SKILLS | DevOps and Cloud Computing

Advanced Ansible

Objective

- Organize complex playbooks using Ansible Roles.
- Create dynamic templates using Jinja2 and the template module.
- Encrypt and manage sensitive information using Ansible Vault.
- Debug playbooks and handle errors effectively.

PW SKILLS

Explaining the purpose of roles in simplifying large playbooks.

Let’s see

Ansible roles simplify large playbooks by organizing automation tasks into modular, reusable components. Roles break down complex playbooks into smaller, structured files, grouping related tasks, variables, handlers, templates, and other assets within a standardized directory structure.

- Roles enable reusability by allowing developers to use predefined configurations in multiple playbooks or projects without rewriting code.
- They also streamline workflows by ensuring logical separation of tasks while maintaining scalability and consistency in automation.

Describing the default directory structure of a role

Directory structure

The default directory structure of an Ansible role includes several standardized directories, each serving a specific purpose. Below is the structure:

- tasks/
- handlers/
- templates/
- files/
- vars/
- defaults/
- meta/

PW SKILLS

Demonstrating how to create a role using Ansible

Let’s do it

To create and use a role in Ansible, follow these steps:

Step 1: Create the Role Structure
Use the ‘ansible-galaxy’ command to initialize the role with its default directory structure:

```
ansible-galaxy role init my_role
```

This creates a directory ‘my-role’ with subdirectories like tasks/, handlers/, templates/, etc., for organizing automation components.

Let’s do it

Step 2: Define Tasks in the Role
Edit the tasks/main.yml file inside the role to define tasks. For example:

```yaml
- name: Install Apache
  ansible.builtin.package:
    name: apache2
    state: present
```

This task installs Apache on managed nodes.

Let’s do it

Step 3: Create a Playbook to Use the Role

Write a playbook (e.g., site.yml) that references the role:

```
- name: Web Server Setup
  hosts: webservers
  roles:
    - my_role
```

Alternatively, you can dynamically include the role using:

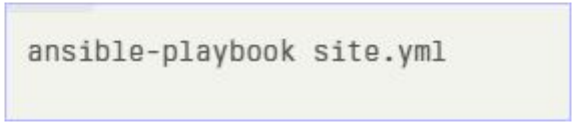
```
- name: Include my_role dynamically
  import_role:
    name: my_role
```

Both approaches allow you to execute the tasks defined in the role

Let’s do it

Step 4: Run the Playbook

Execute the playbook to apply the role's tasks to the target hosts:



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

This modular approach simplifies automation workflows, ensures reusability, and organizes tasks effectively.

Explaining how tasks, handlers, and variables are structured in roles.

Let’s see

In Ansible roles, tasks, handlers, and variables are structured to ensure modularity and reusability:

- **Tasks:** Defined in the tasks/main.yml file, they contain the core automation logic. Tasks are executed sequentially to perform specific operations, such as installing software or configuring files.
- **Handlers:** Stored in the handlers/main.yml file, they define actions triggered by tasks (e.g., restarting a service). Handlers only run when notified by a task.
- **Variables:** Organized into two directories:

defaults/main.yml: Contains default variables with the lowest precedence, allowing them to be overridden easily.

vars/main.yml: Holds higher-priority variables specific to the role.

Demonstrating writing a role for installing and configuring Nginx.

Let’s do it

Step 1: Create the Role Structure
Generate the role structure using ansible-galaxy:

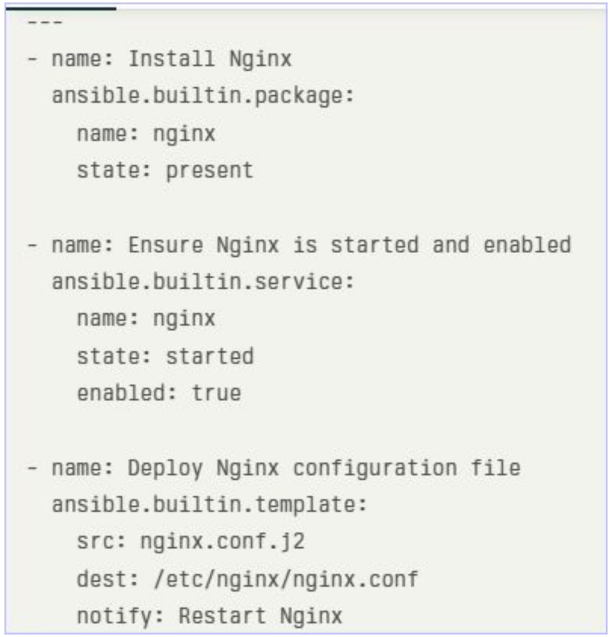
```
ansible-galaxy role init nginx_role
```

