Week - 7 Ansible

Introduction to Ansible and Configuration Management (CM):

- Defining Ansible and Understanding the Need for Configuration Management (CM).
- In-Depth Exploration of Ansible Architecture:
- Analyzing the Architecture of Ansible for Efficient Configuration Management.
- Detailed Steps for Installing and Setting Up Ansible.
- Exploring Essential Components such as Ansible Roles, Ansible Collections, Ad-hoc Commands, and Playbooks Setup.

Automation with Ansible Playbooks:

- Creating Playbooks for Automation.
- Building Playbooks to Copy Files with Special Variables.
- Utilizing Ansible Handlers and Notifiers for Effective Automation.
- Implementing Playbooks for Downloading Artifacts and Unzipping Files.

Advanced Automation Scenarios with Ansible:

- Leveraging Ansible Tags for Targeted Deployment to Servers.
- Automating the Installation of Apache and Configuring the Corresponding Configuration Files
- Configure Multi node k8s cluster with Ansible
- Manage Variable and Ansible Facts

Ansible

- Initializing Ansible Configuration
  To initialize an Ansible configuration file, use the following command:
  #ansible-config init --disabled > ansible.cfg

  This command initializes a new ansible.cfg file in the current directory, disabling all configurations by default. You can edit this file to customize Ansible's behavior for your project.

- Managing SSH Keys
  When managing SSH keys for Ansible, ensure the private key (key.pem in this case) is secured properly: *#chmod 400 key.pem*

  This command restricts permissions on key.pem to be readable only by the owner, enhancing security.

- Executing Ad-hoc Commands
  Ad-hoc commands in Ansible allow you to perform quick tasks on remote hosts. For example, to display the current date on all hosts: *#ansible all -m command -a "date"*

Explanation:

- all: Targets all hosts defined in your Ansible inventory.
- -m command: Specifies the module to use (command in this case).
- -a "date": Passes the argument to the module (in this case, the date command).

- Running Ansible Playbooks
  Playbooks in Ansible are YAML files that define a set of tasks to be executed on managed hosts. To run a playbook*: ansible-playbook <playbook-name>.yml*

  Replace <playbook-name> with the actual name of your playbook file (playbook.yml for example). Playbooks offer a structured way to automate tasks and configurations across multiple servers.

- Jinja Templating in Ansible
  Ansible uses Jinja2 templating engine for dynamic expressions and template rendering within playbooks and templates. For detailed documentation on Jinja templates, refer to:

Why Ansible

Managing configurations and deployments across multiple servers manually is time-consuming and error-prone. Ansible automates these tasks, ensuring consistency and efficiency in server management.

What is Ansible

Ansible is an open-source automation tool that simplifies software provisioning, configuration management, and application deployment.

Advantages of Ansible

- Declarative: Ansible allows us to specify the desired state of the system, and it ensures that the system configuration matches that state.
- Agentless: No agent needs to be installed on the managed nodes, simplifying deployment and reducing overhead.
- Idempotent: Running Ansible playbooks multiple times yields the same result, making it safe and predictable.

Ansible is written in Python and uses YAML syntax for defining playbooks.

Command to install Ansible on Amazon Linux 3: *#yum install ansible-core -y*
This command installs Ansible on Amazon Linux 3 using yum.

Ansible Configuration File

The default Ansible configuration file is located at /etc/ansible/ansible.cfg.
To initialize a new configuration file in the current directory, use:

```
#ansible-config init --disabled > ansible.cfg
```

Customize ansible.cfg to tailor Ansible's behavior, such as defining the inventory location or disabling host key checking.

Inventory File in Ansible

The inventory file (hosts) lists all servers (target nodes) Ansible manages.
By default, it resides at /etc/ansible/hosts.
Specify a custom inventory file location in ansible.cfg:

```
inventory = ./inventory
```

```
172.31.33.111 ansible_user=ec2-user ansible_ssh_private_key_file=key.pem
```

Additional ansible.cfg Options

host_key_checking = False: Disables host key checking, useful for automated setups.
become, become_user, become_method: Configures privilege escalation settings.

Ansible Playbook
Playbooks are YAML files where tasks for remote servers are defined.

```
- hosts: all
  tasks:

    - name: Installing HTTPD Package
      package:
        name: httpd
        state: present

    - name: Copying Files from Localhost to Remote
      copy:
        src: index.html
        dest: /var/www/html
        owner: root
        mode: '666'

    - name: Starting HTTPD Service
      service:
        name: httpd
        state: started
```

Explanation:

This playbook installs the httpd package, copies index.html from the localhost to remote servers, and ensures the httpd service is started.

