**Ansible In-Depth: Course Outline**

**Module 1: Introduction to Ansible**

* **Section 1.1: Course Welcome & Overview**
  + Introduction by Rahul.
  + Brief mention of topics to be covered (detailed later).
  + Recommendation for beginners to follow sequentially.
  + Guidance for experienced users to use timestamps for specific chapters.
  + Highlight of the final real-world project: Deploying a blog with a custom Ansible Playbook.
* **Section 1.2: What is Ansible?**
  + Definition: Ansible is an open-source automation tool.
  + **Core Capabilities:**
    1. **Software Installation:** Automate software installation across multiple servers.
    2. **Configuration Management:** Add, update, or delete server configurations.
    3. **Application Deployment:** Deploy web applications or any other applications, automating the process.
* **Section 1.3: The "Why Ansible?" Demo - Manual vs. Automated Installation**
  + **Scenario:** Install Python, create a directory, and install Apache2 on a remote server.
  + **Part 1: The Manual Way (Without Ansible)**
    1. Method: SSH into the remote server.
    2. Commands executed:
       - **apt-get update**
       - **apt-get install python3** (or verify existing installation)
       - **mkdir basic\_http\_dir**
       - **apt-get install apache2 -y**
       - **systemctl enable apache2**
       - **systemctl status apache2**
    3. Observation: Effective for one server, but tedious and error-prone for many. Scripts can be non-intuitive.
  + **Part 2: The Ansible Way (A Glimpse)**
    1. Concept: Using an Ansible Playbook (**.yml** file).
    2. Playbook Structure (brief overview):
       - Tasks with descriptive names (e.g., "Install Python," "Create Directory," "Install Apache2").
       - Self-explanatory and intuitive compared to raw shell commands.
    3. Execution: Single **ansible-playbook** command.
    4. Benefits: Automation, readability, consistency.
    5. Verification: SSH into the server post-playbook execution to check Python, directory, and Apache2 status.
  + **Key Takeaway:** Ansible simplifies and automates repetitive tasks, making them more manageable and understandable.

**Module 2: Getting Started with Ansible**

* **Section 2.1: Why Ansible is "Agentless"**
  + Diagram explanation: Developer machine (with Ansible installed) <-> SSH <-> Remote Servers.
  + Ansible is installed on the control node (e.g., developer's laptop).
  + It connects to remote servers using SSH (Secure Shell).
  + No Ansible agent software needs to be installed or running on the managed/remote servers.
  + Playbook instructions are executed over SSH.
* **Section 2.2: Ansible Playbooks & YAML Basics**
  + Playbook file extension: **.yml** (or **.yaml**).
  + Written using YAML (YAML Ain't Markup Language) conventions.
  + Brief example: Snippet for installing Apache server.
* **Section 2.3: Installing Ansible**
  + **On Windows:**
    - Recommendation: Use WSL (Windows Subsystem for Linux).
    - Steps:
      1. Install WSL utility (referencing Canonical Ubuntu documentation for Windows 10/11).
      2. Install a Linux distribution (e.g., Ubuntu LTS) from the Microsoft Store.
      3. Configure the Linux distro (username/password).
      4. Once Ubuntu (or other distro) is running in WSL, follow its specific Ansible installation steps.
  + **On Ubuntu/Debian-based systems:**
    - **sudo apt update**
    - **sudo apt install software-properties-common** (to manage repositories)
    - **sudo add-apt-repository --yes --update ppa:ansible/ansible** (adds Ansible PPA)
    - **sudo apt update** (refresh package list after adding PPA)
    - **sudo apt install ansible**
    - Verify: **ansible --version**
  + **On CentOS/RHEL-based systems:**
    - **sudo yum update** (or **dnf update** for newer versions)
    - **sudo yum install epel-release** (Extra Packages for Enterprise Linux)
    - **sudo yum install ansible** (or **dnf install ansible**)
    - Verify: **ansible --version**
  + **On macOS:**
    - Prerequisite: Homebrew package manager.
    - Command: **brew install ansible**
    - Verify: **ansible --version**

**Module 3: Ansible Project Structure**

* **Section 3.1: Overview of a Typical Project**
  + Key directories/files:
    - **inventory/**: Contains host definitions.
    - **roles/**: Contains reusable sets of tasks, variables, handlers.
    - **<playbook\_name>.yml**: The main playbook file.
* **Section 3.2: The Inventory File**
  + Purpose: Defines the hosts (servers) Ansible will manage.
  + Location: Typically **inventory/<project\_name>/hosts** or just **inventory/hosts**.
  + Format:
    - Can be simple IP addresses or domain names.
    - Can be grouped (e.g., **[gcp\_hosts]**).
    - Example:

ini

[gcp\_host]

