Learning Objectives

Learners will be able to...

- **Run Commands in Ansible**
- **Understand play results**
- **Catch errors in tasks**

|||info
Make Sure You Know
- Basic shell scripting

Run a Command

Executing Shell Commands

Ansible allows you to execute shell commands on remote hosts using the shell module, `ansible.builtin.shell`. This provides a way to perform a wide range of operations on remote hosts, including: running system commands, managing files, and interacting with services.

```
- name: Run /bin/true
  ansible.builtin.shell: /bin/true
```

The command by default uses `/bin/sh` shell, though
it is possible to use other shells by defining `shell` attribute.

Parameters: Creates and Removes

The command will be executed every time Ansible is executed unless `creates` or `removes` is used with the filename.

A good example of these parameters is considering the case where file should be created/removed by your executable. Essentially, `creates/removes` file is used as flag to execute or skip execution.

If your script is intended to create a file, and the specified file name exists, `creates` will stop the script from executing. This is shown in the example below:

```
- name: Execute a script with log
  ansible.builtin.shell: somescript.sh >> somelog.txt
  args:
    creates: somelog.txt
```

Alternatively, `removes` serves the opposite function: if the file exists it will be removed, and the script won't execute if the file does not exist.

Ignore Errors

important

As with all infrastructure as code, our playbooks should be designed to execute without failures. Writing scripts to avoid execution failures is generally not considered best practice. However, there can be unavoidable complications in real world applications, so it's important to discuss error handling.

Ignore Failed Commands

In some circumstances you may want different behavior resulting from errors. Sometimes a non-zero return code indicates success; you may want a failure on one host to stop execution on all hosts.

When a task fails on a host, by default Ansible stops executing all subsequent tasks on that host. Using the `ignore_errors` directive will execute tasks even after a failure.

```
- name: Does not count as failure
  ansible.builtin.command: /bin/false
  ignore_errors: true
```

warning

`ignore_errors` only works when the task is able to execute but returns a 'failed' status. It does not cause Ansible to disregard errors resulting from undefined variables, connection failures, execution issues such as missing packages, or syntax errors.

`ignore_errors` does not only relate to tasks executing shell commands, but any task errors your play may encounter.

Ignore Unreachable hosts

In a situation where some hosts specified in your inventory cannot be reached due to network issues or other factors, you may not want the playbook to fail or wait for those hosts.

In order for Ansible to disregard connection failures, `ignore_unreachable` can be used for unreachable hosts

```
- name: This task executes, fails, ignores failure
  ansible.builtin.command: /bin/true
  ignore_unreachable: true
```

warning

Similarly to `ignore_errors`, `ignore_unreachable` only applies to unreachable hosts and connection failures. It does not apply to other types of errors.

Recovery From Error

Controlling Errors With Blocks

Similar to exception handling (i.e. try-catch) in other programming languages, rescue blocks define tasks to be executed if any preceding tasks fail.

> warning
>
> Ansible executes rescue blocks only when a task returns a status of failed; rescue blocks are **not** triggered by bad task definitions or unreachable hosts!

```
- name: Play with rescue block
  block:
    - name: Print message
      ansible.builtin.debug:
        msg: 'I execute normally'

    - name: Failure is inevitable
      ansible.builtin.command: /bin/false

    - name: Will not execute
      ansible.builtin.debug:
        msg: 'I didn't execute because previous task failed'
  rescue:
    - name: Print upon error
      ansible.builtin.debug:
        msg: 'I could fix the error if programmed to do so'
```

The Always section

Similar to other programming languages, you can use `always` section with or without rescue, which will be executed regardless of the previous block's state

```
- name: Play with always section
  block:
    - name: Print message
      ansible.builtin.debug:
        msg: 'I execute normally'

    - name: Failure is inevitable
      ansible.builtin.command: /bin/false

    - name: will not execute
      ansible.builtin.debug:
        msg: ':-('
  always:
    - name: Always executes
      ansible.builtin.debug:
        msg: ":-)"
```

Used in conjunction, `rescue` and `always` can offer complex error handling.

Retry

Retrying a Task

As mentioned previously, there are sometimes real-world challenges that can complicated a playbook's execution.

It's easy to imagine a situation where our playbook executes a command to alter a database structure, but the database is restarting so we are unable to connect to it for the time being. We fail to connect but still anticipate being able to after some time.

> topic
>
> Ansible offers robust loop functionality, useful for performing repetitive tasks on multiple hosts or multiple items on a single host, and can help improve the efficiency of infrastructure management.
>
> Check out the Ansible docs here to learn more.

The `until` keyword allows an action to be repeated several times, with or without a specified delay between retries, until success.

```
- name: Retry until target host is reached
  ansible.builtin.ping:
  register: result
  until: result == 0
  retries: 3
  delay: 10
```

- The `register` parameter stores the result of the ping command in the `result` variable
- The `until` parameter specifies the required conditions, in this case until the `result` variable equals 0
- The `retries` parameter specifies the maximum number of retries (3 attempts) before failure
- The `delay` parameter specifies time (10 seconds) between retries

Errors and handlers

definition

Handlers

In Ansible, **handlers** are a type of task that are used to trigger specific actions when notified by another task. They are typically used to perform actions that should only be executed once, such as restarting a service or reloading a configuration file, after one or more other tasks have made changes that require them.

Handlers will be covered in greater detail in the upcoming File Management assignment.

Force Handlers

Since handlers are executed last, when an error happens, the handlers won't be executed. This means that even if a play has changed a service file, it won't reload Systemd or restart the service, leading to unexpected issues or behaviors in the host's state.

Handlers' behavior can be adjusted to avoid these issues in the event of a failure by:

- Using the command-line option `--force-handlers`
- Including the flag `force_handlers: true` in the play
- Adding `force_handlers = True` to ansible.cfg

warning

Be aware that certain errors, such as hosts becoming unreachable, can cause handlers not to run, even if forced.

Code Quality

Ansible Lint

Ansible is a tool for infrastructure as code. As with all written code, quality, consistent style, and avoid bad practices are important.

For this reason using a linter is common practice for software developers, and the same is true for Ansible - using Ansible Lint is good practice.

As its name implies, `ansible-lint` is a YAML linter specified to Ansible playbook conventions. That means when you lint a playbook, it's not just looking at the markup syntax but also how you're using Ansible modules within your tasks.

Using `ansible-lint`

The `main.yaml` playbook in the window on the left is intended to run a script that installs `node.js`, but has several syntactical errors.

Use the button bellow (or run `ansible-lint` in the terminal) and check the output:

Try to fix the errors presented.