DevOps Shack

Ansible Notes

1. Introduction to Ansible

What is Ansible?

Ansible is an open-source automation platform used for IT orchestration, configuration management, and application deployment. It's particularly popular for DevOps workflows, enabling administrators to automate daily tasks across various servers in a reliable and repeatable manner. Unlike traditional configuration management tools like Puppet or Chef, Ansible is agentless, meaning that no software needs to be installed on the systems being managed, apart from SSH for Linux/Unix or WinRM for Windows.

Ansible was designed with simplicity in mind, using a declarative approach to configuration management, so instead of scripting every step of the configuration, you describe the desired state of your infrastructure, and Ansible makes sure it gets there.

Why is Ansible Popular?

- **Agentless:** No need to install agents on the systems being managed. Ansible uses native SSH or WinRM to communicate with servers.
- **Declarative and Idempotent:** You describe the desired state, and Ansible ensures that this state is achieved in an idempotent way, meaning running the playbook multiple times produces the same result.
- **Simplicity:** YAML-based playbooks are human-readable and easy to understand. It's an automation language that's simple to write and can be understood even by those who are new to automation.
- **Modular Architecture:** Ansible modules allow easy customization and extensions to the platform. It supports out-of-the-box modules and allows users to write their own.

Core Components of Ansible:

- **Playbooks:** YAML files that describe the set of steps or configurations you want to apply to your systems.
- **Modules:** Reusable code that Ansible runs on managed hosts. These are the units of action in Ansible, from installing packages to restarting services.
- **Inventory:** A list of hosts or servers that Ansible manages. It can be static (defined in a file) or dynamic (fetched from a cloud provider or external source).

- **Roles:** A way to organize playbooks into reusable components, often used to manage complex configurations.

---

2. Core Concepts of Ansible

To fully understand Ansible, you need to get familiar with its core concepts, such as **control nodes**, **managed nodes**, **inventory**, and **modules**.

Control Node vs. Managed Nodes

- **Control Node:** The machine where Ansible is installed. This node sends commands to managed nodes via SSH or WinRM.
    - **Example**: A control node could be your laptop or a server from which you run all your automation tasks.
- **Managed Nodes:** These are the machines that Ansible will be controlling. These can be servers, VMs, cloud instances, or even network devices like switches.
    - **Example**: All the servers in your infrastructure (like a web server, database server, or load balancer) are managed nodes.

Ansible Inventory

An inventory defines the list of hosts or nodes Ansible will manage. These nodes can be grouped together for organizational purposes. Inventory can either be **static** or **dynamic**.

Static Inventory Example

A static inventory is a simple text file that lists hosts and organizes them into groups:

```
[webservers]
web01.example.com
web02.example.com

[dbservers]
db01.example.com
db02.example.com

[all:vars]
ansible_user=ubuntu
ansible_ssh_private_key_file=~/.ssh/id_rsa
```

Dynamic Inventory

Dynamic inventory scripts query infrastructure services (like AWS, GCP, or Azure) in real-time to get an updated list of hosts.

For AWS EC2, you can use a dynamic inventory file like:

```
plugin: aws_ec2
regions:
  - us-west-2
filters:
  instance-state-name: running
```

Ansible Modules

Modules are the actual units of work in Ansible. They are executed on the managed nodes and perform actions such as installing packages, copying files, or managing services.

How Modules Work

Modules are transferred to the managed nodes over SSH (or WinRM for Windows), executed locally, and then return the results to the control node. The results can indicate success, failure, or changes made.

Popular Built-in Modules

1. **Package Management Modules**:
    - apt: For managing packages on Debian-based systems (Ubuntu).
    - yum: For managing packages on RedHat-based systems (RHEL/CentOS).

Example:

```
- name: Install NGINX on Ubuntu
  apt:
    name: nginx
    state: present
```

2. **Service Management**:
    - service: For managing services (start/stop/restart).

Example:

```
- name: Ensure NGINX is running
  service:
    name: nginx
    state: started
```

3. **File/Directory Management**:
    - file: For managing file permissions, ownership, and state.
    - copy: For copying files from the control node to the managed node.