10.X.X.X ansible\_ssh\_private\_key\_file=/path/to/private\_key

* + Connects Ansible to the target remote server(s).
* **Section 3.3: Roles and Tasks**
  + **Roles:**
    - A way to organize playbooks into reusable components.
    - Directory structure: **roles/<role\_name>/**
    - Inside a role:
      * **tasks/main.yml**: Defines the main tasks for this role.
      * (Other optional directories: **vars/**, **handlers/**, **templates/**, **files/**)
  + **Tasks (within tasks/main.yml):**
    - Individual units of work to be performed on remote hosts.
    - Example: A role named **python** might have tasks in **roles/python/tasks/main.yml** to:
      * Install Python.
      * Create a specific directory.
      * Install Apache.
* **Section 3.4: The Main Playbook File (<playbook\_name>.yml)**
  + Orchestrates the execution of roles and tasks on specified hosts.
  + Key elements:
    - **name:**: A descriptive name for the play.
    - **hosts:**: Specifies which hosts or groups from the inventory to target (e.g., **all**, **gcp\_host**).
    - **remote\_user:**: The username to use when connecting to remote hosts via SSH (e.g., **ubuntu**, **rahulv**).
    - **roles:**: A list of roles to apply to the hosts.
      * Example:

yaml

- hosts: all

remote\_user: rahulv

roles:

- python

* **Section 3.5: Executing the Playbook**
  + Command: **ansible-playbook -i <path\_to\_inventory\_host\_file> <path\_to\_main\_playbook.yml>**
  + Example: **ansible-playbook -i inventory/vm\_setup\_playbook/hosts vm\_setup\_playbook.yml**
  + Process:
    - Ansible reads the playbook.
    - Connects to hosts defined in inventory (using **remote\_user** and SSH keys if configured).
    - Executes tasks defined in the specified roles.
    - Output shows status: **ok** (no change), **changed** (task made a change), **failed**, **skipped**.
  + Verification: Log into the server to confirm changes (Python installed, directory created, Apache running).

**Module 4: Secure Connections with SSH Keys**

* **Section 4.1: Understanding SSH Key Pairs**
  + Two keys: Public key and Private key.
  + Private Key: Kept secure on the Ansible control node (your local machine).
  + Public Key: Copied to the remote server(s) into the **~/.ssh/authorized\_keys** file of the target user.
  + Enables passwordless SSH authentication.
* **Section 4.2: Generating SSH Keys**
  + Command: **ssh-keygen**
  + Options:
    - **-t rsa** (type of key, RSA is common)
    - **-b 4096** (bit length for security)
    - **-f /path/to/your/key\_name** (e.g., **~/.ssh/ansible\_demo**)
  + Prompts for a file to save the key (defaults to **~/.ssh/id\_rsa**).
  + Prompts for a passphrase (optional but recommended for private key security).
  + Generates two files: **ansible\_demo** (private key) and **ansible\_demo.pub** (public key).
* **Section 4.3: Copying the Public Key to Remote Server(s)**
  + Command: **ssh-copy-id -i /path/to/your/public\_key.pub user@remote\_server\_ip**
  + Example: **ssh-copy-id -i ~/.ssh/ansible\_demo.pub rahulv@10.X.X.X**
  + Prompts for the remote user's password once to copy the key.
* **Section 4.4: Configuring Ansible to Use the Private Key**
  + In the inventory file (**hosts**):
    - Add **ansible\_ssh\_private\_key\_file=/path/to/your/private\_key** next to the host entry.
    - Example:

ini

[my\_server]

10.X.X.X ansible\_ssh\_private\_key\_file=~/.ssh/ansible\_demo

* + When **ansible-playbook** runs, it will use this specified private key for SSH authentication.

**Module 5: YAML for Ansible Playbooks**

* **Section 5.1: What is YAML?**
  + Full form: "YAML Ain't Markup Language." (Recursive acronym)
  + Data serialization language, human-readable.
  + Not a markup language like XML/HTML (no tags).
* **Section 5.2: The Importance of Indentation**
  + YAML uses indentation (spaces, typically 2) to define structure and hierarchy.
  + Consistent indentation is critical.
  + Defines parent-child relationships between data items.
  + Different horizontal positions define nesting levels.
* **Section 5.3: Basic YAML Syntax for Ansible**
  + **Document Start:** **---** (three dashes) often used to indicate the start of a YAML document.
  + **Lists/Sequences:** Items start with a hyphen and a space (**-**).
    - Example (task definition):

yaml

- name: Create a directory

*# ... other attributes*

* + **Dictionaries/Mappings (Key-Value Pairs):**
    - **key: value**
    - Nested dictionaries are indented.
    - Example (file module):

yaml

- name: Create a directory

file: *# Parent attribute (module name)*

path: /srv/myapp/somedir *# Child attribute*

state: directory *# Child attribute*

* + - The **name** attribute in a task is for description.
    - Module names (like **file**) are parent attributes, their parameters (**path**, **state**) are child attributes.
  + **Comments:** Lines starting with **#** are comments.
* **Section 5.4: YAML in Inventory Files**
  + Inventory files (**hosts** or **hosts.ini**) can be in INI format (as shown previously) or YAML format.
  + INI format:

ini

[gcp\_host]

10.X.X.X

* + YAML format (allows more complex structures and variable definitions):

yaml

all:

hosts:

gcp\_host\_yaml:

ansible\_host: 10.X.X.X

*# other host-specific variables*

children:

*# define groups*

* + Key is to be consistent and choose a format that suits the complexity.

