1. What is Ansible?

Ans ->

Ansible is an **open-source automation tool (written in Python language)** used for configuration management, application deployment, and task automation. It simplifies IT processes by allowing users to define infrastructure as code (IaC) in a declarative manner.

Key Features:

- **Agentless:** Unlike other automation tools, Ansible does not require agents on remote machines; it uses SSH for communication.
- **Simple YAML Syntax:** Uses human-readable YAML-based playbooks for automation.
- **Idempotency:** Ensures tasks run only when necessary, avoiding redundant changes.
- **Scalability:** Can manage thousands of nodes efficiently.
- **Extensibility:** Supports plugins and modules for various cloud providers, databases, and services.

Common Use Cases:

- Configuration Management (e.g., setting up web servers)
- Application Deployment (e.g., deploying Docker containers)
- Infrastructure Automation (e.g., provisioning cloud resources)
- Security and Compliance (e.g., enforcing firewall rules)

2. What are the benefits of Ansible?

Ans ->

Agentless Architecture – No need to install software or agents on remote systems; it uses SSH for communication.

Simple & Human-Readable – Uses YAML-based playbooks, making it easy to learn and write automation scripts.

Idempotency – Ensures tasks are executed only when changes are needed, preventing unnecessary actions.

Cross-Platform Support – Works on Linux, Windows, cloud environments, and networking devices.

Scalability & Efficiency – Can manage thousands of nodes with minimal resource consumption.

Extensive Modules & Integrations – Supports various cloud platforms (AWS, Azure, GCP), databases, and applications.

Security & Compliance – Automates security policies, firewall rules, and system hardening.

Orchestration & Workflow Automation – Coordinates multi-tier applications and complex workflows.

Community Support – Large open-source community providing continuous updates and improvements.

3. What are the important terminologies in Ansible?

Ans ->

Important Terminologies in Ansible:

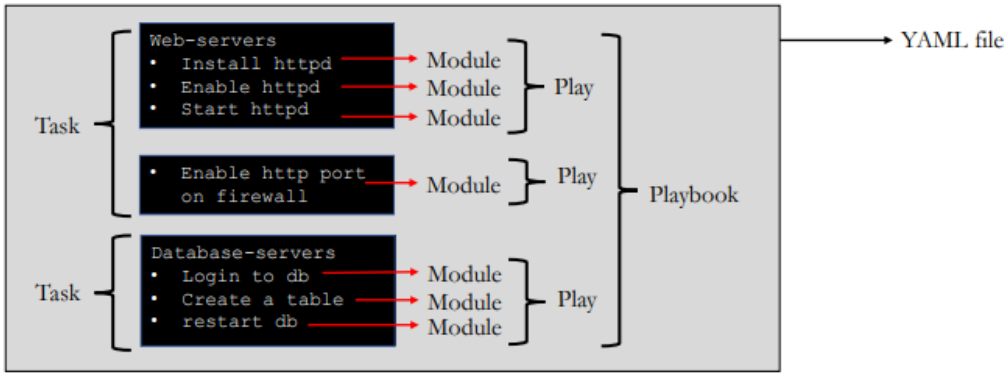
- **Control Node** – The machine where Ansible is installed and from which automation tasks are executed.
- **Managed Nodes** – The remote systems that Ansible manages and configures.
- **Inventory** – A file containing a list of managed nodes, categorized into groups for better organization. File that has information about remote clients where tasks are executed.
- **Playbook** – A YAML file with step-by-step execution of multiple tasks, configurations, and automation workflows.
- **Task** – A single unit of work in a playbook, such as installing software or updating configurations. A task can have multiple modules.
- **Module** – Predefined scripts used to perform specific tasks (e.g., file management, package installation).
- **Role** – A structured way to group tasks, variables, templates, and handlers for reusability. In another word, splitting of playbook into smaller groups.
- **Handler** – A special task triggered by other tasks when changes occur (e.g., restarting a service).
- **Variable** – A way to store dynamic values that can be reused in playbooks.
- **Facts** – System information automatically gathered by Ansible about managed nodes.
- **Template** – A Jinja2-based file used to dynamically generate configuration files.

- **Galaxy** – A repository for sharing Ansible roles and collections.
- **Vault** – A security feature used to encrypt sensitive data like passwords and API keys.
- **Ad-hoc Commands** – One-time commands run directly in Ansible without using a playbook.
- **Collection** – A packaged set of modules, roles, and plugins that extend Ansible's functionality.
- **Tag** - A reference or alias to a specific task.

4. How does Ansible works?

Ans ->

- **Install Ansible**
  - Ansible is installed on a Control Node (Linux/macOS).
  - No agents are required on managed nodes.
- **Define Inventory**
  - Create an inventory file (hosts) listing the remote machines to manage.
- **Write Playbooks**
  - Use YAML-based playbooks to define tasks and configurations.
- **Establish Connection**
  - Ansible connects to managed nodes via SSH (Linux) or WinRM (Windows).
- **Execute Tasks**
  - Runs tasks using modules (e.g., install software, configure files).
- **Ensure Idempotency**
  - Only applies changes if needed, avoiding unnecessary modifications.
- **Generate Reports**
  - Displays execution results (successful, changed, failed tasks).



5. Explain about Ansible configuration files.

Ans ->

- **Main Configuration File (ansible.cfg):**
  - **Default Path:** /etc/ansible/ansible.cfg or ~/.ansible.cfg (user-specific)
- **Inventory File (hosts):**
  - **Default Path:** /etc/ansible/hosts or custom-defined in ansible.cfg
- **Playbooks (.yml files):**
  - User-defined, usually stored in project directories (e.g., /home/user/playbook.yml)
- **Roles Directory (roles/):**
  - **Default Path:** /etc/ansible/roles/ or project-specific (e.g., /home/user/ansible/roles/)
- **Vault File (vault.yml)**
  - Stored in a secure location, e.g., /home/user/secrets/vault.yml

6. Free Tier Ansible VS Red Hat Ansible.

Ans ->

1. Free Ansible (Community Version)

- Open-source and free to use.
- Uses command-line interface (CLI) with YAML playbooks.
- No official support; relies on community help.
- Lacks advanced enterprise features like automation analytics and governance.
- Best suited for small teams, developers, and simple automation tasks.

