1. Introduction to Python Fabric Fabric is a Python module designed to automate administration tasks using SSH. It simplifies running command-line tasks both locally and remotely.

2. Setting Up Fabric To begin using Fabric: 1. Install Fabric: `bash pip install fabric<2.0` 2. Create a directory for your Fabric script: `bash mkdir fabric cd fabric/ vim fabfile.py` 3. To see available Fabric commands: `bash fab -l`

3. Basic Fabric Functions

- **Greeting Function**: Simple function to print a greeting message. "'python from fabric.api import *

  def greeting(msg): print("Good %s" % msg) "'

- **System Information Function**: Prints system disk space, RAM size, and uptime. `python def system_info(): print("Disk Space") local("df -h") print("RAM size") local("free -m") print("System uptime.") local("uptime")`

- **Remote Execution Function**: Runs commands on a remote server using SSH. `python def remote_exec(): print("Get System info") run("hostname") run("uptime") run("df -h") run("free -m") sudo("yum install mariadb-server") sudo("systemctl start mariadb") sudo("systemctl enable mariadb")`

4. Website Setup Using Fabric Fabric can automate setting up a web server:

```
def web_setup(WEBURL, DIRNAME):
    local("apt install zip unzip -y")
    sudo("yum install httpd wget unzip -y")
    sudo("systemctl start httpd")
    sudo("systemctl enable httpd")
    local("wget -o website.zip %s" % WEBURL)
    local("unzip -o website.zip")
    with lcd(DIRNAME):
        local("zip -r tooplate.zip *")
    put("tooplate.zip", "/var/www/html/", use_sudo=True)
    with cd("/var/www/html/"):
        sudo("unzip tooplate.zip")
```

```
    sudo("systemctl restart httpd")
    print("Website setup is done")
```

5. Executing Fabric Functions To execute functions in the `fabfile.py` script, use the following commands:

- To execute the greeting function: `bash fab greeting:Evening`
- To execute system information: `bash fab system_info`

6. Running Fabric Commands on Remote Server

- To run the `remote_exec` function on a remote server: `bash fab remote_exec`

Note: You need to set up user credentials and ensure the user has `sudo` privileges. Use the `visudo` command to grant root access:

```
devops ALL=(ALL) NOPASSWD:ALL
```

Enable password authentication in `/etc/ssh/sshd_config`:

```
PasswordAuthentication yes
```

7. Passwordless SSH Login To set up passwordless login: 1. Generate an SSH key: `bash ssh-keygen` 2. Copy the SSH key to the remote server: `bash ssh-copy-id devops@192.168.10.3` After this, you can log in without a password.

8. Running Fabric Commands on Remote Server with Passwordless SSH To run Fabric functions on a remote server using passwordless login:

```
fab -H <ip-address> -u <user-name> <function-to-exec>
```

9. Example: Remote Web Setup Using Fabric, you can run commands both locally and remotely with root privileges. For example, to set up a website remotely:

```
fab web_setup:http://example.com,website_directory
```

10. Virtual Environment in Python To create and use a Python virtual environment: 1. Install virtualenv: `bash pip install virtualenv` 2. Create a virtual environment: `bash virtualenv <virtual-env-name>` 3. Activate the virtual environment: `bash source ./<virtual-env-name>/bin/activate` 4. Deactivate the virtual environment: `bash deactivate`

These notes summarize essential Python Fabric tasks, allowing you to automate command-line tasks locally or remotely in a structured and reusable way.

DevOps Notes: Continuous Integration with Jenkins and Docker

1. Prerequisites for Connecting Jenkins to AWS ECR To integrate Jenkins with AWS Elastic Container Registry (ECR), the following steps and resources are required:

1. **IAM User with ECR Permissions**:
   - Create an IAM user with permissions to access ECR.
2. **Store AWS Credentials in Jenkins**:
   - Store the AWS credentials (access key and secret key) in Jenkins using Jenkins' credentials store.
3. **Create ECR Repository**:
   - In AWS, create a repository where Docker images will be stored.
4. **Environment Variables**:
   - Set the following environment variables in the Jenkins pipeline:

```
environment {
    registryCredential = 'ecr:us-east-2:awscreds' // AWS credentials for ECR
    appRegistry = "951401132355.dkr.ecr.us-east-2.amazonaws.com/vprofileappimg" //
    vprofileRegistry = "https://951401132355.dkr.ecr.us-east-2.amazonaws.com" // H
}
```

2. Jenkins Setup for Docker CI To set up Docker on Jenkins, follow these steps:

1. **Install Docker Pipeline Plugin**:
   - Jenkins requires the Docker Pipeline plugin to manage Docker containers in the pipeline.
2. **Install Docker Engine**:
   - Install Docker on the Jenkins server and add the Jenkins user to the Docker group. Then reboot the server.

```
sudo usermod -aG docker jenkins
sudo systemctl reboot
```

3. **Install AWS CLI**:
   - AWS CLI is used to interact with AWS services like ECR.
4. **Create ECR Repository**:
   - Create an ECR repository in AWS where Docker images will be stored.
5. **Install Required Plugins**:
   - Install plugins such as **ECR**, **Docker Pipeline**, and **AWS SDK** in Jenkins to handle AWS credentials and Docker images.
6. **Store AWS Credentials in Jenkins**:
   - Add AWS access credentials in Jenkins under **Credentials** and reference them in the pipeline.
7. **Run the Pipeline**:
   - Execute the Jenkins pipeline to build and push Docker images to ECR.

3. Jenkins Pipeline for Docker CI/CD

```
pipeline {
    agent any
    tools {
        maven "MAVEN3"
        jdk "OracleJDK8"
    }
    environment {
        registryCredential = 'ecr:us-east-2:awscreds'
        appRegistry = "951401132355.dkr.ecr.us-east-2.amazonaws.com/vprofileappimg"
        vprofileRegistry = "https://951401132355.dkr.ecr.us-east-2.amazonaws.com"
    }
    stages {
        stage('Fetch Code') {
            steps {
                git branch: 'docker', url: 'https://github.com/devopshydclub/vprofileproject
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Code Analysis with Checkstyle') {
            steps {
                sh 'mvn checkstyle:checkstyle'
            }
            post {
                success {
                    echo 'Generated Analysis Result'
                }
            }
        }
        stage('Build & SonarQube Analysis') {
            environment {
                scannerHome = tool 'sonar4.7'
            }
            steps {
                withSonarQubeEnv('sonar') {
                    sh '''
                    ${scannerHome}/bin/sonar-scanner \
                    -Dsonar.projectKey=vprofile \
                    -Dsonar.projectName=vprofile-repo \
                    -Dsonar.projectVersion=1.0 \
                    -Dsonar.sources=src/ \
```

```
                    -Dsonar.java.binaries=target/test-classes/ \
                    -Dsonar.junit.reportsPath=target/surefire-reports/ \
                    -Dsonar.jacoco.reportsPath=target/jacoco.exec \
                    -Dsonar.java.checkstyle.reportPaths=target/checkstyle-result.xml
                    '''
                }
            }
        }
        stage('Quality Gate') {
            steps {
                timeout(time: 1, unit: 'HOURS') {
                    waitForQualityGate abortPipeline: true
                }
            }
        }
        stage('Build App Image') {
            steps {
                script {
                    dockerImage = docker.build("${appRegistry}:${BUILD_NUMBER}", "./Docker-f
                }
            }
        }
        stage('Upload App Image') {
            steps {
                script {
                    docker.withRegistry(vprofileRegistry, registryCredential) {
                        dockerImage.push("${BUILD_NUMBER}")
                        dockerImage.push('latest')
                    }
                }
            }
        }
    }
}
```

4. **Explanation of Pipeline Stages**

- **Fetch Code**: Fetch the latest code from the Git repository. `groovy git branch: 'docker', url: 'https://github.com/devopshydclub/vprofileproject.git'`
- **Test**: Run unit tests using Maven. `groovy sh 'mvn test'`
- **Code Analysis (Checkstyle)**: Perform code analysis using Checkstyle. `groovy sh 'mvn checkstyle:checkstyle'`
- **SonarQube Analysis**: Analyze the code quality using SonarQube. `groovy withSonarQubeEnv('sonar') { sh '''`

```
${scannerHome}/bin/sonar-scanner \      -Dsonar.projectKey=vprofile
\      -Dsonar.projectName=vprofile-repo \      -Dsonar.projectVersion=1.0
\      -Dsonar.sources=src/ \      -Dsonar.java.binaries=target/test-classes/
\      -Dsonar.junit.reportsPath=target/surefire-reports/
\      -Dsonar.jacoco.reportsPath=target/jacoco.exec \
-Dsonar.java.checkstyle.reportPaths=target/checkstyle-result.xml
'''  }
```

- **Quality Gate**: Ensure the quality gate is passed before proceeding. `groovy  waitForQualityGate abortPipeline: true`
- **Build App Image**: Build the Docker image for the application. `groovy dockerImage = docker.build("${appRegistry}:${BUILD_NUMBER}", "./Docker-files/app/multistage/")`
- **Upload App Image**: Push the Docker image to ECR. `groovy docker.withRegistry(vprofileRegistry, registryCredential) { dockerImage.push("${BUILD_NUMBER}")      dockerImage.push('latest') }`

This pipeline demonstrates how to automate Continuous Integration (CI) and Docker-based Continuous Delivery (CD) using Jenkins, AWS ECR, and Docker.

DevOps Notes: Continuous Integration with Jenkins and Docker

1. Setting Up Jenkins and Docker

- First, log in to the Jenkins server and install Docker by following the official Docker documentation:
  Docker Installation on Ubuntu
- Ensure you are a root user while installing Docker.

2. Configuring Jenkins to Use Docker

- After installing Docker, map the Jenkins user to the Docker group so that Jenkins can execute Docker commands.
  Check if the Jenkins user belongs to the Docker group: `bash  id jenkins` If Docker group is not listed, add it using: `bash  usermod -a -G docker jenkins`
- Reboot or restart Jenkins after this change to apply the new Docker group membership.

3. Installing AWS CLI

- Install AWS CLI to allow pushing Docker images to AWS ECR (Elastic Container Registry): `bash  apt install awscli -y`

4. **Creating an IAM User and ECR Repository in AWS**

- **IAM User Creation**:
  1. Go to AWS services and search for **IAM**.
  2. In Access Management, navigate to **Users** and click **Add User**.
  3. Provide a username and select **Access key - programmatic access**.
  4. On the **Set Permissions** page, select "Attach existing policies directly" and assign the following permissions:
     - `AmazonEC2ContainerRegistryFullAccess`
     - `AmazonECS_FullAccess`
  5. After creating the user, download the access key and secret key as a CSV file.
- **ECR Repository Creation**:
  1. Go to AWS services and search for **Elastic Container Registry**.
  2. Click **Create Repository** and provide a repository name.
  3. Copy the repository URI for use in Jenkins.

5. **Installing Jenkins Plugins for Docker and AWS Integration**

- Go to the Jenkins **Plugin Manager** and install the following plugins:
  1. **Docker Pipeline**: To build and use Docker containers in pipelines.
  2. **Amazon ECR**: Generates Docker authentication tokens from AWS credentials to access Amazon ECR.
  3. **Amazon Web Services SDK**: Provides AWS SDK for Java modules.
  4. **CloudBees Docker Build and Publish**: Enables building Dockerfile-based projects and publishing images to Docker registries.

6. **Adding AWS Credentials in Jenkins**

- Navigate to: `Dashboard > Credentials > System > Global Credentials (unrestricted) > Add Credentials`
- Choose **AWS Credentials** and input the access key ID and secret access key obtained from the IAM user creation step.

7. **Setting Up Jenkins Pipeline**

- Update your pipeline script with the necessary environment variables:

```
groovy      pipeline {          environment {              registryCredential
= 'ecr:<region>:<credential-id>'              appRegistry =
'<REGISTRY_URL>/registryname'              vprofileRegistry
= '<REGISTRY_PATH>'          }          stages {              stage('Build')
{                  steps {                      script {
// Pipeline steps for building and pushing Docker images
}                  }              }          }      }
```

- **Key Variables**:

- `registryCredential`: Credential ID for the ECR registry.
- `appRegistry`: The full Docker registry URL including the image name.
- `vprofileRegistry`: Full path to the ECR registry.

8. Running the Jenkins Pipeline

- After configuring the pipeline script, run the job. AWS credentials will automatically be used for authentication, allowing passwordless access to AWS services.

9. Push Docker Images to AWS ECR

- Edit the pipeline to include the steps to build, tag, and push Docker images to the ECR repository. Make sure to use the `ECR` registry details and tags in your pipeline configuration. Build the pipeline to deploy your Docker images into AWS ECR.

This guide helps set up a Jenkins pipeline with Docker and AWS integration, including necessary configurations, AWS setup, and Jenkins plugin installations.

DevOps Notes: Continuous Integration and Delivery with Jenkins, Docker, and AWS ECS

1. Extending CI to Continuous Delivery

- The goal is to extend the Jenkins Continuous Integration (CI) pipeline to include Continuous Delivery (CD).
- At the final stage of CI, Docker images are published to Amazon ECR (Elastic Container Registry).
- These images are then hosted on **Amazon ECS (Elastic Container Service)**, a Docker container hosting platform that pulls the latest image from ECR and hosts it on ECS.

2. Docker Hosting Platforms

- **Local Hosting**:
  Docker images can be run locally for development and testing using: `bash docker run <image-name>` However, local hosting lacks production features like high availability, self-healing, etc.
- **Production Hosting**:
  For production environments, container orchestration platforms like **Kubernetes** are used. Available options include:
  - AWS: **EKS (Elastic Kubernetes Service)**
  - Azure: **AKS (Azure Kubernetes Service)**

  - Google Cloud: **GKE (Google Kubernetes Engine)**
  - Red Hat: **OpenShift**

- **Amazon ECS**:
  The current solution utilizes **Amazon ECS** for hosting containerized applications. ECS provides a fully managed container orchestration service that integrates with other AWS services.

3. ECS Clusters

- **Cluster Creation**:
  An ECS cluster is a logical grouping of **EC2 instances** or **Fargate tasks** that run containerized applications. When creating an ECS cluster, you specify:
  - The number of EC2 instances or Fargate tasks.
  - Configuration details like instance type, AMI (Amazon Machine Image), and networking settings.
- **Task Definitions and Services**:
  - A **task definition** describes how to run containers, including the container's resources and networking requirements.
  - An **ECS service** ensures a specified number of instances of a task definition run simultaneously in a cluster. Services can be configured with auto-scaling and fault tolerance.

4. ECS Services and Scaling

- **Service Management**:
  ECS services can dynamically scale and update as needed. For example:
  - Update a service to use a new version of a task definition.
  - Scale the service up or down by adjusting the number of running tasks.
- **Integration with Other AWS Services**:
  ECS integrates with services like **Elastic Load Balancing (ELB)** and **Auto Scaling** for increased functionality and scalability.

5. Jenkins Pipeline Configuration for ECS Deployment

- **Environment Variables**:
  The Jenkins pipeline defines the following environment variables: `groovy environment { registryCredential = 'ecr:us-east-2:awscreds' appRegistry = "951401132355.dkr.ecr.us-east-2.amazona vprofileRegistry = "https://951401132355.dkr.ecr.us-east-2.amazonaws.com" cluster = "vprofile" service = "vprofileappsvc" }` These variables store essential information, such as the ECR registry, cluster, and service details.

- **Deploy to ECS Stage**:
  The following pipeline stage deploys the Docker image to ECS: `groovy`

```
stage('Deploy to ecs') {          steps {              withAWS(credentials:
'awscreds', region: 'us-east-2') {                  sh 'aws
ecs update-service --cluster ${cluster} --service ${service}
--force-new-deployment'              }          }      }
```

  - This step updates the ECS service with the latest image.
  - The `--force-new-deployment` flag forces a new task deployment, pulling the latest image from ECR and updating the running container.

6. Deploying Docker Images to ECS

- **Cluster and Service Setup**:
  Before deploying, ensure the ECS cluster and service are created in the AWS Elastic Container Service.
- **Image Deployment Process**:
  - Jenkins uses AWS credentials and region details to trigger the ECS service update.
  - The `--force-new-deployment` command creates a new task in ECS, pulls the latest Docker image from ECR, and replaces the old image in the service with the new one.

Summary: In this pipeline, Jenkins integrates with AWS ECS for a CI/CD solution. Docker images are built and stored in ECR, and then deployed to ECS clusters. The pipeline automates service updates, ensuring the latest images are deployed with every release.

DevOps Notes: AWS ECS Setup and Jenkins Integration

1. Introduction to Amazon ECS

- **Amazon ECS (Elastic Container Service)** is a scalable and fast container management service that allows you to run, stop, and manage containers on a cluster.
- **Jenkins Plugin**: To use ECS with Jenkins, install the necessary plugin.
- **Setup Steps**: Access ECS via AWS services and switch to the "new ECS experience" if prompted.

2. Creating an ECS Cluster

- **Networking Configuration**:
  - When creating a cluster, provide a **cluster name**.
  - ECS tasks and services run in **default subnets** of the default **VPC** unless custom VPC and subnets are specified.
  - In production, it's recommended to use **at least three subnets** for high availability.

- **Infrastructure Options**:
  - **AWS Fargate (Serverless)**: A serverless, pay-as-you-go service that manages containers without requiring manual EC2 configuration. Ideal for small, batch, or burst workloads.
  - **EC2 Instances**: Manually configured for consistent workloads with high resource demands.
  - **ECS Anywhere**: Allows manual configuration of EC2 instances across various locations for running containers anywhere.

3. Enabling Monitoring for ECS

- **CloudWatch Monitoring**:
  - Enable monitoring to track container metrics such as **CPU**, **memory**, **disk**, and **network**.
  - **Container Insights**: Provides diagnostic information like container restart failures.
  - **CloudWatch Alarms**: You can set alarms on the collected metrics to proactively manage issues.

4. Creating ECS Task Definition

- After creating a cluster, the next step is creating a **task definition**.
- **Task Definition Parameters**:
  - **Task Name**: Provide a name for the task definition.
  - **Container Name** and **Image URI**: Specify the Docker image URI from **Elastic Container Registry (ECR)**.
  - **Container Port**: Set the port as `8080` and protocol as `TCP`.

```
Task Definition:
- Container Name: my-container
- Image URI: <your-ECR-image-URI>
- Port: 8080
- Protocol: TCP
```

- **Environment & Infrastructure Setup**:
  - Assign **AWS Fargate** as the infrastructure.
  - Configure environment variables, storage, monitoring, and tags.
  - Set CPU and memory requirements for the task.

5. Creating a Service for Task Deployment

- Once the task definition is created, go to the **Services** tab in ECS to create a service.
- **Service Configuration**:
  - Select the task definition from the dropdown.
  - Set the **service name** and **number of tasks** to run (Desired, Minimum, Maximum).

6. Configuring Load Balancer

- **Load Balancer Setup**:
  - Create an **Application Load Balancer** to distribute traffic across tasks running in the service.
  - Set the **load balancer port** as `80` (listens on port 80) and protocol as `HTTP`.
- **Security Group**:
  - Configure a security group to control inbound traffic.
  - Allow traffic for specific ports, such as `8080` for the application, by adding inbound rules.

```
Inbound Rules:
- Port: 8080
- Protocol: TCP
- Source: <IP address or CIDR>
```

7. Editing Inbound Rules for EC2 Instance

- If hosting an application on port `8080`, update the inbound rules to allow traffic on that port.
- For example, to allow access to port `8080`, add the rule for both **IPv4** and **IPv6**.

8. Load Balancer Listener and Target Group

- **Listener**: Defines the port and protocol that the load balancer listens on (Port `80` for HTTP).
- **Target Group**: The load balancer routes traffic to the tasks based on this group.
  - The **target port** (e.g., `8080`) is where the application is running, while the **load balancer listens on port 80**.

9. Accessing the Application

- Once the service is deployed, you can access the application using:
  - **Load balancer's output service address** or
  - **Task's IP address** on port `8080`.

Summary: This ECS setup involves creating clusters, defining tasks, configuring services, and setting up load balancers to ensure seamless deployment of Dockerized applications using AWS ECS and Jenkins.

DevOps Notes: Docker CICD with Jenkins and AWS ECS

1. Integrating AWS ECS with Jenkins Pipeline

- After creating an **ECS cluster**, tasks, services, and load balancer, the next step is to connect this setup to the **Jenkins pipeline** for automation.
- To enable this, you need to install the **'Pipeline: AWS Steps' plugin** from Jenkins' Plugin Manager.
  - Go to **Manage Jenkins** > **Plugin Manager**, and search for the plugin to install it.

2. Jenkins Pipeline Stages In the Jenkinsfile, the following stages are commonly used in a Docker CI/CD pipeline:

```
pipeline {
    agent any
    stages {
        stage('Fetch Code') {
            steps {
                // Fetch the application code from the repository
            }
        }
        stage('Test') {
            steps {
                // Run tests
            }
        }
        stage('Code Analysis with Checkstyle') {
            steps {
                // Code analysis for code quality
            }
        }
        stage('Build & SonarQube Analysis') {
            steps {
                // Build the application and run SonarQube analysis
            }
        }
        stage('Quality Gate') {
            steps {
                // Check SonarQube Quality Gate results
            }
        }
        stage('Build App Image') {
            steps {
                // Build the Docker image of the application
            }
        }
        stage('Upload App Image') {
            steps {
                // Upload the built Docker image to Amazon ECR
```

```
            }
        }
        stage('Deploy to ECS') {
            steps {
                // Deploy the Docker image to ECS
                sh 'aws ecs update-service --cluster ${cluster} --service ${service} --force
            }
        }
    }
}
```

- **Key Stages**:
    1. **Fetch Code**: Pull the source code from the repository.
    2. **Test**: Run unit tests.
    3. **Code Analysis with Checkstyle**: Perform static code analysis to maintain code quality.
    4. **Build & SonarQube Analysis**: Build the application and run **SonarQube** analysis.
    5. **Quality Gate**: Enforce quality gates based on the SonarQube results.
    6. **Build App Image**: Build the Docker image for the application.
    7. **Upload App Image**: Push the built image to **Amazon ECR**.
    8. **Deploy to ECS**: Update the ECS service using the latest Docker image by forcing a new deployment.
- The key command to update the ECS service is:

    ```
    aws ecs update-service --cluster ${cluster} --service ${service} --force-new-deployment
    ```

This command ensures that the ECS service fetches the latest Docker image, runs a new task, and replaces the old container with the new one.

3. Cleanup ECS

- To delete an ECS cluster, you must follow a specific order:
    1. Set the number of tasks in the service to **0**.
    2. Delete the **ECS service**.
    3. Delete the **ECS cluster**.

ECS clusters cannot be deleted if there are still active tasks running in the service.

4. Build Triggers

- **Build Triggers** automate the Jenkins pipeline execution process. Instead of manually running the build with 'Build Now' or 'Build Now with Parameters', triggers enable automatic execution of the pipeline.
- Triggers can be set up based on events like:

– Changes in the code repository (e.g., a push to Git).
– Time-based schedules (e.g., a nightly build).
– Webhooks from external services.

DevOps Notes: Jenkins Build Triggers & Automation

1. Overview of Jenkins Build Triggers Jenkins build triggers are mechanisms that automate the execution of Jenkins jobs. Triggers help streamline the CI/CD process by automating the pipeline execution based on specific events or schedules.

2. Popular Build Triggers

1. **Git Webhook**
   - Triggers a Jenkins job whenever there's a commit to the Git repository.
   - GitHub sends a JSON payload to Jenkins, initiating the build automatically.
2. **Poll SCM**
   - Jenkins periodically checks the Git repository for new commits based on a predefined interval.
   - If new commits are detected, the job is triggered.

   ```
   Poll SCM schedule: H/5 * * * *
   ```

   The above cron schedule checks every 5 minutes.
3. **Scheduled Jobs**
   - Jenkins jobs are scheduled to run at specific times or intervals, similar to cron jobs.
   - For example, you can schedule a job to run at midnight every day:

   ```
   0 0 * * *
   ```
4. **Remote Triggers**
   - Jobs can be triggered externally through API calls, scripts, or tools like Ansible.
   - Remote triggers often use authentication tokens or secrets to ensure secure access.

   Example: Use a script to trigger the Jenkins job via API.

   ```
   curl -X POST http://jenkins-url/job/job-name/build?token=your-token
   ```
5. **Build After Other Projects Are Built**

- Automatically triggers a job once another job has successfully completed. Useful for multi-step pipelines where one job depends on the output of another.

3. Automating Git Repository Access in Jenkins

Steps to Set Up GitHub Repository and SSH Access:

1. **Create a GitHub Repository**
   - Create a new GitHub repository (e.g., named `jenkinstriggers`).
   - Choose whether the repository should be public or private.
2. **Generate SSH Keys**
   - Generate SSH keys on your terminal to set up passwordless access to your GitHub repository:
     ```
     ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
     ```
   - The public key (`id_rsa.pub`) is added to your GitHub account under **SSH and GPG keys**.
   - The private key (`id_rsa`) will be used in Jenkins for secure access.
3. **Clone the Repository**
   - After adding the public key to GitHub, clone the repository locally using SSH:
     ```
     git clone git@github.com:username/jenkinstriggers.git
     ```
4. **Jenkins Configuration for SSH Access**
   - Go to **Manage Jenkins** > **Configure Global Security**.
   - Under **Git Host Key Verification**, set the option to **Accept First Connection** to avoid SSH fingerprint verification issues.

4. Creating and Configuring a Jenkins Pipeline

1. **Create a Jenkinsfile**
   - Create a `Jenkinsfile` inside your Git repository to define your pipeline stages. An example pipeline for demonstration:
     ```
     pipeline {
         agent any
         stages {
             stage('Build') {
                 steps {
                     sh 'echo "Build completed."'
                 }
             }
         }
     }
     ```
2. **Configure Jenkins Job**
   - In Jenkins, create a new **Pipeline** job.
   - In the job configuration, select **Pipeline script from SCM**.

- Set the **Repository URL** (use the SSH URL) and provide the credentials (**SSH username with private key**).
  - The private key is stored in Jenkins to securely access the Git repository.
  - The public key is added to GitHub for passwordless access.
- Provide the branch name and specify the path to the `Jenkinsfile` (e.g., `Jenkinsfile`).

3. **Save and Test the Job**
   - After setting up the job, save the configuration.
   - You can manually trigger the build or set up one of the popular triggers, such as a Git webhook or poll SCM.

5. Testing Triggers

- After configuring Jenkins to pull the Jenkinsfile from GitHub, test the automation by committing and pushing changes to the repository. `bash git commit -m "Added Jenkinsfile"  git push origin master`
- If the triggers are set correctly (e.g., via GitHub Webhook), Jenkins will automatically run the pipeline.

DevOps Notes: Jenkins Build Triggers Demonstration

1. Adding a GitHub Webhook to Jenkins To trigger Jenkins jobs automatically upon code pushes to a GitHub repository, a webhook needs to be set up.