Example:

```
- name: Ensure a directory exists
  file:
    path: /var/www/html
    state: directory
    mode: '0755'
```

4. **User and Group Management**:
    - user: For managing users on a system.

Example:

```
- name: Create a new user
  user:
    name: john
    state: present
    groups: sudo
```

---

3. Installation and Configuration

Step-by-Step Installation of Ansible

You can install Ansible on various platforms like Linux, macOS, or even Windows (via WSL). Here are step-by-step instructions for different platforms.

Installing Ansible on Ubuntu/Debian

```
sudo apt update
```

```
sudo apt install ansible -y
```

Installing Ansible on CentOS/RHEL

First, enable the **EPEL** repository, then install Ansible:

```
sudo yum install epel-release -y
```

```
sudo yum install ansible -y
```

Installing Ansible on macOS (Using Homebrew)

```
brew install ansible
```

Verifying Installation

Once installed, check the version to ensure Ansible is installed correctly:

```
ansible --version
```

You should see output displaying the installed version and path.

Configuration Using ansible.cfg

Ansible uses a configuration file (ansible.cfg) to control its behavior. This file can be placed in your project directory, home directory, or system-wide in /etc/ansible/.

Sample ansible.cfg File:

```
[defaults]
inventory = ./inventory          # Path to your inventory file
remote_user = ubuntu             # Default SSH user
host_key_checking = False        # Disable SSH key checking for ease
retry_files_enabled = False      # Disable retry files after failed runs
```

Important Options:

- **inventory**: Specifies the inventory file location.
- **remote_user**: The user to connect as by default (can be overridden in the playbook).

- **host_key_checking**: Disables SSH host key checking, which prevents warnings about unknown hosts.
- **retry_files_enabled**: Prevents Ansible from creating .retry files on failures.

SSH Configuration for Ansible

Since Ansible uses SSH to connect to managed nodes, it's essential to set up passwordless SSH using public/private keys.

Generating an SSH Key Pair

You can generate a new SSH key pair using the following command:

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

Copying the Public Key to the Managed Node

Once you have your SSH key pair, you need to copy the public key to the managed nodes:

```
ssh-copy-id user@host
```

With this configuration, you can now connect to your servers without being prompted for a password.

---

4. Ansible Inventory Management

Ansible inventory defines which hosts are under management and groups them for different tasks. An inventory file can include **host variables**, **group variables**, and **global variables**.

Static Inventory

A static inventory is just a text file listing all your hosts and organizing them into groups.

Basic Inventory Example

```
[webservers]
web01.example.com
web02.example.com

[dbservers]
db01.example.com
db02.example.com

[all:vars]
ansible_user=ubuntu
ansible_ssh_private_key_file=~/.ssh/id_rsa
```

- **[webservers]**: Group containing web server hosts.
- **[dbservers]**: Group containing database server hosts.
- **[all]**: Global variables that apply to all hosts, such as the user for SSH login and the SSH key file to use.

Host Variables

You can assign variables directly to specific hosts.

[webservers]

web01.example.com ansible_host=192.168.1.10 ansible_user=ubuntu

This ensures that when connecting to web01.example.com, Ansible uses 192.168.1.10 as the actual IP and ubuntu as the user.

Dynamic Inventory

In environments like cloud infrastructures, where hosts are dynamically created and destroyed, using a static inventory can become a challenge. Dynamic inventories solve this by querying a data source (like AWS, Azure, or GCP) to get the list of hosts in real-time.

Example: AWS EC2 Dynamic Inventory

To use a dynamic inventory for AWS EC2 instances, you'll need to install the **amazon.aws** collection:

```
ansible-galaxy collection install amazon.aws
```

Then, create a YAML file that uses the dynamic inventory plugin:

```
plugin: amazon.aws.ec2
regions:
  - us-west-1
filters:
  instance-state-name: running
```

This inventory will dynamically fetch all running EC2 instances in the us-west-1 region.

Host Groups and Variables in Dynamic Inventory

You can organize your dynamic inventory into groups and assign variables at the group level. This allows you to run specific tasks on different sets of servers.

Example:

```
plugin: amazon.aws.ec2
keyed_groups:
  - key: tags.Name
    prefix: "tag_"
  - key: instance-state-name
    prefix: "state_"
```

This configuration creates dynamic groups based on EC2 tags and instance states.