Creating EC2 instances using Ansible

An Ansible playbook (ec2.yml) is used to define tasks for provisioning EC2 instances. Below is an expanded explanation of the playbook:

- We can use the `amazon.aws.ec2_instance` module to create the EC2 instances using Ansible.
- Installing the AWS Collection
    - The amazon.aws collection includes Ansible modules specifically designed to interact with AWS services. Install it using Ansible Galaxy:
    - *ansible-galaxy collection install amazon.aws*
    - This command downloads and installs the AWS collection, making AWS-specific modules available for use in Ansible playbooks.

ec2.yml

```
- hosts: localhost
  vars_files:
    - credentials.yml
    - ec2-config.yml
  tasks:
    - package:
        name: python3-pip
        state: present

    - pip:
        name: boto3

    - amazon.aws.ec2_instance:
        aws_access_key: "{{ access_key }}"
        aws_secret_key: "{{ secret_access_key }}"
        region: "{{ region }}"
        image_id: "{{ ami_id }}"
        instance_type: "{{ instance_type }}"
        security_group: "{{ sg }}"
        count: "{{ count }}"
        key_name: "{{ key_pair }}"
        tags:
          name: ansible-demo
      register: ec2

    - debug:
        var: ec2
    - set_fact:
        private_ip: "{{ ec2.instances | map(attribute='private_ip_address') | list }}"
```

```
- debug:
    var: private_ip

- template:
    src: inventory.j2
    dest: inventory
```

Explanation of the Playbook:

- hosts: localhost: Specifies that tasks are executed on the local machine where Ansible is running (localhost).

- vars_files: Loads variables from external YAML files (credentials.yml and ec2-config.yml):

- credentials.yml: Contains AWS access credentials (access_key and secret_access_key).

- ec2-config.yml: Defines configuration parameters for EC2 instances (e.g., region, instance_type, ami_id, key_pair, sg, count).

- Tasks:
    - Install Python3 and pip: Ensures Python3 and pip are installed on the local machine.
    - Install boto3 library: Uses pip to install the boto3 library, a dependency for interacting with AWS via Python.
    - Uses the amazon.aws.ec2_instance module to provision EC2 instances on AWS.
        - Parameters:
            - aws_access_key and aws_secret_key: AWS credentials to authenticate with AWS.
            - region: AWS region where instances will be created.
            - image_id: ID of the Amazon Machine Image (AMI) used for the instances.
            - instance_type: Type of EC2 instance to create (e.g., t2.micro).
            - security_group: Security group for the instances.
            - count: Number of instances to create.
            - key_name: Name of the SSH key pair used to access instances.
            - tags: Optional tags to apply to the instances.
            - register: ec2_instances: Stores the output (instance information) in the ec2_instances variable.

    - debug: Displays the content of ec2_instances, providing details about the created instances.
    - set_fact: Sets the private_ips variable to store a list of private IP addresses of the created instances.

- template: Generates a dynamic inventory file (inventory) based on the private IP addresses of the created instances, using the Jinja2 template inventory.j2.

- In the above playbook, we used one option like vars_files which allows us to define the variables in the separate files and use them in the playbook. We can define the variables in the `credentials.yml` and `ec2-config.yml` files and use them in the playbook.

Variables Files

- Let's create the `credentials.yml` file and define the variables in it.

  ```
  access_key: Your_Access_Key
  secret_access_key: Your_Secret_Access_Key
  ```

- Let's create the `ec2-config.yml` file and define the variables in it.

  ```
  region: us-east-1
  instance_type: t2.micro
  count: 0
  ami_id: ami-0e731c8a588258d0d
  key_pair: key
  sg: launch-wizard-13
  ```

Dynamic Inventory in Ansible

Ansible allows for dynamic inventory management, which means you can generate inventory information on the fly rather than statically defining it in a file. This is particularly useful in cloud environments where instances are frequently created, terminated, or scaled.

After creating these instances, it's common to want Ansible to immediately start managing these instances without manually updating an inventory file.

Using the template Module for Dynamic Inventory
In the playbook (ec2.yml), the template module is used to generate a dynamic inventory file (inventory) based on the private IP addresses of the created EC2 instances.

Step 1: template Module Usage

```
{% for ip in private_ip %}
{{ ip }}
{% endfor %}
```

Explanation:

- Jinja2 Templating: Ansible uses Jinja2 templates for dynamic content generation.
- {% for ip in private_ips %}: Iterates over the private_ips variable, which contains the private IP addresses of the newly created EC2 instances.
- {{ ip }}: Outputs each private IP address in the inventory file.