This creates the following directories:
tasks/
handlers/
templates/
files/
vars/
defaults/

Let’s do it

Step 2: Define Tasks

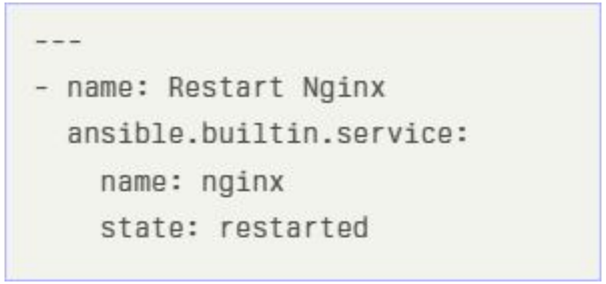
Edit tasks/main.yml to install and configure Nginx:



```yaml
---
- name: Install Nginx
  ansible.builtin.package:
    name: nginx
    state: present

- name: Ensure Nginx is started and enabled
  ansible.builtin.service:
    name: nginx
    state: started
    enabled: true

- name: Deploy Nginx configuration file
  ansible.builtin.template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
    notify: Restart Nginx
```

Let’s do it

Step 3: Define Handlers
Edit handlers/main.yml to restart Nginx when notified:

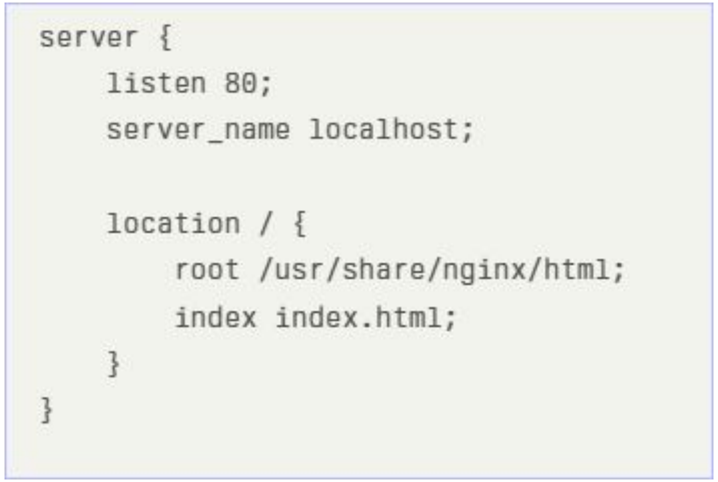


```
---
- name: Restart Nginx
  ansible.builtin.service:
    name: nginx
    state: restarted
```

Let’s do it

Step 4: Add Template

Create a Jinja2 template (templates/nginx.conf.j2) for the Nginx configuration:

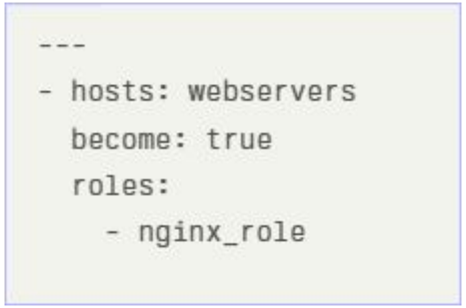


```
server {
    listen 80;
    server_name localhost;

    location / {
        root /usr/share/nginx/html;
        index index.html;
    }
}
```

Let’s do it

Step 5: Create a Playbook to Use the Role

Write a playbook (e.g., site.yml) to apply the role:



```
---
- hosts: webservers
  become: true
  roles:
    - nginx_role
```

Let’s do it

Step 6: Run the Playbook

Execute the playbook to install and configure Nginx:

```
ansible-playbook site.yml -i inventory.ini
```

This role installs Nginx, configures it using a template, and ensures it is running, providing modular automation for web server setup.