---

5. Playbooks: Anatomy and Execution

Playbooks are where the real power of Ansible lies. Playbooks are written in YAML and contain plays, which consist of tasks applied to managed hosts.

What is a Play?

A play is a collection of tasks that target a group of hosts. A play ensures that the desired state is achieved for the hosts it is run against.

Basic Structure of a Playbook

```
---
- name: Configure Web Servers
  hosts: webservers
  become: yes
  tasks:
    - name: Install NGINX
      apt:
        name: nginx
        state: present

    - name: Copy custom NGINX configuration
      template:
        src: templates/nginx.conf.j2
        dest: /etc/nginx/nginx.conf

  handlers:
    - name: Restart NGINX
      service:
        name: nginx
        state: restarted
```

Key Playbook Components:

- **hosts**: The group of hosts on which the play will run.
- **become**: Runs tasks with elevated privileges (like sudo).
- **tasks**: Defines the steps to be executed on the target hosts.
- **handlers**: Special tasks that are triggered when notified by other tasks, like restarting a service after configuration changes.

Running a Playbook

To run a playbook, you use the ansible-playbook command:

```
ansible-playbook playbook.yml
```

To increase verbosity for troubleshooting:

```
ansible-playbook playbook.yml -vvv
```

This will output more detailed logs, helping you debug any issues during execution.

---

6. Ansible Modules Deep Dive

Ansible modules are the building blocks for task execution. Each module is a small piece of code that performs a specific task, such as managing packages, files, users, or services on a system.

Module Execution Flow

Ansible modules are executed on the managed nodes over SSH or WinRM. Here's how it works:

1. **Task Execution**: Ansible sends the module code and any arguments to the managed node.
2. **Module Execution**: The module is executed on the managed node in isolation.
3. **Result Return**: The module returns the result back to the control node, indicating whether the task succeeded, failed, or made changes.

Writing Custom Modules

If the built-in modules don't meet your requirements, you can write custom modules in any language, though Python is the most common choice because Ansible's internals are Python-based.

Custom Module in Python

```
#!/usr/bin/python

from ansible.module_utils.basic import AnsibleModule
import os

def main():
    module = AnsibleModule(
        argument_spec=dict(
            path=dict(type='str', required=True)
        )
    )

    path = module.params['path']
    if os.path.exists(path):
        result = dict(changed=False, message="Path exists")
    else:
        result = dict(changed=True, message="Path does not exist")

    module.exit_json(**result)

if __name__ == '__main__':
    main()
```

This custom module checks whether a file or directory exists on the target system.

How to Use Custom Modules

Custom modules are placed in a directory named library/ within your playbook's directory structure. Here's how to use the above module in a playbook:

```
---
- name: Check if a path exists using custom module
  hosts: localhost
  tasks:
    - name: Check for /etc/nginx
      my_custom_module:
        path: /etc/nginx
```

7. Advanced Ansible Roles

Roles allow you to break down your playbooks into reusable components. This is especially useful for larger projects with complex configurations. Each role contains everything necessary to configure a part of your infrastructure: tasks, handlers, templates, files, and variables.

Creating a Role

You can create a role manually by organizing tasks, handlers, files, and variables, or you can use Ansible Galaxy to initialize a role:

```
ansible-galaxy init my_role
```

This will generate the following structure:

```
my_role/
  ├── defaults/      # Default variables
  ├── files/         # Static files to copy
  ├── handlers/      # Handlers that respond to task changes
  ├── meta/          # Metadata (like dependencies)
  ├── tasks/         # Main list of tasks
  ├── templates/     # Jinja2 templates
  ├── vars/          # Role-specific variables
```

Example: NGINX Role

Here's an example of an NGINX role structure:

```
roles/
  nginx/
    tasks/
      main.yml
    templates/
      nginx.conf.j2
    handlers/
      main.yml
```

Breaking Down the NGINX Role

Tasks (tasks/main.yml)

```
---
- name: Install NGINX
  apt:
    name: nginx
    state: present

- name: Deploy NGINX config
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
  notify:
    - Restart NGINX
```

Handlers (handlers/main.yml)

```
---
- name: Restart NGINX
  service:
    name: nginx
    state: restarted
```

Templates (templates/nginx.conf.j2)

```
server {
    listen 80;
    server_name {{ server_name }};
    root /var/www/html;
}
```

This structure ensures that the NGINX server is installed, configured, and restarted if the configuration changes.