2. Red Hat Ansible (Ansible Automation Platform)

- Enterprise-grade version with additional features.
- Includes **Ansible Tower** (now called **Red Hat Automation Controller**) for web-based management.
- Provides role-based access control (RBAC), logging, and automation analytics.
- Official Red Hat support and regular security updates.
- Best for large enterprises needing automation at scale with governance and compliance.

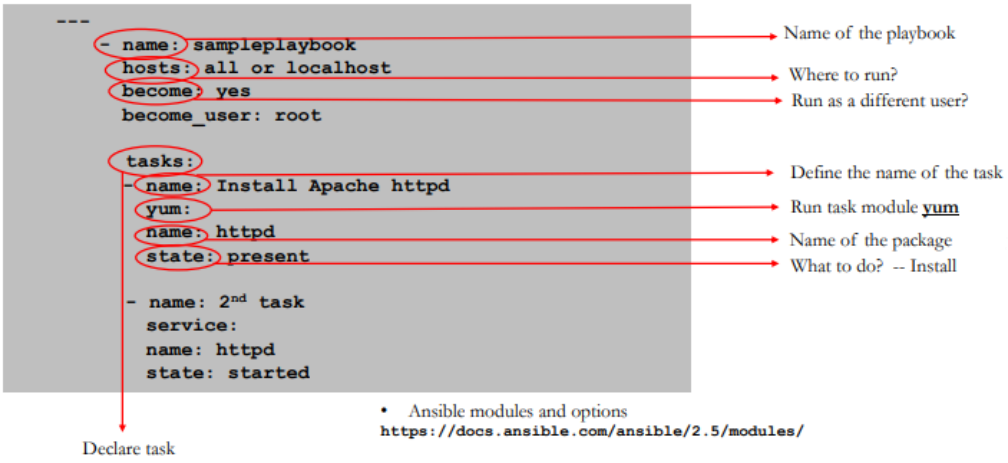
7. Important things to remember about YAML File Syntax in the context of Ansible.

Ans ->

- **Task Execution Order:** Tasks run sequentially, one at a time, from top to bottom.
- **Indentation:** Use spaces only (**never tabs**); incorrect indentation leads to errors.
- **Empty lines have no value:** Blank lines in a YAML file are ignored.
- File extensions are usually **.yml** or **.yaml** – Both extensions work for Ansible playbooks.
- Task names can be written with or without quotes.
- **YAML playbook files can be placed anywhere** – As long as they are executed with an absolute path, location doesn't matter.
- **No need to modify file permissions** – Playbook files do not require executable permissions (chmod +x).
- **Comments:** Start with # and are ignored during execution.
- When a flat file is written in YAML format to execute tasks/plays then it is called playbook
- **Key-Value Pairs:** Written as key: value with a space after ':'

8. Example Ansible Playbook Illustrating YAML Syntax Rules

Ans ->



Key Points

- Uses .yaml extension and proper indentation (spaces only).
- Sequential execution – tasks run one by one.
- Minimal yet functional – installs Apache, starts it, and deploys a simple HTML page.

9. Create and run First Playbook.

Ans ->

Before we begin to create our first playbook we need to follow the below steps in terminal:

- First we need to change the user as **root**: su - root
- Then we must create a directory for playbooks: mkdir /etc/ansible/playbooks
- And go to that directory: cd /etc/ansible/playbooks
- Creating the playbook: vi myFirstPlaybook.yml

Now we write the below contents in the playbook file:

```
---
- name: "My first playbook"
  hosts: localhost

  tasks:
  - name: "test connectivity"
    ping:
```

Note: this playbook will test the connectivity of localhost means it will ping the localhost

To check the syntax of playbook run the below command:

- ansible-playbook --syntax-check myFirstPlaybook.yml

To do a dry run:

- ansible-playbook --check myFirstPlaybook.yml

Run the playbook:

- ansible-playbook myFirstPlaybook.yml

10. Playbook example for printing output in the screen.

Ans ->

This playbook will print "Hello World" on localhost

```
# cd /etc/ansible/playbook
# vim helloworld.yml

---
- name: My Second playbook
  hosts: localhost

  tasks:
  - name: Print Hello World
    debug: msg="Hello World"
```

Name of the play or playbook
Run on localhost
Run the following task
Name of the task
Run **debug** module which prints statements during execution

Run the playbook

```
# anisble-playbook helloworld.yml
```

11. Multitask playbook example.

Ans ->

The playbook will ping localhost and print "Hello World"

```
# vim mtask.yml

---
- name: Running 2 tasks
  hosts: localhost

  tasks:
  - name: Test connectivity
    ping:

  - name: Print Hello World
    debug: msg="Hello World"
```

Name of the play
Run it on local host
Run the following task
Name of the task
Run the **ping** module
Name of the 2nd task
Run the **debug** module

Run the playbook

```
# anisble-playbook mtask.yml
```

12. Explain about modules in Ansible.

Ans ->

In **Ansible**, a **module** is a piece of code that performs a specific task on a remote system. Think of it like a small tool or command that Ansible uses to do things like **install software**, **copy files**, or **configure settings** on servers.

Modules are the building blocks of Ansible playbooks and are executed on the target machines. You don't need to worry about how the module works internally, Ansible takes care of that for you. You simply call the module with the right parameters.

For example:

- The '**yum**' module installs or removes software on a Red Hat-based system.
- The '**copy**' module copies files from your local machine to remote servers.
- The '**service**' module manages services like starting or stopping them.

Modules help Ansible to automate tasks without needing to write custom scripts or commands manually.

13. Explain about Roles in Ansible.

Ans ->

Ansible roles are a structured way to organize playbooks by **separating tasks**, **variables**, **files**, **templates**, and **handlers** into a predefined folder layout, making automation cleaner, reusable, and easier to maintain.

Think of a role as a **packaged unit** containing everything needed to configure one specific part of your system; like installing **nginx**, **setting up users**, **configuring firewalls**, etc.

Key points:

- **Reusability** – Write once and use the role in many playbooks.
- **Maintainability** – Code becomes clean and well-structured.
- **Scalability** – Easy to manage complex automation projects.
- **Standardization** – Follows a fixed directory layout that everyone understands.