Running the playbook : *#ansible-playbook ec2.yml*

Load Balancer using Ansible

A load balancer is a crucial component in computer networks and server architecture designed to efficiently distribute incoming network or application traffic across multiple servers. This distribution helps in optimizing resource utilization, maximizing throughput, minimizing response time, and ensuring high availability of services by preventing any single server from becoming overwhelmed or failing due to excessive load.

HAProxy (High Availability Proxy) is a popular open-source software load balancer and proxy server that operates at the application layer (Layer 7) of the OSI model.

Step 1: Installing HAProxy
First, create a playbook (load-balancer.yml) to install HAProxy on your localhost:

```
- hosts: localhost
  tasks:
    - name: Install HAProxy package
      package:
        name: haproxy
        state: present
```

Explanation:

- The package module is used to ensure that the haproxy package is installed (state: present).
- This playbook targets localhost because HAProxy will run locally as a load balancer.

Running the Playbook
Execute the playbook to install HAProxy:
#ansible-playbook load-balancer.yml

Step 2: Configuring HAProxy
After installing HAProxy, you need to configure it to act as a load balancer for your backend servers. Normally, the HAProxy configuration file (haproxy.cfg) is located at /etc/haproxy/haproxy.cfg.

Example Template HAProxy Configuration file is present here -
https://github.com/sudhanshuvlog/GFG-Devops18/blob/main/Ansible/haproxy.j2

Explanation:

- Global and Defaults: Sets global and default options for HAProxy.
- Frontend and Backend: Defines how HAProxy should handle incoming requests and route them to backend servers.
- {% for ip in private_ips %}: Jinja2 loop to dynamically generate backend server entries based on private_ips.
- server app{{ loop.index }} {{ ip }}:80 check: Defines each backend server (app1, app2, etc.) with its private IP and port (:80).

Step 3: Dynamic IP Update in HAProxy Configuration

To dynamically update the HAProxy configuration with backend server IPs, modify your Ansible playbook (load-balancer.yml):

```
- hosts: localhost
  vars_files:
    - credentials.yml
    - ec2-config.yml
  tasks:
    - package:
        name: haproxy
        state: present

    - amazon.aws.ec2_instance_info:
        aws_access_key: "{{ access_key }}"
        aws_secret_key: "{{ secret_access_key }}"
        region: "{{ region }}"
        filters:
          "tag:Name": "ansible-demo"
          instance-state-name: ["running"]
      register: ec2

    - set_fact:
        private_ips: "{{ ec2.instances | map(attribute='private_ip_address') | list }}"

    - template:
        src: haproxy.j2
        dest: /etc/haproxy/haproxy.cfg

    - service:
        name: haproxy
        state: restarted
```

Explanation:

- amazon.aws.ec2_instance_info: Ansible module to fetch information about EC2 instances tagged with ansible-demo and running state.
- set_fact: Stores the list of private IPs (private_ips) of EC2 instances in a variable.
- template: Updates the HAProxy configuration file (haproxy.cfg) using a Jinja2 template (haproxy.j2).
- service: Restarts the HAProxy service to apply the updated configuration.

Ansible roles

- Roles let you automatically load related vars, files, tasks, handlers, and other Ansible artifacts based on a known file structure. After you group your content into roles, you can easily reuse them and share them with other users.
- Run `ansible-galaxy init role-name` to create a new role.
- After running the above command, You can see that a new directory called `role-name` is created with the following structure.

```
role-name
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml
```

- We just need the arrange the files in the above structure and then we can use the role in our playbook.

Ansible Vault

- Ansible Vault is a feature of Ansible that allows you to keep sensitive data such as passwords or keys in encrypted files, rather than as plaintext in your playbooks or roles.
- You can use ansible-vault to encrypt the file and then you can use the file in your playbook.
- Run `ansible-vault create file-name` to create a new encrypted file. It will ask for a password to encrypt the file.

- You can also put that password in another file and use that file to encrypt the file.
- Run `ansible-vault create file_name --vault-password-file file_name_where_your_passwdIsStored` to encrypt the file.
- If we used any encrypted file in our playbook, We need to use `ansible-playbook --vault-password-file playbook.yml` to run the playbook.

- **Note**: You can also use `ansible-vault edit file_name` to edit the encrypted file.