**Module 6: Ansible Handlers**

* **Section 6.1: What are Handlers?**
  + Tasks that are run only when "notified" by another task.
  + Typically used to restart services or trigger actions after a configuration change.
  + Handlers are run only once at the end of the play, even if notified multiple times.
  + They only run if the notifying task reports a "changed" state.
* **Section 6.2: Use Case: Restarting a Service**
  + Scenario: Update website content (e.g., **index.html**) and then restart the webserver (e.g., Nginx) for changes to take effect.
  + Tasks:
    1. Install Nginx (if not present).
    2. Copy the new **index.html** to the server.
    3. Copy updated HTML content (simulating an update).
* **Section 6.3: Implementing Handlers in a Playbook**
  + **notify keyword:** Used in a task to trigger a handler.

yaml

- name: Copy updated HTML to destination

copy:

src: update.html

dest: /var/www/html/index.html

notify: Restart Nginx Server *# Name of the handler*

* + **handlers section:** Defines the handler tasks. Usually at the same level as **hosts**, **vars**, **roles**.

yaml

handlers:

- name: Restart Nginx Server *# Must match the name in 'notify'*

service:

name: nginx

state: restarted

* + Can also be defined in **roles/<role\_name>/handlers/main.yml**.
  + The playbook execution shows the handler being run after the notifying task completes with a change.

**Module 7: Ansible Variables**

* **Section 7.1: Introduction to Variables**
  + Purpose: Store values that can be reused throughout playbooks, making them dynamic and flexible.
  + Project structure for examples: **ansible\_variables\_project/** with playbook and **my\_vars.yml**.
* **Section 7.2: Defining and Using Basic Data Types**
  + **Defining in Playbook:** Using the **vars:** section.

yaml

- hosts: localhost

vars:

main\_playbook\_variable: "Hello World from main Playbook"

is\_enabled: true

tasks:

- name: String variable from Main Playbook

debug:

msg: "{{ main\_playbook\_variable }}"

* + **String Variables:**
    - Example: **main\_playbook\_variable: "Hello World"**
    - Usage: **{{ main\_playbook\_variable }}** (Jinja2 templating syntax)
  + **Boolean Variables:**
    - Example: **is\_enabled: true** (or **false**, **yes**, **no**)
    - Usage with **when** condition:

yaml

- name: Boolean variable task

debug:

msg: "Variable is true"

when: is\_enabled *# or when: is\_enabled == true*

* + **List Variables:**
    - Example:

yaml

fruits:

- apple

- banana

- orange

* + - Usage: **{{ fruits }}** (prints the whole list)
    - Accessing elements by index: **{{ fruits[0] }}** (prints "apple")
  + **Dictionary Variables:**
    - Example:

yaml

fruit\_prices:

apple: 0.50

banana: 0.25

orange: 0.75

* + - Usage: **{{ fruit\_prices }}** (prints the whole dictionary)
    - Accessing values by key: **{{ fruit\_prices.apple }}** or **{{ fruit\_prices['apple'] }}** (prints 0.50)
* **Section 7.3: Registering Variables**
  + Purpose: Capture the output of a task into a variable.
  + Keyword: **register:**
  + Example:

yaml

- name: Get the price of an apple (from dictionary)

debug:

msg: "{{ fruit\_prices.apple }}"

register: apple\_price\_variable *# Registers the entire debug task output*

- name: Print the value of the registered variable

debug:

*# Accessing the specific message from the registered debug output*

msg: "Registered Apple Price: {{ apple\_price\_variable.msg }}"

*Note: The video's example of registering a debug output is a bit meta. More commonly, you register output from****command****,****shell****, or other modules that produce meaningful results.*

* **Section 7.4: Working with Nested Variables & Complex Data Structures**
  + Example:

yaml

fruit\_basket:

- name: John

fruits: [apple, orange]

- name: Yen

fruits: [banana, apple]

* + Accessing nested elements:
    - **{{ fruit\_basket[0].name }}** (John)
    - **{{ fruit\_basket[0].fruits[0] }}** (apple from John's basket)
  + Combining with other dictionaries:
    - Goal: Get price of John's first fruit (apple) using **fruit\_prices** dictionary.
    - **{{ fruit\_prices[fruit\_basket[0].fruits[0]] }}** (evaluates to **fruit\_prices['apple']**)
* **Section 7.5: Jinja2 Filters for Variable Manipulation**
  + Purpose: Transform variable data within templates.
  + Syntax: **{{ variable | filter1 | filter2 }}**
  + Examples:
    - **{{ fruit\_prices.keys() | list }}** (get dictionary keys, convert to a list)
    - **{{ "hello" | upper }}** (converts to "HELLO")
    - **{{ my\_list\_of\_strings | map('upper') | list }}** (convert all strings in a list to uppercase)
    - Video example: **{{ fruit\_prices.keys() | list | map('upper') | list }}** (Get keys, make list, uppercase each, ensure final output is list)
* **Section 7.6: Using External Variable Files (vars\_files)**
  + Purpose: Keep variables separate from the main playbook for better organization.
  + Create a YAML file (e.g., **my\_vars.yml**):

yaml

*# my\_vars.yml*

var\_from\_my\_vars\_yml: "Value from external var file"