Example role usage in a playbook:

```yaml
- hosts: web
  roles:
    - nginx
```

Example role folder structure: roles/nginx/tasks/main.yml

14. What are the differences between Roles and Modules in Ansible?

Ans ->

- **Roles** are used to **organize and structure automation**. They group related tasks, variables, files, templates, and handlers needed to configure a specific service or component.
- **Modules** are the **actual tools that perform actions**, such as installing packages, creating users, copying files, or managing services.
- **Roles** are used **inside playbooks**, while **modules** are used **inside tasks**.
- A **role does not perform actions by itself**; it runs tasks that call modules.
- Modules perform **one specific operation at a tim**e, whereas roles manage complete configurations.
- Roles are designed for **reuse and maintainability**, while modules focus on **execution**.

In short:

Modules do the work, and roles organize the work in Ansible.

15. Explain about the Ansible inventory file (hosts) and its syntax.

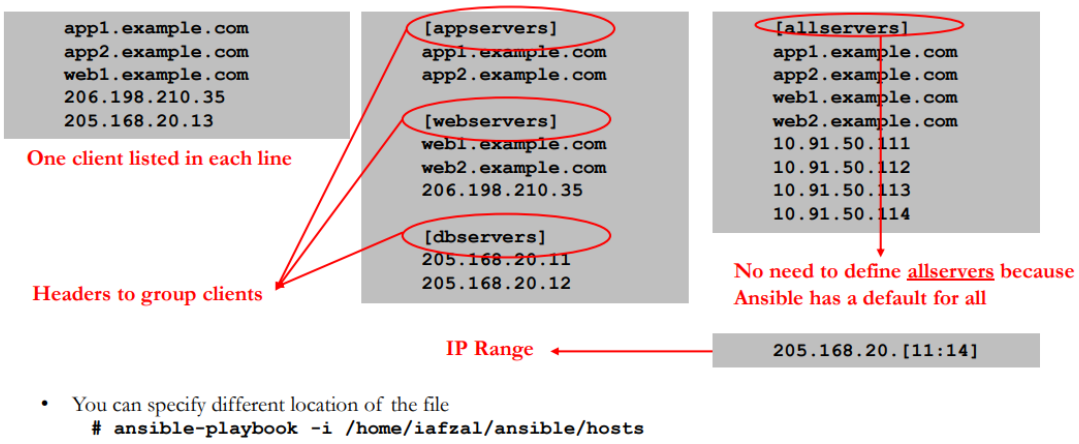
Ans ->

The **Ansible inventory file**, usually located at /etc/ansible/hosts, is where you define the hosts (servers or nodes) that Ansible will manage. It lists all the systems Ansible can communicate with and is essential in defining which hosts will be affected by Ansible playbooks or ad-hoc commands.

Key Points:

- **Location:** Default is /etc/ansible/hosts, customizable with -i.
- **Hosts & Groups:** Organize hosts into groups; hosts can be in multiple groups.
- **Variables:** Define host/group-specific variables.
- **Static/Dynamic:** Static by default, dynamic inventories for changing environments.
- **Formats:** Supports INI (default) and YAML.
- **Patterns:** Target hosts/groups with patterns or wildcards.

Syntax:



- You can specify different location of the file

```
# ansible-playbook -i /home/iafzal/ansible/hosts
```

16. Some commands related to host file.

Ans ->

Ping all hosts:

- ansible all -m ping => Tests connectivity to all hosts in the inventory.

Ping specific group:

- ansible group_name -m ping => Pings hosts in a specific group.

Specify a custom inventory file with playbook:

- ansible-playbook -i /path/to/inventory <playbook_name> => Uses a custom inventory file instead of the default.

List hosts in the inventory:

- ansible-inventory --list -i /etc/ansible/hosts => Shows all hosts and groups in the inventory.

Show detailed information for a specific host:

- ansible -m setup host_name => Gathers and displays facts about a specific host.

Run a command on all hosts:

- ansible all -m shell -a "uptime" => Runs the uptime command on all hosts in the inventory.

17. How to setup password-less SSH for Ansible Clients?

Ans ->

Step1: Populate Ansible Hosts File

Define a group [labclients] or mention as all in the inventory file with client IPs:

Example:

[labclients] = For Grouping

10.253.1.18

10.253.1.20

Step2: Generate SSH Keys on the Control Node

Run ssh-keygen, press Enter to accept defaults until the image key is generated, by default the location of the key is ~/.ssh/.

Step3: Copy the SSH Key to Clients

Use ssh-copy-id to send the public key to each client, in this step we must provide the root password (one time) of the client which we want to connect to:

Example:

ssh-copy-id 10.253.1.18

ssh-copy-id 10.253.1.20

Note: you will have to copy this key to each client listed in the inventory/host file

Step4: Test Password-less SSH

Verify access with: ssh 10.253.1.18

Once successful, Ansible can manage these clients without needing a password.

18. Example playbook for copying files from compute node to client node.

Ans ->

This Ansible playbook copies a file from the control node to all target hosts. It uses the **copy** module to place the file in /tmp with the specified owner, group, and permissions.

```
---
- name: Copy file from control node to client node    # Play description
  hosts: all                                          # Target all hosts

  tasks:
  - name: Copying file                                # Task description
    become: true                                      # Use sudo

    copy:                                             # This is copy module
      src: /etc/ansible/test/some.txt                 # Source file path
      dest: /tmp                                      # Destination path
      owner: samim                                    # File owner
      group: samim                                    # File group
      mode: 0644                                      # File permissions (rw-r--r--)
```

19. Example playbook for change file permission

Ans ->

This Ansible playbook changes the permissions of a specific file on **all target hosts**. It uses the **file** module to add write access **(a+w)** for all users on **/tmp/example.txt**.

```
---
- name: Change file permission          # Play description
  hosts: all                            # Run on all hosts

  tasks:
  - name: File permission                # Task description
    file:
      path: /tmp/example.txt             # Target file path
      mode: a+w                          # Add write permission for all users
```