K8s MultiNode Cluster Configuration With Ansible

Follow the Github -
https://github.com/sudhanshuvlog/kubernetes-multinode-cluster-ansible-aws

Overview of rhel_common.yml

Basic Overview of --> rhel_common.yaml

- Disable Swap: Swapping is disabled using the swapoff -a command to prevent Kubernetes from using swap space.

- Add Swapoff to Crontab for Reboot: This step ensures that swap remains disabled even after a system reboot by adding swapoff -a to the crontab to run at reboot.

- Install Required Packages: Installs necessary packages like iproute-tc and git using the DNF package manager.

- Load Kernel Modules: Loads kernel modules required for Kubernetes, specifically overlay and br_netfilter.

- Configure Kernel Modules: Configures kernel modules by creating configuration files in /etc/modules-load.d/k8s.conf and /etc/sysctl.d/k8s.conf to set up required network settings.

- Apply Sysctl Settings: Applies sysctl settings to the system.

- Disable SELinux: Disables SELinux by setting it to permissive mode.

- Add Kubernetes Repository: Adds repositories for Kubernetes and CRI-O, enabling the system to install packages from those sources.

- Install Kubernetes Packages: Installs Kubernetes-related packages like cri-o, kubelet, kubeadm, kubectl, and cri-tools using the DNF package manager.

- Enable and Start CRI-O Service: Enables and starts the CRI-O service, which is the container runtime for Kubernetes.

- Enable and Start Kubelet Service: Enables and starts the Kubelet service, which is responsible for managing the state of the Kubernetes node.

Basic Overview of --> rhel_master.yaml
For K8s Master Node Configuration:

- Get the Master Node Private IP: Sets the MASTER_PRIVATE_IP variable having the private IP address of the master node using shell commands.

- Define the POD_CIDR Variable: Sets the POD_CIDR variable to 192.168.0.0/16 in the shell.

- Define the Master Node Name: Sets the NODENAME variable to the short hostname of the master node using shell commands.

- Initialize the Kubernetes Cluster: Initializes the Kubernetes cluster on the master node using kubeadm init command with various options such as --apiserver-advertise-address, --apiserver-cert-extra-sans, --pod-network-cidr, --node-name, and --ignore-preflight-errors Swap.

- Output of the Kubeadm Init Command: Prints the output of the kubeadm init command for debugging purposes.

- Create .kube Directory: Ensures the .kube directory exists in the user's home directory.

- Copy Admin.conf to User's .kube Directory: Copies the admin.conf file from /etc/kubernetes to the user's .kube/config directory.

- Change Ownership of .kube/config: Changes the ownership of the config file to the user.

- Configure the CNI using Calico: Applies the Calico CNI (Container Network Interface) configuration using kubectl apply command.

- Generate Join Command: Generates the join command for worker nodes to join the cluster using kubeadm token create --print-join-command.

- Copy Join Command to Local File: Copies the join command to a local file named join-command.

For Worker Node Configuration:

- Copy Join Command to Server Location: Copies the join command file from the local machine to the /tmp directory on the worker node.

- Join the Node to Cluster: Executes the join command script to join the worker node to the Kubernetes cluster.

Tags

Tags in Ansible allow you to categorize tasks and selectively run them using the --tags or --skip-tags options from the command line. Tags are useful for running specific parts of a playbook or skipping certain tasks altogether.

Handlers
Handlers in Ansible are special tasks that are only executed when notified by other tasks in the playbook. They are typically used to restart services or perform other actions that should only occur when certain changes have been made.

Handlers are automatically triggered by tasks that notify them, ensuring that critical actions like service restarts occur only when necessary. Tags provide flexibility in playbook execution, allowing you to focus on specific parts of a playbook based on their categorization. Together, they enhance the control and manageability of Ansible playbooks in various deployment scenarios.

```
- hosts: localhost
  tasks:
  - package:
      name: httpd
      state: absent
    tags:
      - uninstall
  - copy:
      src: index.html
      dest: /var/www/html
  - copy:
      src: httpd.conf
      dest: /etc/httpd/conf/
    notify:
      - Restart apache
    tags:
      - configuration
  handlers:
    - name: Restart apache
      service:
        name: httpd
        state: restarted
```

Explanation:

- handlers define a list of handler tasks.
- name: Restart apache is the name of the handler.

- service module is used to manage services.
- name: httpd specifies the name of the service to manage.
- state: restarted ensures the service (Apache in this case) is restarted.
- tags: uninstall on the first task (package module) means you can run only this task with --tags uninstall.