PW SKILLS

Showing how to include roles in a playbook

Let’s see

To include roles in an Ansible playbook, you can use either the roles keyword for static inclusion or the include role/import role modules for dynamic inclusion. Here's how:

Static Inclusion

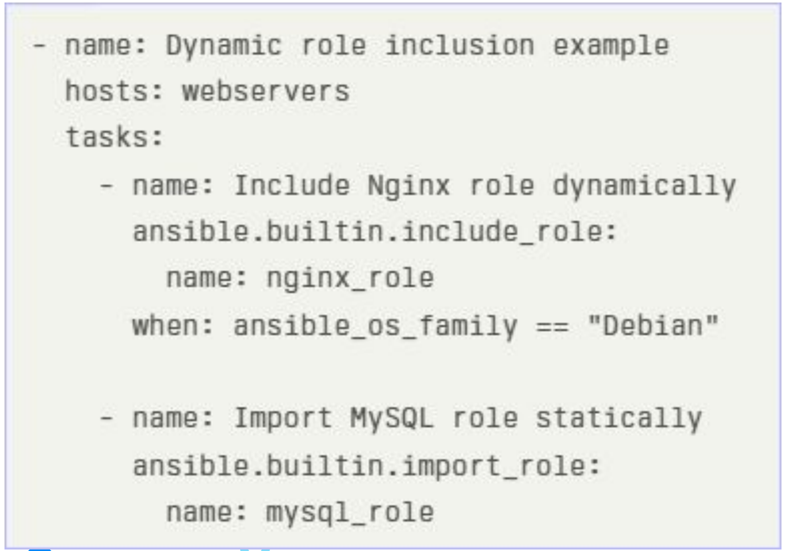
Use the roles keyword to include roles directly in a play:

```
- name: Static role inclusion example
  hosts: webservers
  roles:
    - nginx_role
    - mysql_role
```

Let’s see

Dynamic Inclusion

Use the include role or import role module to include roles dynamically within tasks. This allows more control, such as conditional execution:



```
- name: Dynamic role inclusion example
  hosts: webservers
  tasks:
    - name: Include Nginx role dynamically
      ansible.builtin.include_role:
        name: nginx_role
      when: ansible_os_family == "Debian"

    - name: Import MySQL role statically
      ansible.builtin.import_role:
        name: mysql_role
```

Pop Quiz

Q. Which file does Ansible look for when a role is included in a playbook?

A

tasks.yml

B

main.yml

Pop Quiz

Q. Which file does Ansible look for when a role is included in a playbook?

A

tasks.yml

B

main.yml

Explaining what Ansible Galaxy is and how to find reusable roles.

Ansible Galaxy

Ansible Galaxy is a platform for discovering, sharing, and downloading community-developed roles and collections to accelerate automation projects. It provides pre-packaged units of work, such as roles, playbooks, modules, and plugins, enabling users to reuse existing automation content.

Finding Reusable Roles

1. Visit the Ansible Galaxy website: https://galaxy.ansible.com.
2. Use the search bar to find roles based on keywords, tags, or namespaces.
3. Filter results to refine your search criteria.
4. Install roles using the ansible-galaxy install <role name> command.
5. Installed roles are stored in the default directory /etc/ansible/roles.

PW SKILLS

Demonstrating installing a role from Galaxy

Let’s do it

Step 1: Use the ansible-galaxy install Command
Run the following command to install a specific role:

```
ansible-galaxy role install geerlingguy.nginx
```

Step 2: Specify a Custom Directory (Optional)
To install the role into a specific directory, use the --roles-path option:

```
ansible-galaxy role install geerlingguy.nginx --roles-path ./my_roles
```

Let’s do it

Step 3: Install Multiple Roles Using requirements.yml
Create a requirements.yml file listing all required roles:

```
- name: geerlingguy.nginx
- name: geerlingguy.mysql
```

Install all roles in the file using:

```
ansible-galaxy role install -r requirements.yml
```

Let’s do it

Step 4: Verify Installed Roles
List installed roles with:

```
ansible-galaxy list
```

These steps simplify downloading and managing reusable roles for automation projects.