Steps to Add a GitHub Webhook:

1. **Access Jenkins URL:**
   - Copy the Jenkins instance URL, such as `http://18.221.221.216:8080/`, from the browser.
2. **GitHub Webhook Setup:**
   - In the GitHub repository settings, navigate to **Webhooks**.
   - Set the **Payload URL** as `http://18.221.221.216:8080/github-webhook/`.
   - Set the **Content type** to `application/json`.
   - Choose the **Push** event as the triggering event for the webhook.
3. **Troubleshooting:**
   - If the webhook fails to trigger, check:
     - The Jenkins URL matches the Payload URL.
     - The Jenkins server's security group allows access from the internet (GitHub needs access).

2. Configuring Jenkins to Use the Webhook

1. **Jenkins Job Configuration:**
   - In Jenkins, configure the job to use the **GitHub hook trigger for GITScm polling**.

- This ensures that any code push to the specified repository triggers the Jenkins job.
- Jenkins will auto-build whenever a commit is pushed to the repository.

3. Poll SCM Trigger Instead of waiting for GitHub to send a webhook, Jenkins can periodically poll the repository for new commits.

Setting Up Poll SCM:

1. **Configure Poll SCM:**
   - In Jenkins, set a polling schedule in **cron format** under the **Build triggers** section.
     - Example: Run every minute: `* * * * *`.

   `Schedule: * * * * *`
2. **Logs:**
   - View the **Git polling log** under the build section of the Jenkins dashboard to see when the last poll occurred and if any new commits were found.

4. Comparison of Triggers: Webhook vs. Poll SCM

- **Git Webhook**: GitHub sends a JSON payload to Jenkins as soon as a commit happens.
- **Poll SCM**: Jenkins periodically checks the GitHub repository for new commits, and if found, it triggers the job.

5. Scheduled Jobs Jobs can be configured to run periodically, regardless of repository changes.

Setting a Schedule for Periodic Jobs:

1. In Jenkins, go to **Job Configuration** > **Build triggers**.
2. Select **Build periodically** and set a schedule using cron job syntax.
   - Example: To schedule the job to run Monday through Friday:

   `0 0 * * 1-5`

6. Remote Triggers You can trigger Jenkins jobs remotely using API calls, scripts, or other Jenkins servers.

Setting Up Remote Triggers:

1. **Enable Remote Triggers:**
   - Go to **Job Configuration** > **Build triggers**.
   - Check **Trigger builds remotely** and provide a token name (e.g., `testtoken`).

2. **Generate API Token:**
   - Under your username in Jenkins, navigate to **Configure** > **API Token**.
   - Generate and save the API token for future use.
3. **Generate Jenkins Crumb:**
   - Use `wget` to obtain a Jenkins crumb:

```
wget -q --auth-no-challenge --user username --password password \
--output-document - 'http://JENKINS_IP:8080/crumbIssuer/api/xml?xpath=concat(//crumbReq
```

   - Save the generated crumb for remote triggering.

Example of Triggering a Job Remotely via Curl:

```
curl -I -X POST http://admin:116ce8f1ae914b477d0c74a68ffcc9777c@52.15.216.180:8080/job/vprof
```

7. Job Dependency Trigger If one Jenkins job depends on the successful execution of another job, you can configure a dependency trigger.

Configuring a Dependent Job:

1. **In Job B (dependent job):**
   - Go to **Job Configuration** > **Build triggers**.
   - Check **Build after other projects are built**.
   - Specify **Job A** (the upstream job) as the dependency. After **Job A** completes, **Job B** will automatically start.

These are key Jenkins build triggers to automate pipeline execution, from webhooks and Poll SCM to scheduled jobs and remote API triggers, ensuring efficient CI/CD practices.

DevOps Notes: Variables, JSON, YAML, and Docker

1. Understanding Variables, JSON, and YAML In DevOps, data is often exchanged between tools using **JSON** and **YAML** formats. As a DevOps engineer, it's important to be comfortable with these formats for configuration and data exchange.

Variables in Shell Scripts:

- When using variables in shell scripts, it's important to note how quotes affect variable interpolation:

```
name="Jeevan"
echo "Hello $name"    # Output: Hello Jeevan
echo 'Hello $name'    # Output: Hello $name (no interpolation)
```

2. Introduction to Docker Docker is an open platform for developing, shipping, and running applications. It allows you to package applications in isolated environments called **containers**, which can run simultaneously on a single host without interference.

Key Concepts:

- **Docker Image**: A lightweight, standalone, and executable package that contains everything needed to run an application.
- **Docker Container**: A running instance of a Docker image. Containers provide isolation and security for running applications.
- **Docker Hub**: An official Docker registry where you can find pre-built images.

Docker Workflow:

- **Client-Server Model**: Docker commands are run on the client, and the **Docker Daemon** on the host executes them.
- If an image is not available locally, Docker fetches it from the **Docker Registry**.

3. Docker Installation on Linux To install Docker on a Linux machine (e.g., Ubuntu), follow these steps:

```
sudo apt-get update
sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release -y

# Add Docker's official GPG key
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/k

# Set up the stable repository
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyri
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Install Docker Engine
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io -y
```

For the complete Docker installation documentation, refer to Docker's official documentation.

4. Essential Docker Commands

- **View Images**: List all Docker images stored locally.

  ```
  docker images
  ```

- **View Running Containers**: List all active containers.

  ```
  docker ps
  ```

- **View All Containers**: List containers in any state (running, stopped, or exited).

  ```
  docker ps -a
  ```

- **Remove a Container**: Remove a container by its name.

  ```
  docker rm <container-name>
  ```

- **Remove an Image**: Remove an image by its ID.

  ```
  docker rmi <image-id>
  ```

5. Running Docker Containers with Port Mapping When running a Docker container, port mapping is essential to access services running inside the container from the host machine.

```
docker run --name web01 -d -p 9080:80 nginx
```

- This command runs an **Nginx** container named `web01`, mapping the host's port **9080** to the container's port **80**. Any request sent to `localhost:9080` on the host will be forwarded to port **80** inside the container.

Example of Port Mapping Output:

```
0.0.0.0:9080->80/tcp, :::9080->80/tcp
```

6. Inspecting Docker Containers To find the **IP address** of a running container:

```
docker inspect <container-name>
```

- The IP address can be found under the `IPAddress` field in the output. For example, inside the container, you can access the application via `http://172.17.0.2:80`.
- To access this from the host, use the host machine's IP address and the mapped port:

  ```
  http://192.168.0.127:9080
  ```

In this case, **192.168.0.127** is the host machine's IP, and **9080** is the port mapped to the container's port **80**.

These notes cover essential concepts of variables, JSON, YAML, and Docker, with examples and code snippets to help in understanding and practical implementation.

DevOps Notes: Containers, Microservices, and Docker-Compose

1. Vprofile Project on Containers In this project, **Vagrant** is used to dynamically launch multiple virtual machines (VMs). However, the focus is on using **Docker-Compose** to manage containerized services such as **MySQL**, **Memcached**, **RabbitMQ**, **Tomcat**, and **Nginx**.

Key Docker-Compose Command:

- To launch all the services defined in the `docker-compose.yml` file:

```
docker compose up -d
```

- Ensure that you run this command from the directory where the `docker-compose.yml` file is located.

2. Monolithic vs. Microservices Architecture

Monolithic Architecture:

- A **monolithic application** integrates all features (e.g., User Interface, Chat, Posts, Notifications) into a single codebase, often hosted on a single server.
- Example: A Java application with multiple features running under a single **Tomcat server**. Modifying any feature requires stopping the entire application, leading to potential downtime and code conflict.

Microservices Architecture:

- **Microservices** split the application into small, independent services. Each service focuses on a single feature and communicates with others via an **API Gateway**.
- Services are loosely coupled, allowing each to be developed, deployed, and maintained independently. This architecture reduces the risk of one service impacting another.

Advantages of Microservices:

- **Modularity**: Each service can be built using different technologies.
- **Scalability**: Easier to scale specific services.
- **Independent Deployment**: Teams can work independently on different services.
- **Resilience**: A failure in one service doesn't affect others.

Microservices and Containers: Instead of maintaining separate servers for each microservice, which increases costs, **containers** provide an efficient solution. Containers allow services to run in isolated environments on the same host, significantly reducing infrastructure costs.

3. Microservices with Docker Containers provide a dedicated environment for each microservice without the need for multiple Linux servers. Each service is packaged into a **Docker image**, which, when run, becomes a **Docker container**. This approach preserves modularity while optimizing resource use.

Key Points:

- Containers allow for a lightweight and efficient microservices architecture.
- Each feature of the application is run in a separate container.

4. Microservices Project: Docker-Compose Example Below is a sample **docker-compose.yml** file that demonstrates how multiple services are launched using Docker-Compose. The services include **MySQL**, **Memcached**, **RabbitMQ**, **Tomcat**, and **Nginx**.

```yaml
version: '3.8'
services:
  vprodb:
    image: vprocontainers/vprofiledb
    ports:
      - "3306:3306"
    volumes:
      - vprodbdata:/var/lib/mysql
    environment:
      - MYSQL_ROOT_PASSWORD=vprodbpass

  vprocache01:
    image: memcached
    ports:
      - "11211:11211"

  vpromq01:
    image: rabbitmq
    ports:
      - "15672:15672"
    environment:
      - RABBITMQ_DEFAULT_USER=guest
      - RABBITMQ_DEFAULT_PASS=guest

  vproapp:
    image: vprocontainers/vprofileapp
```

```
    ports:
      - "8080:8080"
    volumes:
      - vproappdata:/usr/local/tomcat/webapps

  vproweb:
    image: vprocontainers/vprofileweb
    ports:
      - "80:80"

volumes:
  vprodbdata: {}
  vproappdata: {}
```

Key Components of the Docker-Compose File:

- **Services**: Each block under `services:` defines a containerized service such as **MySQL**, **Memcached**, **RabbitMQ**, **Tomcat**, or **Nginx**.
- **Ports**: Ports are mapped between the host and the container. For example, **3306** is mapped for MySQL, **8080** for the application, and **80** for the web service.
- **Volumes**: Data persistence is achieved using Docker volumes, such as `vprodbdata` for MySQL and `vproappdata` for Tomcat.

5. Conclusion This example highlights the power of **Docker-Compose** in managing multiple containerized services. By using **containers**, microservices architectures can be efficiently deployed with isolated, dedicated environments, reducing the overhead of managing multiple physical servers.

AWS and Cloud Computing Concepts

1. What is Cloud Computing? Cloud computing allows businesses to use resources such as servers, storage, and applications over the internet, rather than managing them locally. This provides flexibility, scalability, and cost-efficiency.

Key Benefits of Cloud Computing:

- **Elasticity**:
  - You don't need to over-provision resources in anticipation of scaling. You can scale up or down based on real-time needs. Cloud providers offer resources on a **pay-as-you-go** basis, preventing the need to invest in costly hardware.
- **Cost Savings**:
  - By scaling resources only as needed, you can avoid unnecessary costs, maximizing savings.
- **Global Deployment in Minutes**:

  - Cloud services allow you to deploy your infrastructure and applications globally within minutes, improving speed and flexibility in operations.

2. Cloud Service Models Cloud computing services are categorized into three primary models:

a) Infrastructure as a Service (IaaS):

- **IaaS** provides access to the fundamental building blocks of IT infrastructure, including virtual machines, networking, and storage.
- **Advantages**:
  - You get complete control over your infrastructure, allowing for customization and scalability.
  - Familiarity with traditional on-premise infrastructure, making it easier for IT departments to adapt.
- **Example Providers**:
  - Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP).

b) Platform as a Service (PaaS):

- **PaaS** removes the need to manage underlying infrastructure such as hardware, operating systems, or storage, enabling developers to focus on building and managing applications.
- **Advantages**:
  - Efficiency in application development without worrying about infrastructure maintenance, capacity planning, or patching.
  - Infrastructure automatically scales based on the demand for the application.
- **Example Providers**:
  - AWS Elastic Beanstalk, Google App Engine, Heroku.

c) Software as a Service (SaaS):

- **SaaS** provides fully managed software solutions over the internet. The cloud provider manages everything, including infrastructure, application code, and updates.
- **Advantages**:
  - Users only need to focus on how they will use the application, without concern for maintenance or infrastructure management.
- **Examples**:

– Web-based email (e.g., Gmail), CRM platforms (e.g., Salesforce), collaboration tools (e.g., Microsoft 365).

3. Comparison of IaaS, PaaS, and SaaS

Service Model	Control	Focus Area	Examples
IaaS (Infrastructure)	Full control over infrastructure (e.g., configuring network ports, security settings)	Networking, storage, compute	AWS EC2, Azure VMs, Google Compute Engine
PaaS (Platform)	Application-level management, infrastructure auto-scales	Application development	AWS Elastic Beanstalk, Google App Engine, Heroku
SaaS (Software)	No control over underlying infrastructure, focus solely on using the application	End-user application usage	Gmail, Salesforce, Microsoft 365

4. Summary Cloud computing offers three core models—**IaaS**, **PaaS**, and **SaaS**—each offering varying levels of control and flexibility over infrastructure, platform, and software services. These services provide businesses with significant advantages in terms of scalability, cost-efficiency, and global reach, making cloud computing a critical component of modern IT strategy.

AWS EC2 Introduction

1. Tags in AWS EC2 Tags in AWS help you manage and organize your resources efficiently. These are key-value pairs that provide metadata for your resources. For instance, you can use tags to name your EC2 instances or to specify the region they are deployed in.

- **Key**: Descriptive term (e.g., "Name", "Region")
- **Value**: Specific value for the key (e.g., "WebServer", "US-East")

Example:

```
Key: Name     Value: WebServer01
Key: Project  Value: Alpha
```

Tags help in searching and filtering resources.

2. Security Groups Security Groups act as virtual firewalls for your EC2 instances. They control both inbound and outbound traffic. For each security group, you can define rules that allow or deny specific types of traffic based on protocol and IP addresses.

- **Inbound Traffic**: Requests coming into the instance.
- **Outbound Traffic**: Requests going out from the instance.

3. EC2 Login Using Key Pairs To log in to an EC2 instance remotely, you need a key pair, which uses public-key cryptography for secure authentication. This consists of: - **Public Key**: Stored in AWS. - **Private Key**: Stored on your local machine.

You use the private key to authenticate when connecting to your instance via SSH.

4. Launching an EC2 Instance

- Go to **Services > EC2** from the AWS dashboard.
- Select **Launch Instance**.
- Choose an **Amazon Machine Image (AMI)**, which is a template with the operating system, application server, and applications pre-configured for launching an instance.

5. Key Pair Types When creating an EC2 instance, you need to generate a key pair for secure login. The key pair types are: - **RSA**: RSA-encrypted private and public key pair. - **ED25519**: An encryption method not supported for Windows instances.

Key pair file formats: - **.pem**: For OpenSSH. - **.ppk**: For PuTTY.

Store the private key securely, as it will be required to access the instance later.

6. Amazon Machine Images (AMIs) Amazon Machine Images (AMIs) are templates that include an operating system and required software to launch EC2 instances.

Types of AMIs: - **AWS Marketplace AMIs**: These are secure and verified by AWS and trusted third parties. - **Community AMIs**: Open-source AMIs that can be created by anyone and are not verified.

Example:

```
Key: Name          Value: WebServer01
Key: Project       Value: Alpha
```

7. EC2 Instance Types When launching an EC2 instance, you can choose from several instance types based on: - **CPU capacity**. - **Memory**. - **Storage**.

For general purposes, a **t2.micro** instance is free-tier eligible. For compute-intensive workloads, like machine learning tasks, use **C-series** instances.

You can find more details on instance types here: AWS EC2 Instance Types.

8. Network Settings and Security During EC2 instance creation, you can modify the network settings, such as: - **VPC Network**: Choose a virtual private cloud for your instance. - **Subnet**: If no preference is selected, the default subnet for the region will be used. - **Auto-assign Public IP**: Allocates a public IP for the instance (enabled by default). - **Security Groups**: Customize inbound and outbound traffic rules. By default, SSH traffic is allowed from anywhere, but you can restrict it to specific IPs for enhanced security.

Example:

```
# Example configuration with SSH access restricted
SecurityGroup:
  Allow SSH: MyIP
  Deny: AllOthers
```

9. Summary AWS EC2 offers scalable and customizable virtual machines, allowing you to tailor instances to your needs with specific configurations, security, and management. The use of key pairs, AMIs, security groups, and network settings ensures the security and efficiency of your cloud infrastructure.

AWS EC2 Quick Start Notes

1. Configuring EC2 Storage

- EC2 instances allow configuration of storage options during setup.
- Free tier users are eligible for up to 30 GB of EBS (Elastic Block Store) General Purpose SSD or Magnetic storage.

2. User Data for Initial Setup

- In the "Advanced Details" section, there's a **User Data** field for shell commands to execute automatically after the EC2 instance launches.
- Example of user data for installing and starting the Apache HTTP server:

```
#!/bin/bash
sudo yum install httpd -y
sudo systemctl start httpd
sudo systemctl enable httpd
mkdir /tmp/test1
```

3. Checking EC2 Instance Details

- Navigate to the **Instances** tab to view:
  - Name, Instance ID, State, Type, Status Checks, Alarm Status, Availability Zone, Public IPv4 Address, Private IPv4 Address.
- **Health Checks**: EC2 instances undergo two health checks — hardware and the instance's launch script.

Address Usage:

- **Public IPv4**: Used for remote connections to the instance.
- **Private IPv4**: Used within the subnet for communication.

4. Connecting to an EC2 Instance via SSH Steps to connect: 1. Open an SSH client. 2. Locate the private key file (`web-dev-key.pem`). 3. Change file permissions to ensure the key is not publicly viewable:

```
chmod 400 web-dev-key.pem
```

4. Connect to the instance using its Public DNS:

```
ssh -i "web-dev-key.pem" ec2-user@ec2-3-87-210-157.compute-1.amazonaws.com
```

5. Verifying Apache HTTP Server on EC2

- After connecting via SSH, check the status of the Apache HTTP server using the following command:

  ```
  sudo systemctl status httpd
  ```

  Example output:

  ```
  httpd.service - The Apache HTTP Server
  Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; preset: disabled)
  Active: active (running) since Sat 2023-06-24
  Main PID: 13174 (httpd)
  ```

6. Checking Network Ports

- Use the following command to check which processes are listening on port 80 (HTTP):

  ```
  ss -tunlp | grep 80
  ```

  Explanation:

  - `ss` = Socket statistics.
  - Options:
    - `-t`: Filter by TCP.
    - `-u`: Filter by UDP.
    - `-n`: Show numerical addresses.
    - `-l`: Show listening sockets.
    - `-p`: Show process information.
  - `grep 80`: Searches for port 80, used by HTTP protocol.

7. Modifying Security Groups to Allow HTTP Access

- EC2 instances, by default, may only allow SSH (Port 22) access. To allow HTTP traffic on Port 80:

1. Go to the **Instance** page and select the instance.
2. Under the **Security Groups** tab, modify inbound rules.
3. Add a new rule:
   - **Type**: Custom TCP.
   - **Port**: 80.
   - **Source**: Choose either `My IP` (for personal access) or `Anywhere` (for global access).
4. Save the changes to apply them.

8. Managing EC2 Instance States You can manage the state of your EC2 instance using the following options: 1. **Stop**: Stops the instance but retains its configuration. 2. **Start**: Boots the instance again after stopping. 3. **Reboot**: Restarts the instance. 4. **Hibernate**: Pauses the instance while saving its state to disk. 5. **Terminate**: Completely removes the instance and clears the allocated volume.

These notes summarize the key concepts of launching and managing EC2 instances, along with relevant commands for setup and troubleshooting.

AWS EC2 Advanced Notes

1. EC2 Instance Creation Process

- **Key Steps**:
  1. **Requirement Gathering**
  2. **Key Pairs**
  3. **Security Group**
  4. **Instance Launch**

2. Requirement Gathering for EC2 To create an EC2 instance, the following information is required: - **Operating System (OS)**: Choose OS for the application (e.g., CentOS). - **Size**: Define RAM, CPU, network requirements. - **Storage Size**: Specify disk size (e.g., 10 GB). - **Project Details**: Specify the software and the desired outcome. - **Services/Apps**: Define services (e.g., SSH, HTTP, MySQL) to ensure proper inbound/outbound rules. - **Environment**: Specify environment (Dev, QA, Staging, Prod). - **Login User/Owner**: Control access based on the role of the user.

3. Security Groups

- **Purpose**: Acts as a virtual firewall, controlling traffic to/from instances.
- **Statefulness**: Security groups are stateful, meaning return traffic is automatically allowed.
- **Inbound Rules**: Controls incoming connections. Example for SSH:

```
Type: SSH
Source: My IP (restricts SSH access to specific IP)
```

- **Outbound Rules**: Controls outgoing traffic between the EC2 instance and the internet.

4. Handling Security Groups

- If a security group is attached to resources, it cannot be deleted until all dependencies are removed.
- Security groups can be shared across multiple instances.

5. Key Pairs

- **Definition**: A key pair consists of a private key (used for login) and a public key (injected into the instance).
- Multiple key pairs can be created for different environments to isolate access.

6. Launching Multiple Instances

- You can launch several instances simultaneously, each with unique configurations.
- **Tags**: Tags are helpful for filtering instances globally.

7. Hosting a Website on an EC2 Instance After creating an instance, you can host a website using the following commands:

```
sudo apt update
sudo apt install apache2 wget unzip -y
wget https://www.tooplate.com/zip-templates/2128_tween_agency.zip
unzip 2128_tween_agency.zip
cp -r 2128_tween_agency/# /var/www/html/
systemctl restart apache2
```

- To check the Apache server status:

```
systemctl status apache2
```

8. Accessing Port 80 To allow access to your hosted website via port 80: 1. Modify **Inbound Rules** in the security group: - Type: **Custom TCP** - Port: **80** - Source: **My IP** (or `Anywhere` for global access). 2. Access the website using your public IP.

9. Elastic IP Addresses

- **Public IP Address**: Changes with each instance restart. To maintain a persistent IP, use an **Elastic IP**.
- **Creating an Elastic IP**:

- Go to **Network & Security > Elastic IPs > Create**.
- Allocate the Elastic IP to your instance.

10. Network Interfaces and Volumes

- **Network Interfaces**: Can be created and assigned to an instance from **Network & Security**.
- **Volumes**: Elastic Block Store volumes can be created and attached to instances.

11. Quick Actions on Instances From the **Instance Tab**, you can quickly perform actions like: - **Connect**: Access via SSH. - **Manage Instance State**: Start, stop, or reboot the instance. - **Networking**: View and edit network settings. - **Security**: Modify security groups and key pairs. - **Monitoring & Troubleshooting**: Analyze performance and resolve issues.

12. Cloning Instances

- You can clone an instance to replicate its configuration but not its data.

13. Cost Management

- Release unused resources (e.g., Elastic IPs, volumes) to avoid charges, as AWS operates on a pay-as-you-go basis.

These notes summarize the key steps and best practices for advanced EC2 instance management, including security, networking, and hosting applications.

AWS CLI Notes

1. Introduction to AWS CLI

- AWS resources can be managed through both the **AWS Management Console (GUI)** and **AWS CLI** (Command Line Interface).
- AWS CLI allows programmatic access to AWS services, useful for automation.
- To install AWS CLI on Windows:

```
choco install awscli
```

2. Creating an IAM User for CLI Access

- To use AWS CLI, you need an **IAM user** with proper permissions.
- Steps to create an IAM user:
  1. Go to the **IAM Dashboard** on AWS.
  2. Click **Add User** to create a new user.

3. Assign permissions:
   - **Add User to Group**: Manage permissions based on the job function.
   - **Copy Permissions**: Copy from an existing user.
   - **Attach Policies Directly**: Directly assign policies (best for specific use cases, e.g., admin access).

3. Granting Permissions to IAM Users

- You can assign policies to a user or group, depending on the level of access required.
- For admin-level access, use the **AdministratorAccess** policy. For limited access (e.g., EC2 only), assign specific policies like **AmazonEC2FullAccess**.

4. Generating Access Keys

- After creating the user, navigate to **Security Credentials**.
- Create **Access Keys** to use AWS CLI (these are like a username/password for programmatic access).
- **Important**: Store the access key and secret key safely (e.g., download as a CSV file). Never share them, as they grant admin access.

5. Configuring AWS CLI Once you have the access keys, configure AWS CLI using the `aws configure` command:

```
$ aws configure
AWS Access Key ID [None]: <Your Access Key>
AWS Secret Access Key [None]: <Your Secret Key>
Default region name [None]: us-east-1
Default output format [None]: json
```

- These values are stored in configuration files located at `~/.aws/` (Linux/Mac) or `%UserProfile%\.aws\` (Windows).

```
$ cat ~/.aws/config
[default]
region = us-east-1
output = json
```

```
$ cat ~/.aws/credentials
[default]
aws_access_key_id = <Your Access Key>
aws_secret_access_key = <Your Secret Key>
```

6. Managing Access Keys

- If your keys are compromised, delete or deactivate them instead of deleting the IAM user.

```
aws iam delete-access-key --access-key-id <Your Access Key>
```

7. **Basic AWS CLI Commands**
   - Get current user identity (User ID and Account ID):

     ```
     aws sts get-caller-identity
     ```

   - Describe EC2 instances in your account:

     ```
     aws ec2 describe-instances
     ```

8. **Useful Resources**
   - Official AWS CLI documentation: AWS CLI Documentation
   - For help with AWS CLI commands, you can also refer to tools like ChatGPT to generate the commands based on your needs.

9. **General Tips**
   - While AWS CLI provides automation and flexibility, the AWS GUI is always available for more intuitive resource management.

These notes cover setting up AWS CLI, creating an IAM user with the appropriate permissions, and executing basic AWS commands for managing resources.

AWS Elastic Block Storage (EBS) Notes

1. **Introduction to Elastic Block Store (EBS)**
   - **EBS** is a virtual hard disk for EC2 instances.
   - EBS is **block-based storage**, where data is stored and retrieved in blocks. It is automatically replicated within the same Availability Zone (AZ) to provide fault tolerance.
   - **Snapshot**: A backup of the EBS volume that can be restored later.

2. **Types of EBS Volumes**
   - **General Purpose (SSD)**: Used for most general workloads, e.g., `gp2` or `gp3`.
   - **Provisioned IOPS (SSD)**: Used for workloads where high Input/Output operations per second (IOPS) are a priority, e.g., large databases.
   - **Throughput Optimized HDD (st1)**: Ideal for processing large datasets, e.g., Big Data applications and data warehouses.
   - **Cold HDD (sc1)**: Suitable for infrequent access, often used in file servers.
   - **Magnetic**: A low-cost option, primarily used for backups and archiving.