* + Include in playbook:

yaml

- hosts: localhost

vars\_files:

- my\_vars.yml *# Path to the variable file*

tasks:

- name: Access variable from vars\_files

debug:

msg: "{{ var\_from\_my\_vars\_yml }}"

* **Section 7.7: Passing Variables at Runtime (--extra-vars)**
  + Purpose: Provide or override variables when running the playbook from the command line.
  + Syntax: **ansible-playbook playbook.yml --extra-vars "version=1.0 other\_var=foo"**
  + Can also pass a JSON string or a YAML file:
    - **--extra-vars '{"key": "value"}'**
    - **--extra-vars "@path/to/vars\_file.yml"** (loads variables from this file)
  + Access in playbook: **{{ version }}**, **{{ other\_var }}**.
  + Example:

yaml

*# playbook.yml*

- hosts: localhost

tasks:

- name: Get value from runtime

debug:

msg: "Version: {{ version }}, Who: {{ who }}"

Run with: **ansible-playbook playbook.yml --extra-vars "version=1.0 who=rahul"**

* **Section 7.8: Variable Precedence**
  + Ansible has a well-defined order of precedence for variables if defined in multiple places.
  + Key takeaway from video: Variables passed via **--extra-vars** (command line) generally have the highest precedence and will override variables defined elsewhere (e.g., in playbook **vars:**, **vars\_files:**).

**Module 8: Environment Variables in Ansible**

* **Section 8.1: Understanding Environment Variables in Ansible Context**
  + Refers to environment variables available to the processes run by Ansible tasks (e.g., **shell** or **command** modules).
  + Can be system environment variables (PATH, HOME, USER) or custom ones.
* **Section 8.2: Setting Environment Variables**
  + **Playbook-level (Global for all tasks in the play):**

yaml

- hosts: localhost

environment:

EXAMPLE\_GLOBAL\_VAR: "foobar\_global"

MY\_VAR\_ONE: "variable\_value\_one\_global"

tasks:

- name: Access global env var

command: echo $EXAMPLE\_GLOBAL\_VAR

register: output

- debug: var=output.stdout

* + **Task-level (Specific to one task):**

yaml

- hosts: localhost

tasks:

- name: Task with its own env var

command: echo $MY\_TASK\_LEVEL\_VAR

environment:

MY\_TASK\_LEVEL\_VAR: "hello\_world\_task\_local"

register: output

- debug: var=output.stdout

* + **Scope:**
    - Playbook-level variables are accessible by all tasks within that play.
    - Task-level variables are only accessible within that specific task and override playbook-level variables with the same name for that task.
  + Demonstration of scope: A task trying to access another task's local environment variable will fail to find it.

**Module 9: Ansible Conditionals (when statement)**

* **Section 9.1: Introduction to Conditionals**
  + Analogous to **if** statements in programming languages.
  + Allows tasks to be executed or skipped based on whether a condition is met.
  + Keyword: **when**
* **Section 9.2: Basic Conditional Logic**
  + **Using a Boolean Variable:**
    - Define a variable: **install\_apache\_flag: true** (in **vars/main.yml** or playbook **vars:**)
    - Task:

yaml

- name: Install Apache web server based on flag

apt:

name: apache2

state: present

when: install\_apache\_flag *# Evaluates if true*

* + - If **install\_apache\_flag** is **false**, the task is skipped.
  + **Using Ansible Facts:**
    - Ansible gathers "facts" (system information) about managed hosts by default.
    - Examples: **ansible\_os\_family**, **ansible\_distribution\_major\_version**.
    - Task:

yaml

- name: Print kernel if Debian and version 20

debug:

var: ansible\_kernel

when:

- ansible\_facts['os\_family'] == "Debian" *# or ansible\_os\_family == "Debian"*

- ansible\_facts['distribution\_major\_version'] == "20"

(Note: The video used **ansible\_distribution\_major\_version > "20"** in one example, string comparison for versions can be tricky, often better to cast to int if possible or use **version\_compare** filter if available.)

* **Section 9.3: Multiple Conditions**
  + **Implicit AND (list format):**

yaml

when:

- condition1

- condition2 *# Both must be true*

* + **Explicit AND/OR Operators:**

yaml

when: condition1 and condition2

when: condition1 or condition2

when: (condition1 and condition2) or condition3

* + - Example from video (using **and**):

yaml

when: ansible\_os\_family == "Debian" and ansible\_distribution\_major\_version >= "20"

* **Section 9.4: Conditionals with Registered Variables**
  + Register the output of a command.
  + Use the registered variable in a **when** condition.
  + Example: Check if a file contains specific text.

yaml

- name: Read content of test file

command: cat /home/ubuntu/test\_file.txt *# Assuming test\_file.txt contains "hi"*

register: test\_file\_content

- name: Print file content if it contains 'hi'

debug:

var: test\_file\_content.stdout

when: "'hi' in test\_file\_content.stdout" *# String 'hi' found in stdout*

* **Section 9.5: Conditionals in Loops**
  + Apply a condition to each item in a loop.
  + Example: Print only numbers greater than 5.

yaml

- name: Loop with conditional

debug:

msg: "Item: {{ item }}"

loop: [0, 2, 4, 6, 8, 10]

when: item > 5

Output will show tasks for 0, 2, 4 being skipped, and tasks for 6, 8, 10 being executed.