Using the Role in a Playbook

```
---
- name: Apply NGINX role
  hosts: webservers
  roles:
    - nginx
```

Role Dependencies

Roles can have dependencies on other roles, which is defined in the meta/main.yml file. For example, if your NGINX role depends on a common role that sets up the basic environment, you can specify that dependency:

```
dependencies:

  - role: common
```

8. Ansible Variables: Advanced Techniques

Variables in Ansible allow you to parameterize your playbooks and roles. You can define variables in many places, such as inventory files, playbooks, roles, or as extra vars passed from the command line.

Variable Precedence

Ansible has a well-defined hierarchy for variable precedence, from highest to lowest:

1. **Extra vars** (passed via command-line --extra-vars).
2. **Task vars** (defined within tasks).
3. **Block vars** (defined in blocks).
4. **Play vars** (defined in the playbook).
5. **Role vars** (defined in roles).
6. **Inventory vars** (defined in the inventory file).
7. **Role defaults** (defined in the role defaults).

8. **Ansible built-in defaults**.

Defining Variables

You can define variables in many places:

In a Playbook:

```
---
- hosts: webservers
  vars:
    nginx_port: 8080
  tasks:
    - name: Ensure NGINX is listening on port 8080
      template:
        src: templates/nginx.conf.j2
        dest: /etc/nginx/nginx.conf
```

In the Inventory File:

```
[webservers]
web01 ansible_host=192.168.1.10 nginx_port=8080
web02 ansible_host=192.168.1.11 nginx_port=8081
```

Each host can have its own set of variables defined in the inventory.

Dynamic Variables with Jinja2

Jinja2 is a powerful templating language used in Ansible to generate dynamic content. It allows you to manipulate variables inside playbooks and templates.

Example: Using Jinja2 for Conditional Configuration

```
---
- hosts: webservers
  vars:
    nginx_port: "{{ ansible_default_ipv4['address'] == '192.168.1.10' | ternary('8080', '80') }}"
```

This dynamically sets the NGINX port based on the IP address of the host.

---

9. Conditionals, Loops, and Filters in Depth

Ansible provides powerful features for handling complex tasks, including conditionals (when), loops, and filters.

Conditionals with when

The when clause is used to conditionally execute a task based on a fact or variable.

Example: Run a Task Only on Debian Systems

```
---
- name: Install NGINX only on Debian systems
  apt:
    name: nginx
    state: present
  when: ansible_facts['os_family'] == 'Debian'
```

Loops in Ansible

Ansible supports several looping constructs, such as loop, with_items, and with_dict.

Looping Through a List of Items

```
---
- name: Install multiple packages
  apt:
    name: "{{ item }}"
    state: present
  loop:
    - nginx
    - git
    - curl
```

Looping Through a Dictionary

```
---
- name: Create multiple users
  user:
    name: "{{ item.key }}"
    state: present
  with_dict:
    user1: "admin"
    user2: "guest"
```

Filters for Data Manipulation

Ansible provides a large set of Jinja2 filters for transforming and manipulating data.

Example: Using Filters for String Manipulation

```
---
- name: Convert the string to uppercase
  debug:
    msg: "{{ 'ansible' | upper }}"
```

This will output ANSIBLE.

---

10. Ansible Vault: Managing Secrets

Ansible Vault is used to encrypt sensitive data, such as passwords, API keys, or SSH private keys, that you don't want to store as plain text.

Encrypting a File

To encrypt a file (e.g., secret_vars.yml), run:

```
ansible-vault encrypt secret_vars.yml
```

You will be prompted to enter a password, which will be required for decryption.