3. EBS Volume Performance

- **General Purpose SSD (gp2, gp3)**:
  - Baseline throughput of 128 MiB/s per TiB, burstable to higher speeds depending on volume size and IOPS.
- **Provisioned IOPS SSD (io1, io2)**:
  - Maximum throughput of 1,000 MiB/s per volume.
- **Throughput Optimized HDD (st1)**:
  - Maximum throughput of 500 MiB/s.
- **Cold HDD (sc1)**:
  - Maximum throughput of 250 MiB/s.

4. Monitoring EBS Performance

- Use **Amazon CloudWatch** to monitor and calculate throughput over time.
- **EBS Performance Test Tool** can be used for benchmarking EBS volumes under different workloads.

5. Setting up EBS on EC2 To test EBS on an EC2 instance, follow these steps: 1. Launch a **CentOS EC2** instance. 2. Run the following script in the instance or provide it as **user data**:

```
yum install httpd wget unzip -y
systemctl start httpd
systemctl enable httpd
cd /tmp
wget https://www.tooplate.com/zip-templates/2128_tween_agency.zip
unzip -o 2128_tween_agency.zip
cp -r 2128_tween_agency/* /var/www/html/
systemctl restart httpd
```

- This sets up a basic web server on EC2 using **gp2** (General Purpose SSD) as the default storage.

6. Storage Availability

- Both the instance and its attached volume must be in the same **Availability Zone**. You cannot attach an EBS volume from a different AZ.

7. EBS Partitioning and Disk Information

- List disk details:

  ```
  fdisk -l
  ```

- Example output:

  ```
  Disk /dev/xvda: 8589 MB, 8589934592 bytes, 16777216 sectors
  ```

- To partition a volume, use the following command:

  ```
  fdisk /dev/xvda
  ```

8. Formatting EBS Volumes

- Format a volume with a file system (ext4 or xfs):

  For ext4:

  ```
  sudo mkfs -t ext4 /dev/sdb1
  ```

  For xfs:

  ```
  sudo mkfs -t xfs /dev/sdb1
  ```

- **Note**: Formatting a volume erases all the data on it.

9. Mounting EBS Volumes

- To mount the volume to a directory (e.g., `/var/www/html/images`):

  ```
  mount /dev/xvdf1 /var/www/html/images/
  ```

- To unmount the volume:

  ```
  umount /var/www/html/images/
  ```

- **Persistent Mount**: To make the mount permanent, edit the `/etc/fstab` file and add the following line:

  ```
  /dev/xvdf1 /var/www/html/images ext4 defaults 0 0
  ```

- Apply the changes with:

  ```
  mount -a
  ```

10. EBS in AWS Free Tier

- The AWS free tier provides **up to 30GB** of EBS storage.

These notes cover key concepts of EBS, including volume types, partitioning, formatting, and mounting, as well as practical use cases on EC2 instances.

AWS Elastic Block Store (EBS) Snapshots Notes

1. Managing Folders and Processes in Linux

- If a folder cannot be deleted due to being used by a process, you can use the following command to identify the process using the folder:

  ```
  lsof /var/www/html/images
  ```

- The command shows the process IDs using the folder. You can decide to terminate these processes if they are unnecessary, allowing you to delete the folder.

2. **Managing Unused EBS Volumes**
   - It's important to **delete unused volumes** to avoid paying for them.
   - After attaching a new volume to an EC2 instance, you can verify it with the following command:

     ```
     fdisk -l
     ```

3. **Dedicated Partitions for Applications**
   - For applications like MySQL, it's useful to create dedicated partitions on the EBS volume. For example, mount a partition to `/var/lib/mysql` to provide dedicated storage for MySQL data.

4. **EBS Snapshots: Backup and Restore**
   - **Snapshots** are backups of EBS volumes at a specific point in time.
   - To create a snapshot:
     1. Navigate to the **Volumes** section under Elastic Block Store.
     2. Select the volume.
     3. Choose **Create Snapshot** under the **Actions** tab.
     4. The snapshot will appear under the **Snapshots** section.
   - A snapshot captures all files on the volume until the time of creation, making it easy to restore data if needed.

5. **Restoring Data from a Snapshot**
   - Example: Recovering data from a MySQL database using a snapshot.

     Steps:

     1. Stop the MariaDB service:

        ```
        systemctl stop mariadb
        ```

     2. Unmount the MySQL directory:

        ```
        umount /var/lib/mysql
        ```

     3. Detach the volume in the AWS Console by selecting the volume under **Volumes** and choosing **Detach Volume**.

     4. Go to the **Snapshots** section and create a new volume from the snapshot via **Create Volume** under the **Actions** tab.

     5. Attach the newly created volume to the EC2 instance.

6. Snapshot Capabilities

- **Modify Volume Properties**: Snapshots allow you to change the volume size, switch between **Availability Zones**, or increase storage capacity.
- **Copying Snapshots Across Regions**: You can copy a snapshot to different regions, allowing for cross-region replication of file structures.
- **Creating AMIs from Snapshots**: Snapshots can also be used to create Amazon Machine Images (AMIs), enabling you to launch new instances based on the volume configuration.
- **Public Snapshots**: Snapshots can be made public or shared with other AWS accounts by modifying permissions under the **Actions** tab.

7. Code Snippet for Snapshot-Based Volume Restoration

```
# Stop the MariaDB service
systemctl stop mariadb

# Unmount the MySQL directory
umount /var/lib/mysql

# Detach the volume from the instance (performed in AWS Console)

# Create a new volume from the snapshot (performed in AWS Console)

# Attach the new volume to the instance (performed in AWS Console)
```

By leveraging EBS snapshots, you can effectively manage backups, restore data, migrate across regions, and share resources between AWS accounts.

AWS Elastic Load Balancer (ELB) Notes

1. Overview of Load Balancers in AWS A **load balancer** in AWS automatically distributes incoming traffic across multiple targets such as **EC2 instances**, containers, and IP addresses. Load balancing enhances application scalability, availability, and performance.

Types of AWS Load Balancers: 1. **Application Load Balancer (ALB)** 2. **Network Load Balancer (NLB)** 3. **Classic Load Balancer (CLB)** 4. **Gateway Load Balancer (GWLB)**

---

2. Elastic Load Balancing (ELB) Overview

- **Elastic Load Balancing (ELB)** automatically distributes application or network traffic across multiple **targets** in one or more **Availability Zones**.

- Supports key features like **SSL/TLS termination**, **health checks**, **real-time monitoring**, and **integrated certificate management**.

  Key benefits:

  - Automatically scales to handle changes in incoming traffic.
  - Distributes traffic across targets for better performance and high availability.

3. Frontend and Backend Ports

- **Frontend Ports (Listeners)**: Ports where the load balancer listens for incoming user requests.
- **Backend Ports**: Ports where the applications running on the backend services (such as EC2 instances) listen for traffic.

  Example of a typical setup:

  - **Frontend**: Receives user requests (e.g., HTTP or HTTPS).
  - **Backend**: Routes traffic to services running on EC2 or containers.

---

4. Types of Load Balancers in Detail

Classic Load Balancer (CLB)

- **Works at OSI Layer 4 (Transport Layer)**.
- Routes traffic based on application or network-level information.
- Primarily used for simple load balancing of traffic across EC2 instances.
- **Cheapest option**, ideal for straightforward traffic distribution needs.

Application Load Balancer (ALB)

- **Operates at OSI Layer 7 (Application Layer)**.
- Routes traffic based on advanced application-level information, such as HTTP headers, methods, paths, and query strings.
- Ideal for handling **HTTP/HTTPS requests** and for routing traffic based on URL paths (e.g., directing requests for `www.example.com/images` to a specific set of resources).

  Example setup for ALB:

  - Routes traffic based on path or host-based routing.

```
# ALB setup example to forward different URLs to different targets
Target group 1: www.example.com → Target EC2 Group A
Target group 2: www.example.com/images → Target EC2 Group B
```

Network Load Balancer (NLB)

- **Operates at OSI Layer 4 (Transport Layer)**.
- Designed to handle **millions of requests per second** and provides **ultra-high performance**.
- Assigns a **static IP** address, which is useful for applications that require fixed IP endpoints (unlike ALB and CLB which have dynamic IPs).

  Example use case for NLB:

  - High-performance applications requiring fast, consistent network traffic handling.
  - Useful when a static IP is necessary.

Gateway Load Balancer (GWLB)

- Enables the deployment and scaling of **virtual appliances** such as firewalls, intrusion detection/prevention systems, and deep packet inspection.
- Like other load balancers, it distributes traffic across multiple targets in multiple Availability Zones.

---

5. Code Snippet for Checking Health of ELB AWS provides built-in health checks that monitor the health of the registered targets. Below is a command that helps monitor your load balancer using the **AWS CLI**.

```
# Describe the health of the targets in the load balancer target group
aws elbv2 describe-target-health --target-group-arn <target-group-arn>
```

This will return the health status of your targets, enabling automatic removal of unhealthy instances.

---

6. Choosing the Right Load Balancer

- **Application Load Balancer (ALB)**: Best for **HTTP/HTTPS traffic**, web traffic, and advanced routing.
- **Network Load Balancer (NLB)**: Ideal for high-performance, low-latency applications with a need for **static IP** addresses.
- **Classic Load Balancer (CLB)**: Suitable for **simple traffic balancing** and backward compatibility.
- **Gateway Load Balancer (GWLB)**: Perfect for **virtual network appliances** such as firewalls and intrusion prevention systems.

By understanding the various types of load balancers and their use cases, you can select the most appropriate one based on your application's architecture, performance requirements, and security needs.

AWS Elastic Load Balancer (ELB) Hands-On

This section provides a hands-on approach to setting up an **EC2 instance** and configuring a simple website that can be accessed via the instance's IP address. The steps include setting up Apache (HTTPD) or NGINX, depending on the Linux distribution (CentOS or Ubuntu), downloading a web template, and deploying it to the web server.

1. Bash Script for EC2 Instance Setup The following script will launch and configure an EC2 instance for testing the ELB (Elastic Load Balancer) storage by deploying a web application on either **CentOS** or **Ubuntu**. The script installs necessary packages, deploys a website, and starts the web server.

```
#!/bin/bash

# Variable Declaration
PACKAGE=""
SVC=""
URL='https://www.tooplate.com/zip-templates/2098_health.zip'
ART_NAME='2098_health'
TEMPDIR="/tmp/webfiles"

# Check if running on CentOS or Ubuntu
yum --help &> /dev/null

if [ $? -eq 0 ]
then
    # Set Variables for CentOS
    PACKAGE="httpd wget unzip"
    SVC="httpd"

    echo "Running Setup on CentOS"
    # Installing Dependencies
    echo "Installing packages."
    sudo yum install $PACKAGE -y > /dev/null
    echo

    # Start & Enable Service
    echo "Start & Enable HTTPD Service"
    sudo systemctl start $SVC
    sudo systemctl enable $SVC
    echo

    # Creating Temporary Directory
    echo "Starting Artifact Deployment"
    mkdir -p $TEMPDIR
```

```
    cd $TEMPDIR
    echo

    # Download and Extract Web Template
    wget $URL > /dev/null
    unzip $ART_NAME.zip > /dev/null
    sudo cp -r $ART_NAME/* /var/www/html/
    echo

    # Restart HTTPD Service
    echo "Restarting HTTPD service"
    sudo systemctl restart $SVC
    echo

    # Clean Up Temporary Files
    echo "Removing Temporary Files"
    rm -rf $TEMPDIR
    echo

    # Check Status of the Service
    sudo systemctl status $SVC
    ls /var/www/html/

else
    # Set Variables for Ubuntu
    PACKAGE="apache2 wget unzip"
    SVC="apache2"

    echo "Running Setup on Ubuntu"
    # Installing Dependencies
    echo "Installing packages."
    sudo apt update
    sudo apt install $PACKAGE -y > /dev/null
    echo

    # Start & Enable Service
    echo "Start & Enable Apache2 Service"
    sudo systemctl start $SVC
    sudo systemctl enable $SVC
    echo

    # Creating Temporary Directory
    echo "Starting Artifact Deployment"
    mkdir -p $TEMPDIR
    cd $TEMPDIR
    echo
```

```
    # Download and Extract Web Template
    wget $URL > /dev/null
    unzip $ART_NAME.zip > /dev/null
    sudo cp -r $ART_NAME/* /var/www/html/
    echo

    # Restart Apache2 Service
    echo "Restarting Apache2 service"
    sudo systemctl restart $SVC
    echo

    # Clean Up Temporary Files
    echo "Removing Temporary Files"
    rm -rf $TEMPDIR
    echo

    # Check Status of the Service
    sudo systemctl status $SVC
    ls /var/www/html/

fi
```

---

2. Steps Explained

1. **Variable Declaration:**
   - Variables like `PACKAGE`, `SVC`, `URL`, and `ART_NAME` are defined for different distributions (CentOS vs Ubuntu) and the URL of the web template to be downloaded.
2. **Dependency Installation:**
   - For **CentOS**, it installs `httpd` (Apache) and other required packages (`wget`, `unzip`).
   - For **Ubuntu**, it installs `apache2` and the required packages.
3. **Starting the Web Server:**
   - Once the packages are installed, the web server (`httpd` or `apache2`) is started and enabled to run on boot.
   - This is done using `systemctl` commands:
     `sudo systemctl start httpd`
     `sudo systemctl enable httpd`
4. **Artifact Deployment:**
   - The web template is downloaded from the given URL and unzipped in a temporary directory.
   - The extracted files are copied to the default web directory `/var/www/html/`.

5. **Service Restart and Cleanup:**
   - The web service is restarted after deployment to reflect changes: `sudo systemctl restart httpd`
   - Temporary files are removed, and the script checks the status of the web service.

---

3. Accessing the Deployed Website Once the EC2 instance is running and the script is executed, the website will be available at the following URL:

`http://<EC2-IP>:8080`

To troubleshoot if the website is not working: 1. SSH into the EC2 instance. 2. Check the status of the Apache or httpd service using: `bash sudo systemctl status httpd # For CentOS sudo systemctl status apache2 # For Ubuntu` 3. If the services are not running, manually install them and re-execute the script.

AWS Elastic Load Balancer (ELB) Hands-On

This section details the process of setting up Elastic Load Balancers (ELBs) on AWS, working with EC2 instances, and using target groups for health checks. It also includes instructions on creating AMIs (Amazon Machine Images), automating image creation with EC2 Image Builder, and using launch templates for instance deployment.

---

1. Creating an AMI from an EC2 Instance

To scale up your application, you can create multiple instances of an EC2 instance by creating an **Amazon Machine Image (AMI)**.

- **Steps to create an AMI:**
  1. Go to the **EC2 Instances** page.
  2. Select the instance and choose **Create Image** from the instance actions.
  3. The image will be listed under the **AMI** section.
- **Options for AMIs:**
  - You can **copy AMIs** from one region to another.
  - **Edit Permissions**: You can make the AMI public, private, or share it with specific AWS accounts.

---

2. Automating AMI Creation with EC2 Image Builder

You can automate the process of AMI creation by using **EC2 Image Builder**. This allows you to create a **pipeline** for AMI creation, which automates the

entire process.

- Steps:
  - Set up a pipeline to build an AMI based on a specified EC2 instance configuration.

---

3. Using Launch Templates

Creating **Launch Templates** saves time by pre-filling fields that are commonly used during EC2 instance launches. Instead of filling out all the details every time, you can use these templates to launch new instances quickly.

- **Steps to create a Launch Template:**
  1. Go to **Launch Templates**.
  2. Save the configurations like instance type, security group, etc.
  3. When launching an instance, select the **Launch Instance from Template** option and modify the necessary fields.

---

4. Load Balancers and Target Groups

When launching multiple instances of your application, instead of interacting with each instance individually, you can use a **Load Balancer** to distribute traffic between instances.

Target Group: A **Target Group** allows you to define a group of EC2 instances and perform **health checks** to ensure only healthy instances receive traffic.

- **Health Checks:**
  - **Protocol:** Select the protocol (HTTP/HTTPS) to perform the health check.
  - **Port:** Define the port (e.g., `80` or `8080`).
  - **Health Check Path:** This is the URL path to perform the health check.
    - Example: `http://<IP_Address>:<Port>/<URL-Path>`
    - Only the `<URL-Path>` is needed in the health check configuration.
- **Health Check Parameters:**
  - **Healthy Threshold:** Number of consecutive successes required for an instance to be considered healthy.
  - **Unhealthy Threshold:** Number of consecutive failures required for an instance to be marked unhealthy.
  - **Timeout:** Time, in seconds, to wait for a response.
  - **Interval:** Time between health checks.
  - **Success Codes:** HTTP success codes, such as `200`, to indicate a healthy instance.

---

5. Creating a Load Balancer

Once your target group is set up, you can create a **Load Balancer** to distribute traffic among your EC2 instances.

Types of Load Balancers:

1. **Application Load Balancer** (Layer 7 - HTTP/HTTPS)
2. **Network Load Balancer** (Layer 4 - TCP/UDP)
3. **Gateway Load Balancer** (for virtual appliances like firewalls)

Steps to Create a Load Balancer:

1. Go to the **Load Balancer** section.
2. Choose **Create Load Balancer**.
3. Select the type of Load Balancer (e.g., **Application Load Balancer** for HTTP traffic).
4. Provide basic details:
   - **Load Balancer Name**
   - **Scheme:**
     - **Internet-Facing:** Routes requests over the internet to targets in public subnets.
     - **Internal:** Routes requests within a VPC using private IP addresses.
5. Attach the **Target Group** created earlier to perform health checks on the instances.

---

6. Example Load Balancer Setup

Here's an example of how to configure a Load Balancer for your application:

- **Name:** MyApp-ALB
- **Scheme:** Internet-facing
- **Target Group:** MyApp-TargetGroup
  - **Protocol:** HTTP
  - **Port:** 80
  - **Health Check Path:** `/healthcheck`
  - **Healthy Threshold:** 3
  - **Unhealthy Threshold:** 2
  - **Timeout:** 5 seconds
  - **Interval:** 30 seconds
  - **Success Codes:** `200-299`

This setup ensures that only healthy instances receive user traffic and improves the availability and fault tolerance of your application.

AWS Elastic Load Balancer (ELB) Overview

Elastic Load Balancer (ELB) is a service that distributes incoming traffic across multiple EC2 instances, enhancing the scalability, availability, and fault tolerance of applications. ELBs support two main types of load balancers: **Internet-facing** and **Internal**.

---

1. Types of Load Balancers

Internet-facing Load Balancer

- **Public IP Address**: Can receive requests from clients over the internet.
- **Use Case**: Suitable for applications like web servers or e-commerce sites that need to serve external users.

Internal Load Balancer

- **Private IP Address**: Only receives requests from within the same VPC or through a VPC-connected network (via VPC Peering, VPN, or Cloud Interconnect).
- **Use Case**: Ideal for internal applications like database servers or microservices.

Key Difference:

- **Internet-facing**: Accessible by anyone over the internet.
- **Internal**: Accessible only by clients within the VPC or connected networks.

---

2. Virtual Private Cloud (VPC) Network

A **VPC network** is a virtual, isolated network in the cloud, customizable to user needs, and it allows you to securely launch and manage cloud resources like EC2 instances, load balancers, and VPNs.

---

3. Load Balancer Configuration

IP Address Types

- **IPv4 or IPv6**: Choose the type of IP address for the load balancer, based on your traffic requirements.

Availability Zones

- **Mapping to Availability Zones**: Select at least two availability zones to ensure high availability.
- **Subnets**: Ensure that subnets are selected for the availability zones, as traffic will only be routed to targets within the selected zones.

Security Groups

- **Security Group Configuration**: Define the security group for the load balancer with rules for **inbound** and **outbound** traffic.
  - **Inbound Rule Example**: Allow traffic on port 80 for HTTP from **any** source (IPv4 and IPv6), making your application accessible to all users.

```
# Example Inbound Rule for Security Group (Port 80)
Inbound Rule:
- Type: HTTP
- Protocol: TCP
- Port Range: 80
- Source: 0.0.0.0/0 (IPv4) or ::/0 (IPv6)
```

---

4. Listeners and Routing

Listeners Listeners define how incoming requests are routed to the target group. For an HTTP listener on port 80, set up routing to direct traffic to the target group.

Target Group

- The **target group** contains the EC2 instances behind the load balancer.
- Ensure that the load balancer forwards requests received on port 80 to the instances in the target group for health checks and request handling.

```
# Listener Example
Listener:
- Protocol: HTTP
- Port: 80
Target Group:
- Protocol: HTTP
- Port: 80
- Health Check Path: / (or /healthcheck)
```

---

5. Troubleshooting and Health Checks

Health Check Failures If you're unable to access the website hosted behind the load balancer, check if: 1. The individual instances in the target group are **healthy**. 2. **Security Group Rules**: Ensure that the instances' security group allows inbound traffic on port 80 from the load balancer's security group.

```
# Security Group Rule for Instances
Inbound Rule:
- Type: HTTP
- Protocol: TCP
- Port Range: 80
- Source: Load Balancer Security Group
```

---

6. Load Balancer Maintenance

Deregistering Instances for Maintenance When performing maintenance on instances: 1. **Deregister the instance** from the target group to prevent it from receiving new traffic. 2. The load balancer will drain existing requests before removing the instance. 3. Once maintenance is complete, **re-register the instance** to bring it back online.

```
# Deregister Instance
aws elbv2 deregister-targets --target-group-arn <target-group-arn> --targets Id=<instance-id

# Re-register Instance
aws elbv2 register-targets --target-group-arn <target-group-arn> --targets Id=<instance-id>
```

- Only **healthy instances** will receive traffic during maintenance periods.
- You can monitor the health of instances from the **Target Group** section in the Load Balancer dashboard.

Summary

Elastic Load Balancers are essential for distributing traffic across EC2 instances, ensuring scalability and availability. Whether for external applications or internal microservices, understanding how to configure and manage both **internet-facing** and **internal load balancers**, alongside security rules and health checks, is key to maintaining a robust cloud architecture.

AWS CloudWatch Overview

AWS CloudWatch is a monitoring and management service that provides visibility into the performance and health of your AWS resources and applications. It allows you to collect metrics, create alarms, trigger automated actions, and

visualize data using dashboards. CloudWatch helps ensure your systems operate efficiently and within desired thresholds.

Key CloudWatch Components

1. CloudWatch Metrics CloudWatch allows you to monitor the utilization and status of AWS resources like EC2, RDS, S3, and more by default. You can also define **custom metrics** for your specific applications.

Example code to publish a custom metric using AWS SDK for Python (Boto3):

```
import boto3

# Create CloudWatch client
cloudwatch = boto3.client('cloudwatch')

# Put custom metric data
response = cloudwatch.put_metric_data(
    MetricData=[
        {
            'MetricName': 'PageViews',
            'Dimensions': [
                {
                    'Name': 'PageName',
                    'Value': 'Homepage'
                },
            ],
            'Unit': 'Count',
            'Value': 100.0
        },
    ],
    Namespace='MyCustomNamespace'
)

print("Metric submitted: ", response)
```

2. CloudWatch Alarms Alarms notify you when a specified threshold is crossed for a metric, and can trigger actions like sending notifications via **Amazon SNS**.

Example of creating a CloudWatch Alarm using Boto3:

```
cloudwatch.put_metric_alarm(
    AlarmName='HighCPUAlarm',
    ComparisonOperator='GreaterThanThreshold',
    EvaluationPeriods=1,
```

```
    MetricName='CPUUtilization',
    Namespace='AWS/EC2',
    Period=300,
    Statistic='Average',
    Threshold=80.0,
    ActionsEnabled=True,
    AlarmActions=['arn:aws:sns:REGION:ACCOUNT_ID:MyTopic'],
    AlarmDescription='Alarm when server CPU exceeds 80%',
    Dimensions=[
        {
            'Name': 'InstanceId',
            'Value': 'i-0123456789abcdef0'
        },
    ],
    Unit='Seconds'
)
```

3. CloudWatch Events CloudWatch Events deliver near real-time events from AWS services (e.g., EC2, Lambda). You can set up rules that trigger actions when an event occurs. For example, triggering a Lambda function when an EC2 instance state changes.

Example of setting up a CloudWatch Event rule:

```
{
  "source": ["aws.ec2"],
  "detail-type": ["EC2 Instance State-change Notification"],
  "detail": {
    "state": ["stopped"]
  }
}
```

4. CloudWatch Logs CloudWatch Logs allow you to store, access, and monitor log data from AWS resources such as EC2, Lambda, and CloudTrail. To stream logs from an EC2 instance, you need to install and configure the **CloudWatch Logs Agent**.

Sample command to install the CloudWatch Logs agent on an EC2 instance:

```
sudo yum install -y awslogs
sudo service awslogs start
sudo chkconfig awslogs on
```

You will also need to configure the `/etc/awslogs/awslogs.conf` file to define which log files to monitor.

5. CloudWatch Dashboards CloudWatch Dashboards offer a customizable visual display of your metrics, alarms, and logs. These dashboards can be shared

across teams and embedded into other applications or websites.

Use Case Example: Automating EBS Storage Expansion

Using CloudWatch Metrics, you can monitor the utilization of an Amazon EBS volume. When the storage usage exceeds a certain threshold (e.g., 75%), CloudWatch can trigger an Amazon SNS notification, which could notify an admin or automatically initiate an action (such as increasing EBS volume size).

Steps:

1. **Set up a CloudWatch metric** to monitor EBS storage utilization.
2. **Create a CloudWatch Alarm** to trigger when usage exceeds 75%.
3. **Set up an Amazon SNS topic** to send a notification or trigger an automated action, such as extending the storage.

AWS CloudWatch Hands-On

1. Monitoring Key Metrics in CloudWatch CloudWatch automatically tracks several key metrics for AWS resources, particularly for EC2 instances. Here are some metrics you can monitor: - **CPU Utilization** - **Network In/Out (Bytes)** - **Network Packets In/Out (Count)** - **Disk Reads/Writes (Bytes & Operations)** - **CPU Credit Usage/Balance**

If you need to monitor additional metrics, like **RAM Utilization** or **Storage Utilization**, you'll need to create **custom metrics** in CloudWatch.

By default, CloudWatch updates these metrics every 5 minutes. For more detailed monitoring, you can enable **Detailed Monitoring**, which updates every minute. While **basic monitoring** is free, **detailed monitoring** is a paid service with a low cost.

2. Installing Stress Library on EC2 for Load Testing The **stress** tool is useful for simulating high CPU, memory, disk, or network usage on an EC2 instance. This helps test application performance under load. To install the stress tool on an EC2 instance (Amazon Linux), use the following commands:

```
# Install EPEL repository
sudo amazon-linux-extras install epel -y

# Install stress tool
sudo yum install stress -y
```

The **stress** tool is used to test system resilience by generating different types of loads, which helps you observe system performance.

3. Using Stress Tool with AWS Fault Injection Simulator (FIS) AWS Fault Injection Simulator (FIS) can run fault injection experiments, including **CPU stress**, **memory stress**, **disk stress**, and **network stress** on EC2 instances. FIS helps simulate failures and monitor the resilience and observability of your systems.