20. Example playbook for http setup

Ans ->

This Ansible playbook **installs** and **starts** the **Apache HTTP web server (httpd)** on **all target hosts** and then configures the firewall to **allow incoming traffic on port 80 (HTTP)**. Finally, it **reloads the firewall** service, so the new rules take effect.

```
---
- name: Setup httpd and open firewall port
  hosts: all
  become: true
  tasks:
  - name: Install apache packages
    yum:
        name: httpd
        state: present

  - name: Start httpd
    service:
        name: httpd
        state: started

  - name: Open port 80 for http access
    firewalld:
        service: http
        permanent: true
        state: enabled

  - name: Reload firewalld

    service:

            name: firewalld

            state: reloaded
```

21. Example playbook for running a shell script file inside a remote client from the control node machine.

Ans ->

This playbook connects to **all remote hosts** and **runs a shell script** located at /home/samim/scriptfile.sh on each machine using the shell module, allowing any commands inside the script to execute directly on the remote client systems.

```
---
- name: playbook for running shell script inside remote client
  hosts: all
  tasks:
  - name: Run shell script
    shell: "/home/samim/scriptfile.sh"
```

22. Example playbook for scheduling a cron job (crontab) in remote client machine.

Ans ->

The below Ansible playbook will do the followings:

- Run on **all target hosts** defined in the inventory.
- **Create or manage a cron job** for the **root user**.
- Schedule the job to run at **10:00 AM**.
- Run it **every day of the month** and **every month (day: "*", *month: "*"*)**.
- Run the job **every Thursday (weekday: "4")**.
- Execute the script **/home/samim/scriptfile.sh**.
- Add a **description/comment** to the cron entry: "**This job is scheduled by Ansible**".

```
---
- name: Create a cron job
  hosts: all
  tasks:
  - name: Schedule cron:
    cron:
      name: This job is scheduled by Ansible
      minute: "0"
      hour: "10"
      day: "*"
      month: "*"
      weekday: "4"
      user: root
      job: "/home/samim/scriptfile.sh"
```

23. Example playbook for creating a user in the remote client machine.

Ans ->

This below Ansible playbook will do the followings:

- **Run on all managed hosts** with **root** privileges (become: yes).
- Create and ensure the presence of a user named **george**.
- Set /home/george as the home directory and create it if missing.
- Assign /bin/bash as the login shell.
- Use **ansible.builtin.user**, the official built-in Ansible module for user management.

```
---
- name: Playbook for creating users
  hosts: all
  become: yes

  tasks:
  - name: Create user 'george'
    ansible.builtin.user:
        name: george
        home: /home/george
        shell: /bin/bash
        create_home: yes
        state: present
```

24. Password encryption using ansible-vault for an ansible playbook to use.

Ans ->

Ansible does not allow cleartext passwords in the playbook, Instead, we use **hashed password** passed securely through **Ansible Vault**, to do this we follow the below steps inside the control node machine:

Step 1: Create a new yml file called **vault.yml** using the command: **vim vault.yml**

Step 2: Give a **variable name** and the actual **password** you want to add/update, for example: **newpassword: Abc123p@$$w0rd**

Step 3: Now to secure the **vault.yml** file using ansible vault, **run the command**:

ansible-vault encrypt vault.yml --> this will prompt you to enter a vault password,

so set the vault password as you want and press enter, then it will ask you to confirm the password so retype again and press enter. Now the **vault.yml** file is encrypted.

25. Example playbook for add/update passwords in a remote client machine.

Ans ->

The below Ansible playbook will do the followings:

- Run on **all managed hosts** with **root privileges** (**become: yes**).
- **Update or set the password** for the user **george**.
- Always **force a password update** (**update_password: always**).
- Use a **SHA-512 hashed password** stored inside **ansible-vault** in a variable called **newpassword** variable.
- Use **ansible.builtin.user**, the official built-in Ansible module for managing users.
- **Note:** while running this playbook use the command:

  ansible-playbook <playbook_name>.yml --ask-vault-pass

  This will prompt you to enter the vault password.

```
---
- name: Add or update user password
  hosts: all
  become: yes
  vars_files:
    - vault.yml

  tasks:
  - name: Change "george" password
    ansible.builtin.user:
        name: george

        update_password: always

        password: "{{ newpassword | password_hash('sha512') }}"
```

26. Example playbook for downloading package from a URL.

Ans ->

The below Ansible playbook will do the followings:

- Run on **localhost only**.
- **Create the directory /opt/tomcat** with specified permissions and ownership.
- Ensure the directory exists before downloading files.
- **Download the Tomcat 9 archive** from the given URL.
- Save the file inside **/opt/tomcat**.
- **Set file permissions and ownership** for the downloaded archive.

```
---
- name: Create a directory and download tomcat from a URL
  hosts: localhost
  tasks:
  - name: Create a directory
    file:
         path: /opt/tomcat
         state: directory
         mode: 0775
         owner: root
         group: root
- name: Download tomcat from a URL
  get_url:
          url: https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.113/bin/apache-
tomcat-9.0.113.tar.gz
          dest: /opt/tomcat
          mode: 0755
          owner: samim
          group: samim
```

27. Example playbook for killing a running process inside a remote client machine.

Ans ->

The below Ansible playbook will perform the below steps:

- Run on **all managed hosts**.
- **Find running processes named top**, **ignoring errors if found**.
- **Store the process IDs (PIDs)** in a variable for later use.
- **Loop through the stored PIDs** and attempt to kill each process.
- **Continue execution even if errors occur** during process lookup or termination.

```yaml
---
- name: Find a running process and kill it
  hosts: all

  tasks:
  - name: Get running processes from remote hosts
    ignore_errors: yes
    shell: "ps -efw | grep top | awk '{print $2}'"
    register: running_process

  - name: Kill running processes
    ignore_errors: yes
    shell: "kill {{ item }}"
    with_items: "{{ running_process.stdout_lines }}"
```

28. Explain the different pick-and-choose execution options in Ansible using a single playbook example.

Ans ->

In Ansible, “**pick and choose**” means running only specific parts of a playbook instead of the whole playbook. This is commonly done using tags.