SKILLS

Take A 5-Minute Break!

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

Explaining Jinja2 templates and their role in dynamic configuration.

Jinja2

Jinja2 is a powerful templating engine for Python that enables dynamic content generation by combining templates with contextual data. It is widely used in web development, automation tools like Ansible, and configuration management systems.

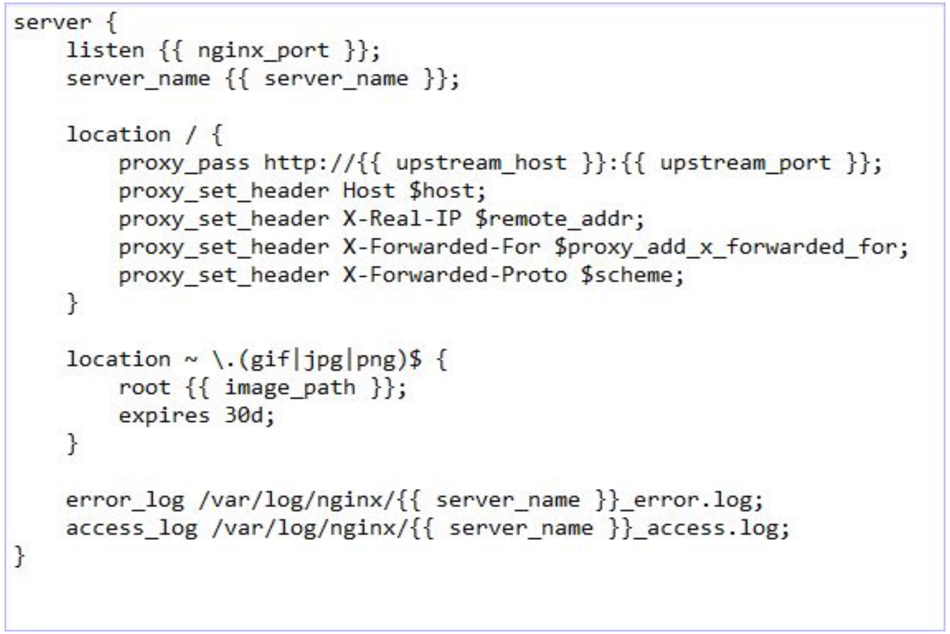
Role of Jinja2 Templates in Dynamic Configuration:

- Dynamic Content Rendering
- Integration with Automation Tools
- Interaction with Context Data
- Flexibility and Reusability

Showing an example template (nginx.conf.j2)

Let’s do it

Here is an example of an Nginx configuration template using Jinja2, named nginx.conf.j2. This template incorporates variables to dynamically configure the server settings.



```
server {
    listen {{ nginx_port }};
    server_name {{ server_name }};

    location / {
        proxy_pass http://{{ upstream_host }}:{{ upstream_port }};
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    location ~ \.(gif|jpg|png)$ {
        root {{ image_path }};
        expires 30d;
    }

    error_log /var/log/nginx/{{ server_name }}_error.log;
    access_log /var/log/nginx/{{ server_name }}_access.log;
}
```

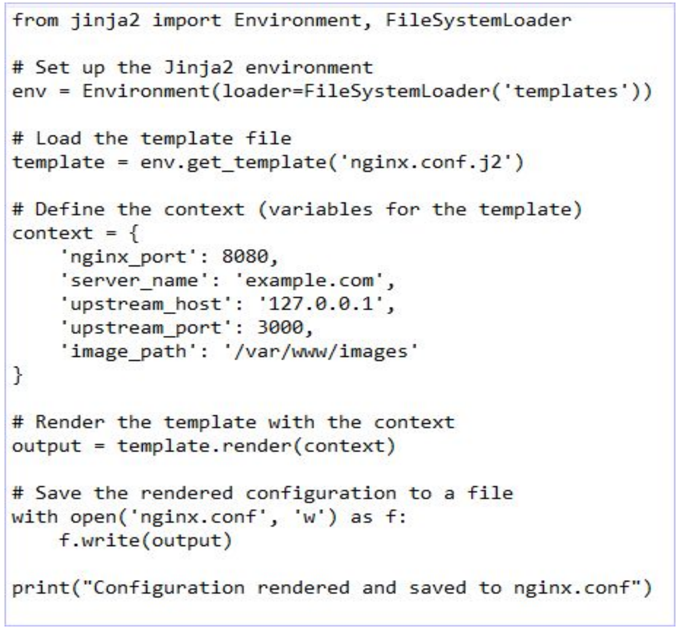
PW SKILLS

Demonstrating using the template module to render the file

Let’s do it

To render a Jinja2 template file using Python, you can use the template module as demonstrated below:

Example Code to Render ‘nginx.conf.j2’



```
from jinja2 import Environment, FileSystemLoader

# Set up the Jinja2 environment
env = Environment(loader=FileSystemLoader('templates'))

# Load the template file
template = env.get_template('nginx.conf.j2')

# Define the context (variables for the template)
context = {
    'nginx_port': 8080,
    'server_name': 'example.com',
    'upstream_host': '127.0.0.1',
    'upstream_port': 3000,
    'image_path': '/var/www/images'
}

# Render the template with the context
output = template.render(context)

# Save the rendered configuration to a file
with open('nginx.conf', 'w') as f:
    f.write(output)

print("Configuration rendered and saved to nginx.conf")
```

PW SKILLS

Explaining the need for encrypting passwords, API keys, and secrets.

Let’s see

Encrypting passwords, API keys, and secrets is essential for safeguarding sensitive data and preventing unauthorized access. Here are the key reasons:

1. Prevent Unauthorized Access
2. Protect Against Interception
3. Compliance and Legal Requirements
4. Mitigate Damage from Breaches
5. Enhance Security Practices

PW SKILLS

Demonstrating creating an encrypted vault file

Let’s do it

To create an encrypted vault file with sensitive data, follow these steps:

1. **Prerequisite:** Ensure Ansible is installed, and optionally install the cryptography package:

```
pip install cryptography
```

2. **Create the Vault File:**

Use the ansible-vault create command to create a new encrypted file:

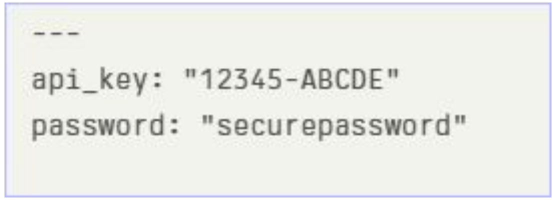
```
EDITOR=nano ansible-vault create secrets.yml
```

- You will be prompted to enter a password twice.
- The specified editor (e.g., nano) will open for you to add content.

Let’s do it

3. Add Sensitive Data:

Inside the editor, add your sensitive information:



```
---
api_key: "12345-ABCDE"
password: "securepassword"
```

4. Save and Exit:

- In nano, press Ctrl-O to save and Ctrl-X to exit.
- The file will be automatically encrypted upon saving.

Let’s do it

5. Verify Encryption:

View the encrypted content of the file using:

```
cat secrets.yml
```

Example output:

```
$ANSIBLE_VAULT;1.1;AES256
353535316566353639663963613966326264356239353633373464...
```

Let’s do it

6. View Decrypted Content:

To view the decrypted content, use:

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

This approach ensures sensitive data is securely stored and accessible only with the vault password.

PW SKILLS

Showing how to encrypt/decrypt an existing file

Let’s see

Encrypt a File

1. Open a terminal and navigate to the directory containing the file:

```
cd /path/to/your/file
```

2. Use the gpg command to encrypt the file:

```
gpg -c filename
```

- The -c option specifies symmetric encryption.
- Enter and confirm a passphrase when prompted.
- This creates an encrypted file named filename.gpg.

Let’s see

Decrypt a File

1. To decrypt the file, use the following command:

```
gpg filename.gpg
```

2. Enter the passphrase used during encryption to decrypt the file.

- The decrypted file will be restored with its original name.