**Module 10: Ansible Roles and Task Imports (Deeper Dive)**

* **Section 10.1: Recap of Role Structure**
  + Standard directory structure: **roles/<role\_name>/tasks/main.yml**, **vars/**, **handlers/**, etc.
  + **roles/custom\_role/tasks/main.yml**
  + **roles/install\_apache/tasks/main.yml**
* **Section 10.2: Including Roles in a Playbook**
  + Main playbook (**ansible\_import\_roles\_playbook.yml**):

yaml

- hosts: your\_target\_host

remote\_user: ubuntu

roles:

- custom\_role *# Executes tasks from custom\_role*

- install\_apache *# Executes tasks from install\_apache*

* **Section 10.3: Importing Tasks within a Role (import\_tasks)**
  + Allows breaking down tasks within a single role into multiple files for better organization.
  + **import\_tasks** is static: The content of the imported file is included at playbook parse time.
  + Example: **roles/custom\_role/tasks/main.yml**:

yaml

- name: Import package installation tasks

import\_tasks: packages.yml *# Looks for packages.yml in the same directory*

- name: Import message handling tasks

import\_tasks: messages.yml

- name: Start Nginx server (example main task)

service:

name: nginx

state: started

* + **roles/custom\_role/tasks/packages.yml**:

yaml

- name: Install Nginx

apt: name=nginx state=present

- name: Install Git

apt: name=git state=present

*# ... other package installations*

* + **roles/custom\_role/tasks/messages.yml**:

yaml

- name: Copy HTML file (example message task)

copy:

src: some\_html\_file.html

dest: /var/www/html/

* + *Note:****include\_tasks****is an alternative for dynamic inclusion at runtime, allowing loops and conditionals on the include itself.*
* **Section 10.4: Execution Flow**
  + When the main playbook runs **custom\_role**:
    1. **main.yml** in **custom\_role** starts.
    2. It imports and executes tasks from **packages.yml**.
    3. Then, it imports and executes tasks from **messages.yml**.
    4. Finally, it executes any remaining tasks in **main.yml** (like starting Nginx).
  + Then, the playbook moves to the next role, **install\_apache**, and executes its tasks.

**Module 11: Jinja2 Templating in Ansible**

* **Section 11.1: What is Jinja2 Templating?**
  + A templating engine for Python, widely used by Ansible.
  + Allows creation of dynamic configuration files or scripts.
  + Benefits:
    1. **Dynamic Configuration:** Generate config files with variable values.
    2. **Dynamic Scripts:** Generate scripts with variable content.
* **Section 11.2: Creating a Jinja2 Template File**
  + File extension: **.j2** (e.g., **my\_config.j2**).
  + Content: Base text of the file (e.g., server config) with placeholders for variables.
  + **Variable Placeholder (Interpolation):** **{{ variable\_name }}**
  + Example (**my\_config.j2** for Lighttpd):
  + # ... other lighttpd config ...
  + server.port = {{ server\_port }}

# ... other lighttpd config ...

* **Section 11.3: Using the template Module in Ansible**
  + Purpose: Renders a Jinja2 template file with provided variables and copies it to the remote host.
  + Key parameters:
    1. **src**: Path to the **.j2** template file on the control node.
    2. **dest**: Path where the rendered file will be placed on the remote host.
  + Passing variables: Use the **vars:** section in the task or playbook, or any other Ansible variable definition method.
  + Example task (within **roles/install\_lighttpd/tasks/main.yml**):

yaml

- name: Install Lighttpd web server

apt:

name: lighttpd

state: present

- name: Create custom configuration for Lighttpd from template

template:

src: my\_config.j2 *# Assumes my\_config.j2 is in roles/install\_lighttpd/templates/*

*# or relative path if in same dir as task file as per video*

dest: /etc/lighttpd/lighttpd.conf

vars:

server\_port: 80 *# Value for {{ server\_port }} in the template*

notify: Restart Lighttpd *# If config changes, restart service*

- name: Start Lighttpd server

service:

name: lighttpd

state: started

enabled: yes

*(Note: Conventionally, template files go into a****templates/****subdirectory within the role, and****src****would be just****my\_config.j2****. The video keeps it in the****tasks/****dir for simplicity).*

* **Section 11.4: Demonstration and Benefits**
  + Run playbook: Lighttpd installed, **lighttpd.conf** created with **server.port = 80**.
  + Modify **server\_port** in playbook (e.g., to **9090**), rerun playbook.
  + The **template** module detects the change (content of rendered file would differ), updates **lighttpd.conf** on the server, and notifies the handler to restart Lighttpd.
  + Verification: SSH into server, **cat /etc/lighttpd/lighttpd.conf**, check port.
  + Power: Easily manage and update configurations across servers by changing variables.