Using Encrypted Files in Playbooks

You can reference encrypted files in your playbooks using the vars_files keyword:

```
---
- hosts: all
  vars_files:
    - secret_vars.yml
  tasks:
    - name: Use the secret variable
      debug:
        var: secret_password
```

When running the playbook, Ansible will prompt you for the vault password:

```
ansible-playbook playbook.yml --ask-vault-pass
```

Encrypting Variables Inline

You can encrypt individual variables within a YAML file using ansible-vault:

```
ansible-vault encrypt_string 'supersecretpassword' --name 'db_password'
```

This command will output an encrypted string that you can use directly in your playbooks:

```
db_password: !vault |
          $ANSIBLE_VAULT;1.1;AES256

61366437653033336432393933230613433353566313637616234643231363731353838313132 31
          ...
```

---

11. Error Handling and Debugging

Ansible provides several mechanisms for handling errors and debugging playbooks. It's crucial to implement proper error handling to ensure your playbooks run reliably in production environments.

Error Handling with failed_when

You can customize failure conditions for a task using failed_when. This is useful for tasks where a non-zero exit code doesn't necessarily mean failure.

Example: Custom Failure Condition

```
---
- name: Check disk space
  command: df -h /home
  register: disk_output
  failed_when: "'100%' in disk_output.stdout"
```

In this task, the task will fail if the disk space usage is 100%.

Ignoring Errors with ignore_errors

You can ignore errors in tasks using the ignore_errors directive, allowing the playbook to continue executing even if a task fails.

```
---
- name: Try to install a package (ignore failures)
  apt:
    name: nonexistent-package
    state: present
  ignore_errors: yes
```

Using rescue and always Blocks

Ansible allows for structured error handling using the block, rescue, and always keywords.

Example: Handling Failures with rescue

```
---
- hosts: all
  tasks:
    - block:
        - name: Try to install a package
          apt:
            name: nonexistent-package
            state: present
      rescue:
        - name: Print an error message
          debug:
            msg: "Failed to install the package"
      always:
        - name: Cleanup temp files
          file:
            path: /tmp/tempfile
            state: absent
```

In this example, if the package installation fails, the rescue block will be executed to handle the failure, and the always block will be executed regardless of success or failure.

---

12. Optimizing Ansible for Large-Scale Environments

Ansible performs well in large environments, but there are optimizations you can make to improve performance and scalability.

Parallelism with Forks

By default, Ansible runs tasks on 5 hosts at a time. You can increase the number of parallel tasks using the -f flag or by modifying ansible.cfg:

ansible-playbook playbook.yml -f 10

To make this change permanent, edit ansible.cfg:

```
[defaults]
forks = 10
```

Managing Large Inventories with Dynamic Inventory

Dynamic inventories are critical in large environments where the number of hosts changes frequently. You can query cloud providers like AWS or Azure to fetch the list of active hosts.

Example: Using AWS EC2 as Dynamic Inventory

Install the AWS EC2 dynamic inventory plugin:

```
ansible-galaxy collection install amazon.aws
```

Then, configure your inventory as:

```
plugin: amazon.aws.ec2
regions:
  - us-west-2
filters:
  instance-state-name: running
keyed_groups:
  - key: tags.Environment
    prefix: "env_"
```

Managing Multiple Environments

In large organizations, you might have separate environments for development, testing, and production. You can handle this by maintaining separate inventories for each environment and controlling playbook behavior using environment-specific variables.

Example directory structure for multi-environment management:

```
inventories/
  production/
    hosts
    group_vars/
  staging/
    hosts
    group_vars/
  development/
    hosts
    group_vars/
```

---

13. Integrating Ansible with DevOps Workflows

Ansible fits well into DevOps workflows, especially when integrated with CI/CD pipelines for continuous deployment, infrastructure as code, and configuration management.

CI/CD Integration with Jenkins

Jenkins is commonly used to automate builds, tests, and deployments. You can integrate Ansible into Jenkins pipelines to automate the deployment of infrastructure and applications.

Example: Jenkins Pipeline with Ansible

```
pipeline {
  agent any
  stages {
    stage('Checkout') {
      steps {
        git 'https://github.com/yourrepo.git'
      }
```

```
    }
    stage('Run Ansible Playbook') {
      steps {
        ansiblePlaybook credentialsId: 'ssh-key', inventory: 'inventory/hosts', playbook: 'deploy.yml'
      }
    }
  }
}
```

In this pipeline, Jenkins will check out the code from the Git repository and run the Ansible playbook.