FIS allows you to define: - **Stop conditions**: To control the safety of experiments. - **Rollbacks**: To restore the system to its normal state after the experiment.

4. Generating Stress on EC2 Instances To simulate a high CPU load on your EC2 instance, you can use the following command to create 4 fake processes for 300 seconds:

```
nohup stress -c 4 -t 300 &
```

This command runs 4 CPU-intensive tasks for 300 seconds in the background, simulating a CPU load on the instance. You can check this using the `top` command on your instance.

5. Advanced Stress Simulation on EC2 To simulate varying levels of stress on an EC2 instance, you can combine **sleep** and **stress** commands to create different load conditions:

```
sleep 60 && stress -c 4 -t 60 && sleep 60 && stress -c 4 -t 30 && sleep 30 && stress -c 4 -t
```

This command alternates between periods of stress and inactivity, helping you observe different load patterns in CloudWatch.

6. Creating Alarms in CloudWatch You can set up CloudWatch alarms to monitor specific metrics and trigger actions when a threshold is crossed. For example, you can create an alarm for **CPU Utilization** and set it to trigger an email notification when CPU usage exceeds a certain percentage.

Steps to create an alarm: 1. Go to the **CloudWatch dashboard**. 2. Select your EC2 instance and choose the metric (e.g., **CPU Utilization**). 3. Set a threshold and configure your email or SNS to receive notifications. 4. Optional: Under **EC2 actions**, choose whether to **Stop**, **Terminate**, **Reboot**, or **Auto Scale** the instance when the alarm is triggered.

AWS Elastic File System (EFS) Overview

1. Introduction to EFS Amazon Elastic File System (EFS) is a scalable, elastic, and fully managed **NFS file system** for use with AWS services and on-premises resources. It supports **shared storage** across multiple EC2 instances, making it ideal for scenarios where data needs to be centralized for multiple servers.

Key features: - **Scalable storage**: Automatically grows and shrinks based on the files stored. - **Two storage classes**: - **Standard**: For frequently accessed files. - **Infrequent Access (IA)**: Cost-optimized for files not accessed frequently.

EFS supports shared storage, unlike **EBS**, which can only be attached to one instance at a time.

2. EFS Use Cases Some common use cases of AWS EFS include: - **Web Content Serving**: EFS can store and serve web content (e.g., images, videos) while providing high availability. - **Enterprise Applications**: EFS can serve as a shared file system for enterprise workloads. - **Media Processing**: Useful for storing and processing large media files, such as videos, for tasks like transcoding and rendering. - **Shared Directories**: EFS is suitable for creating shared or home directories across multiple EC2 instances. - **Database Backups**: Ideal for storing backups of databases and restoring them quickly when needed. - **Development Tools**: Store and share code, configuration files, and logs for development environments.

3. Setting Up EFS

a. Security Group for EFS When setting up EFS, you must configure a security group with the following rules: - **Inbound Rule**: - Type: **NFS**. - Source: Custom IP, which should accept requests from the security group of the EC2 instance (where the website is hosted).

b. EFS Performance Modes

- **General Purpose**: Free under the free tier, suitable for most use cases.
- **Max I/O**: Paid feature, useful for high-performance needs.

c. EFS Throughput Modes

- **Bursting**: Free and works based on a burst credit system.
- **Provisioned**: Paid option for consistent, high throughput.

d. Mount Targets Assign the security group created for EFS to all **availability zones**. This ensures that the EC2 instances can communicate with EFS using the correct security groups.

4. Accessing EFS from EC2

a. Installing EFS Utilities To access EFS from an Amazon Linux EC2 instance, you need to install the `amazon-efs-utils` package:

```
sudo yum install -y amazon-efs-utils
```

For other Linux distributions: - **CentOS**: `bash  sudo yum -y install git  git clone https://github.com/aws/efs-utils  sudo yum -y install make rpm-build  sudo make rpm  sudo yum -y install ./build/amazon-efs-utils*rpm`

- **Debian**:

```
sudo apt-get -y install binutils
./build-deb.sh
sudo apt-get -y install ./build/amazon-efs-utils*deb
```

b. Creating Mount Point To create a mount point from the EC2 instance to EFS, add the following entry to the `/etc/fstab` file:

```
file-system-id efs-mount-point efs _netdev,tls,accesspoint=access-point-id 0 0
```

Replace: - `file-system-id`: The ID of the file system, obtained from EFS details. - `access-point-id`: The ID of the access point, obtained from the access point details.

Example:

```
fs-47a7ccb2 /var/www/html/img efs _netdev,tls,accesspoint=fsap-03f6334520365d2d7 0 0
```

This ensures that the mount point is automatically created when the EC2 instance reboots.

5. Troubleshooting

- **Timeout Errors**: Check the security groups if you encounter timeouts.
- **File System Errors**: Verify the correctness of the file system ID and access point ID.

6. Creating an AMI with EFS Setup You can create an **AMI** from an EC2 instance that has EFS storage enabled. This preserves the configuration of security groups and instances, making it easier to replicate the setup without reconfiguring all the settings.

By creating an AMI, you ensure that all security group settings and EFS configurations are stored and can be reused without repeating the setup.

AWS Auto Scaling Group (ASG) Overview

1. Introduction to Auto Scaling AWS Auto Scaling is a service that automatically adjusts the number of EC2 instances to maintain performance and optimize costs for applications. Auto Scaling works closely with **CloudWatch Alarms** to monitor metrics and dynamically upscale or downscale instances based on demand.

Key Benefits: - **Upscaling**: Automatically adds instances when demand increases to ensure high availability. - **Downscaling**: Removes instances during low demand to reduce costs.

Auto Scaling utilizes a **Launch Configuration/Template** to create EC2 instances, and scaling policies determine when to increase or decrease the number of instances.

2. Key Terminologies

- **Minimum Size**: The minimum number of instances that must always be running.
- **Desired Capacity**: The optimal number of instances to run based on expected load.
- **Maximum Size**: The maximum number of instances that Auto Scaling can launch to handle traffic spikes, ensuring cost control.

3. Hands-On: Setting Up Auto Scaling

a. Pre-requisites

1. **Create a Target Group**: This group allows requests from the **Load Balancer**. Add the security group of the load balancer to the inbound rules of the target group.
2. **Create a Load Balancer**: Ensure that the load balancer is set up to distribute traffic to the instances in the Auto Scaling Group (ASG).
3. **Create an AMI**: Use an AMI (Amazon Machine Image) that contains your desired instance configuration.
4. **Launch Template**: Customize a launch template that includes:
    - Inbound and outbound rules.
    - AMI details.
    - Desired instance configuration.

b. Creating an Auto Scaling Group (ASG) To create an Auto Scaling Group: - Provide the **Auto Scaling Group name**. - Select the **Launch Template** to be used for creating new instances dynamically. - Choose the **Availability Zones** where the ASG should operate. It's recommended to use at least two availability zones for high availability. - Optionally, attach a **Load Balancer** and select the target groups created earlier.

c. Health Check Auto Scaling Groups provide two health check types: - **EC2 Health Check**: Checks the hardware and software state of the EC2 instance. - **ELB Health Check**: Checks the application's state by sending requests via the **Elastic Load Balancer (ELB)** and verifying the response status.

Auto Scaling uses health checks to determine if an instance needs to be replaced. If an instance is deemed unhealthy, Auto Scaling will terminate it and launch a

new one.

d. Group Size Configuration Configure the group size by defining: - **Minimum Capacity**: The least number of instances that must always run. - **Desired Capacity**: The ideal number of instances to meet the load. - **Maximum Capacity**: The upper limit of instances that can be launched.

The relation between these capacities:

```
Minimum Capacity <= Desired Capacity <= Maximum Capacity
```

e. Scaling Policies Scaling policies adjust the number of instances dynamically between the minimum and maximum capacity. Notifications can be set up for events like instance launches, terminations, or failures.

4. Instance Management and Protection

a. Instance States In the **Instance Management** section of the Auto Scaling Group, instances can be: - **Detached**: Removed from the ASG. - **Standby**: Temporarily removed from service. - **Inservice**: Actively serving traffic. - **Scale-in Protection**: Instances with this protection cannot be terminated by the ASG during scale-in.

b. Modifying Instances Direct changes to running instances within an Auto Scaling Group are temporary. Any manual changes will be lost when the ASG launches new instances. To make persistent changes: - **Update the Launch Template**: Modify the template and refresh the Auto Scaling Group. This ensures that future instances inherit the changes.

c. Refreshing Instances After updating the launch template, use the **Refresh Instances** option to apply changes across instances. To minimize downtime, set the **minimum healthy percentage** to a value like 90%, which ensures that at least 90% of the instances remain in service during the update.

5. Data Persistence Since EC2 instances in an Auto Scaling Group are dynamic, data stored locally on the instances may be lost during scaling events or refreshes. To ensure data persistence: - **Mount EBS or EFS volumes** to store data outside the instance.

6. Deleting an Auto Scaling Group To delete instances created by an Auto Scaling Group, the group itself must be deleted. Otherwise, new instances will continue to be created based on the scaling policy. Additionally, if a load balancer is attached, it must be removed to complete project cleanup.

Example: Scaling Policy Setup in CLI

```
aws autoscaling put-scaling-policy \
--auto-scaling-group-name my-asg \
--policy-name scale-out \
--scaling-adjustment 1 \
--adjustment-type ChangeInCapacity
```

This creates a scaling policy to increase the capacity by one instance when triggered.

AWS Relational Database Service (RDS) Overview

1. Introduction to RDS Amazon Relational Database Service (RDS) is a managed service that handles the administrative tasks of relational databases. It simplifies tasks like: - **Database Installation** - **Patching** - **Monitoring** - **Performance Tuning** - **Backups** - **Scaling** - **Security** and **Privacy** - **Hardware Upgrades** - **Storage Management**

RDS helps reduce the operational burden by automating these tasks, allowing developers to focus on application development.

2. Key Features of RDS

- **High Availability with Multi-AZ Deployments**: RDS supports Multi-AZ deployments, hosting databases across two availability zones. The primary instance handles traffic, and a secondary instance is kept in sync. If the primary fails, the secondary automatically takes over, ensuring high availability.
- **Effortless Scaling**: Scaling RDS up or down based on demand is seamless. You can adjust the compute and storage capacity with minimal disruption.
- **Read Replicas for Performance**: RDS supports **Read Replicas** to improve performance by offloading read operations from the main database. Read replicas can also be used for disaster recovery, business reporting, or cross-region replication.

3. Read Replicas

- **Purpose**: Read replicas are ideal for offloading read operations, improving performance, and providing additional redundancy.
- **Supported Databases**: MySQL, MariaDB, PostgreSQL, Oracle, SQL Server, and Amazon Aurora.
- **Replication**: The replication process is **asynchronous**, meaning there might be slight delays in data synchronization between the primary database and the read replica.

- **Cross-Region Replication**: Read replicas can be created across different AWS regions for added redundancy and disaster recovery.

Promoting a Read Replica: If needed, you can promote a read replica to become a standalone instance.

Code Snippet for Creating Read Replica:

```
aws rds create-db-instance-read-replica \
    --db-instance-identifier my-read-replica \
    --source-db-instance-identifier my-primary-db
```

4. Basic RDS Architecture Use Case In a typical use case, RDS is private to a **VPC** (Virtual Private Cloud) while an **EC2 instance** hosts a web application. The web application is publicly accessible through an **Internet Gateway**, while RDS remains private, enhancing security. The application interacts with RDS, but external access to the database is restricted.

Architecture Example: - **EC2 (Web Application)** → Public Access - **RDS** → Private in VPC, only accessible via EC2 instance

5. Creating an RDS Instance

a. Database Creation Methods

- **Standard Create**: Configure all options, including availability, security, backups, and maintenance.
- **Easy Create**: Uses AWS-recommended configurations for simplicity. Options can be modified later if necessary.

b. Recommended Database Engine: Amazon Aurora AWS recommends using **Amazon Aurora** for MySQL or PostgreSQL workloads. Aurora offers: - **64TB of Auto-Scaling Storage** - **6-way replication** across three availability zones - **15 low-latency read replicas** - **5x faster than MySQL** and **3x faster than PostgreSQL**

Aurora also supports **serverless architecture**, providing flexibility in handling variable workloads.

c. Configuring the Database When creating an RDS instance, the following settings must be configured: - **Database Engine**: Select from Amazon Aurora, MySQL, MariaDB, PostgreSQL, Oracle, Microsoft SQL Server. For this example, choose **MySQL**. - **Engine Version**: Choose the desired version of the database engine. - **Database Template**: Select from: - **Production**: Optimized for high availability and consistent performance. - **Dev/Test**: For development and testing environments. - **Free Tier**: Suitable for learning and testing purposes under AWS Free Tier.

d. DB Instance Classes Choose the instance class based on your use case: - **Standard Classes** (e.g., m-class) - **Memory Optimized Classes** (e.g., r-class, x-class) - **Burstable Classes** (e.g., t-class, suitable for free tier usage)

e. Instance Identifier and Credentials

- **DB Instance Identifier**: Provide a unique name for your RDS instance in your AWS region.
- **Credentials**: Set the username and password for accessing the database. Optionally, you can use auto-generated passwords.

Code Snippet for Creating an RDS Instance (CLI):

```
aws rds create-db-instance \
    --db-instance-identifier mydbinstance \
    --db-instance-class db.t2.micro \
    --engine mysql \
    --allocated-storage 20 \
    --master-username admin \
    --master-user-password password123
```

6. Best Practices for RDS

- **Security**: Keep RDS private within the VPC. Use IAM roles and security groups to manage access.
- **Backups**: Enable automated backups to ensure data recovery.
- **Performance**: Use read replicas to optimize read-heavy workloads.
- **Monitoring**: Use Amazon CloudWatch to monitor performance metrics and set up alarms.

7. Conclusion Amazon RDS simplifies database management by handling administrative tasks like patching, backups, and scaling. With high availability, read replicas, and seamless scaling, it is a reliable solution for managing relational databases in the cloud.

AWS RDS (Relational Database Service) Overview

---

1. Storage Types in RDS When setting up Amazon RDS, you have three storage options to choose from:

- **General Purpose (SSD)**: Standard storage option suitable for a wide range of workloads.
- **Provisioned IOPS (SSD)**: Optimized for I/O-intensive workloads, offering high-performance storage.
- **Magnetic**: Legacy option, now mostly replaced by SSD options.

You can also configure **Storage Autoscaling** to dynamically adjust the storage size based on demand. It's important to set an upper limit to control costs.

---

2. Multi-AZ Deployment for High Availability **Multi-AZ Deployment** ensures high availability, durability, and fault tolerance by replicating data synchronously across multiple Availability Zones (AZs). It automatically fails over to a standby instance in case of primary instance failure, minimizing downtime and avoiding data loss. Multi-AZ can be enabled via:

- **AWS Management Console**
- **AWS CLI**
- **RDS API**

Supported database engines include **MySQL, PostgreSQL, MariaDB, Oracle, and SQL Server**.

```
# Command to create Multi-AZ RDS using AWS CLI
aws rds modify-db-instance \
    --db-instance-identifier mydbinstance \
    --multi-az \
    --apply-immediately
```

Note: Multi-AZ deployments are more expensive due to the additional standby instance and data transfer costs.

---

3. RDS Replicas for Read Throughput RDS supports **Read Replicas** to scale read-intensive workloads. Replicas can be created to offload read operations from the primary instance, improving overall performance.

- **Main RDS instance**: Used for write operations.
- **Read Replicas**: Serve read queries, enhancing throughput.

---

4. Public and Private Subnet Configuration You can set RDS to be accessible via a **Public** or **Private Subnet**:

- **Public Subnet**: Allows access over the internet.
- **Private Subnet**: Restricts access within the **VPC (Virtual Private Cloud)**.

In a private subnet configuration, you'll need to define VPC **Security Groups** to specify inbound and outbound rules for your database.

---

5. RDS Authentication Options RDS allows you to authenticate using:

- **Username and Password**
- **IAM-based Authentication**: Secure access using AWS IAM users.

---

6. Backup and Recovery RDS provides automatic backups with a configurable **Backup Retention Period** (up to 35 days). You can also set up **Backup Windows** to define when backups are initiated.

- After the retention period, backups are deleted.
- You can enable **Cross-Region Backup Replication** to store backups in other regions for disaster recovery.

---

7. Monitoring and Logs RDS can be monitored in real time, with intervals as low as 1 second. Key logs you can export to **Amazon CloudWatch** include:

- **Audit Logs**
- **Error Logs**
- **General Logs**
- **Slow Query Logs**

```
# Exporting RDS logs to CloudWatch using AWS CLI
aws rds modify-db-instance \
    --db-instance-identifier mydbinstance \
    --cloudwatch-logs-export-configuration '{"EnableLogTypes":["audit","error","general","sl
```

---

8. Automatic Updates and Protection

- **Minor Version Upgrades**: Can be enabled to automatically apply updates that don't involve major changes.
- **Deletion Protection**: Helps prevent accidental deletion of RDS instances.

---

9. RDS Costs Key factors contributing to RDS costs include:

- **DB Instance** cost
- **Storage** cost
- **Multi-AZ standby instance** costs

---

10. Connecting to RDS via Command Line You can connect to an RDS instance using the MySQL client via the command line:

```
mysql -h <RDS-endpoint> -u <user-name> -p
```

To troubleshoot connection issues, use **telnet** to verify if the RDS instance is reachable:

```
telnet <RDS-endpoint> <port-num>
```

If connection fails, check the **Security Group** settings and ensure the instance's private IP address is allowed.

---

11. Creating Read Replicas You can create a **Read Replica** from the RDS console or CLI. If you've created a snapshot of your RDS instance, it can be used to restore deleted instances or create new ones.

```
# Command to create a read replica
aws rds create-db-instance-read-replica \
    --db-instance-identifier mydbinstance \
    --source-db-instance-identifier mydbinstance
```

---

12. RDS Automation AWS RDS provides a high level of automation, which reduces the need for a dedicated **Database Administrator (DBA)**. A basic understanding of **AWS** and **SQL** is enough to manage RDS instances effectively.

This summarized version of AWS RDS gives an overview of key concepts, features, and management techniques for relational databases in AWS. The provided command snippets help in using the AWS CLI for setting up and managing RDS instances.

AWS VPC (Virtual Private Cloud) Overview

---

1. Introduction to VPC and Networking Components In traditional data centers, physical networking components like **Switches, Routers, and Firewalls** are used to manage network traffic between different **subnets**. A subnet could be dedicated to a front-end service, while another might be assigned to a back-end service.

AWS introduced **VPC (Virtual Private Cloud)** to allow users to create and manage a virtual network in the cloud, with full control over network architecture.

---

2. What is a VPC? A **VPC (Virtual Private Cloud)** is a logically isolated virtual network on AWS where you can launch AWS resources such as **EC2 instances, RDS databases**, and more. It mimics a traditional **LAN (Local Area Network)** but is created within the AWS cloud, giving you control over IP addressing, subnets, route tables, and security.

- **Custom VPC**: Offers more control over configurations.
- **Default VPC**: Automatically created by AWS for each region.

---

3. VPC Features

- **Full control over network components**: Decide the **IP address range**, subnets, and configure routing and security.
- **Private environment**: Resources inside the VPC are logically isolated from other AWS users, ensuring security.
- **High Availability**: Distribute subnets across multiple **Availability Zones** for better resilience and uptime.

```
# AWS CLI to create a VPC
aws ec2 create-vpc --cidr-block 10.0.0.0/16
```

---

4. IPv4 Addressing in VPC

- **IPv4 addresses** are 32-bit addresses, represented in four octets.
  - Example: `192.168.100.1`
  - Range: `0.0.0.0` to `255.255.255.255`
- **Public IP Addresses**: Used for public-facing services, e.g., websites (`54.86.23.90`).
- **Private IP Addresses**: Used within private networks and not exposed to the public internet.

Private IP Ranges:

- **Class A**: `10.0.0.0 - 10.255.255.255`
- **Class B**: `172.16.0.0 - 172.31.255.255`
- **Class C**: `192.168.0.0 - 192.168.255.255`

---

5. Classes of IP Addresses

1. **Class D** (Multicast Addresses):
   - **Purpose**: Used for multicast communication (one-to-many).

- **Example**: Used in video conferencing, streaming services.
- Cannot be used for regular unicast traffic.

2. **Class E** (Reserved Addresses):
   - **Purpose**: Reserved for future or experimental use.
   - Not used for general communication on the internet.

---

6. Subnet Mask and Network Segmentation

- A **Subnet Mask** is a 32-bit number that divides an IP address into:
  - **Network bits**: Identify the network.
  - **Host bits**: Identify specific devices within the network.

The subnet mask allows you to define smaller networks (**subnets**) within a larger network. This segmentation enhances network management and security.

- Example of Subnet Mask: `255.255.255.0` for a subnet in `192.168.1.0/24`

```
# AWS CLI to create a subnet in a VPC
aws ec2 create-subnet --vpc-id vpc-12345678 --cidr-block 10.0.1.0/24
```

---

7. VPC Use Cases and Benefits

- **Custom Network Architectures**: You can define subnets for different layers of your application (e.g., public for web servers, private for databases).
- **Security and Isolation**: Traffic between subnets can be filtered using **Network ACLs (Access Control Lists)** and **Security Groups**.
- **Cost-Efficiency and Performance**: By reducing the need for a public internet gateway and enabling private connections, VPCs lower data transfer costs and improve performance with low-latency access.

Summary

AWS **VPC** enables you to create a secure, isolated network environment in the cloud where you have full control over IP addressing, subnets, routing, and security. By understanding **IP addressing**, **subnet masks**, and the classes of IP addresses, you can effectively design robust and scalable network architectures for your AWS resources.

AWS VPC: Subnet Masks and CIDR Notation

---

1. Purpose of Subnet Masks **Subnet Masks** are used to divide a large IP network into smaller, manageable subnetworks. They help in organizing and managing IP addresses efficiently within a network and improve overall security and network performance.

Key Functions of Subnet Masks:

- **Network Segmentation**: Splits a larger network into smaller subnets.
- **Efficient IP Allocation**: Assigns IP addresses to different departments or teams.
- **Broadcast Domain Control**: Limits unnecessary broadcast traffic within the subnet.
- **Security**: Isolates different parts of a network for better security.
- **Routing Decisions**: Helps routers decide whether the IP is local or needs to be routed externally.

---

2. CIDR Notation (Classless Inter-Domain Routing) CIDR is a flexible method for IP address allocation and routing, replacing the traditional IP class system (A, B, C). It allows for efficient and granular control over subnet sizes and address assignments.

- **CIDR Notation**: Represents the IP address followed by a slash `/` and a number that specifies the number of bits allocated to the network prefix.
  - Example: `192.168.1.10/24` (24 bits are for the network portion, and the remaining are for hosts).

```
# Example of a CIDR block for a VPC
10.0.0.0/16
```

Advantages of CIDR: - Allows flexible subnet size control. - Aggregates routes, reducing the size of routing tables. - Efficient IP address allocation, slowing down the exhaustion of IPv4 addresses.

---

3. Subnet Mask and Usable IPs Subnet masks determine which portion of the IP address belongs to the network and which part is for host identification.

Examples:

- **192.168.0.174/24** (Subnet Mask: `255.255.255.0`)
  - **Network IP**: `192.168.0.0`
  - **Usable IPs**: `192.168.0.1 - 192.168.0.254`
  - **Broadcast IP**: `192.168.0.255`
  - **Total Usable IPs**: 254
- **172.16.12.36/16** (Subnet Mask: `255.255.0.0`)
  - **Network IP**: `172.16.0.0`

  - **Usable IPs**: `172.16.0.1 - 172.16.255.254`
  - **Broadcast IP**: `172.16.255.255`
  - **Total Usable IPs**: 65534
- **10.23.12.56/8** (Subnet Mask: `255.0.0.0`)
  - **Network IP**: `10.0.0.0`
  - **Usable IPs**: `10.0.0.1 - 10.255.255.254`
  - **Broadcast IP**: `10.255.255.255`
  - **Total Usable IPs**: 16,777,214

---

4. CIDR and Subnet Mask Conversion CIDR notation simplifies subnet mask representation:

CIDR	Subnet Mask	Binary Representation
/8	255.0.0.0	11111111.00000000.00000000.00000000
/16	255.255.0.0	11111111.11111111.00000000.00000000
/24	255.255.255.0	11111111.11111111.11111111.00000000

Example: - **172.20.0.0/16**: - Network: `172.20` - Host Address: `0.0` - Subnet Mask: `255.255.0.0` - Wildcard: `0.0.255.255` (opposite of the subnet mask)

Summary

Understanding **Subnet Masks** and **CIDR Notation** is essential for effectively managing and organizing IP networks. Subnet masks help in breaking a large network into smaller subnets, while CIDR provides flexibility in IP address allocation and routing, making networks more efficient and scalable.

AWS VPC Design & Components

---

1. VPC Overview A **Virtual Private Cloud (VPC)** is the main network container where multiple subnets are created. VPCs are isolated from each other, allowing you to define your own network space within AWS. Subnets within a VPC can be classified into two types:

- **Public Subnets**: Instances within public subnets can access the internet through an **Internet Gateway**.
- **Private Subnets**: Instances in private subnets cannot access the internet directly as they lack public IPs. However, they can communicate through a **NAT Gateway** to access external services when necessary.

---

2. Public and Private Subnets

- **Public Subnets**:
  - Instances in public subnets have public IP addresses and access the internet via the **Internet Gateway (IGW)**.
  - Suitable for services that need to be accessible from the internet, such as web servers.
- **Private Subnets**:
  - Instances in private subnets lack public IP addresses, so they cannot connect to the internet directly.
  - To access the internet for updates or installations, private subnets use a **NAT Gateway** placed in a public subnet.

---

3. Subnet Availability Zones Each subnet is mapped to a specific **Availability Zone (AZ)**, ensuring that the network spans multiple data centers for redundancy. To enhance **high availability**, subnets are distributed across multiple AZs.

```
# Example VPC with 4 Subnets in two Availability Zones
VPC CIDR: 172.20.0.0/16
Public Subnet 1: 172.20.1.0/24 (AZ: us-west-1a)
Public Subnet 2: 172.20.2.0/24 (AZ: us-west-1b)
Private Subnet 1: 172.20.3.0/24 (AZ: us-west-1a)
Private Subnet 2: 172.20.4.0/24 (AZ: us-west-1b)
```

---

4. Internet Gateway and NAT Gateway

- **Internet Gateway (IGW)**:
  - A highly available, horizontally scaled VPC component.
  - It allows instances in public subnets to communicate with the internet.
- **NAT Gateway**:
  - Enables instances in private subnets to connect to the internet or AWS services, but prevents the internet from directly initiating a connection to those instances.
  - Requests from private subnets are routed through the NAT Gateway to the Internet Gateway.

```
# Routing table for Public Subnet
0.0.0.0/0 -> Internet Gateway

# Routing table for Private Subnet
0.0.0.0/0 -> NAT Gateway
```

---

5. Route Tables and Network ACLs Each subnet has an associated **route table** that controls where traffic is directed. **Network ACLs (NACLs)** are applied to control inbound and outbound traffic at the subnet level, similar to how security groups work at the instance level.