**Module 12: Ansible Deployment Strategies**

* **Section 12.1: Single Server Deployment**
  + Inventory file contains a single host or a group with one host.
  + Playbook targets this host/group.
  + Example: **ansible\_deployment\_strategy.yml** installing Python.
    - Host file:

ini

[default]

XX.XX.XX.XX

* + - Playbook command: **ansible-playbook -i inventory/hosts ansible\_deployment\_strategy.yml** (no **--limit** needed if **hosts: all** or **hosts: default** is used in playbook).
* **Section 12.2: Multi-Server Deployment and Grouping**
  + Inventory file with multiple hosts, categorized into groups.
  + Example for Blue/Green:

ini

[blue]

IP\_BLUE\_SERVER\_1

IP\_BLUE\_SERVER\_2

[green]

IP\_GREEN\_SERVER\_1

IP\_GREEN\_SERVER\_2

* + Playbook can target **all**, or specific groups like **blue** or **green**.
* **Section 12.3: Limiting Execution with the --limit Flag**
  + Purpose: Run a playbook on a subset of hosts defined in the inventory, regardless of what **hosts:** is set to in the playbook.
  + Syntax: **ansible-playbook -i inventory/hosts playbook.yml --limit <group\_name\_or\_host\_pattern>**
  + Example: **ansible-playbook -i inventory/hosts playbook.yml --limit blue**
    - This will run the playbook *only* on servers in the **[blue]** group.
  + Useful for phased rollouts, testing on specific servers, or strategies like Blue/Green.
* **Section 12.4: Blue/Green Deployment Strategy with Ansible**
  + **Concept:**
    - Two identical production environments:
      * **Green:** The currently live environment, serving active customer requests.
      * **Blue:** An identical, idle environment.
    - Deployment Process:
      * Deploy the new application version to the Blue (idle) environment.
      * Thoroughly test the Blue environment.
      * Switch traffic (e.g., via load balancer or DNS) from Green to Blue. Blue is now live.
      * Green becomes idle. It can be used for rollback or as the next Blue environment.
  + **Ansible Implementation:**
    - Define **[blue]** and **[green]** groups in inventory.
    - When deploying a new release: **ansible-playbook -i inventory/hosts deploy\_app.yml --limit blue**
    - After successful deployment and testing on Blue, manually or via another playbook, switch traffic.
  + **Benefits:** Minimizes downtime, allows for easy rollback (switch traffic back to Green).

**Module 13: Error Handling in Ansible**

* **Section 13.1: ignore\_errors: true**
  + Purpose: Allows a playbook to continue executing even if a specific task fails.
  + Usage: Add **ignore\_errors: true** to a task.
  + Scenario: Trying to copy **index.html** which is missing. Without **ignore\_errors**, the play would halt. With it, the task is marked as failed, but the play continues.

yaml

- name: Copy index.html (might be missing)

copy:

src: index.html *# This file does not exist on control node*

dest: /home/ubuntu/index.html

ignore\_errors: true

* + Playbook summary will show **failed=X** but also **ignored=Y**.
  + Use with caution, as it can mask real problems.
* **Section 13.2: any\_errors\_fatal: true**
  + Purpose: If any task fails on any host, the entire playbook execution stops for *all* hosts in the current play. This is stricter than the default behavior.
  + Usage: Add **any\_errors\_fatal: true** at the play level.

yaml

- hosts: all

any\_errors\_fatal: true

tasks:

*# ...*

* + Scenario: Copying a critical config file. If it fails on one server, you want to stop deployment everywhere to investigate.
  + Combined with **failed\_when** for custom failure.
* **Section 13.3: failed\_when: <condition>**
  + Purpose: Define custom conditions that will cause a task to be marked as failed, even if the module itself technically succeeded.
  + Usage: Add **failed\_when:** with a Jinja2 conditional expression to a task.
  + Scenario: A command should produce specific output. If it doesn't, consider it a failure.

yaml

- name: Copy index.html and check for specific error

copy:

src: index.html *# Missing file*

dest: /home/ubuntu/index.html

register: copy\_result

ignore\_errors: true *# Ignore initial module failure to evaluate failed\_when*

- name: Force failure if copy\_result indicates specific error

debug:

msg: "Deliberately failing based on previous task's error."