Using Ansible with Docker and Kubernetes

Ansible can be used to orchestrate Docker and Kubernetes deployments. For example, you can use Ansible to define Kubernetes manifests and apply them to your cluster.

Deploying Docker Containers with Ansible

```
---
- name: Deploy a Docker container
  hosts: all
  tasks:
    - name: Run an NGINX container
      docker_container:
        name: nginx
        image: nginx
        state: started
        ports:
          - "80:80"
```

Managing Kubernetes with Ansible

To interact with Kubernetes, you can use the kubernetes.core collection. Install it using:

```
ansible-galaxy collection install kubernetes.core
```

Now you can manage Kubernetes resources:

```
---
- name: Apply Kubernetes manifests
  hosts: all
  tasks:
    - name: Apply the deployment
      kubernetes.core.k8s:
        state: present
        definition:
          apiVersion: apps/v1
          kind: Deployment
          metadata:
            name: nginx
          spec:
            replicas: 3
            selector:
              matchLabels:
                app: nginx
            template:
```

```
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
```

---

14. Ansible Best Practices: Code and Infrastructure Design

Modularize Playbooks with Roles

Instead of writing massive playbooks with many tasks, modularize them using roles. This makes your Ansible code more maintainable and reusable.

Manage Idempotency

Always ensure your playbooks are idempotent, meaning running the playbook multiple times should not change the system's state after the first run. This is crucial for reliability in automation.

Version Control with Git

Store all your Ansible code in a version control system like Git. Use branching strategies (e.g., master, develop, feature/*) to manage changes to your infrastructure.

Testing Ansible Playbooks with Molecule

Use Molecule to test your Ansible roles and playbooks in isolated environments before running them in production.

```
molecule init role my_role
cd my_role
molecule test
```

This command will set up a test environment, run your playbook, and verify that it works as expected.

---

15. Advanced Usage of Ansible with Cloud Platforms

Ansible integrates well with cloud platforms like AWS, Azure, and GCP. By leveraging their APIs, you can automate the provisioning, configuration, and management of cloud infrastructure.

AWS with Ansible

To interact with AWS, install the amazon.aws collection:

```
ansible-galaxy collection install amazon.aws
```

Example: Launching EC2 Instances

```
---
- name: Launch EC2 instance
  hosts: localhost
  tasks:
    - name: Launch an EC2 instance
      amazon.aws.ec2_instance:
        key_name: my_key
        instance_type: t2.micro
        image_id: ami-0abcdef1234567890
        wait: yes
```

Azure with Ansible

To manage Azure resources, install the azure.azcollection:

```
ansible-galaxy collection install azure.azcollection
```

Example: Provisioning an Azure VM

```
---
- name: Create an Azure VM
  hosts: localhost
  tasks:
    - name: Create VM
      azure.azcollection.azure_rm_virtualmachine:
        resource_group: myResourceGroup
        name: myVM
        vm_size: Standard_B1ms
        admin_username: azureuser
        ssh_password_enabled: false
```

Google Cloud with Ansible

To interact with Google Cloud, use the google.cloud collection:

```
ansible-galaxy collection install google.cloud
```

Example: Provisioning a Google Compute Engine Instance

```
---
- name: Launch a GCE instance
  hosts: localhost
  tasks:
    - name: Create an instance
      google.cloud.gcp_compute_instance:
        name: "test-instance"
        machine_type: "n1-standard-1"
        zone: "us-central1-a"
        disks:
          - auto_delete: true
            boot: true
            initialize_params:
              source_image: "projects/debian-cloud/global/images/debian-10-buster-v20200902
```

Conclusion

This guide has provided a deep dive into the world of Ansible, covering everything from installation, configuration, and playbooks, to advanced topics like roles, modules, error handling, and cloud integrations. By mastering these concepts and best practices, you'll be well-equipped to use Ansible for automating tasks in any environment—be it small-scale server management or large-scale cloud infrastructure.

With Ansible, you can automate repetitive tasks, ensure consistent configurations, and streamline your DevOps workflows, resulting in more efficient and reliable IT operations.