- Public subnet route tables forward traffic to the Internet Gateway.
- Private subnet route tables forward traffic to the NAT Gateway.

---

6. High Availability with VPC To ensure high availability, at least one public and private subnet should be deployed in each availability zone. In case of a failure in one zone, traffic can be handled by other zones.

- **Bastion Host**: A bastion host is placed in a public subnet and is used to securely access instances in private subnets that do not have direct access to the internet.

```
# Example Setup
Region: us-west-1
VPC CIDR: 172.20.0.0/16
Public Subnet 1: 172.20.1.0/24
Private Subnet 1: 172.20.3.0/24
Bastion Host in Public Subnet: 172.20.1.10
```

---

7. Default VPC and Custom VPC Each AWS region comes with a **default VPC**, which includes default subnets. These subnets are public, and if traffic is routed through the Internet Gateway, they allow access to the internet.

To check whether a subnet is public or private: - If the route table directs traffic to `0.0.0.0/0` through an Internet Gateway, the subnet is public. - If the traffic is routed to a NAT Gateway, it is a private subnet.

---

8. Architecture for VPC Setup For setting up a VPC architecture, here's an example structure:

- **Region**: `us-west-1`
- **VPC Range**: `172.20.0.0/16`
- **4 Subnets**:
  - `172.20.1.0/24` (Public Subnet in us-west-1a)
  - `172.20.2.0/24` (Public Subnet in us-west-1b)
  - `172.20.3.0/24` (Private Subnet in us-west-1a)
  - `172.20.4.0/24` (Private Subnet in us-west-1b)
- **Internet Gateway**: 1
- **NAT Gateways**: 2 (one per availability zone)

- **Elastic IPs (EIP)**: 2 for NAT Gateways
- **Route Tables**: Separate route tables for public and private subnets
- **Bastion Host**: Used to connect from the public subnet to instances in the private subnet.

To save cost, some setups may use only 1 NAT Gateway and Elastic IP.

Summary

Understanding **VPC architecture** is essential for building scalable, secure, and highly available cloud environments. Public subnets, private subnets, Internet Gateways, and NAT Gateways work together to handle different networking needs, while route tables and NACLs control how traffic flows in and out of the VPC.

AWS VPC Creation and Configuration

---

1. Creating a VPC

- To create a VPC, navigate to the VPC creation window.
- **VPC Only**: This option is used when creating only the VPC without subnets or gateways.
  - Provide a **name** and a **CIDR range** (e.g., `x.x.x.x/x`).
  - Click **Create**.
- **VPC and More**: Use this option to create an entire VPC architecture, including public/private subnets, NAT gateways, and an Internet Gateway.
  - Add tags for the VPC, set **IPv4 CIDR range**, and choose **availability zones**.
  - Define the number of **public and private subnets**.
  - If you need dedicated hardware, set the **tenancy** to "Dedicated" (optional).

```
# Example VPC setup
VPC CIDR: 172.20.0.0/16
Public Subnet 1 CIDR: 172.20.1.0/24
Public Subnet 2 CIDR: 172.20.2.0/24
Private Subnet 1 CIDR: 172.20.3.0/24
Private Subnet 2 CIDR: 172.20.4.0/24
```

---

2. NAT Gateway Options When creating the VPC, you will have options for setting up a NAT Gateway: - **None**: No NAT Gateway. - **In 1 AZ**: One

NAT Gateway in a public subnet for all private subnets. - **1 per AZ**: One NAT Gateway per availability zone for high availability, but it's more costly.

To avoid using a NAT Gateway when accessing **S3** (which can incur costs), use **VPC Endpoints** and configure an **S3 Gateway**. This reduces NAT Gateway charges and improves security.

---

3. Subnets Subnets can be created individually under the **Subnet Menu**: - Provide a **name**, **CIDR block**, and select the **availability zone**. - Subnets should be associated with the main VPC's CIDR block. - Subnets by default have **private IP addresses** and are not internet-accessible unless public.

---

4. Internet Gateway An **Internet Gateway (IGW)** connects your VPC to the internet. - Create an IGW and name it. - The IGW will initially be in a "detached" state. - Attach the IGW to your VPC to allow communication with the internet.

---

5. Route Tables **Route Tables** define the pathways for your subnets to communicate with the internet or NAT Gateways.

- Create a route table and associate it with the VPC.
- Under the route table, **edit subnet associations** to associate the public subnets.
- For public subnets, create a rule that routes traffic with `0.0.0.0/0` to the Internet Gateway.
- For private subnets, route traffic to the NAT Gateway.

```
# Example for public subnet route table
Destination: 0.0.0.0/0 -> Target: Internet Gateway

# Example for private subnet route table
Destination: 0.0.0.0/0 -> Target: NAT Gateway
```

---

6. NAT Gateway Setup

- Allocate a **static IP (Elastic IP)** to the NAT Gateway.
- When creating the NAT Gateway, assign the Elastic IP and select a **public subnet**.
- Private subnets' traffic is routed through the NAT Gateway, which forwards it to the internet.

---

7. VPC Endpoints for S3 To reduce NAT Gateway costs for accessing S3: - Create a **VPC endpoint** for S3, which allows private subnets to directly access S3 without routing through the internet. - Customize the **endpoint policy** for security and access control.

---

8. Bastion Host A **Bastion Host** provides a secure way to access resources in private subnets via SSH: - The Bastion Host resides in a **public subnet** and allows access to instances in private subnets. - Create a **security group** for the Bastion Host, allowing **SSH access** from your IP (`My IP`). - Generate a **key pair** for SSH access.

When launching an EC2 instance in a private subnet, use the **key pair** and **security group** created for the Bastion Host.

```
# Example security group inbound rule for Bastion Host
Type: SSH
Protocol: TCP
Port: 22
Source: My IP
```

---

9. Launching EC2 Instances

- When creating an EC2 instance, select the custom VPC instead of the default VPC.
- Attach the **key pair** and **security group** created earlier.
- After launching, the instance will have a **public IP** if it's in a public subnet, or will be accessible via the Bastion Host if in a private subnet.

```
# Steps for EC2 instance
- Select AMI
- Choose custom VPC
- Assign security group and key pair
- Launch the instance
```

Once the instance is launched, you can SSH into the instance using the public IP or through the Bastion Host if the instance resides in a private subnet.

By following these steps, you can create a well-architected AWS VPC with public and private subnets, route tables, gateways, and secure access using Bastion Hosts and VPC endpoints.

AWS VPC Setup and Architecture

Creating a VPC

- To **create a VPC**, choose the "VPC only" option, provide the **name** and the **CIDR block** (e.g., `x.x.x.x/x`), and then create the VPC.
- If you want to create a complete architecture (public/private subnets, NAT gateway, Internet gateway), select "VPC and more." Provide the **IPv4 CIDR range** and other configurations such as:
  - Tenancy: Default or Dedicated
  - Number of **Availability Zones (AZs)**
  - Number of public/private **subnets**

Example CIDR ranges: - VPC: `172.20.0.0/24` - Public Subnet 1: `172.20.1.0/24` - Private Subnet 1: `172.20.3.0/24`

Internet Gateway (IGW)

- An **Internet Gateway** connects a VPC to the internet.
- After creating an IGW, attach it to your VPC to change its state from *detached* to *attached*.

Subnets

- **Subnets** can be created individually under the VPC by specifying the subnet name, CIDR block, and the availability zone.
- Public subnets require an IGW, while **private subnets** use a **NAT Gateway** to access the internet.

Route Tables

- Create **Route Tables** for each subnet (public/private).
- For **public subnets**, add a route to forward traffic (e.g., `0.0.0.0/0`) to the IGW.
- For **private subnets**, create a route that directs non-local traffic to the **NAT Gateway**.

```
# Example route table entry for public subnets
Destination: 0.0.0.0/0
Target: Internet Gateway
```

NAT Gateway

- A **NAT Gateway** is used to allow instances in private subnets to access the internet while keeping them unreachable from the outside.
- Allocate an **Elastic IP** for the NAT Gateway and assign it to a public subnet.
- Update the **route tables** of the private subnets to route traffic to the NAT Gateway.

```
# Route for private subnets
Destination: 0.0.0.0/0
Target: NAT Gateway
```

Bastion Host Setup

- A **Bastion Host** is a server that provides secure access to instances in a private subnet.
- Create a security group for the Bastion Host and allow inbound **SSH** connections only from your IP.

```
# SSH to bastion host
ssh -i bastion-key.pem ec2-user@<Bastion-IP>
```

Accessing Instances in Private Subnets To access an instance in a private subnet through the Bastion Host: 1. **Copy the key** used to access the private instance from your local machine to the Bastion Host. 2. **SSH into the private instance** using the copied key:

```
# Copy key to Bastion Host
scp -i bastion-key.pem web-key.pem ec2-user@<Bastion-IP>:/home/ec2-user/

# Access private instance from Bastion Host
ssh -i web-key.pem ec2-user@<Private-Instance-IP>
```

Hosting a Website in a Private Subnet

Setting Up EC2 Instance for the Website

- Create an **EC2 instance** in a private subnet and use **Bastion Host** for access.
- Install necessary software packages (Apache, wget, unzip):

```
yum install httpd wget unzip -y
```

- Download a website template from a provider like **tooplate.com** using `wget`, unzip it, and copy the contents to `/var/www/html/`.

```
wget https://www.tooplate.com/zip-templates/2136_kool_form_pack.zip
unzip 2136_kool_form_pack.zip
cp -r 2136_kool_form_pack/* /var/www/html/
```

- Restart Apache to serve the website:

```
systemctl restart httpd
systemctl enable httpd
```

Application Load Balancer (ALB)

- To make the website accessible, set up an **Application Load Balancer** in the public subnet.
- Create a **target group** for the instances in the private subnet and configure the load balancer to forward HTTP requests to the target group.
- Ensure that security groups for both the EC2 instance and the load balancer allow traffic on port 80.

```
# Sample DNS name for accessing the website
vpro-web-elb-401942659.us-west-1.elb.amazonaws.com
```

VPC Peering

- **VPC Peering** enables communication between two VPCs (either in the same or different regions/accounts).
- Ensure that the CIDR blocks of the VPCs do not overlap.
- After creating a **peering connection**, establish routes in the route tables of both VPCs to forward traffic to the peering connection.

```
# Example route for VPC peering
Destination: <VPC2-CIDR>
Target: Peering Connection
```

Here's a structured summary of your DevOps notes on AWS EC2 logs, including headings and code snippets for clarity:

AWS Part 2: EC2 Logs and CloudWatch Integration

1. Overview of EC2 Logs

When hosting a website on an EC2 instance, two main types of logs are generated: - **Access Log**: Located at `/var/log/httpd/access_log`, it records all actions on the website. - **Error Log**: Located at `/var/log/httpd/error_log`, it captures errors encountered during operations.

Importance of Log Management

For public-facing websites, the logs can grow quickly, consuming significant disk space. To manage this effectively, it's essential to archive these logs and store them in a location like an S3 bucket.

2. Archiving Logs

Step 1: Create an Archive

To archive the logs, execute the following commands:

```
# Navigate to the log directory
cd /var/log/httpd

# List existing log files
ls
# Output: access_log error_log
```

```
# Create a compressed archive of the logs
tar czvf wave-web01-httpdlogs19122020.tar.gz *
```

Step 2: Move Archive to a Temporary Folder

After creating the archive, move it to a designated temporary folder for S3 upload:

```
# Create a temporary directory for logs
mkdir /tmp/logs-wave

# Move the archive to the temporary folder
mv wave-web01-httpdlogs19122020.tar.gz /tmp/logs-wave/
```

Step 3: Clear Local Log Files

To free up local disk space, clear the contents of the logs:

```
# Clear the contents of access_log
cat /dev/null > access_log

# Clear the contents of error_log
cat /dev/null > error_log
```

3. Installing and Configuring AWS CLI

To interact with S3 and other AWS services from the terminal, install the AWS Command Line Interface (CLI):

```
# Install AWS CLI
yum install awscli
```

Step 1: Create IAM User

1. Create an IAM user with programmable access.
2. Attach a policy that provides S3 read/write access.

Step 2: Configure AWS CLI

Use the following command to configure the AWS CLI with your Access Key ID and Secret Access Key:

```
# Configure AWS CLI
aws configure
```

Provide the following details when prompted:

```
AWS Access Key ID [None]: <Your_Access_Key_ID>
AWS Secret Access Key [None]: <Your_Secret_Access_Key>
```

```
Default region name [None]: us-east-2
Default output format [None]: json
```

4. Uploading Logs to S3

To copy the archived logs to your S3 bucket, use:

```
# Copy the archive to S3
aws s3 cp /tmp/logs-wave/wave-web01-httpdlogs19122020.tar.gz s3://<your-s3-bucket-name>/
```

Keeping Source and S3 in Sync

To synchronize local files with the S3 bucket, use:

```
# Sync the local folder with the S3 destination
aws s3 sync /tmp/logs-wave/ s3://wave-web-logs-321/
```

This command only copies files that are present on one side but not the other. You can automate this with a cron job to keep the folders in sync.

5. Using CloudWatch Logs

For real-time access to logs without sharing production credentials, utilize AWS CloudWatch Logs. This service helps monitor, analyze, and store logs centrally.

Creating IAM Roles for CloudWatch and S3

1. Create a role with full access to S3 and CloudWatch.
2. Assign this role to the IAM user created for log operations.

Removing Old Credentials

After creating the IAM user and roles, remove the old AWS CLI credentials:

```
# Remove old AWS credentials
rm -rf ~/.aws/credentials
```

6. Installing CloudWatch Agent

To install the AWS CloudWatch agent for log management, execute:

```
# Install AWS CloudWatch agent
yum install awslogs -y
```

This structure provides a clear overview and step-by-step instructions on managing EC2 logs, configuring AWS CLI, and utilizing CloudWatch for monitoring.

Here's a structured summary of your DevOps notes on streamlining EC2 logs to CloudWatch and managing load balancer access logs with S3, including headings and code snippets for clarity:

AWS Part 2: Streamlining EC2 Logs to CloudWatch and S3

1. Streaming Logs to CloudWatch

To send logs to CloudWatch using the `awslogs` agent, you'll need to edit the configuration file located at `/etc/awslogs/awslogs.conf`.

Configuration for Log Streaming

Edit the `/etc/awslogs/awslogs.conf` file and add the following configurations for `messages` and `access_log`:

```
# Log configuration for /var/log/messages
[/var/log/messages]
datetime_format = %b %d %H:%M:%S
file = /var/log/messages
buffer_duration = 5000
log_stream_name = web01-sys-logs
initial_position = start_of_file
log_group_name = wave-web

# Log configuration for /var/log/httpd/access_log
[/var/log/httpd/access_log]
datetime_format = %b %d %H:%M:%S
file = /var/log/httpd/access_log
buffer_duration = 5000
log_stream_name = web01-httpd-access
initial_position = start_of_file
log_group_name = wave-web
```

Configuration Parameters Explained

- **Configuration Name**: Identifies the log file being configured.
- **file**: Path to the log file you want to stream to CloudWatch.
- **log_stream_name**: Unique identifier for the log stream.
- **log_group_name**: Logical grouping for related log streams.

Restarting the AWS Logs Daemon

After making changes, restart and enable the `awslogs` service with the following commands:

```
# Restart the awslogs daemon
systemctl restart awslogsd

# Enable awslogs service to start on boot
systemctl enable awslogsd
```

2. Viewing Logs in CloudWatch

After restarting the service, logs should start streaming to CloudWatch. To view them: 1. Navigate to the **CloudWatch Service** in the AWS Management Console. 2. Check the **Log Groups** section for the group named `wave-web`. 3. Under **Log Streams**, locate `web01-sys-logs` to view the logs created from `access_log`.

3. Additional Configuration

You also need to edit the configuration in `/var/awslogs/etc/awslogs.conf` to ensure proper access to the logs.

```
# Edit the main awslogs configuration file
vi /var/awslogs/etc/awslogs.conf

# Restart the awslogs service
service awslogs restart
```

4. Exporting Logs to S3 and Other Services

You can export streamlined logs to Amazon S3 and integrate with other AWS services like Elasticsearch and Lambda.

Creating Metric Filters

Create metric filters to analyze log content based on specific patterns such as `ERROR`, `INFO`, or `WARNING`. This allows you to trigger alarms based on these metrics.

5. Managing Load Balancer Access Logs

To store load balancer access logs in S3, you must adjust the S3 bucket policy to allow access from the Elastic Load Balancer (ELB).

Steps to Enable Access Logs

1. **Edit S3 Bucket Policy**: Ensure your S3 bucket policy allows the ELB to write logs. Refer to the AWS documentation for detailed instructions.
2. **Enable ELB Access Logs**: After configuring the S3 bucket policy, enable access logging on your ELB and specify the S3 bucket for log storage.

Accessing ELB Logs

Provide the necessary details from the Load Balancer to access the logs stored in your S3 bucket:

```
S3 Bucket: s3://<Your-S3Bucket>
```

This structured format organizes your notes into clear sections, making it easier to follow the steps for managing EC2 logs and integrating them with CloudWatch and S3.

Here's a summarized version of your DevOps notes on Terraform, organized with headings, explanations, and code snippets for clarity:

Terraform Tutorial: Introduction and Exercise 1

1. What is Terraform?

- **Terraform** is an Infrastructure-as-Code (IaC) tool that automates the provisioning and management of cloud infrastructure.
- Allows you to define your infrastructure's desired state, and it ensures resources are created and managed accordingly.
- Terraform uses **HashiCorp Configuration Language (HCL)**, which is declarative and simple to learn.

Benefits:

- Automates infrastructure provisioning and management.
- Works across multiple cloud providers (AWS, Azure, Google Cloud).
- Maintains infrastructure state and ensures it is kept consistent.

2. Comparing Terraform with Other Automation Tools

Tool	Purpose	Approach	Strengths
Terraform	Cloud infrastructure provisioning	Declarative	Cloud resource management across providers

Tool	Purpose	Approach	Strengths
Ansible	IT automation, configuration management	YAML playbooks	Simple, agentless, ideal for configuration tasks
Puppet	Manages large infrastructures	Declarative	Consistency enforcement, strong reporting
Chef	Configuration management	Convergent	Flexible and scalable for complex infrastructures

Key Differences:

- Terraform focuses on **infrastructure provisioning**.
- Ansible, Puppet, and Chef focus on **configuration management** of software within infrastructure.

---

3. Terraform Installation

Steps to Install Terraform

1. Download the **Terraform binary** from the official website for your OS.
2. Add the binary to your system's `PATH`. For Linux, the default location is `/usr/local/bin`.

```
# Example for adding Terraform to PATH on Linux
export PATH=$PATH:/usr/local/bin
```

On Windows:

- Install Terraform via **choco** in PowerShell:

```
choco install terraform
```

---

4. Launching an EC2 Instance with Terraform

Prerequisites:

- An **AWS account**.
- Create an **IAM User** with access keys.

Terraform Configuration File (instance.tf)

Write the Terraform configuration file (`instance.tf`) to launch an EC2 instance:

```
provider "aws" {
  region = "us-east-2"
}

resource "aws_instance" "intro" {
  ami = "ami-03657b56516ab7912"
  instance_type = "t2.micro"
  availability_zone = "us-east-2a"
  key_name = "dove-key"
  vpc_security_group_ids = ["sg-0780815f55104be8a"]

  tags = {
    Name = "Dove-Instance"
  }
}
```

Explanation:

- **provider**: Specifies the cloud service (in this case, AWS) and region.
- **resource**: Defines the AWS EC2 instance with attributes like AMI ID, instance type, availability zone, key pair, and security group.

Running Terraform Commands

1. **Initialize Terraform**:

   ```
   terraform init
   ```

2. **Apply the Terraform configuration** to launch the instance:

   ```
   terraform apply
   ```

---

5. Using AWS Access Keys with Terraform

Providing AWS Credentials

While you can directly embed credentials in the Terraform configuration, this is not recommended for security reasons. Instead, use the AWS CLI to configure credentials:

```
aws configure
```

This command will prompt for the AWS Access Key ID, Secret Access Key, default region, and output format.

Safer Provider Block

Instead of hardcoding keys, use the following `provider` block:

```
provider "aws" {
  region = "us-east-2"
}
```

---

6. Making Changes to EC2 Instance

Modifying the `instance.tf` file:

- You can modify resources, such as changing the instance type or adding tags, and reapply the configuration.
- Example of changing instance type:

```
instance_type = "t2.medium"
```

After making changes, run:

```
terraform apply
```

---

7. Exercise

Task:

- Write an `instance.tf` file to launch an EC2 instance.
- Modify the file and reapply changes using the following steps:
  1. Create the `instance.tf` file.
  2. Launch the instance with `terraform apply`.
  3. Make changes (e.g., change instance type).
  4. Reapply changes using `terraform apply`.

By following these steps, you can automate the process of creating and managing infrastructure using Terraform, while ensuring security and efficiency through proper practices.

Here's a structured summary of your Terraform notes, complete with headings, explanations, and code snippets for better understanding:

Terraform Tutorial: Commands, Variables, and Provisioners

1. Running Terraform Commands

Initializing Terraform

The first command to run when using Terraform is:

`terraform init`

- **Purpose**: This command initializes the working directory containing the Terraform configuration files.
- **Outcome**: It checks the resource provider and downloads the necessary plugins, creating a hidden `.terraform` directory where these plugins are stored:

  `.terraform/plugins/`

Validating and Formatting Terraform Scripts

- **Validate**: Use the following command to check for syntax errors in your configuration:

  `terraform validate`

- **Format**: To automatically format the configuration files for readability, use:

  `terraform fmt`

Planning and Applying Changes

- **Plan**: Before applying changes, you can review the expected modifications by running:

  `terraform plan`

- **Apply**: To apply the configuration and create resources, use:

  `terraform apply`

State Files

Terraform tracks the state of your infrastructure in a file called `terraform.tfstate`. It contains details about the current state of the resources, and a backup file (`terraform.tfstate.backup`) is also maintained.

Destroying Resources

To delete all resources defined in your configuration, run:

```
terraform destroy
```

---

2. Working with Variables in Terraform

Variables allow you to manage sensitive information and frequently changing values without hardcoding them in the main configuration files. They also enable reusability.

Declaring Variables

Variables can be declared in separate files like `vars.tf` and referenced in the configuration files.

Example:

- **providers.tf**:

```
provider "aws" {
  region = var.REGION
}
```

- **vars.tf**:

```
variable "AWS_ACCESS_KEY" {}
variable "AWS_SECRET_KEY" {}

variable "REGION" {
  default = "us-west-1"
}

variable "AMIS" {
  type = "map"
  default = {
    us-west-1 = "ami-06397100adf427136"
    us-west-2 = "ami-0a42f4d8"
  }
}
```

- **terraform.tfvars** (Sensitive information should be stored here):

```
AWS_ACCESS_KEY = "your-access-key"
AWS_SECRET_KEY = "your-secret-key"
```

Using Variables in Terraform Files

- **instance.tf**:

```
resource "aws_instance" "intro" {
  ami             = var.AMIS[var.REGION]
  instance_type   = "t2.micro"
  availability_zone = "us-west-1a"
}
```

---

3. Example of Variables

Example:

- **vars.tf**:

```
variable "REGION" {
  default = "us-east-2"
}

variable "ZONE1" {
  default = "us-east-2a"
}

variable "AMIS" {
  type = "map"
  default = {
    us-east-2 = "ami-03657b56516ab7912"
    us-east-1 = "ami-0947d2ba12ee1ff75"
  }
}
```

- **providers.tf**:

```
provider "aws" {
  region = var.REGION
}
```

- **instance.tf**:

```
resource "aws_instance" "dove-inst" {
  ami = var.AMIS[var.REGION]
  instance_type = "t2.micro"
  availability_zone = var.ZONE1
  key_name = "new-dove"
  vpc_security_group_ids = ["sg-0780815f55104be8a"]

  tags = {
```

```
    Name = "Dove-Instance"
    Project = "Dove"
  }
}
```

Important:

- Public IPs for EC2 instances cannot be changed unless the instance is terminated and recreated.

---

4. Provisioners in Terraform

Provisioners in Terraform help set up resources by executing scripts or commands after the resource is created.

Types of Provisioning:

1. **Custom Images**: Build custom images using tools like **Packer**.
2. **Standard Images**: Use provisioners to install software and upload files after the resource is launched.

Example of `remote-exec`: You can install software on a newly created EC2 instance using the `remote-exec` provisioner.

```
provisioner "remote-exec" {
  inline = [
    "sudo yum install -y sqlite"
  ]
}
```

Connecting Terraform with Other Configuration Management Tools

- Terraform can connect with **Ansible**, **Puppet**, or **Chef** to further configure resources.
- **Ansible** can be used without an agent, while **Puppet** and **Chef** require agents on the resources.

Example of File Provisioning:

You can use the `file` provisioner to copy files to a remote instance.

```
provisioner "file" {
  source      = "files/test.conf"
  destination = "/etc/test.conf"
```

```
  connection {
    type     = "ssh"
    user     = "root"
    password = var.root_password
  }
}
```

This script copies `test.conf` from your local machine to `/etc/test.conf` on the EC2 instance via SSH.

This structured approach should help you better understand and implement Terraform configurations, variables, and provisioning techniques in your DevOps projects.

Here's a summarized version of your Terraform notes, organized with headings, explanations, and code snippets to better understand the concepts:

Terraform Tutorial: Provisioners, Key Pairs, and File Transfers

1. Provisioners in Terraform

Provisioners allow you to execute scripts, copy files, or perform other tasks after a resource is created.

Common Provisioners:

1. **File Provisioner**: Used to copy files or directories to a remote machine.

   ```
   provisioner "file" {
     source      = "conf/myapp.conf"
     destination = "C:/App/myapp.conf"
     connection {
       type     = "winrm"
       user     = "Administrator"
       password = var.admin_password
     }
   }
   ```

2. **Remote-Exec Provisioner**: Invokes commands or scripts on a remote resource.

   ```
   provisioner "remote-exec" {
     inline = [
       "sudo yum install -y nginx"
   ```

```
  ]
}
```

3. **Local-Exec Provisioner**: Executes local commands after the resource creation.
4. **Puppet Provisioner**: Installs and configures Puppet on a remote resource. Supports both SSH and WinRM connections.
5. **Chef Provisioner**: Installs and configures Chef on a remote resource. Supports SSH and WinRM connections.
6. **Ansible Integration**: Run Terraform, output the IP address, and run Ansible playbooks using `local-exec`.

---

2. Using SSH Keys with Terraform

You can define variables for private and public keys and reuse them across configurations.