*# This task itself will be marked as failed by failed\_when*

failed\_when:

- "'Could not find or access' in copy\_result.msg" *# Check error message*

- "copy\_result.failed == true" *# Check if the copy task itself reported failure*

*# To make the whole play stop if this custom failure occurs:*

*# any\_errors\_fatal: true (at play level, or handle differently)*

*The video demonstrates registering the output of a failing****copy****task (with****ignore\_errors: true****on that copy task itself, so the play doesn't stop there), then using****failed\_when****on a subsequent task to evaluate the****copy\_result****. If****any\_errors\_fatal: true****is set at the play level, this custom failure will halt the play.*

* **Section 13.4: changed\_when: <condition>**
  + Purpose: Define custom conditions for when a task should report a "changed" state. Ansible modules usually determine this, but for **shell** or **command** modules, Ansible doesn't know if a change occurred.
  + Usage: Add **changed\_when:** with a Jinja2 conditional expression.
  + Scenario: A script creates a file. Report "changed" if the script's return code (**rc**) is 0 (success).

yaml

- name: Create a file if it does not exist using 'touch'

command: touch /home/ubuntu/my\_file.txt

register: touch\_result

changed\_when: "touch\_result.rc == 0" *# If command successful, consider it a change*

*# (though 'touch' is idempotent, this illustrates the concept)*

If **my\_file.txt** already exists, **touch** updates timestamp, **rc** is 0. If it doesn't exist, it's created, **rc** is 0. A better example for **changed\_when** would be a script that *conditionally* makes a change.

**Module 14: Ansible Vault for Secrets Management**

* **Section 14.1: What is Ansible Vault?**
  + A feature to encrypt sensitive data (passwords, API keys, certificates) within Ansible projects.
  + Allows storing secrets in version control (e.g., Git) in an encrypted form.
* **Section 14.2: Core ansible-vault Commands**
  + **ansible-vault create <filename.yml>**:
    - Creates a new, encrypted YAML file.
    - Prompts for a new vault password (and confirmation).
    - Opens the file in an editor (default: **vi**) to add content.
    - Example: **ansible-vault create group\_vars/all/my\_vault.yml**
      * Inside editor: **my\_secret: "This is the first secret from Ansible Vault"**
  + **ansible-vault view <encrypted\_filename.yml>**:
    - Displays the decrypted content of a vault file.
    - Prompts for the vault password.
  + **ansible-vault edit <encrypted\_filename.yml>**:
    - Decrypts, opens in editor, and re-encrypts on save.
    - Prompts for vault password.
  + **ansible-vault rekey <encrypted\_filename.yml>**:
    - Changes the password of an existing vault file.
    - Prompts for the old password, then new password (and confirmation).
  + **ansible-vault encrypt <unencrypted\_filename.yml>**: Encrypts an existing plain-text YAML file.
  + **ansible-vault decrypt <encrypted\_filename.yml>**: Decrypts a vault file to plain text.
* **Section 14.3: Using Vaulted Variables in Playbooks**
  + Define variables in the encrypted vault file (e.g., **my\_vault.yml** containing **my\_secret**).
  + Access in playbook like any other variable: **{{ my\_secret }}**.
  + Ansible needs the vault password at runtime to decrypt these variables.
* **Section 14.4: Providing the Vault Password at Runtime**
  + **Prompt for Password (--ask-vault-pass):**
    - Command: **ansible-playbook playbook.yml --ask-vault-pass**
    - Ansible will prompt "Vault password:" in the terminal.
  + **Vault Password File (--vault-password-file or --vault-id @prompt):**
    - Store the vault password in a plain text file (e.g., **~/.vault\_pass.txt**).
      * **SECURITY RISK:** Protect this file strictly (e.g., **chmod 600**).
    - Command: **ansible-playbook playbook.yml --vault-password-file ~/.vault\_pass.txt**
    - Generating a strong password for the file: **openssl rand -base64 32 > ~/.ansible\_vault\_pass.txt**
    - Creating a new vault using this password file: **ansible-vault create new\_vault.yml --vault-password-file ~/.ansible\_vault\_pass.txt**
  + **Environment Variable (ANSIBLE\_VAULT\_PASSWORD\_FILE):**
    - Set an environment variable pointing to the vault password file: **export ANSIBLE\_VAULT\_PASSWORD\_FILE=~/.vault\_pass.txt**
    - Run playbook: **ansible-playbook playbook.yml** (no extra vault flags needed).
    - Ansible automatically detects and uses the password from the file specified by the env var. This is often the most convenient for CI/CD or automated scripts.
* **Section 14.5: Specifying Which Vault File to Use (if not in standard locations like group\_vars or host\_vars)**
  + The video shows using **-e @path/to/my\_vault.yml** (extra vars from file) when running the playbook, if the vault file isn't automatically picked up. This loads variables from that file, and if it's vaulted, Ansible will need the password.
  + Standard practice is to put vaulted files in **group\_vars/all/vault.yml** or similar, which Ansible loads automatically.