In short it:

- Allows you to select **specific tasks** or **plays** to run.
- Useful for **testing**, **debugging**, or **partial execution**.
- Prevents unnecessary changes to the system.

```yaml
---
- name: Pick and choose demo playbook
  hosts: all
  become: yes

  tasks:
  - name: Install Apache
    yum:
      name: httpd
      state: present
    tags: install

  - name: Start Apache service
    service:
      name: httpd
      state: started
    tags: start

  - name: Copy index file
    copy:
      src: index.html
      dest: /var/www/html/index.html
    tags: config
```

Tags (--tags):

- Run only specific tagged tasks or roles
- Most common and recommended method
- **Example: ansible-playbook site.yml --tags install**

Skip tags (--skip-tags):

- Run everything except the specified tagged tasks
- **Example: ansible-playbook site.yml --skip-tags config**

Start at task (--start-at-task):

- Start execution from a specific task name
- Skips all tasks before it
- Useful for resuming after failure
- **Example: ansible-playbook site.yml --start-at-task="Start Apache service"**

Limit (--limit):

- Run the playbook on selected hosts only
- **Example: ansible-playbook site.yml --limit webservers**

Check mode (--check):

- Perform a dry run to see what would change
- No actual changes are made
- **Example: ansible-playbook site.yml --check**

Step mode (--step):

- Ask for confirmation before each task
- Useful for learning and debugging
- **Example: ansible-playbook site.yml --step**

Task selection with --list-*:

- **--list-tasks** → show tasks without running them
- **--list-tags** → show available tags
- **--list-hosts** → show targeted hosts

- **Examples:**
  - ansible-playbook site.yml --list-tasks
  - ansible-playbook site.yml --list-tags
  - ansible-playbook site.yml --list-hosts

In summary:

Ansible “**pick and choose**” options allow you to **control what runs**, **where it runs**, and **how it runs**, without modifying the playbook itself.

29. Example playbook for creating and mounting new storage.

Ans ->

```
---
- name: Create and mount new storage
  hosts: localhost

  tasks:
   - name: Create new partition
     parted:
           name: files
           label: gpt
           device: /dev/sdb
           number: 1
           state: present
           part_start: 1MiB
           part_end: 1GiB

   - name: Create xfs file system
     filesystem:
          dev: /dev/sdb1
          fstype: xfs

   - name: Create mount directory
     file:
          path: /data
          state: directory

   - name: Mount the filesystem
     mount:
          src: /dev/sdb1
          fstype: xfs
          state: mounted
```

In the above playbook we are using "**parted**" and "**mount**" module, and some Ansible distribution does not come with "**parted**" and "**mount**" module, so to install those modules, **run the below commands**:

```
$ ansible-galaxy collection install community.general
$ ansible-galaxy collection install ansible.posix
```

/dev/sdb represents the **entire raw disk**, not a usable storage area by itself. The partition **/dev/sdb1 does not exist initially** and must be created first by partitioning the disk. Once the partition **/dev/sdb1** is created, it becomes a proper block device that can be **formatted with a filesystem and mounted**. Creating a partition before formatting is standard Linux practice because it makes disk management safer, structured, and easier to maintain.

The above playbook does the followings:

- Run the playbook **on localhost** only.
- **Create a new GPT partition** (**/dev/sdb1**) on the disk **/dev/sdb**.
- **Format the partition /dev/sdb1** with the **XFS filesystem**.
- **Create the mount directory /data** if it does not already exist.
- **Mount the filesystem /dev/sdb1** at **/data**, making the storage available for use.

30. What are Ansible Ad-Hoc Commands?

Ans ->

Ansible ad-hoc commands are **one-line Ansible commands** used to **perform quick, simple tasks** on remote hosts without writing a playbook. They are mainly used for **day-to-day administration**, troubleshooting, or checking system status, such as pinging hosts, checking uptime, restarting a service, or copying a file. Ad-hoc commands are **fast and easy**, but they are **not meant for complex or repeatable automation**, where playbooks are preferred.

Syntax for Ad-Hoc commands: ansible [target] -m [module] -a "[module options]"

Example Ad-Hoc commands:

- **Ping localhost:** ansible localhost -m ping
- **Creating a file on all remote clients:**
  $ ansible all -m file -a "path=/home/samim/adhoc1 state=touch mode=700"
- **Deleting a file on all remote clients:**
  $ ansible all -m file -a "path=/home/samim/adhoc1 state=absent"
- **Copying a file to all remote clients:**
  $ ansible all -m copy -a "src=/tmp/adhoc2 dest=/home/iafzal/adhoc2"
- **Installing package (telnet and httpd-manual) in all remote clients:**
  $ ansible all -m yum -a "name=telnet state=present"
  $ ansible all -m yum -a "name=httpd-manual state=present"
- **Starting httpd packae service:**
  $ ansible all -m service -a "name=httpd state=started"

- **Start httpd and enable at boot time:**
  $ ansible all -m service -a "name=httpd state=started enabled=yes"
- **Checking httpd service status on all remote client:**
  $ ansible all -m shell -a "systemctl status httpd"
- **Remove httpd package from all remote clients:**
  $ ansible all -m yum -a "name=httpd state=absent"
  Or
  $ ansible all -m shell -a "yum remove httpd"
- **Creatin a user on remote clients:**
  $ ansible all -m user -a "name=hello home=/home/hello shell=/bin/bash state=present"
- **To add a user to a different group:**
  $ ansible all -m user -a "name=hello group=samim"
- **Deleting a user on remote clients:**
  $ ansible all -m user -a "name=hello home=/home/hello shell=/bin/bash state=absent"
  Or
  $ ansible all -m shell -a "userdel hello"
- **Getting system information from remote clients:**
  $ ansible all -m setup
- **You can run commands on the remote host without a shell module e.g. reboot client1:**
  $ ansible client1 -a "/sbin/reboot"

31. Explain about Roles in Ansible.

Ans ->

Writing **Ansible** code to manage the same service for multiple environments creates more complexity, and it becomes difficult to manage everything in one Ansible playbook. Also sharing code among other teams becomes difficult. That is where **Ansible Role** helps solve these problems.

Ansible roles are a way to organize playbooks into a **clean**, **reusable** structure. Instead of writing everything in one big playbook, **roles** let you group related **tasks**, **files**, **variables**, **templates**, and **handlers** for a specific purpose (like **installing Apache or configuring a database**) in one place. This makes automation easier to **read**, **reuse**, **share**, and **maintain**, especially in large or complex environments.