Example: Key Pair Variables

- **vars.tf**:

```
variable "PRIV_KEY_PATH" {
  default = "infi-inst_key"
}

variable "PUB_KEY_PATH" {
  default = "infi-inst_key.pub"
}

variable "USER" {
  default = "ubuntu"
}
```

Upload Public Key to AWS

The `aws_key_pair` resource is used to upload a public key to AWS.

```
resource "aws_key_pair" "dove-key" {
  key_name   = "dovekey"
  public_key = file("dovekey.pub")
}
```

You can then reference this key in your EC2 instance configuration:

```
resource "aws_instance" "intro" {
  ami                     = var.AMIS[var.REGION]
```

```
  instance_type            = "t2.micro"
  availability_zone        = var.ZONE1
  key_name                 = aws_key_pair.dove-key.key_name
  vpc_security_group_ids   = ["sg-833e24fd"]
}
```

---

3. File Provisioning and Script Execution

You can use the **file provisioner** to push files from the local machine to a remote instance and the **remote-exec provisioner** to execute scripts on the remote instance.

Example: File Provisioning

```
provisioner "file" {
  source      = "web.sh"
  destination = "/tmp/web.sh"

  connection {
    user        = var.USER
    private_key = file(var.PRIV_KEY_PATH)
    host        = self.public_ip
  }
}
```

Example: Remote Execution

Once the file is copied, use the **remote-exec** provisioner to run the script:

```
provisioner "remote-exec" {
  inline = [
    "chmod u+x /tmp/web.sh",
    "sudo /tmp/web.sh"
  ]
}
```

---

4. End-to-End Terraform Workflow

Here's the general workflow to set up and configure infrastructure with Terraform:

1. **Generate Key Pair**: Create your SSH key pair and define them as variables.
2. **Write Script**: Write a shell script (e.g., `web.sh`) that installs software or configures your resources.

3. **Write Providers and Variables**: Define providers (`providers.tf`) and variables (`vars.tf`).
4. **Define EC2 Instance**: Write `instance.tf` and reference your key pairs, AMIs, and other resources.
5. **Use Provisioners**: Use file provisioners to copy files and remote-exec to run scripts.
6. **Apply Terraform**: Run `terraform apply` to create and configure the resources.

---

5. Example Files for Complete Setup

vars.tf:

```
variable "REGION" {
  default = "us-east-2"
}

variable "ZONE1" {
  default = "us-east-2a"
}

variable "AMIS" {
  type = map
  default = {
    us-east-2 = "ami-03657b56516ab7912"
    us-east-1 = "ami-0947d2ba12ee1ff75"
  }
}
```

providers.tf:

```
provider "aws" {
  region = var.REGION
}
```

web.sh (Shell Script):

```
#!/bin/bash
yum install wget unzip httpd -y
systemctl start httpd
systemctl enable httpd
wget https://www.tooplate.com/zip-templates/2117_infinite_loop.zip
unzip -o 2117_infinite_loop.zip
cp -r 2117_infinite_loop/* /var/www/html/
systemctl restart httpd
```

This structured approach highlights how to use Terraform provisioners, key pairs, and remote commands effectively in your infrastructure automation tasks.

Here's a summary of your DevOps notes with relevant headings, explanations, and code snippets:

Terraform: Key Concepts and Practical Examples

1. Defining AWS Resources with Terraform

AWS Key Pair and EC2 Instance Setup

Terraform allows you to create and manage AWS resources like EC2 instances and key pairs. Below is an example of how to create an AWS key pair and launch an EC2 instance:

Example: EC2 Instance and Key Pair

`instance.tf`:

```
resource "aws_key_pair" "dove-key" {
  key_name   = "dovekey"
  public_key = file("dovekey.pub")
}

resource "aws_instance" "dove-inst" {
  ami                    = var.AMIS[var.REGION]
  instance_type          = "t2.micro"
  availability_zone      = var.ZONE1
  key_name               = aws_key_pair.dove-key.key_name
  vpc_security_group_ids = ["sg-0780815f55104be8a"]

  tags = {
    Name    = "Dove-Instance"
    Project = "Dove"
  }

  provisioner "file" {
    source      = "web.sh"
    destination = "/tmp/web.sh"
  }

  provisioner "remote-exec" {
```

```
    inline = [
      "chmod u+x /tmp/web.sh",
      "sudo /tmp/web.sh"
    ]
  }

  connection {
    user        = var.USER
    private_key = file("dovekey")
    host        = self.public_ip
  }
}
```

Steps to Apply Changes

To apply the above Terraform configuration, follow these commands:

```
terraform init       # Initialize the configuration
terraform validate   # Validate the syntax of the Terraform files
terraform fmt        # Format the Terraform code
terraform plan       # Preview the changes
terraform apply      # Apply the changes and create the resources
```

Destroy Resources

To delete all resources created by Terraform:

```
terraform destroy
```

---

2. Outputting Instance Details

To retrieve instance details like Public and Private IPs after resource creation, you can use the **output** block.

Example: Output Block for IP Addresses

`instance.tf`:

```
output "PublicIP" {
  value = aws_instance.dove-inst.public_ip
}

output "PrivateIP" {
  value = aws_instance.dove-inst.private_ip
}
```

When you run `terraform apply`, the outputs will look like this:

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:
PrivateIP = 172.31.1.145
PublicIP  = 18.189.145.9
```

---

3. Managing Terraform State with Backend

In a collaborative environment, it's essential to store the **terraform state** centrally, so multiple team members can work on the same environment. Terraform can store its state in an **S3 bucket**.

Example: S3 Backend Configuration

`backend.tf`:

```
terraform {
  backend "s3" {
    bucket = "terra-state-dove"
    key    = "terraform/backend"
    region = "us-east-2"
  }
}
```

Steps to Apply Changes with Backend

To use this backend configuration, run:

```
terraform init
terraform validate
terraform fmt
terraform plan
terraform apply
```

---

4. Multi-Cloud Resource Management

Terraform supports multiple cloud providers such as AWS, Azure, GCP, Kubernetes, and Oracle Cloud Infrastructure.

Resources for Multi-Cloud Examples:

- **Terraform Provider Registry**: registry.terraform.io
- **AWS Provider Documentation**: Terraform AWS Provider

Here, you can find various code snippets and examples for managing cloud resources with Terraform across different cloud platforms.

This summarized guide provides an overview of essential Terraform commands, resource creation (like EC2 instances and key pairs), output management, and backend configuration with S3.

Creating a VPC using Terraform

In this exercise, we will create a Virtual Private Cloud (VPC) along with subnets and necessary resources using Terraform. Below is the breakdown of the steps:

---

1. Define Variables The variables are defined in a `vars.tf` file to store region, availability zones, AMIs, and user keys.

```
# vars.tf
variable "REGION" {
  default = "us-east-2"
}

variable "ZONE1" {
  default = "us-east-2a"
}

variable "ZONE2" {
  default = "us-east-2b"
}

variable "ZONE3" {
  default = "us-east-2c"
}

variable "AMIS" {
  type = map
  default = {
    us-east-2 = "ami-03657b56516ab7912"
    us-east-1 = "ami-0947d2ba12ee1ff75"
  }
}

variable "USER" {
  default = "ec2-user"
}
```

```
variable "PUB_KEY" {
  default = "dovekey.pub"
}

variable "PRIV_KEY" {
  default = "dovekey"
}
```

---

2. AWS Provider Configuration In the `providers.tf` file, we configure AWS as the provider and set the region dynamically using the defined variable.

```
# providers.tf
provider "aws" {
  region = var.REGION
}
```

---

3. Creating a VPC The VPC is created in the `vpc.tf` file. This example shows the creation of a VPC with the CIDR block of `10.0.0.0/16`.

```
# vpc.tf
resource "aws_vpc" "dove" {
  cidr_block = "10.0.0.0/16"
  instance_tenancy = "default"
  enable_dns_support = true
  enable_dns_hostnames = true

  tags = {
    Name = "dove-vpc"
  }
}
```

---

4. Creating Public Subnets Three public subnets are created, one in each availability zone defined earlier.

```
resource "aws_subnet" "dove-pub-1" {
  vpc_id = aws_vpc.dove.id
  cidr_block = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone = var.ZONE1

  tags = {
    Name = "dove-pub-1"
```

```
  }
}

resource "aws_subnet" "dove-pub-2" {
  vpc_id = aws_vpc.dove.id
  cidr_block = "10.0.2.0/24"
  map_public_ip_on_launch = true
  availability_zone = var.ZONE2

  tags = {
    Name = "dove-pub-2"
  }
}

resource "aws_subnet" "dove-pub-3" {
  vpc_id = aws_vpc.dove.id
  cidr_block = "10.0.3.0/24"
  map_public_ip_on_launch = true
  availability_zone = var.ZONE3

  tags = {
    Name = "dove-pub-3"
  }
}
```

---

5. Creating Private Subnets Similar to public subnets, private subnets are created for each availability zone.

```
resource "aws_subnet" "dove-priv-1" {
  vpc_id = aws_vpc.dove.id
  cidr_block = "10.0.4.0/24"
  map_public_ip_on_launch = false
  availability_zone = var.ZONE1

  tags = {
    Name = "dove-priv-1"
  }
}

resource "aws_subnet" "dove-priv-2" {
  vpc_id = aws_vpc.dove.id
  cidr_block = "10.0.5.0/24"
  map_public_ip_on_launch = false
  availability_zone = var.ZONE2
```

```
  tags = {
    Name = "dove-priv-2"
  }
}

resource "aws_subnet" "dove-priv-3" {
  vpc_id = aws_vpc.dove.id
  cidr_block = "10.0.6.0/24"
  map_public_ip_on_launch = false
  availability_zone = var.ZONE3

  tags = {
    Name = "dove-priv-3"
  }
}
```

---

6. Creating an Internet Gateway An Internet Gateway is created to allow internet access to the public subnets.

```
resource "aws_internet_gateway" "dove-IGW" {
  vpc_id = aws_vpc.dove.id

  tags = {
    Name = "dove-IGW"
  }
}
```

---

7. Creating a Route Table A route table is created to route internet traffic through the Internet Gateway for the public subnets.

```
resource "aws_route_table" "dove-pub-RT" {
  vpc_id = aws_vpc.dove.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.dove-IGW.id
  }

  tags = {
    Name = "dove-pub-RT"
  }
}
```

---

8. Associating Route Table with Subnets Lastly, the route table is associated with each of the public subnets using `aws_route_table_association`.

```
resource "aws_route_table_association" "dove-pub-RT-assoc-1" {
  subnet_id = aws_subnet.dove-pub-1.id
  route_table_id = aws_route_table.dove-pub-RT.id
}

resource "aws_route_table_association" "dove-pub-RT-assoc-2" {
  subnet_id = aws_subnet.dove-pub-2.id
  route_table_id = aws_route_table.dove-pub-RT.id
}

resource "aws_route_table_association" "dove-pub-RT-assoc-3" {
  subnet_id = aws_subnet.dove-pub-3.id
  route_table_id = aws_route_table.dove-pub-RT.id
}
```

This concludes the creation of a VPC with public and private subnets, an Internet Gateway, a route table, and the required associations using Terraform.

Terraform Tutorial 292 - Exercise 6: Multi-Resource Deployment

This tutorial focuses on creating a Virtual Private Cloud (VPC) and related resources using Terraform, including route table associations, security groups, EC2 instances, and EBS volumes.

---

1. Route Table Associations We create route table associations for three public subnets. These associations ensure that requests from the subnets are routed through the route table and ultimately to the internet gateway.

```
# Route Table Associations
resource "aws_route_table_association" "dove-pub-1-a" {
  subnet_id      = aws_subnet.dove-pub-1.id
  route_table_id = aws_route_table.dove-pub-RT.id
}

resource "aws_route_table_association" "dove-pub-2-a" {
  subnet_id      = aws_subnet.dove-pub-2.id
  route_table_id = aws_route_table.dove-pub-RT.id
}
```

```
resource "aws_route_table_association" "dove-pub-3-a" {
  subnet_id      = aws_subnet.dove-pub-3.id
  route_table_id = aws_route_table.dove-pub-RT.id
}
```

Note: Each subnet has a route table association that redirects traffic to the route table, which further routes the request to the Internet Gateway.

---

2. Security Group Configuration A security group is created for the VPC, allowing only SSH access from a specific IP address (MYIP) while allowing all outbound traffic.

```
# Security Group (secgrp.tf)
resource "aws_security_group" "dove_stack_sg" {
  vpc_id      = aws_vpc.dove.id
  name        = "dove-stack-sg"
  description = "Security Group for dove SSH"

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = [var.MYIP]
  }

  tags = {
    Name = "allow-ssh"
  }
}

# Variable for MYIP
variable "MYIP" {
  default = "183.83.39.203/32"
}
```

Note: The security group allows SSH access only from the specified IP address while allowing all outbound traffic.

---

3. Storing Terraform State in S3 We configure Terraform to store its state file (`terraform.tfstate`) in an S3 bucket for safe and centralized storage.

```
# Terraform Backend Configuration
terraform {
  backend "s3" {
    bucket = "terra-state-dove"
    key    = "terraform/backend_exercise6"
    region = "us-east-2"
  }
}
```

---

4. Provisioning with a Shell Script The `web.sh` script provisions an EC2 instance by installing necessary packages and deploying a simple website.

```
# web.sh
#!/bin/bash
yum install wget unzip httpd -y
systemctl start httpd
systemctl enable httpd
wget https://www.tooplate.com/zip-templates/2117_infinite_loop.zip
unzip -o 2117_infinite_loop.zip
cp -r 2117_infinite_loop/* /var/www/html/
systemctl restart httpd
```

> **Note:** This script installs Apache, downloads a web template, and sets up the web server.

---

5. EC2 Instance and EBS Volume We create an EC2 instance with a key pair, associate a security group, and attach an Elastic Block Store (EBS) volume.

```
# EC2 Instance and Key Pair
resource "aws_key_pair" "dove-key" {
  key_name   = "dovekey"
  public_key = file(var.PUB_KEY)
}

resource "aws_instance" "dove-web" {
  ami                    = var.AMIS[var.REGION]
  instance_type          = "t2.micro"
  subnet_id              = aws_subnet.dove-pub-1.id
```

```
  key_name               = aws_key_pair.dove-key.key_name
  vpc_security_group_ids = [aws_security_group.dove_stack_sg.id]

  tags = {
    Name = "my-dove"
  }
}

# EBS Volume
resource "aws_ebs_volume" "vol_4_dove" {
  availability_zone = var.ZONE1
  size              = 3  # 3GB Volume

  tags = {
    Name = "extr-vol-4-dove"
  }
}

# Volume Attachment
resource "aws_volume_attachment" "atch_vol_dove" {
  device_name = "/dev/xvdh"
  volume_id   = aws_ebs_volume.vol_4_dove.id
  instance_id = aws_instance.dove-web.id
}
```

> **Note:** An EBS volume is created and attached to the EC2 instance as an additional storage device.

---

6. Output Variables We can define output variables to capture and display values like the public IP of the EC2 instance after resource creation.

```
# Output the Public IP
output "PublicIP" {
  value = aws_instance.dove-web.public_ip
}
```

> **Note:** The public IP of the EC2 instance will be displayed after running `terraform apply`.

This concludes the multi-resource Terraform exercise, demonstrating how to deploy VPCs, subnets, security groups, EC2 instances, EBS volumes, and other AWS resources programmatically.

Terraform Tutorial 293: AWS Elastic Kubernetes Service (EKS) Setup

This guide outlines how to set up an Amazon Elastic Kubernetes Service (EKS) cluster using Terraform. We will create the required infrastructure, including VPC, EKS cluster, and worker nodes, while leveraging Terraform modules and resources.

---

1. Overview of Amazon EKS

Amazon EKS provides a managed Kubernetes service, simplifying cluster administration. Unlike `kops` or `kubeadm`, Amazon EKS takes care of many aspects such as automatic scaling, secure control plane, and integrations with AWS services like EC2, VPC, IAM, and EBS. Although EKS is more expensive, it reduces the complexity of cluster management.

---

2. Creating an EKS Cluster with Terraform

To set up the EKS cluster and associated resources like VPC, we utilize predefined Terraform modules from the Terraform Registry.

Code to Create a VPC for EKS:

```
# VPC Module
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"

  name            = "my-vpc"
  cidr            = "10.0.0.0/16"
  azs             = ["eu-west-1a", "eu-west-1b", "eu-west-1c"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]

  enable_nat_gateway = true
  enable_vpn_gateway = true

  tags = {
    Terraform   = "true"
    Environment = "dev"
  }
}
```

> **Note:** The module `terraform-aws-modules/vpc/aws` simplifies the VPC creation process. We pass necessary arguments like CIDR blocks, subnets, and availability zones (AZs).

Terraform Backend Configuration for Storing State:

```
# terraform.tf
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.46.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "~> 3.4.3"
    }
    tls = {
      source  = "hashicorp/tls"
      version = "~> 4.0.4"
    }
    cloudinit = {
      source  = "hashicorp/cloudinit"
      version = "~> 2.2.0"
    }
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = "~> 2.16.1"
    }
  }

  backend "s3" {
    bucket = "terra-eks12"
    key    = "state/terraform.tfstate"
    region = "us-east-1"
  }

  required_version = "~> 1.3"
}
```

> **Note:** We specify the required providers for AWS, Kubernetes, and other dependencies. We also configure the backend to store the Terraform state in an S3 bucket.

---

3. Defining Variables for Cluster Setup

We create variables for essential configuration like the AWS region and the EKS cluster name.

```
# variables.tf
variable "region" {
  description = "AWS region"
  type        = string
  default     = "us-east-1"
}

variable "clusterName" {
  description = "Name of the EKS cluster"
  type        = string
  default     = "vpro-eks"
}
```

> **Note:** These variables help maintain flexibility in configuring the region and cluster name for the EKS setup.

---

4. EKS Cluster and Kubernetes Provider

We configure the EKS and AWS providers to interact with the cluster and manage resources.

```
# main.tf
provider "kubernetes" {
  host                   = module.eks.cluster_endpoint
  cluster_ca_certificate = base64decode(module.eks.cluster_certificate_authority_data)
}

provider "aws" {
  region = var.region
}

# Fetch available AWS availability zones
data "aws_availability_zones" "available" {}

locals {
  cluster_name = var.clusterName
}
```

> **Note:** The `kubernetes` provider needs the cluster's endpoint and certificate authority data, which are provided by the EKS module. The `aws_availability_zones` data source retrieves a list of available zones in the specified region.

---

5. Using Predefined Terraform Modules

Terraform modules simplify the creation of complex infrastructure. For instance, the `terraform-aws-modules/vpc/aws` module creates a VPC with public and private subnets, NAT gateways, and more.

You can find predefined modules in the Terraform Registry, which contains reusable code for a variety of infrastructure components.

Example of Using a Module:

```
# Usage of a predefined module
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"

  name            = "my-vpc"
  cidr            = "10.0.0.0/16"
  azs             = ["eu-west-1a", "eu-west-1b", "eu-west-1c"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]

  enable_nat_gateway = true
  enable_vpn_gateway = true

  tags = {
    Terraform   = "true"
    Environment = "dev"
  }
}
```

Conclusion

Setting up an EKS cluster using Terraform allows for an efficient and scalable Kubernetes environment with AWS's managed services. By leveraging Terraform modules and defining resources like VPCs, clusters, and nodes, you can automate the deployment and management of your infrastructure.

Terraform Tutorial 293: AWS Elastic Kubernetes Service (EKS) Setup

This guide explains how to set up Amazon Elastic Kubernetes Service (EKS) using Terraform. It covers configuring VPCs, subnets, EKS cluster, and worker nodes, as well as accessing the cluster using `kubectl`.

---

1. VPC Configuration

We begin by creating a Virtual Private Cloud (VPC) using Terraform's AWS VPC module. This VPC will contain both private and public subnets.

VPC Code Example:

```
# vpc.tf
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.14.2"

  name            = "vprofile-eks"
  cidr            = "172.20.0.0/16"
  azs             = slice(data.aws_availability_zones.available.names, 0, 3)

  private_subnets = ["172.20.1.0/24", "172.20.2.0/24", "172.20.3.0/24"]
  public_subnets  = ["172.20.4.0/24", "172.20.5.0/24", "172.20.6.0/24"]

  enable_nat_gateway     = true
  single_nat_gateway     = true
  enable_dns_hostnames   = true

  public_subnet_tags = {
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/elb"                      = 1
  }

  private_subnet_tags = {
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/internal-elb"             = 1
  }
}
```

Explanation: - The `module "vpc"` configures a VPC with private and public subnets. - The availability zones are sliced from the list `data.aws_availability_zones.available.names`. - NAT Gateway is enabled, but only one instance is used for all private subnets (`single_nat_gateway` is true).

---

2. EKS Cluster Configuration

Once the VPC is set up, we configure the Amazon EKS cluster and its node groups using the `terraform-aws-modules/eks/aws` module.

EKS Cluster Code Example:

```
# eks-cluster.tf
module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "19.0.4"

  cluster_name    = local.cluster_name
  cluster_version = "1.27"

  vpc_id     = module.vpc.vpc_id
  subnet_ids = module.vpc.private_subnets

  cluster_endpoint_public_access = true

  eks_managed_node_group_defaults = {
    ami_type = "AL2_x86_64"
  }

  eks_managed_node_groups = {
    one = {
      name          = "node-group-1"
      instance_types = ["t3.small"]
      min_size       = 1
      max_size       = 3
      desired_size   = 2
    }
    two = {
      name          = "node-group-2"
      instance_types = ["t3.small"]
      min_size       = 1
      max_size       = 2
      desired_size   = 1
    }
  }
}
```

Explanation: - The EKS cluster is created with the VPC's private subnets. - We define two managed node groups with different desired capacities and instance types (`t3.small`).

---

3. AWS CLI Configuration

To interact with AWS from a Linux terminal, use `aws configure` to set up credentials, region, and output format.

AWS CLI Configuration Example:

```
$ aws configure
AWS Access Key ID [ ********** CX6S]:
AWS Secret Access Key [ ********** 4pcC]:
Default region name [us-east-1]:
Default output format [json]:
```

---

4. Updating Kubeconfig

After creating the EKS cluster, update the kubeconfig file to allow `kubectl` to interact with the EKS cluster.

Kubeconfig Update Command:

```
$ aws eks update-kubeconfig --region us-east-1 --name vprof-eks
```

> **Explanation**: This command adds the EKS cluster's configuration to the kubeconfig file, enabling `kubectl` to communicate with the Kubernetes cluster.

---

5. Installing `kubectl` on Windows

To manage the Kubernetes cluster on Windows, install `kubectl` using Chocolatey.

Kubectl Installation:

```
choco install kubernetes-cli -y
```

> **Explanation**: This installs `kubectl` on a Windows machine, allowing you to manage Kubernetes resources.

Conclusion

This setup demonstrates how to configure an Amazon EKS cluster using Terraform, deploy it with managed node groups, and connect to it using `kubectl`.

Here is a summarized version of the provided transcript with appropriate headings and code snippets:

Ansible: Introduction

Ansible is one of the most popular DevOps tools for automation and configuration management. It evolved after scripting tools like Bash (Linux) and Batch (Windows), which were followed by languages such as Python, PERL, and Ruby. Configuration management tools like Puppet and SaltStack were also used in infrastructure management. Puppet, written in Ruby, manages configurations via agents on servers, while SaltStack, based on Python, allows remote command execution.

Ansible, however, is simpler and agentless, making it suitable for both Linux and Windows automation, as well as cloud, networking, and database automation.

Ansible Use Cases

- **Automation**: Automates tasks like setting up web services, databases, and configuration management.
- **Change Management**: Manages production changes efficiently.
- **Provisioning**: Automates cloud infrastructure setup and provisioning.
- **Orchestration**: Combines and sequences multiple automations, integrated with CI tools like Jenkins.

Ansible vs. Other Tools

- **Puppet/Chef**: Requires agents on target servers, with master-slave architecture.
- **Ansible**: No agents required. Connects to target systems via SSH (Linux), WinRM (Windows), or APIs (Cloud services). Configuration is managed using YAML and is written to JSON format.

Ansible Playbooks and Architecture

- **Playbooks**: Written in YAML, a structured method for declaring automation tasks.
- **Inventory File**: Contains details about target machines (IP address, credentials).
- **Modules**: Ansible includes a wide range of modules (e.g., for package installation, service management, cloud resource management).

Ansible Setup Example: 1. Define inventory:

```
[webservers]
vprofile-web01 ansible_host=10.0.0.1 ansible_user=ubuntu
vprofile-web02 ansible_host=10.0.0.2 ansible_user=ubuntu
```

```
[dbservers]
vprofile-db01 ansible_host=10.0.0.3 ansible_user=ubuntu
```

2. Basic Playbook to install a web server:

```
---
- hosts: webservers
  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present
    - name: Start Apache service
      service:
        name: apache2
        state: started
```

Ansible Setup: Practical Example

To set up an Ansible control machine and multiple target machines:

1. **Control Machine Setup (Ubuntu EC2 instance)**:
   - Launch an EC2 instance for Ansible.
   - Assign security group rules to allow SSH access on port 22.
   - Download the key-pair for secure login.

```
ssh -i Downloads/control.pem ubuntu@<control-machine-ip>
```

2. **Target Machines (CentOS EC2 instances)**:
   - Launch three EC2 instances for web and DB servers.
   - Assign security group rules allowing SSH access from the control machine.

```
ssh -i Downloads/web.pem centos@<web-machine-ip>
```

3. **Access Control**:
   - Ensure port 22 is open for SSH between the control machine and target machines.