**Module 15: Real-World Project - Deploying a Hugo Static Blog to SiteGround**

* **Section 15.1: Project Overview**
  + Blog: **jhooq.com** (static site generated by Hugo).
  + Hosting Provider: SiteGround.
  + Goal: Automate the deployment of locally generated Hugo site to SiteGround server using Ansible.
* **Section 15.2: Local Hugo Workflow**
  + Content written in Markdown files (**.md**).
  + Hugo framework installed locally.
  + **hugo server**: Runs a local development server for previewing changes (e.g., on **localhost:1313**).
  + **hugo**: Command to build the static site. Generates HTML, CSS, JS files into a **public/** directory.
  + Example: Modify a blog post markdown, change **lastmod** date, save. Hugo dev server auto-reloads. Run **hugo** to build final **public/** dir.
* **Section 15.3: Deployment Strategy to SiteGround**
  + **Local:** Generate HTML files (Hugo **public/** directory).
  + **Local:** Create a ZIP archive of the **public/** directory.
  + **Transfer:** Upload the ZIP file to the SiteGround server.
  + **Remote:** Extract the ZIP file on the SiteGround server into the web root directory.
  + **Remote:** Clean up (remove the uploaded ZIP file).
* **Section 15.4: Prerequisites for Ansible Automation**
  + **SSH Key Authentication:**
    - Generate SSH key pair locally (**ssh-keygen**).
    - Add the public key to SiteGround's SSH Key Manager (SiteGround Dashboard -> Devs -> SSH Keys Manager).
    - Note SiteGround SSH username and port (e.g., port **18765**).
  + **Ansible Inventory Configuration:**
    - **inventory/siteground/hosts**:

ini

[siteground\_server]

jhooq.com ansible\_host=jhooq.com ansible\_user=uXX-siteground\_username ansible\_port=18765 ansible\_ssh\_private\_key\_file=~/.ssh/siteground\_private\_key

* + - Variables (**roles/jhooq/vars/main.yml**):
      * **project\_path**: Local path to the Hugo project.
      * **zip\_file\_name**: e.g., **jhooq.zip**.
      * **destination\_directory**: Remote path on SiteGround (e.g., **/home/uXX-username/jhooq.com/public\_html**).
* **Section 15.5: The Ansible Playbook (jhooq\_playbook.yml)**
  + **hosts: siteground\_server**
  + **remote\_user: uXX-siteground\_username** (can also be defined in inventory)
  + **roles: - jhooq**
  + **Tasks within roles/jhooq/tasks/main.yml:**
    - **Run Hugo Build (Locally):**

yaml

- name: Generate static content with Hugo

command: hugo

args:

chdir: "{{ project\_path }}" *# Run command in the project directory*

delegate\_to: localhost *# Ensure this runs on the Ansible control node*

run\_once: true

* + - **Compress public/ Directory (Locally):**

yaml

- name: Compress the public directory

community.general.archive:

path: "{{ project\_path }}/public/\*"

dest: "{{ project\_path }}/{{ zip\_file\_name }}"

format: zip

delegate\_to: localhost

run\_once: true

* + - **Copy ZIP to Remote Server:**

yaml

- name: Copy compressed file to SiteGround

copy:

src: "{{ project\_path }}/{{ zip\_file\_name }}"

dest: "{{ destination\_directory }}/{{ zip\_file\_name }}"

* + - **Extract ZIP on Remote Server:**

yaml

- name: Extract the compressed file on SiteGround

unarchive:

src: "{{ destination\_directory }}/{{ zip\_file\_name }}"

dest: "{{ destination\_directory }}"

remote\_src: yes *# Indicates src is on the remote machine*

*# Extra opts for unarchive might be needed to overwrite, e.g. --overwrite for unzip*

* + - **Remove ZIP from Remote Server:**

yaml

- name: Remove the compressed file from SiteGround

file:

path: "{{ destination\_directory }}/{{ zip\_file\_name }}"

state: absent

* **Section 15.6: Running the Deployment Playbook**
  + Command: **ansible-playbook -i inventory/siteground/hosts jhooq\_playbook.yml**
  + Verification:
    - Check website in browser.
    - May need to clear cache on SiteGround (SiteGround Dashboard -> Speed -> Caching -> Dynamic Cache -> Flush Cache).
* **Section 15.7: Course Conclusion & Further Learning**
  + Recap of the project's real-world application of Ansible concepts.
  + Encouragement to explore further and subscribe for more DevOps content.
  + Mention of premium membership programs.

This detailed breakdown should provide a solid foundation for your Ansible content creation!

* DevOps Engineer with hands-on experience in automating infrastructure, deployments, and monitoring across various environments.
* Proficient in Linux system administration and shell scripting with Bash and Python for automation tasks.
* Skilled in using version control systems like Git and build tools like Maven.
* Experienced in containerization using Docker and orchestration with Kubernetes to manage scalable microservices.
* Well-versed in creating and managing CI/CD pipelines using Jenkins for streamlined software delivery.
* Knowledgeable in Infrastructure as Code (IaC) with Terraform and configuration management using Ansible.
* Cloud-certified with **AWS Developer Associate**, capable of building and managing cloud-native applications using services like EC2, S3, Lambda, IAM, and CloudWatch.
* Exposure to Azure and Azure DevOps for cloud automation and integration.
* Strong understanding of networking fundamentals and system integration.
* Proficient in setting up monitoring and observability stacks using Prometheus, Grafana, and the ELK Stack.
* Passionate about continuous learning, process automation, and improving system reliability and performance.