Roles are like templates that are most of the time static and can be called by the playbooks

Folder structure for Roles:

/etc/ansible/roles/
→ Main directory where all roles are stored.

/etc/ansible/roles/basicinstall/
→ A single role named basicinstall (role name = folder name).

/etc/ansible/roles/basicinstall/tasks/
→ Contains the list of tasks to be executed for the role

/etc/ansible/roles/basicinstall/tasks/main.yml
→ main.yml is the entry point for tasks.

We **must follow** this above **Ansible role folder structure** to maintain a standard and predictable layout, so **Ansible can automatically locate** tasks, variables, files, templates, and handlers without any extra configuration.

Example roles:

First make **2 directories** for **2 separate roles** with the folder structure and commands given below:

```
$ cd /etc/ansible/
$ mkdir roles
$ cd roles     --> pwd -> /etc/ansible/roles
$ mkdir basicinstall
$ mkdir fullinstall
```

Now create sub directories:

```
$ mkdir basicinstall/tasks
$ mkdir fullinstall/tasks
```

Now create yml files within these sub-directories:

```
$ touch basicinstall/tasks/main.yml
$ touch fullinstall/tasks/main.yml
```

Now write the below configuration inside the file :

/etc/ansible/roles/fullinstall/tasks/main.yml

```
---
- name: Install httpd package
  yum:
     name: httpd
     state: present

- name: Start httpd
  service:
     name: httpd
     state: started

- name: Open port for http
  firewalld:
     service: httpd
     permanent: true
     state: enabled

- name: Restart firewalld
  service:
     name: firewalld
     state: reloaded
```

Now write the below configuration inside the file :

/etc/ansible/roles/basicinstall/tasks/main.yml

```
---
- name: Install httpd package
  yum:
    name: httpd
    state: present

- name: Start httpd
  service:
     name: httpd
     state: started
```

Now create a play book inside the /etc/ansible/playbooks directory and write the below configuration to use the role that we've just created:

```
---
- name: Full install
  hosts: all
  roles:
    - fullinstall

- name: Basic install
  hosts: localhosts
  roles:
    - basicinstall
```

32. What is Ansible Galaxy?

Ans ->

Ansible Galaxy is an **online repository** where you can **find** and **download ready-made roles** and **collections** created by the Ansible community or Red Hat. It helps you automate tasks faster by reusing pre-built configurations instead of writing everything manually.

In short:

It's like a **marketplace** for **Ansible roles/collections** that you can install and use directly in your playbooks.

If you want to install NGINX using a pre-made role:

Install a role from Galaxy: ansible-galaxy install geerlingguy.nginx
This role will then automatically saved under the directory:
/root/.ansible/roles/geerlingguy.nginx
Use it in your playbook like:

```
---
- hosts: all
  roles:
    - geerlingguy.nginx  #Role downloaded from Ansible Galaxy
```

This automatically installs and configures NGINX on your target hosts without writing all the tasks yourself.

33. Explain about variables in Ansible with examples.

Ans ->

Variables in **Ansible** are used to **store values that can be reused in playbooks**, making automation **flexible**, **readable**, and **reusable**. Instead of hard-coding values (like package names, paths, users, or ports), you define variables and reference them wherever needed.

Key points about variables:

- A variable name can **include letters**, **numbers**, and **underscores (_)**.
- Variable names **must always start with a letter**.
- Variable names **cannot contain spaces**, **dots (.)**, or **hyphens (-)**.
- Variables are **case-sensitive**.
- Variables can be defined inside inventory files as well.
- Variables are referenced in playbooks using **Jinja2 syntax**: “{{ variable_name }}”
- Variable references can be written with double quote or single quote, with spaces or without space inside curly braces, like: ‘{{item}}’, ‘{{ item }}’, “{{item}}”, or "{{ item }}" all are valid references.

Different places that we can define a variable and example playbooks to demonstrate them:

1. **Variables inside a playbook (vars section):**

   Variables are defined directly in the playbook and are available to all tasks in that play.

```yaml
---
- hosts: all
  vars:
    pkg_name: httpd

  tasks:
  - yum:
      name: "{{ pkg_name }}"
      state: present
```

2. **Variables at task level:**

   These variables are limited to a single task and override play-level variables for that task.

```yaml
---
- hosts: all
  tasks:
    - yum:
        name: "{{pkg}}"
        state: present
      vars:
        pkg: httpd
```

3. **Variables in inventory files (host & group vars):**

   Variables are defined in the inventory and automatically applied to specific hosts or groups.

   Inside hosts files:

```
[web]
192.168.1.10

[web:vars]
pkg_name=httpd
```

   Inside playbook file:

```yaml
---
- hosts: web
  tasks:
    - yum:
        name: "{{ pkg_name }}"
        state: present
```

4. **Variables from external files (vars_files):**

   Variables are stored in a separate YAML file and imported into the playbook.
   Vars files location: /ansible/vars/vars.yml

```
---
pkg_name: httpd
```

   Inside the playbook file:

```
---
- hosts: all
  vars_files:
    - /ansible/vars/vars.yml

  tasks:
    - name: Install package
      yum:
        name: "{{ pkg_name }}"
        state: present
```

5. **Variables in CLI (command line -e):**

   Variables passed at runtime that have the highest priority and override all others.
   Inside the playbook:

```
---
- hosts: all
  tasks:
    - yum:
        name: "{{ pkg_name }}"
        state: present
```

   When running the playbook use the command:

```
$   ansible-playbook site.yml -e "pkg_name=httpd"
```

6. **Registered variables (task output):**

   Stores the output of a task in a variable that can be reused later in the playbook.

```
---
- hosts: all
  tasks:
    - command: uptime
      register: uptime_output

    - debug:
        msg: "{{ uptime_output.stdout }}"
```

All the above-mentioned Variable Precedence Order (Low → High)

- Inventory variables (group vars, host vars) ← **Lowest priority**
- External variable files (vars_files)
- Playbook variables (vars in playbook)
- Task-level variables
- Registered variables
- Variables in CLI (-e) ← **Highest priority**

Inventory < vars_files < play vars < task vars < registered < Vars in CLI

34. Explain about Handlers in Ansible.

Ans ->

Handlers in Ansible are tasks that run only when something changes. They're mainly used for actions you don't want to run every time, like restarting a service. The most common case is: you change a config file → then you need to restart/reload the service so the change takes effect. Instead of restarting the service on every run, Ansible will restart it only if the config actually changed.