```
cat ~/.ssh/known_hosts
```

Power of Ansible

Ansible can replace many traditional automation tools due to its simplicity, agentless design, and wide range of integrations (e.g., cloud, databases, and networks).

This structure covers the key points of Ansible along with practical code snippets to demonstrate basic operations.

Here's a summarized version of your DevOps notes on Ansible with relevant headings and code snippets:

---

1. Introduction to Ansible

Ansible is a popular DevOps tool designed for configuration management and automation. It evolved from earlier tools like Puppet, Chef, and SaltStack but is simpler and agentless, making it easier to manage infrastructure using SSH or WinRM.

Key Features:

- Agentless: Uses SSH/WinRM for communication.
- Configuration format: Uses YAML for playbooks.
- Cloud, database, and network automation support.

---

2. Use Cases of Ansible

- **Automation**: Automate Linux/Windows configurations, start/stop services, manage web/database services.
- **Change Management**: Manage production servers.
- **Provisioning**: Set up cloud resources and services from scratch.
- **Orchestration**: Integrate with tools like Jenkins to orchestrate larger automation workflows.

---

3. Ansible Architecture

Ansible playbooks are written in YAML and use an inventory file and various modules to target different hosts. The architecture doesn't require master-slave nodes, unlike Puppet or Chef.

- **Inventory file**: Contains information about the target machines (IP, username, password).
- **Modules**: Predefined tasks such as package installation or restarting services.

---

4. Setting Up Ansible on Ubuntu

To install Ansible on an Ubuntu control machine:

```
$ sudo apt update
$ sudo apt install software-properties-common
$ sudo add-apt-repository --yes --update ppa:ansible/ansible
$ sudo apt install ansible
```

---

5. Inventory & Ping Module

An inventory file lists the target machines and their details. It can be written in **INI** or **YAML** format.

INI Format Example:

```
[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
```

YAML Format Example:

```
all:
  hosts:
    web01:
      ansible_host: 172.31.31.178
      ansible_user: ec2-user
      ansible_ssh_private_key_file: clientkey.pem
```

Command to Ping a Host:

```
ansible web01 -m ping -i inventory
```

- `web01`: Hostname from the inventory file.
- `-m`: Specifies the module (ping in this case).
- `-i`: Path to the inventory file.

Disabling Host Key Verification:

Edit the `ansible.cfg` file to disable host key checking:

```
ansible-config init --disabled -t all > ansible.cfg
```

Set `host_key_checking = False` to disable the verification step.

---

6. Ansible Variables

Ansible variables help define host-specific information, such as:

- `ansible_host`: IP or DNS name of the target.
- `ansible_user`: SSH username for connecting.
- `ansible_password`: Password (use vault for secure storage).
- `ansible_ssh_private_key_file`: Path to the private key for SSH connections.

For more information, check the official Ansible Inventory Documentation.

By organizing these notes under relevant headings and including the necessary commands and examples, this summary should help you with your DevOps learning focused on Ansible.

Here's a summarized version of your DevOps notes on Ansible (Inventory Part 2 and YAML & JSON) with relevant headings and code snippets:

---

1. Inventory File in Ansible (Part 2)

Ansible's inventory file defines the hosts or groups of hosts on which playbooks are executed. You can group hosts and even create hierarchical parent-child relationships between them.

Example Inventory File:

```
all:
  hosts:
    web01:
      ansible_host: 172.31.31.178
      ansible_user: ec2-user
      ansible_ssh_private_key_file: clientkey.pem
    web02:
      ansible_host: 172.31.22.225
      ansible_user: ec2-user
      ansible_ssh_private_key_file: clientkey.pem
    db01:
      ansible_host: 172.31.19.215
      ansible_user: ec2-user
      ansible_ssh_private_key_file: clientkey.pem
  children:
    webservers:
```

```
    hosts:
      web01:
      web02:
  dbservers:
    hosts:
      db01:
  dc_oregon:
    children:
      webservers:
      dbservers:
```

Connecting to Hosts Using Ansible:

```
ansible web02 -m ping -i inventory
ansible db01 -m ping -i inventory
```

Grouping Hosts:

Groups like `webservers` and `dbservers` make it easier to manage multiple hosts at once. A parent group like `dc_oregon` can be used to combine both:

```
ansible webservers -m ping -i inventory
ansible dc_oregon -m ping -i inventory
ansible '*' -m ping -i inventory
ansible 'web*' -m ping -i inventory
```

---

2. Variables in Ansible Inventory

Variables can be defined at the **host level** or the **group level**. Group-level variables apply to all hosts within the group, but host-level variables have higher priority.

Example with Group-Level Variables:

```
all:
  hosts:
    web01:
      ansible_host: 172.31.31.178
    web02:
      ansible_host: 172.31.22.225
    db01:
      ansible_host: 172.31.19.215
  children:
    webservers:
      hosts:
        web01:
```

```
        web02:
    dbservers:
      hosts:
        db01:
  vars:
    ansible_user: ec2-user
    ansible_ssh_private_key_file: clientkey.pem
```

To execute the inventory across all hosts:

```
ansible all -m ping -i inventory
```

---

3. Understanding JSON and YAML in Ansible

Ansible uses **YAML** for writing playbooks and **JSON** for response output. YAML is more human-readable, while JSON is often used for machine processing.

JSON Example:

```
{
  "DevOps": [
    "AWS",
    "Jenkins",
    "Python",
    "Ansible"
  ],
  "Development": [
    "Java",
    "NodeJS",
    ".net"
  ],
  "ansible_facts": {
    "python": "/usr/bin/python"
  }
}
```

Equivalent YAML Example:

```
DevOps:
  - AWS
  - Jenkins
  - Python
  - Ansible

Development:
```

```
  - Java
  - NodeJS
  - .net

ansible_facts:
  python: /usr/bin/python
  version: 2.7
```

YAML is designed to be clean and more readable, whereas JSON is structured like a Python dictionary but in a vertical format. Most modern tools, including Ansible, use YAML for configuration files.

---

4. List of Dictionaries in YAML

In YAML, complex data structures like lists and dictionaries can be combined. Here's an example of a **list of dictionaries**:

```
# Employee records
- martin:
    name: Martin D'vloper
    job: Developer
    skills:
      - python
      - perl
      - pascal
- tabitha:
    name: Tabitha Bitumen
    job: Developer
    skills:
      - lisp
      - fortran
      - erlang
```

For more details on YAML syntax, refer to Ansible YAML Syntax Documentation.

This summary organizes your notes into structured sections with code examples and explanations. It covers both inventory management and the use of JSON/YAML in Ansible.

Ansible - Ad Hoc Commands

What Are Ad Hoc Commands? Ad hoc commands in Ansible are simple one-liners used for quick tasks like verification or debugging. These commands are useful when you need immediate results without creating a full playbook.

Example: Ping a server

```
ansible web01 -m ping -i inventory
```

Installing a Package Using Ad Hoc Command To install the `httpd` package on `web01` using the `yum` module:

```
ansible web01 -m ansible.builtin.yum -a "name=httpd state=present" -i inventory
```

For sudo privileges:

```
ansible web01 -m ansible.builtin.yum -a "name=httpd state=present" -i inventory --become
```

Starting a Service Using Ad Hoc Command Start and enable the `httpd` service on all servers in the `webservers` group:

```
ansible webservers -m ansible.builtin.service -a "name=httpd state=started enabled=yes" -i i
```

Equivalent commands in Linux:

```
systemctl start httpd
systemctl enable httpd
```

Copying a File Using Ad Hoc Command Copy the `index.html` file from the local machine to the `/var/www/html` directory on the `webservers` group:

```
ansible webservers -m ansible.builtin.copy -a "src=index.html dest=/var/www/html/index.html"
```

Ansible Playbook & Modules

Example of a Simple Playbook A playbook defines tasks in YAML format to be executed on target servers.

Playbook Example:

```yaml
- hosts: websrvgrp
  tasks:
    - name: Install Apache
      yum:
        name: httpd
        state: present
    - name: Deploy Config
      copy:
        src: file/httpd.conf
        dest: /etc/httpd.conf
- hosts: dbsrvgrp
  tasks:
    - name: Install Postgresql
      yum:
        name: postgresql
        state: latest
```

Explanation of Playbook Elements

- `hosts`: Specifies the group of servers.
- `tasks`: Contains the list of tasks to perform.
- `yum`: Module to install packages.
- `copy`: Module to copy files to remote servers.

Playbook with Multiple Plays and Tasks Example Playbook:

```
- name: Webserver setup
  hosts: webservers
  become: yes
  tasks:
    - name: Install httpd
      ansible.builtin.yum:
        name: httpd
        state: present
    - name: Start service
      ansible.builtin.service:
        name: httpd
        state: started
        enabled: yes

- name: DBserver setup
  hosts: dbservers
  become: yes
  tasks:
    - name: Install mariadb-server
      ansible.builtin.yum:
        name: mariadb-server
        state: present
    - name: Start mariadb service
      ansible.builtin.service:
        name: mariadb
        state: started
        enabled: yes
```

In this example, there are two plays: one for web servers and one for database servers, each performing respective tasks like installing and starting services.

Ansible - Playbook & Modules

Executing Ad Hoc Commands vs Playbooks Ad hoc commands are useful for quick tasks, but for more complex operations, playbooks are preferred. Below is an example of removing a package via an ad hoc command:

```
ansible webservers -m yum -a "name=httpd state=absent" -i inventory --become
```

Running an Ansible Playbook To execute an Ansible playbook:

```
ansible-playbook -i inventory web-db.yaml
```

Debugging Ansible Playbooks with Verbose Mode Ansible supports four levels of verbosity for debugging:

```
ansible-playbook -i inventory web-db.yaml -v    # Basic verbosity
ansible-playbook -i inventory web-db.yaml -vv   # More detailed output
ansible-playbook -i inventory web-db.yaml -vvv  # Even more detailed output
ansible-playbook -i inventory web-db.yaml -vvvv # Maximum verbosity
```

Syntax Check and Dry Run Before executing a playbook, you can check for syntax errors:

```
ansible-playbook -i inventory web-db.yaml --syntax-check
```

To perform a dry run without making actual changes:

```
ansible-playbook -i inventory web-db.yaml -C
```

Dry runs help validate playbooks before applying them to production environments.

Best Practice for Playbook Execution

1. Run a syntax check.
2. Perform a dry run.
3. Execute the actual playbook.

Ansible Modules

Ansible offers a vast library of modules for different tasks across cloud providers and infrastructure setups. You can find a complete list of modules here: Ansible Modules Documentation

Example: Database Server Setup Below is a playbook for setting up a database server, installing `mariadb-server`, and managing a MySQL database and user.

```
- name: DBserver setup
  hosts: dbservers
  become: yes
  tasks:
    - name: Install mariadb-server
      ansible.builtin.yum:
        name: mariadb-server
        state: present
    - name: Install pymysql
```

```
    ansible.builtin.yum:
      name: python3-PyMySQL
      state: present
  - name: Start mariadb service
    ansible.builtin.service:
      name: mariadb
      state: started
      enabled: yes
  - name: Create a new database with name 'accounts'
    community.mysql.mysql_db:
      name: accounts
      state: present
      login_unix_socket: /var/lib/mysql/mysql.sock
  - name: Create database user with name 'vprofile'
    community.mysql.mysql_user:
      name: vprofile
      password: 'admin943'
      priv: '*.*:ALL'
      state: present
      login_unix_socket: /var/lib/mysql/mysql.sock
```

Socket Files in DevOps

What Are Socket Files? Socket files (Unix domain sockets) are special files used for inter-process communication (IPC) on the same machine. Unlike network sockets, they handle communication between local processes.

Common Use Cases

1. **Web Servers and Application Servers:**
   Web servers like Nginx and Apache use socket files to communicate with application servers (e.g., uWSGI, Gunicorn), enhancing performance and security.
2. **Database Connections:**
   Databases like MySQL and PostgreSQL use socket files for local connections, reducing network overhead.
3. **Container Orchestration:**
   Containers in systems like Docker communicate using socket files within the same pod.
4. **System Services:**
   Services like `systemd` use socket files for triggering service activation based on incoming connections.

Ansible Configuration

Changing Ansible Default Behavior To modify Ansible's default behavior, such as connecting to SSH on a different port (e.g., port 2020 instead of the default 22), you need to edit the Ansible configuration settings.

Ansible Configuration File Priority Ansible configuration settings can be changed in different files with the following priority: 1. **Highest Priority**: The file set in the `ANSIBLE_CONFIG` environment variable. 2. **Second Priority**: An `ansible.cfg` file in the project directory (committed to the repository for team-wide use). 3. **Third Priority**: A hidden file in the home directory (`~/.ansible.cfg`). 4. **Last Priority**: The default global configuration file (`/etc/ansible/ansible.cfg`).

Example Ansible Configuration File Here's an example of a simple Ansible configuration file:

```
[defaults]
host_key_checking = False
inventory = ./inventory
forks = 5
log_path = /var/log/ansible.log

[privilege_escalation]
become = True
become_method = sudo
become_ask_pass = False
```

Handling Log File Permission Errors If you encounter an error regarding write permissions for the log file:

```
[WARNING]: log file at /var/log/ansible.log is not writable
```

Fix it with:

```
sudo touch /var/log/ansible.log
sudo chown ubuntu:ubuntu /var/log/ansible.log
```

Variables & Debugging in Ansible

Defining Variables Variables can be defined directly in the playbook after the `hosts` section:

```
- hosts: websrvgrp
  vars:
    http_port: 80
    sqluser: admin
```

In this example, `http_port` and `sqluser` are custom variables. Ansible also has many inbuilt variables such as: - `ansible_os_family`: OS name (e.g., RedHat, Debian) - `ansible_processor_cores`: Number of CPU cores - `ansible_kernel`: Kernel version - `ansible_default_ipv4`: IP, MAC address, and gateway

Registering Variables for Task Output You can store the output of a task using the `register` module:

```
- name: Create database user with name 'vprofile'
  community.mysql.mysql_user:
    name: "{{dbuser}}"
    password: "{{dbpass}}"
    priv: '*.*:ALL'
    state: present
    login_unix_socket: /var/lib/mysql/mysql.sock
  register: dbout

- name: Print dbout variable
  debug:
    var: dbout
```

This registers the output of the task in the `dbout` variable and prints it in the next task.

Example Playbook with Variables Below is an example of using variables in a playbook to make it more generic and reusable:

```
- name: DBserver setup
  hosts: dbservers
  become: yes
  vars:
    dbname: electric
    dbuser: current
    dbpass: tesla
  tasks:
    - debug:
        msg: "The dbname is {{dbname}}"
    - debug:
        var: dbuser
    - name: Install mariadb-server
      ansible.builtin.yum:
        name: mariadb-server
        state: present
    - name: Install pymysql
      ansible.builtin.yum:
        name: python3-PyMySQL
```

```
    state: present
- name: Start mariadb service
  ansible.builtin.service:
    name: mariadb
    state: started
    enabled: yes
- name: Create a new database with name '{{dbname}}'
  mysql_db:
    name: "{{dbname}}"
    state: present
    login_unix_socket: /var/lib/mysql/mysql.sock
- name: Create database user with name '{{dbuser}}'
  community.mysql.mysql_user:
    name: "{{dbuser}}"
    password: "{{dbpass}}"
    priv: '*.*:ALL'
    state: present
    login_unix_socket: /var/lib/mysql/mysql.sock
```

Debugging with Ansible

Use the `debug` module to print variable values or messages during playbook execution, helping with troubleshooting:

```
- debug:
    msg: "The dbname is {{dbname}}"
- debug:
    var: dbuser
```

This approach is useful when you want to verify variable values or output during playbook execution.

Group & Host Variables in Ansible

Defining Variables at the Inventory Level Instead of defining variables within playbooks, you can declare them at the inventory level, making them reusable across multiple playbooks.

1. **Group Variables**: To define variables for a group of hosts, create a `group_vars` directory under your project directory, then create a file named `all` for global variables.

Example:

```
# group_vars/all
dbname: sky
dbuser: pilot
dbpass: aircraft
```

These variables will apply to all hosts unless overridden in the playbook itself, as playbook variables have higher priority.

Priority of Variables

1. **Playbook Variables**: Highest priority.
2. **Host Variables**: Defined in `host_vars` and applied to specific hosts.
3. **Group Variables**: Defined in `group_vars` and applied to a group of hosts.
4. **Global Variables**: Declared in `group_vars/all` and applied to all hosts.

Defining Group Variables For example, to create group-specific variables for webservers, define them under `group_vars/webservers`:

```
# group_vars/webservers
USRNM: webgroup
COMM: variable from group_vars/webservers file
```

Defining Host Variables Host-specific variables can be declared under the `host_vars` directory, with file names corresponding to the inventory file. For example:

```
# host_vars/web02
USRNM: web02user
COMM: variables from host_vars/web02 file
```

Example Playbook Using Variables Below is an example playbook using variables from the `group_vars` directory:

```
- name: Understanding vars
  hosts: all
  become: yes
  vars:
    USRNM: playuser
    COMM: variable from playbook
  tasks:
    - name: create user
      ansible.builtin.user:
        name: "{{USRNM}}"
        comment: "{{COMM}}"
      register: usrout

    - debug:
        var: usrout.name
    - debug:
        var: usrout.comment
```

Registering and Debugging Variables You can register the output of a task and then debug it as follows:

```
- name: create user
  ansible.builtin.user:
    name: "{{USRNM}}"
    comment: "{{COMM}}"
  register: usrout

- debug:
    var: usrout.name
- debug:
    var: usrout.comment
```

Example output:

```
ok: [web01] => {"usrout.name": "playuser"}
ok: [web01] => {"usrout.comment": "variable from playbook"}
```

Using External Variable Files You can also define variables in an external file and use them in the playbooks:

```
- hosts: all
  remote_user: root
  vars:
    favcolor: blue
  vars_files:
    - /vars/external_vars.yml
  tasks:
    - name: This is just a placeholder
      ansible.builtin.command: /bin/echo foo
```

Variable Precedence Variable precedence in Ansible is as follows: 1. **Playbook variables** 2. **Host variables** 3. **Group variables** 4. **All host variables**

Fact Variables in Ansible

What Are Fact Variables? Fact variables are runtime variables generated when the `setup` module is executed. These variables provide system information like OS, hardware details, and network configuration. Examples of fact variables: - `ansible_os_family`: OS name (e.g., RedHat, Debian) - `ansible_processor_cores`: Number of CPU cores - `ansible_kernel`: Kernel version - `ansible_devices`: Connected device information - `ansible_default_ipv4`: IP, MAC address, and gateway - `ansible_architecture`: 64-bit or 32-bit system architecture

When an Ansible playbook runs, the first step is "Gathering Facts," which executes the `setup` module to gather these facts. You can use these variables in your playbooks as needed.

Skipping Fact Gathering To skip the fact-gathering step, you can disable it by setting `gather_facts: False` in the playbook.

Example:

```
- name: Understanding vars
  hosts: all
  become: yes
  gather_facts: False
  vars:
    USRNM: playuser
    COMM: variable from playbook
  tasks:
    - name: create user
      ansible.builtin.user:
        name: "{{USRNM}}"
        comment: "{{COMM}}"
      register: usrout
    - debug:
        var: usrout.name
    - debug:
        var: usrout.comment
```

Gathering Host Information with `setup` Module To fetch configuration and setup details of a host, use the following command:

```
ansible -m setup web01
```

Using and Debugging Fact Variables in Playbooks Example of printing fact variables:

```
- name: Print facts
  hosts: all
  tasks:
    - name: Print OS name
      debug:
        var: ansible_distribution

    - name: Print SELinux mode
      debug:
        var: ansible_selinux.mode

    - name: Print RAM memory
```

```
    debug:
      var: ansible_memory_mb.real.free

  - name: Print Processor name
    debug:
      var: ansible_processor[2]
```

If the variable is not defined, the output will show a message like:

```
ok: [web01] => {"ansible_distribution": "VARIABLE IS NOT DEFINED!"}
```

Defining User and Host in the Inventory File You can declare the user and host information in the inventory file as follows:

```
all:
  hosts:
    web01:
      ansible_host: 172.31.31.178
    web02:
      ansible_host: 172.31.22.225
    web03:
      ansible_host: 172.31.23.53
  vars:
    ansible_user: ubuntu
```

Checking Connectivity with Ping You can check if you can connect to all hosts using the following command:

```
ansible -m ping all
```

Decision Making in Playbooks

Conditional Execution Using `when` In Ansible, you can execute tasks conditionally based on certain facts. For example, to install and start services based on the OS type, use the `when` clause.

Example:

```
- name: Provisioning servers
  hosts: all
  become: yes
  tasks:
    - name: Install NTP agent on CentOS
      yum:
        name: chrony
        state: present
      when: ansible_distribution == "CentOS"
```

```yaml
- name: Install NTP agent on Ubuntu
  apt:
    name: ntp
    state: present
    update_cache: yes
  when: ansible_distribution == "Ubuntu"

- name: Start service on CentOS
  service:
    name: chronyd
    state: started
    enabled: yes
  when: ansible_distribution == "CentOS"

- name: Start service on Ubuntu
  service:
    name: ntp
    state: started
    enabled: yes
  when: ansible_distribution == "Ubuntu"
```

Dry Run of Playbook You can perform a dry run of the playbook without making actual changes by running:

```
ansible-playbook provisioning.yaml -C
```

Decision Making in Ansible

Using `update_cache` The `update_cache: yes` parameter in Ansible's `apt` module ensures that the package manager updates its cache before installing a package. This is commonly used in Ubuntu-based systems.

For more detailed documentation on decision-making in Ansible playbooks, refer to the official guide:
Ansible Playbook Conditionals

Loops in Ansible

Avoiding Code Duplication with Loops When installing multiple packages on a server, you can avoid code duplication by using loops. The `loop` feature allows you to iterate over items, like package names, instead of repeating the task for each package.

Example of installing multiple packages on CentOS:

```yaml
- name: Provisioning servers
  hosts: all
  become: yes
  tasks:
    - name: Install packages on CentOS
      yum:
        name: "{{item}}"
        state: present
      when: ansible_distribution == "CentOS"
      loop:
        - chrony
        - wget
        - git
        - zip
        - unzip
```

Similarly, for Ubuntu:

```yaml
- name: Install packages on Ubuntu
  apt:
    name: "{{item}}"
    state: present
    update_cache: yes
  when: ansible_distribution == "Ubuntu"
  loop:
    - ntp
    - wget
    - git
    - zip
    - unzip
```

Looping Through Dictionaries You can also loop through lists of dictionaries. For example, adding multiple users with specific groups:

```yaml
- name: Add several users
  ansible.builtin.user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

For more on loops, refer to the official documentation:
Ansible Loops

File, Copy, and Template Modules in Ansible

Copying Files to Remote Hosts The `copy` module in Ansible allows you to copy files from your local machine to a remote location. You can either provide the content directly or reference a source file.

Example of copying content to the `/etc/motd` file on a remote server:

```
- name: Banner file
  copy:
    content: '# This server is managed by Ansible. No manual changes please.'
    dest: /etc/motd
```

For dealing with file modules, you can find more detailed information in the official documentation:
Ansible File Modules

Ansible 202: File, Copy, and Template Modules

Ansible provides powerful modules for transferring files to remote servers: the **copy** and **template** modules. Each serves a unique purpose depending on whether the file content is static or dynamic.

---

1. Copy Module The **copy** module is used to copy a file from the local host to a remote server. It is ideal for files that are consistent across all systems, such as login banners, messages of the day (MOTD), or static configuration files.

Example Playbook Using the Copy Module:

```
---
- hosts: nyc1-webserver-1.example.com
  gather_facts: no
  tasks:
    - name: Copy MOTD into place
      copy:
        src: etc/motd
        dest: /etc/motd
        owner: root
        group: root
        mode: '0644'
```

This module copies files as they are, with no templating or dynamic content generation.

---

2. Template Module The **template** module is similar to the copy module but uses **Jinja2 templating** to replace variables within the file. This allows dynamic content generation based on the server or environment-specific data.

Example Playbook Using the Template Module:

```yaml
---
- hosts: nyc1-webserver-1.example.com
  gather_facts: no
  tasks:
    - name: Install and configure keepalived
      template:
        src: templates/etc/keepalived/keepalived.conf.j2
        dest: /etc/keepalived/keepalived.conf
        owner: root
        group: root
        mode: '0644'
```

Here, the `keepalived.conf.j2` file contains placeholders that are replaced by actual values during execution.

Using Jinja2 Templating

Jinja2 is a templating language used to insert dynamic content within configuration files. The template module reads files with placeholders (written in Jinja2 syntax) and replaces them with provided values.

Jinja2 Syntax Example:

```
# nginx.conf.j2
server {
    listen {{ nginx_port }};
    server_name {{ server_name }};
    location / {
        root {{ web_root }};
        index index.html;
    }
}
```

- **Placeholders**: `{{ nginx_port }}`, `{{ server_name }}`, and `{{ web_root }}`.
- During execution, these placeholders are replaced by the values specified in the playbook.

Providing Variables for Templates

Variables can be defined within the playbook or in external files (e.g., `group_vars/all`) to dynamically configure templates.

Example Playbook with Variables:

```
---
- hosts: web_servers
  vars:
    nginx_port: 80
    server_name: example.com
    web_root: /var/www/html
  tasks:
    - name: Configure Nginx
      template:
        src: templates/nginx.conf.j2
        dest: /etc/nginx/nginx.conf
```

When the playbook runs, it substitutes the placeholders in `nginx.conf.j2` with the values defined in the `vars` section.

Dynamic Configuration with the Template Module

The template module excels in generating different configurations for different servers. For instance, using the same template file, you can create unique Nginx configurations by varying the `nginx_port`, `server_name`, and `web_root` values across different hosts.

Example: NTP Configuration with Template Module

In this example, we define a configuration for NTP (Network Time Protocol) based on the operating system (CentOS or Ubuntu) using the **template** module.

```
---
- name: Deploy ntp agent conf on CentOS
  template:
    src: templates/ntpconf_centos
    dest: /etc/chrony.conf
    backup: yes
  when: ansible_distribution == "CentOS"

- name: Deploy ntp agent conf on Ubuntu
  template:
    src: templates/ntpconf_ubuntu
    dest: /etc/ntp.conf
```

```
    backup: yes
  when: ansible_distribution == "Ubuntu"
```

This configuration file will include NTP pool servers, and the placeholders are replaced dynamically.

```
# ntpconf_centos (template example)
pool "{{ ntp0 }}" iburst
pool "{{ ntp1 }}" iburst
pool "{{ ntp2 }}" iburst
pool "{{ ntp3 }}" iburst
```

The placeholders `{{ ntp0 }}`, `{{ ntp1 }}`, etc., will be replaced by the actual pool server addresses defined in a variables file (`group_vars/all`).

Restarting Services After Configuration Change

Once the configuration files are updated, it is important to restart the respective services to apply the changes.

```
---
- name: Restart service on CentOS
  service:
    name: chronyd
    state: restarted
    enabled: yes
  when: ansible_distribution == "CentOS"

- name: Restart service on Ubuntu
  service:
    name: ntp
    state: restarted
    enabled: yes
  when: ansible_distribution == "Ubuntu"
```

Summary: Choosing Between Copy and Template Modules

- **Copy Module**: Use when files are consistent and do not require dynamic content.
- **Template Module**: Use when files need to be dynamically generated using Jinja2 templating (e.g., configuration files that differ per host).

By using the appropriate module, Ansible makes it easy to manage and deploy configuration files across multiple systems.

Ansible 203: Handlers

Handlers in Ansible are tasks that run **only when notified** by other tasks. They are typically used for actions like restarting services after configuration changes.

Defining Handlers Handlers are declared at the same level as tasks in a playbook and are triggered using the `notify` directive.

Example of Handlers:

```yaml
handlers:
  - name: reStart service on centos
    service:
      name: chronyd
      state: restarted
      enabled: yes
    when: ansible_distribution == "CentOS"

  - name: reStart service on ubuntu
    service:
      name: ntp
      state: restarted
      enabled: yes
    when: ansible_distribution == "Ubuntu"
```

Linking Handlers to Tasks To trigger a handler, you must notify it within a task. Handlers are only run if the task changes the state of the system (e.g., if a configuration file is modified).