Demonstrating using a vault file inside a playbook

Let’s do it

To use an encrypted Ansible Vault file in a playbook, follow this concise example:

Step 1: Create an Encrypted Vault File

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

Add sensitive variables to secrets.yml:

```
# vars/secrets.yml
db_password: "s3cur3P@ss!"
api_key: "a1b2c3d4e5"
```

Let’s do it

Step 2: Reference the Vault in a Playbook

```
# deploy_app.yml
- hosts: webservers
  tasks:
    - name: Configure database password
      ansible.builtin.lineinfile:
        path: /etc/app/config.conf
        line: "DB_PASSWORD={{ db_password }}"
    - name: Set API key
      ansible.builtin.lineinfile:
        path: /etc/app/api.conf
        line: "API_KEY={{ api_key }}"
  vars_files:
    - vars/secrets.yml  # Automatically decrypted at runtime
```

Let’s do it

Step 3: Execute the Playbook
Provide the Vault password using one of these methods:

- Interactive prompt:

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

- Password file (store password in ~/.vault pass):

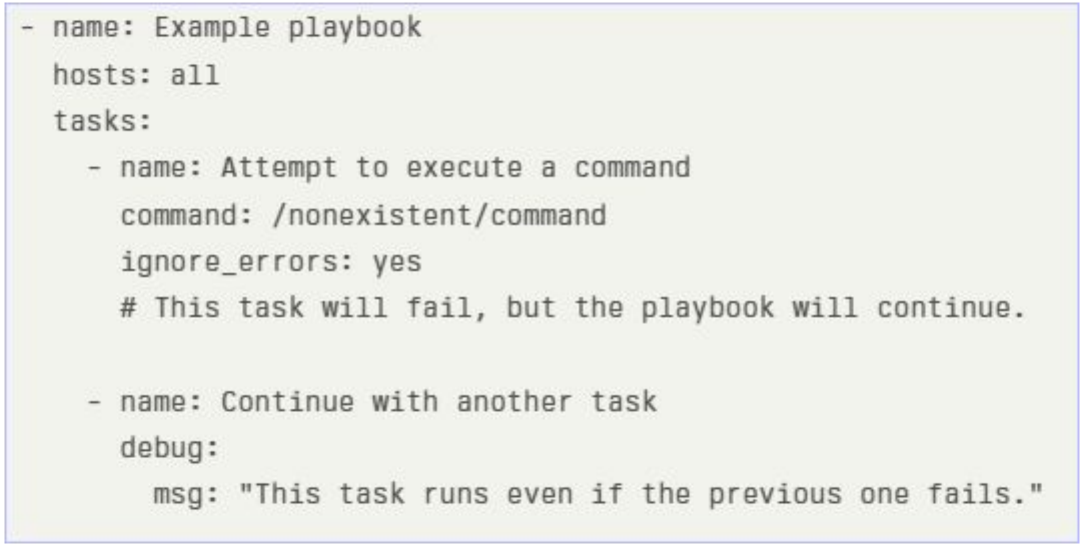
```
ansible-playbook deploy_app.yml --vault-password-file ~/.vault_pass
```

Explaining how to ignore errors using ignore errors: yes:

Let’s see

In Ansible, the ignore errors: yes directive allows a task to fail without stopping the execution of the playbook. Here's how to use it:

Example Playbook Using ignore errors: yes



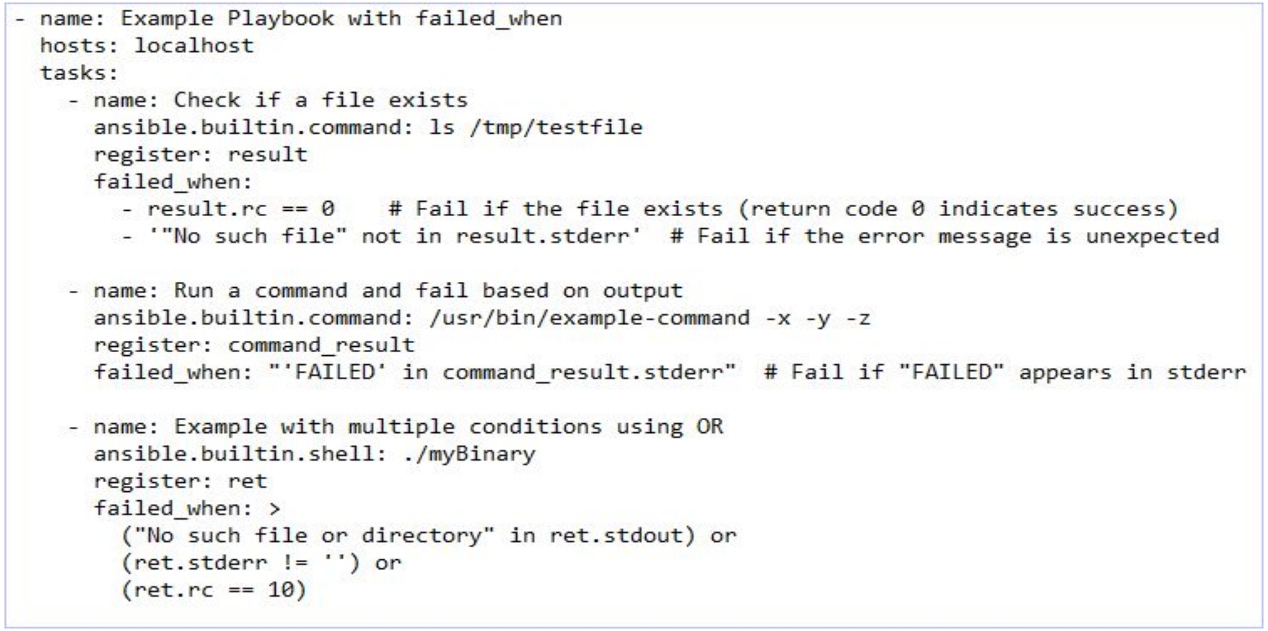
```
- name: Example playbook
  hosts: all
  tasks:
    - name: Attempt to execute a command
      command: /nonexistent/command
      ignore_errors: yes
      # This task will fail, but the playbook will continue.

    - name: Continue with another task
      debug:
        msg: "This task runs even if the previous one fails."
```

Demonstrating failed-when to customize failure conditions:

Let’s do it

The failed-when directive in Ansible allows you to define custom conditions for marking a task as failed. Below is an example demonstrating its usage:

Example Playbook:



```
- name: Example Playbook with failed_when
  hosts: localhost
  tasks:
    - name: Check if a file exists
      ansible.builtin.command: ls /tmp/testfile
      register: result
      failed_when:
        - result.rc == 0    # Fail if the file exists (return code 0 indicates success)
        - '"No such file" not in result.stderr'  # Fail if the error message is unexpected

    - name: Run a command and fail based on output
      ansible.builtin.command: /usr/bin/example-command -x -y -z
      register: command_result
      failed_when: "'FAILED' in command_result.stderr"  # Fail if "FAILED" appears in stderr

    - name: Example with multiple conditions using OR
      ansible.builtin.shell: ./myBinary
      register: ret
      failed_when: >
        ("No such file or directory" in ret.stdout) or
        (ret.stderr != '') or
        (ret.rc == 10)
```

Explaining debugging techniques:

Let’s see

Here are key debugging techniques to troubleshoot Ansible playbooks effectively:

1. Use --check to Simulate Playbook Execution:

- The --check flag runs the playbook in "dry-run" mode, simulating execution without making changes.

Example:

```
ansible-playbook site.yml --check
```

- Useful for verifying tasks before applying changes.

Let’s see

2. Validate YAML Syntax with --syntax-check:

- The --syntax-check flag ensures your playbook's YAML syntax is correct.

Example:

```
ansible-playbook site.yml --syntax-check
```

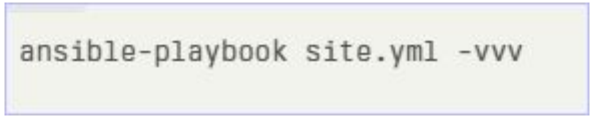
- Prevents errors caused by invalid YAML structure.

Let’s see

3. Increase Verbosity with -vvv:

- Adding verbosity flags (-v, -vv, -vvv, etc.) provides detailed output for troubleshooting.

Example:



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

- Higher verbosity levels (-vvvv) show module arguments and raw outputs, ideal for debugging connectivity or task failures.

PW SKILLS

Time for case study!

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this sessior
- Review your lecture notes
- Note down questions and queries regarding this session ar consult the teaching assistants

Thanks
!