How handlers work (simple words):

- A normal task can **notify** a handler using notify:
- The handler runs **only if the notifying task reports "changed".**
- Even if multiple tasks notify the same handler, the handler runs **only once** (to avoid repeated restarts).
- Handlers usually run **at the end of the play** after all tasks finish.

35. Examples of using handlers in Ansible playbook.

Ans ->

```
---
- name: Verify apache installation
  hosts: localhost
  tasks:
  - name: Ensure apache is at the latest version
    yum:
      name: httpd
      state: latest

  - name: Copy update apache config file
    copy:
      src: /tmp/httpd.conf
      dest: /etc/httpd.conf
    notify:
    - Restart apache

  - name: Ensure apache is running
    service:
      name: httpd
      state: started

  handlers:
    - name: Restart apache
      Service:
        name: httpd
        state: restarted
```

What happens when you run this playbook:

- If httpd.conf is different from what's on the server → Ansible copies it, marks the task as changed, and notifies the handler.
- The handler restarts Apache once, at the end of the play.
- If httpd.conf is already the same → no change → no restart.

36. Explain about condition in Ansible with examples.

Ans ->

In **Ansible**, conditions are used to control whether a task runs or not based on a specific rule or check. They are written using the "**when**" keyword and allow tasks to run only if certain conditions are true, such as checking the **OS type**, a **variable value**, or

the r**esult of a previous task**. Conditions help make playbooks **smarter**, **flexible**, and **safer** by avoiding **unnecessary** or **incorrect** actions on hosts.

Example 1: Run task based on OS

Runs only on RedHat-based systems.

```
---
- name: Install Apache on RedHat systems
  yum:
    name: httpd
    state: present
  when: ansible_os_family == "RedHat"
```

Example 2: Run task if a variable is true

Runs only if apache_enable is set to true

```
---
- name: Start Apache
  service:
    name: httpd
    state: started
  when: apache_enable == true
```

Example 3: Use result of previous task

Restarts Apache only if the file exists.

```
---
- name: Check if file exists
  stat:
    path: /etc/httpd/conf/httpd.conf
  register: conf_file

- name: Restart Apache if config exists
  service:
    name: httpd
    state: restarted
  when: conf_file.stat.exists
```

Example 4: Multiple conditions

Runs only when both conditions are true.

```
---
- name: Install Apache
  yum:
    name: httpd
    state: present
  when:
    - ansible_os_family == "RedHat"
    - install_apache == true
```

37. What are Ansible built-in variable?

Ans ->

Ansible built-in variables (also called facts) are automatically collected system details about managed hosts. They include information like OS type, IP address, hostname, CPU, memory, disks, and more. These variables are commonly used in playbooks for conditions, decisions, and dynamic configuration.

Examples of built-in variables:

- ansible_hostname → Hostname of the system
- ansible_os_family → OS family (RedHat, Debian, etc.)
- ansible_distribution → OS name (CentOS, Ubuntu, RHEL)
- ansible_default_ipv4.address → Primary IP address
- ansible_architecture → System architecture

To check built-in variables, run the command: ansible all -m setup

To check specific facts, run the command:

ansible all -m setup -a "filter=ansible_os_family"

38. Explain about loops in Ansible.

Ans ->

Loops in Ansible allow you to run the same task multiple times with different values, instead of writing the same task again and again. They make playbooks shorter, cleaner, and reusable.

Important points about loops:

- A loop runs one task repeatedly, not the entire playbook
- item is the default variable used inside loops, which represents the current value in each iteration
- Loops can work with lists, dictionaries, files, and sequences
- The modern and recommended syntax is loop (older with_* methods are deprecated)
- Loops can be combined with conditions (when), register, and handlers
- Avoid using loops when a module already supports lists (for example, yum can install multiple packages at once)

39. Different ways of creating multiple users using loops in Ansible playbook

Ans ->

Modern way using loop:

```
---

- name: Create users through loop
  hosts: localhost

  tasks:
  - name: Create users
    user:
       name: "{{item}}"
    loop:
       - jerry
       - kramer
       - eliane
```

Older way using with_*:

```
---

- name: Create users through loop
  hosts: localhost
  vars:
    users: [jerry,kramer,eliane]

  tasks:
  - name: Create users
    user:
       name: '{{ item }}'
    with_items: '{{users}}'
```

- Both playbooks are used to create multiple users by repeating the same task
- The first playbook uses the modern loop keyword, which directly loops over a list of usernames
- The second playbook uses the older with_items method, which works the same but is now less preferred
- In both cases, Ansible runs the task once per username
- The variable item holds the current value in each loop iteration
- Variable references can be written with double quote or single quote, with spaces or without space inside curly braces, like: '{{item}}', '{{ item }}', "{{item}}", or "{{ item }}" all are valid references.
- Spaces inside {{ }} and using single or double quotes do not affect functionality
- Using {{ item }} is recommended for better readability

40. Example playbooks for installing multiple packages

Ans ->

```
---

- name: Install packages thru loop
  hosts: localhost
  vars:
   packages: [ftp,telnet,htop]

 tasks:
 - name: Install packages
   yum:
      name: {{ packages }}
      state: present
```

This above playbook:

- Runs on **localhost** and is used to install multiple packages (ftp, telnet, htop).
- The package names are stored in a variable called packages as a list.
- In the task, the yum module is given the entire list at once using name: "{{ packages }}".
- The yum module natively supports lists, so it installs **all the packages in a single task execution.**

Why a loop is not needed here

- Some Ansible modules (like yum, dnf, apt) can accept a list of values directly.
- Since yum can handle multiple package names at once, there is **no need to loop over each package individually**.
- This makes the playbook **simpler**, **cleaner**, and **more efficient**.

Rule to remember: Use loops only when the module cannot handle lists by itself; if a module supports lists, prefer passing the list directly.