Example of Tasks with Notify:

```yaml
tasks:
  - name: Deploy ntp agent conf on centos
    template:
      src: templates/ntpconf_centos
      dest: /etc/chrony.conf
      backup: yes
    when: ansible_distribution == "CentOS"
    notify:
      - reStart service on centos

  - name: Deploy ntp agent conf on ubuntu
    template:
      src: templates/ntpconf_ubuntu
      dest: /etc/ntp.conf
      backup: yes
    when: ansible_distribution == "Ubuntu"
```

```
    notify:
      - reStart service on ubuntu
```

Multiple handlers can be notified by the same task, if needed.

Ansible 204: Roles

Roles in Ansible allow for **modular and reusable** playbook structures. They help you organize and scale your infrastructure automation.

Key Concepts of Roles

1. **Modularity**: Break down playbooks into smaller, self-contained units. Each role can handle a specific task, such as installing software or configuring a service.
2. **File Structure**: Roles follow a predefined directory structure that includes folders for tasks, variables, handlers, templates, and more.
3. **Reusability**: Roles can be reused across multiple playbooks, reducing code duplication and improving maintainability.
4. **Dependencies**: Roles can depend on other roles. Ansible resolves these dependencies automatically.
5. **Variables**: Roles define their own variables, which can be overridden at the playbook level to customize behavior.
6. **Handlers**: Roles can include handlers that are triggered when certain tasks change the system state.

Role Directory Structure A typical role structure looks like this:

```
roles/
  common-server/
    defaults/
      main.yml
    files/
    handlers/
      main.yml
    meta/
      main.yml
    tasks/
      main.yml
    templates/
    vars/
      main.yml
    tests/
```

```
inventory
test.yml
```

- **defaults/main.yml**: Default variables for the role.
- **tasks/main.yml**: Tasks to be executed by the role.
- **handlers/main.yml**: Handlers to be triggered by tasks.
- **vars/main.yml**: Additional variables for the role.
- **templates/**: Jinja2 templates used in tasks.

This structure provides modularity, reusability, and ease of management.

Example of Role with Tasks and Variables Playbook with a Role:

```yaml
---
- hosts: web_servers
  roles:
    - common-server
  vars:
    mydir: /opt/dir22
```

Role Task Example:

```yaml
tasks:
  - name: Dump file
    copy:
      src: files/myfile.txt
      dest: /tmp/myfile.txt

  - name: Create a folder
    file:
      path: "{{ mydir }}"
      state: directory
```

In this example: - A file is copied to the server. - A folder is created at a location specified by the `mydir` variable, which is defined at the playbook level.

Summary

- **Handlers**: Triggered only when notified by tasks. Used for actions like restarting services after changes.
- **Roles**: Provide a modular way to organize playbooks, making them reusable and scalable. Roles have a predefined directory structure to organize tasks, handlers, templates, and variables.

By adopting roles and handlers, you can build more structured, maintainable, and scalable playbooks for infrastructure automation.

Ansible 204: Roles

Ansible roles provide a way to **organize** and **reuse** playbooks efficiently, making it easier to manage and scale automation.

Installing `tree` to View Directory Structures You can view a directory's structure in tree format by installing the `tree` package:

```
sudo apt install tree -y
```

Then, list the directory structure using the `tree` command:

```
tree exercise14
```

This shows a visual hierarchy of files and directories in the specified path.

Creating Roles from Ansible Playbooks

Converting an Ansible playbook into a role simplifies managing configurations. The structure of a role allows modular organization of variables, tasks, handlers, and templates.

Creating a Role with `ansible-galaxy` To create a role, use the `ansible-galaxy` command:

```
ansible-galaxy init post-install
```

This command creates the basic structure of an Ansible role under a directory called `post-install`.

Moving Playbook Components to the Role Move files, templates, variables, and tasks from your playbook into the respective directories within the role.

- Copy files from the project directory to the role's files folder:

  ```
  cp files/* roles/post-install/files/
  ```

- Copy templates from the project directory to the role's templates folder:

  ```
  cp templates/* roles/post-install/templates/
  ```

- Move variables to `vars/main.yml`:

  ```
  # vars file for post-install
  USRNM: commonuser
  COMM: variable from groupvars_all file
  ntp0: 0.north-america.pool.ntp.org
  ntp1: 1.north-america.pool.ntp.org
  ntp2: 2.north-america.pool.ntp.org
  ```

```
    ntp3: 3.north-america.pool.ntp.org
    mydir: /opt/dir22
```

Defining Handlers in Roles

Move handlers to `handlers/main.yml` under the role directory:

```
# handlers file for post-install
- name: reStart service on centos
  service:
    name: chronyd
    state: restarted
    enabled: yes
  when: ansible_distribution == "CentOS"

- name: reStart service on ubuntu
  service:
    name: ntp
    state: restarted
    enabled: yes
  when: ansible_distribution == "Ubuntu"
```

Using the Role in an Ansible Playbook

Once the role is defined, the playbook becomes simple. Use the `roles` directive to reference the role.

Example Playbook:

```
- name: Provisioning servers
  hosts: all
  become: yes
  roles:
    - post-install
```

In this playbook: - The `post-install` role is executed, and all tasks, handlers, and variables defined in the role will be applied.

Overriding Role Variables in a Playbook Variables defined in a role can be overridden at the playbook level. For example:

```
- name: Provisioning servers
  hosts: all
  become: yes
  roles:
    - role: post-install
```

```
    vars:
      ntp0: 0.in.pool.ntp.org
      ntp1: 1.in.pool.ntp.org
      ntp2: 2.in.pool.ntp.org
      ntp3: 3.in.pool.ntp.org
```

Here, the `ntp` variables defined in the playbook override those in the role.

Predefined Roles from Ansible Galaxy

Instead of manually creating roles, you can use predefined roles available in the Ansible Galaxy community.

Example:

```
- name: Provisioning servers
  hosts: all
  become: yes
  roles:
    - geerlingguy.java
    - role: post-install
      vars:
        ntp0: 0.in.pool.ntp.org
        ntp1: 1.in.pool.ntp.org
        ntp2: 2.in.pool.ntp.org
        ntp3: 3.in.pool.ntp.org
```

In this example, the `geerlingguy.java` role from Ansible Galaxy is executed first, followed by the `post-install` role. Predefined roles are helpful but may offer less customization than user-defined roles.

Organizing Variables in `defaults/main.yml`

While variables can be placed in `vars/main.yml`, it's a good practice to put default variables in `defaults/main.yml` for better modularity. Variables in `defaults/main.yml` have lower precedence compared to those in `vars/main.yml`.

Example Structure:

```
roles/
  post-install/
    defaults/
      main.yml
    vars/
      main.yml
```

```
tasks/
  main.yml
handlers/
  main.yml
files/
templates/
```

Use `include_vars` to import variables from other files into your main role.

Summary

- **Roles** allow for modular and reusable playbook structures.
- You can easily convert a playbook into a role using `ansible-galaxy init`.
- **Handlers** and **tasks** are organized under role directories.
- Variables can be overridden at the playbook level or declared in `defaults/main.yml` for best practices.
- Predefined roles from **Ansible Galaxy** are available, though user-defined roles offer more customization.

Ansible 205: Ansible for AWS

Ansible can effectively manage AWS services by automating tasks such as creating key pairs and launching EC2 instances. This section covers how to configure Ansible to interact with AWS, install necessary dependencies, and execute AWS-related tasks using Ansible playbooks.

---

1. Configuring Ansible to Connect to AWS To enable Ansible to manage AWS resources, you need to provide AWS credentials and ensure Ansible has the necessary libraries to interact with AWS APIs.

a. Create an IAM User with Access Keys

1. **Create IAM User:**
   - Navigate to the AWS Management Console.
   - Go to **IAM** > **Users** > **Add user**.
   - Assign a username and select **Programmatic access**.
2. **Generate Access Keys:**
   - Under **Security credentials**, create an access key.
   - **Download** the access keys as a CSV file for safekeeping.

b. Export AWS Credentials Set the AWS Access Key ID and Secret Access Key as environment variables so Ansible can use them to authenticate with AWS.

```
export AWS_ACCESS_KEY_ID='AK123'
export AWS_SECRET_ACCESS_KEY='abc123'
```

Note: These variables are session-specific. To make them persistent across sessions, add them to your `.bashrc` or `.bash_profile`:

```
echo "export AWS_ACCESS_KEY_ID='AK123'" >> ~/.bashrc
echo "export AWS_SECRET_ACCESS_KEY='abc123'" >> ~/.bashrc
source ~/.bashrc
```

---

2. Installing Required Python Libraries Ansible relies on the `boto3` library to interact with AWS services. Install `boto3` using `pip`.

```
sudo apt update
sudo apt install python3-pip -y
pip3 install boto3
```

---

3. Installing Ansible AWS Collection To access AWS modules, install the `amazon.aws` collection from Ansible Galaxy.

```
ansible-galaxy collection install amazon.aws
```

This collection includes modules for managing various AWS services, such as EC2 instances, S3 buckets, and more.

---

4. Creating an AWS Key Pair with Ansible Use Ansible to create an AWS key pair and save the private key locally.

```yaml
---
- name: Create AWS Key Pair
  hosts: localhost
  gather_facts: false
  tasks:
    - name: Create key pair
      amazon.aws.ec2_key:
        name: sample
        region: us-east-1
      register: keyout

    - name: Save private key
      copy:
        content: "{{ keyout.key.private_key }}"
        dest: ./sample.pem
```

```
    mode: '0600'
  when: keyout.changed
```

Explanation: - **Create key pair:** Uses the `ec2_key` module to create a key pair named `sample` in the `us-east-1` region. - **Save private key:** If a new key pair is created (`keyout.changed` is `true`), the private key is saved to `sample.pem` with restricted permissions.

---

5. Launching an EC2 Instance with Ansible Deploy an EC2 instance using the `amazon.aws.ec2_instance` module.

```
---
- name: Launch EC2 Instance
  hosts: localhost
  gather_facts: false
  tasks:
    - name: Start an EC2 instance
      amazon.aws.ec2_instance:
        name: "public-compute-instance"
        key_name: "sample"
        instance_type: t2.micro
        security_groups: ["default"]
        image_id: ami-02d8bad0a1da4b6fd
        exact_count: 1
        region: us-west-2
        tags:
          Environment: Testing
      register: ec2
```

Explanation: - **name:** Assigns a name to the EC2 instance. - **key_name:** Associates the instance with the previously created key pair (`sample`). - **instance_type:** Specifies the instance type (`t2.micro`). - **security_groups:** Associates the instance with the `default` security group. - **image_id:** Specifies the Amazon Machine Image (AMI) ID to use for the instance. - **exact_count:** Ensures that exactly one instance is running. Re-running the playbook won't create additional instances. - **region:** Specifies the AWS region (`us-west-2`) where the instance will be launched. - **tags:** Adds metadata to the instance for easier identification and management.

---

6. Best Practices

- **Persisting AWS Credentials:**
  - Instead of exporting credentials in `.bashrc`, consider using AWS credentials files (`~/.aws/credentials`) for better security and manage-

ability.

- **Using Variables:**
  - Define AWS-related variables (e.g., `region`, `ami_id`, `instance_type`) in Ansible variables files or inventories for flexibility.
- **Handling Sensitive Data:**
  - Use Ansible Vault to encrypt sensitive information like AWS access keys.
- **Idempotency:**
  - Utilize `exact_count` and other idempotent parameters to ensure playbooks can be safely re-run without unintended side effects.

---

7. Example Complete Playbook Combining the key pair creation and EC2 instance launch into a single playbook:

```yaml
---
- name: Provision AWS Resources
  hosts: localhost
  gather_facts: false
  tasks:
    - name: Create key pair
      amazon.aws.ec2_key:
        name: sample
        region: us-east-1
      register: keyout

    - name: Save private key
      copy:
        content: "{{ keyout.key.private_key }}"
        dest: ./sample.pem
        mode: '0600'
      when: keyout.changed

    - name: Start an EC2 instance
      amazon.aws.ec2_instance:
        name: "public-compute-instance"
        key_name: "sample"
        instance_type: t2.micro
        security_groups: ["default"]
        image_id: ami-02d8bad0a1da4b6fd
        exact_count: 1
        region: us-west-2
        tags:
          Environment: Testing
      register: ec2
```

Usage: 1. **Run the Playbook:** `bash ansible-playbook provision_aws.yml` 2. **Verify Resources:** - Check the AWS Management Console to ensure the key pair and EC2 instance have been created as specified.

Summary

- **AWS Integration:** Configure Ansible with AWS credentials to manage AWS resources.
- **Dependencies:** Install `boto3` and the `amazon.aws` Ansible collection for AWS module support.
- **Key Pair Management:** Use Ansible to create and manage AWS key pairs securely.
- **EC2 Instance Management:** Automate the deployment of EC2 instances with specified configurations.
- **Best Practices:** Ensure idempotency, secure handling of credentials, and modular playbook design for scalable and maintainable infrastructure automation.

By leveraging Ansible's capabilities to interact with AWS, you can streamline and automate the provisioning and management of your cloud infrastructure efficiently.

Containerization with Docker

Containerization is the process of packaging an application along with its dependencies, configurations, and environment settings into a single container, making deployment consistent across environments (e.g., dev, QA, prod). This approach simplifies resource management, reduces deployment errors, and supports microservice architectures by using Docker as the primary tool for containerization.

---

1. Introduction to Containerization

Containerization addresses several issues that arise from traditional VM-based deployments:

- **Manual Deployment Issues:**
  - High CAPEX/OPEX due to manual resource setup.
  - Deployment inconsistency due to human errors.
  - Resource wastage from hosting multiple services, each requiring its own operating system.
- **Microservice Architecture Support:**
  - Containers share the same OS kernel, minimizing resource usage compared to running VMs for each service.

– Containerized applications maintain synchronization across different environments (e.g., dev, prod, QA), avoiding deployment failures.

---

2. Why Docker?

Docker simplifies the containerization process by allowing developers to package applications into Docker images, which can then be deployed consistently across different environments.

- **Docker Images:** Self-contained units that include all necessary components (application code, libraries, environment variables).
- **Docker Containers:** Instances of Docker images running on any machine with Docker installed.
- **Docker Compose:** A tool to manage multi-container applications by defining services in a `docker-compose.yml` file.

Example Workflow:

1. **Pull code from GitHub**: Retrieve the application code, which includes a `Dockerfile` for building the Docker image.
2. **Build Docker Image**:

   ```
   docker build -t imranvisualpath/vproapp .
   ```

3. **Push Image to DockerHub**:

   ```
   docker push imranvisualpath/vproapp
   ```

4. **Run Containers on Multiple Servers**: Use `docker-compose` to spin up containers across multiple servers.

---

3. Using Dockerfile for Customization

A `Dockerfile` defines the steps to build a custom Docker image by pulling base images and installing additional dependencies.

Example of a Simple Dockerfile:

```
# Use base image
FROM ubuntu:latest

# Install necessary packages
RUN apt-get update && apt-get install -y \
    python3 \
    pip
```

```
# Copy application code
COPY . /app

# Set the working directory
WORKDIR /app

# Run the application
CMD ["python3", "app.py"]
```

---

4. Docker Compose for Multi-Container Deployment

`docker-compose.yml` is used to define and manage multiple containers that work together as part of a larger application.

Example `docker-compose.yml`:

```
version: '3'
services:
  web:
    image: myapp/web
    ports:
      - "8080:8080"
  database:
    image: postgres
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
```

---

5. DockerHub Setup

DockerHub is a platform to host and share Docker images publicly or privately. Organizations often use DockerHub for collaboration and version control.

- **Benefits of DockerHub Organization:**
  - Collaboration features for team-based Docker image development.
  - Unlimited private repositories.
  - Up to 5000 image pulls per day.
  - Enhanced container isolation and security.

You can link DockerHub repositories with GitHub or Bitbucket to automatically trigger Docker image builds upon code changes.

---

6. Setting Up Docker Engine on Vagrant

Vagrant is used to manage virtual environments, and you can configure it to run Docker containers by setting up a virtual machine (VM) and installing Docker on it.

a. Configure Vagrantfile To set up networking for the VM, edit the `Vagrantfile` to define private and public network settings:

```
# Configure private network
config.vm.network "private_network", ip: "192.168.56.38"

# Configure public network (bridged)
config.vm.network "public_network"
```

b. Allocate RAM for VirtualBox

```
config.vm.provider "virtualbox" do |vb|
  vb.memory = "2048"
end
```

Allocating 2GB of RAM ensures better VM performance compared to 1GB, which may result in a slower experience.

c. Launching Vagrant VM

1. Initialize the Vagrant environment:

   ```
   vagrant init bento/ubuntu-22.04
   ```

2. Launch the VM:

   ```
   vagrant up
   ```

d. Install Docker on Vagrant VM After launching the VM, follow Docker's installation guide for Ubuntu:

- Docker Engine on Ubuntu Installation

e. Add Vagrant User to Docker Group To run Docker commands inside the VM, add the Vagrant user to the Docker group:

```
usermod -aG docker vagrant
```

Then, logout and re-login to apply the changes. Verify Docker installation by running:

```
docker images
```

---

7. Base Image Setup

For some services, DockerHub provides ready-made images, while others might require customization. For example, to run a Java application on Tomcat, you might need to create a custom Docker image that includes your application artifact.

Summary

- **Containerization** allows consistent, scalable, and resource-efficient deployment across multiple environments.
- **Docker** is the go-to tool for building and running containers using Docker images.
- **Docker Compose** simplifies multi-container setups by managing services with configuration files.
- **Vagrant** can be used to create virtual environments for Docker testing and deployment.
- **DockerHub** offers a platform for sharing Docker images, enabling collaboration among developers.

Containerization is a powerful technique for modern application development, enabling microservice architectures and continuous deployment strategies.

DockerHub & Dockerfile Overview

In this section, we explore the key concepts of DockerHub, Dockerfile, and multi-stage builds, essential for building and optimizing Docker images.

---

1. DockerHub and Image Management

DockerHub is the primary registry for storing and sharing Docker images. To store images in DockerHub, you first create a repository under your username or organization name. When pulling or pushing images, the format used is:

```
docker pull namespace/repository_name:tagname
```

- **Namespace**: Generally, your DockerHub username or organization name.
- **Repository Name**: The name of the image (e.g., `vprofileweb`).
- **Tag Name**: Used to reference different versions of the image (default is `latest`).

Forking in GitHub Forking copies an entire repository, including all branches, from one GitHub account to another. This is useful for making independent modifications.

---

2. Key Dockerfile Instructions

A `Dockerfile` is a set of instructions for building Docker images. Here are the most important instructions used:

- **FROM**: Defines the base image for the Docker image.
- **RUN**: Executes commands during the image build process (e.g., installing software).
- **COPY**: Copies files from the local machine into the container.
- **CMD**: Specifies the default command to run when a container starts.
- **LABEL**: Adds metadata to the image in key-value format.
- **EXPOSE**: Informs Docker that the container listens on specific ports (does not publish the port).
- **ENV**: Sets environment variables in the image.
- **ADD**: Similar to `COPY`, but with additional features like fetching files from URLs or extracting compressed files.
- **ENTRYPOINT**: Specifies the executable that will always run, and takes precedence over `CMD`.

Example Dockerfile:

```
FROM ubuntu:latest
RUN apt-get update && apt-get install -y python3
COPY . /app
WORKDIR /app
EXPOSE 5000
CMD ["python3", "app.py"]
```

- **EXPOSE vs `-p`**: The `EXPOSE` command documents the ports, while the `-p` flag in `docker run` is required to make the port accessible outside the container.

Difference Between `ADD` and `COPY`

- **COPY** is simpler and just copies files or directories.
- **ADD** provides extra functionality, such as extracting archives and fetching files from URLs.

---

3. Multi-Stage Builds

Multi-stage builds allow the creation of smaller, more efficient Docker images by separating the build process into different stages. This approach is beneficial for:

- **Smaller Final Images**: Only essential components are included in the final image, reducing its size.

- **Parallel Build Steps**: Build steps can be executed in parallel, improving efficiency.

Example Multi-Stage Dockerfile:

```
# Stage 1: Build the application
FROM openjdk:11 AS BUILD_IMAGE
RUN apt update && apt install maven -y
RUN git clone https://github.com/devopshydclub/vprofile-project.git
RUN cd vprofile-project && git checkout docker && mvn install

# Stage 2: Deploy the application on Tomcat
FROM tomcat:9-jre11
RUN rm -rf /usr/local/tomcat/webapps/*
COPY --from=BUILD_IMAGE vprofile-project/target/vprofile-v2.war /usr/local/tomcat/webapps/R
EXPOSE 8080
CMD ["catalina.sh", "run"]
```

- **Stage 1**: Uses `openjdk` as the base image to build the Java application with Maven. This stage generates the artifact (`vprofile-v2.war`).
- **Stage 2**: Uses `tomcat:9-jre11` as the base image to deploy the artifact. The default Tomcat web app is removed, and the new artifact is copied into the appropriate location.

The multi-stage build helps eliminate unnecessary build tools and dependencies from the final image, making it smaller and faster to deploy.

---

4. Dockerfile Best Practices

- **Smaller Images**: Use multi-stage builds to minimize the size of the final image.
- **Layer Caching**: Place commands that change infrequently (e.g., installing system packages) near the top of the Dockerfile to take advantage of Docker's layer caching.
- **Organized Commands**: Keep the Dockerfile clean and organized by grouping related commands together.

Summary

- **DockerHub** allows you to store, manage, and share Docker images.
- A **Dockerfile** is a set of instructions to build images, including commands like `FROM`, `RUN`, `COPY`, `CMD`, `EXPOSE`, `ENV`, and `ADD`.
- **Multi-stage builds** optimize Docker images by separating the build process into multiple stages, reducing the final image size and improving efficiency.

- Best practices include using smaller base images, taking advantage of layer caching, and organizing the Dockerfile for better maintainability.

Web, App, and Database Dockerfiles

In this section, we define Dockerfiles for the web, application, and database services. These Dockerfiles are essential components in containerizing the different parts of an application.

Web Dockerfile (`web/Dockerfile`) This Dockerfile is used to configure the web server (Nginx) for the application.

```
# Use Nginx as the base image
FROM nginx

# Add labels for project and author information
LABEL "Project"="Vprofile"
LABEL "Author"="Imran"

# Remove the default Nginx configuration
RUN rm -rf /etc/nginx/conf.d/default.conf

# Copy custom Nginx configuration into the container
COPY nginvproapp.conf /etc/nginx/conf.d/vproapp.conf
```

App Dockerfile (`app/Dockerfile`) This Dockerfile sets up a Tomcat-based application server with the necessary web application artifact.

```
# Use Tomcat as the base image
FROM tomcat:8-jre11

# Add labels for project and author information
LABEL "Project"="Vprofile"
LABEL "Author"="Imran"

# Remove the default Tomcat web applications
RUN rm -rf /usr/local/tomcat/webapps/*

# Copy the WAR file to the Tomcat webapps directory
COPY target/vprofile-v2.war /usr/local/tomcat/webapps/ROOT.war

# Expose the application on port 8080
EXPOSE 8080

# Start Tomcat
CMD ["catalina.sh", "run"]
```

```
# Set working directory and define a volume
WORKDIR /usr/local/tomcat/
VOLUME /usr/local/tomcat/webapps
```

Database Dockerfile (`db/Dockerfile`) This Dockerfile sets up a MySQL database container with an initial database and credentials.

```
# Use MySQL as the base image
FROM mysql:5.7.25

# Add labels for project and author information
LABEL "Project"="Vprofile"
LABEL "Author"="Imran"

# Set environment variables for MySQL
ENV MYSQL_ROOT_PASSWORD="vprodbpass"
ENV MYSQL_DATABASE="accounts"

# Add the database backup to the initialization directory
ADD db_backup.sql docker-entrypoint-initdb.d/db_backup.sql
```

Docker Compose: Managing Multiple Containers

Docker Compose allows you to build and run multiple containers simultaneously. This is useful for running complex applications that require multiple services, such as a web server, app server, and database.

Basic Docker Compose File Here is an example `docker-compose.yml` file that builds and runs several containers for a typical application setup.

```
version: '3.8'
services:
  vprodb:
    build:
      context: ./Docker-files/db
    image: vprocontainers/vprofiledb
    container_name: vprodb
    ports:
      - "3306:3306"
    volumes:
      - vprodbdata:/var/lib/mysql
    environment:
      - MYSQL_ROOT_PASSWORD=vprodbpass

  vprocache01:
```

```
    image: memcached
    ports:
      - "11211:11211"

  vpromq01:
    image: rabbitmq
    ports:
      - "15672:15672"
    environment:
      - RABBITMQ_DEFAULT_USER=guest
      - RABBITMQ_DEFAULT_PASS=guest

  vproapp:
    build:
      context: ./Docker-files/app
    image: vprocontainers/vprofileapp
    container_name: vproapp
    ports:
      - "8080:8080"
    volumes:
      - vproappdata:/usr/local/tomcat/webapps

  vproweb:
    build:
      context: ./Docker-files/web
    image: vprocontainers/vprofileweb
    container_name: vproweb
    ports:
      - "80:80"

volumes:
  vprodbdata: {}
  vproappdata: {}
```

- **Services**: Defines individual containers such as `vprodb` (MySQL), `vprocache01` (Memcached), `vpromq01` (RabbitMQ), `vproapp` (Tomcat), and `vproweb` (Nginx).
- **Ports**: Maps internal container ports to external ports on the host machine.
- **Volumes**: Maps host directories or data volumes to directories within the container.
- **Environment Variables**: Passes environment variables into the containers, such as database credentials.

Docker Compose Commands

- **Start Services**: To build and start all services in the `docker-compose.yml` file, use the following command:

  ```
  docker-compose up -d
  ```

  The `-d` flag runs the services in detached mode (in the background).

- **Stop Services**: To stop and remove all containers defined in the compose file:

  ```
  docker-compose down
  ```

- **View Running Containers**: To see which containers are running:

  ```
  docker-compose ps
  ```

  This only shows containers created using Docker Compose. For all running containers, use:

  ```
  docker ps
  ```

Building and Running the Application

- **Build Images**: To build all the images from the Dockerfiles specified in the compose file:

  ```
  docker-compose build
  ```

- **Run the Application**: After building, use the `docker-compose up` command to start the application:

  ```
  docker-compose up -d
  ```

- **Push Images to DockerHub**: To push a Docker image to DockerHub:

  ```
  docker login
  docker push <your-image-name>
  ```

- **Clean Up**: To remove stopped containers, unused networks, and dangling images, use:

  ```
  docker system prune -a
  ```

Summary

- The **Web, App, and Database Dockerfiles** define the configuration and setup for Nginx, Tomcat, and MySQL containers, respectively.

- **Docker Compose** simplifies the management of multi-container applications, allowing for easy builds, network setup, volume management, and environment configuration.
- Using **docker-compose commands** such as `up`, `down`, and `build`, you can efficiently manage the lifecycle of containers and optimize the deployment process.