Ansible Error Handling – Hands-on Documentation

This document provides a structured, hands-on walkthrough of error handling concepts in Ansible. It is intended for beginners and early DevOps practitioners who want to understand not only *what* to write, but *why* certain patterns are used in real-world automation.

1. Introduction

Ansible is a configuration management and automation tool that executes tasks sequentially. By default, Ansible stops execution as soon as a task fails. While this behavior is safe, real-world automation often requires controlled handling of failures. This document focuses on practical error-handling patterns used in Ansible playbooks.

2. Environment Setup

• Control Node: Ubuntu (WSL) • Managed Nodes: Ubuntu EC2 instances • Authentication: SSH key-based login • Ansible Version: 2.x • Inventory: inventory.ini file with public IPs

3. Task 1 – Understanding Task Failure and ignore_errors

In this task, an intentional failure is introduced by attempting to install a non-existent package. The goal is to observe Ansible's default behavior and then allow the playbook to continue using `ignore_errors: yes`.

Key Learning: Ansible stops on the first unhandled failure unless explicitly told to continue.

4. Task 2 – Using register, rc, and when

This task demonstrates how to capture command execution results using `register` and make decisions based on the return code (`rc`). The nginx version command is used to determine whether nginx is installed.

Key Learning: `register` stores task results, and `rc` provides a reliable way to check success or failure.

5. Task 3 – Conditional Installation (Idempotency)

Building on Task 2, nginx is installed only if it is not already present. This ensures idempotent behavior, where repeated playbook runs do not cause unnecessary changes.

Key Learning: Separate state detection from state enforcement to build safe automation.

6. Task 4 – Custom Failure Logic (failed_when / fail)

This task introduces logical failures based on business rules rather than command return codes. Disk usage is checked, and the playbook is intentionally failed when usage exceeds a defined threshold.

Key Learning: Command success does not always mean task success.

7. Common Mistakes and Best Practices

• Incorrect YAML indentation can break playbooks • `register` and `ignore_errors` serve different purposes • Error handling should be applied to checks, not actions • Avoid unnecessary `ignore_errors` in production

8. Conclusion

By completing these tasks, a strong foundation in Ansible error handling is established. These patterns are commonly used in production automation and CI/CD pipelines. Readers are encouraged to extend this guide with additional scenarios and screenshots from their own practice.