41. Few more examples of using loops in Ansible.

Ans ->

This below playbook loops over files that match a **file pattern** on the control node and copy to the **/tmp** directory:

```
---
- name: copy config files
  hosts: all
  tasks:
    - name: Copy all config files
      copy:
        src: "{{ item }}"
        dest: /tmp/
      with_fileglob:
        - "/etc/*.conf"
```

This below playbook is used when you need a **numeric sequence**, such as file numbering or repeated actions:

```
---
- name: Create files in sequence
  hosts: all
  tasks:
    - name: Create numbered files
      file:
        path: "/tmp/file{{ item }}"
        state: touch
      with_sequence: start=1 end=3
```

This below playbook repeats a task until a condition is met or retries are exhausted:

```
---
- hosts: all
  tasks:
    - name: Wait until service is up
      command: systemctl is-active httpd
      register: result
      until: result.stdout == "active"
      retries: 5
      delay: 3
```

This below playbook used when looping over a **dictionary** instead of a list and working with key-value pairs:

```
---
- name: working with key-value pair (dictionary)
  hosts: all
  tasks:
    - name: Print key-value pairs
      debug:
        msg: "Key={{ item.key }}, Value={{ item.value }}"
      with_dict:
        user1: alice
        user2: bob
```

This playbook used when each loop item has **multiple attributes**, accessed using dot notation:

```
---
- name:
  hosts: all
  tasks:
    - name: Create users
      user:
        name: "{{ item.name }}"
        uid: "{{ item.uid }}"
      loop:
        - { name: alice, uid: 1001 }
        - { name: bob, uid: 1002 }
```

42. Explaina about Ansible vault and it's different options.

Ans ->

Ansible Vault is used to **secure sensitive data** in Ansible, such as **playbooks**, **passwords**, **secrets**, **API keys**, and **private variables**, by **encrypting files** so they cannot be read in plain text.

What Ansible Vault does (simple words):

- Encrypts sensitive files so others **cannot see the content**
- Allows Ansible to **use encrypted data during execution**
- Keeps secrets **safe inside playbooks and roles**

Examples:

First create a playbook with ansible-vault using the command:

```
ansible-vault create secretPlaybook.yml
```

This command will:

- Prompt you to enter a new vault password
- Creates an **encrypted file called** secretPlaybook.yml
- Opens vi editor to add file contents

Once done just save and exit the vi editor. So now the playbook is encrypted and you cannot view the file as plain text, if you want to execute the playbook then you have to use the below command:

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

This will prompt you to enter the vault password that you've set during creation of the secretPlaybook.yml file, once the password is provided then the playbook executes successfully.

But if you run only the command: ansible-playbook secretPlaybook.yml this will through you an error, cause the file is encrypted.

Different options with ansible-vault:

1. Edit an encrypted file:

```
ansible-vault edit secrets.yml
```

- Opens the file after decrypting it
- Saves it back encrypted

2. View encrypted file:

```
ansible-vault view secrets.yml
```

- Displays the content without modifying it

3. Encrypt an existing file:

```
ansible-vault encrypt vars.yml
```

- Converts a normal file into an encrypted file

4. Decrypt a file:

ansible-vault decrypt secrets.yml

- Converts encrypted file back to plain text

5. Change vault password:

ansible-vault rekey secrets.yml

- Changes the encryption password

43. Encrypt string within a playbook.

Ans ->

Encrypting a **string** with **Ansible Vault** is useful when you need to **secure a single sensitive value**, such as a **password**, **API token**, or **secret key**, without encrypting an **entire file**.

This approach is especially helpful when only a few values in a playbook or variable file are confidential and the rest can remain readable, while still keeping the secret safe, encrypted, and which means you can safely commit and push the playbook to GitHub or any version control system without exposing sensitive information, while Ansible securely decrypts it at runtime.

Steps to encrypt a string:

Run the below command from your terminal:

ansible-vault encrypt_string 'mypassword' --name 'db_password'

- You will be asked for a vault password.
- Ansible outputs an encrypted variable.

Example output:

```
db_password: !vault |
      $ANSIBLE_VAULT;1.1;AES256
      616263646566...
```

Use the encrypted string in a playbook:

You can paste the encrypted output directly into a playbook or vars file:

```
---
- name: Vault string example
  hosts: all

  vars:
    db_password: !vault |
      $ANSIBLE_VAULT;1.1;AES256
      616263646566...

  tasks:
    - name: Use encrypted password
      debug:
        var: db_password
```

Run the playbook using the command:

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

Ansible automatically decrypts the string at runtime using the vault password.

44. Additional Ansible commands.

Ans ->

ansible-doc:

- Used to **view documentation** for Ansible **modules** and **plugins**.
- Helps you understand module usage, options, and examples
- Very useful while writing playbooks

Example: ansible-doc yum

ansible-pull:

- Used to **pull playbooks from a Git repository** and run them locally.
- Mostly used in pull-based automation
- Target machine pulls and applies its own configuration

Example: ansible-pull -U https://github.com/user/repo.git site.yml

ansible-config:

- Used to **view and manage Ansible configuration** settings.
- Shows where Ansible gets its config from
- Helps in debugging configuration issues

Example: ansible-config list

ansible-galaxy:

- Used to **download, manage, and create** roles and collections.
- Reuse community roles instead of writing everything from scratch
- Can also create a role skeleton

Example: ansible-galaxy install geerlingguy.apache

ansible-test:

- Used by developers to **test Ansible plugins**, **modules**, and **collections**.
- Mostly for Ansible contributors
- Not commonly used in day-to-day automation

Example: ansible-test sanity

ansible-connection:

- Used to **test and troubleshoot Ansible connection plugins**.
- Helps debug SSH and other connection issues
- Rarely used in normal workflows

Example: ansible-connection localhost

ansible-inventory:

- Used to **view**, **test**, and **debug** inventory files.
- Shows hosts and groups Ansible sees
- Useful when inventory is complex

Example: ansible-inventory --list

ansible-console:

- Used to **run Ansible commands interactively**.
- Opens a shell-like interface
- Useful for quick ad-hoc testing

Example: ansible-console

In short: These commands help you document (ansible-doc), configure (ansible-config), manage roles (ansible-galaxy), inspect inventory (ansible-inventory), test (ansible-test), debug connections (ansible-connection), automate via pull model (ansible-pull), and run Ansible interactively (ansible-console).