Principes des Systèmes d'exploitation

Controlling the workflow

Guillerme Duvillié

1. Workflow overview
2. Pre and post tasks
3. Handlers
4. Error Handling
5. Failure management
6. Exercise

Workflow overview

Execution order

① Pre tasks
② Pre tasks handlers (if triggered)
③ Play level defined roles in the order listed (with dependencies first)
④ Play tasks
⑤ Play handlers (if triggered)
⑥ Post tasks
⑦ Post task handlers if triggered

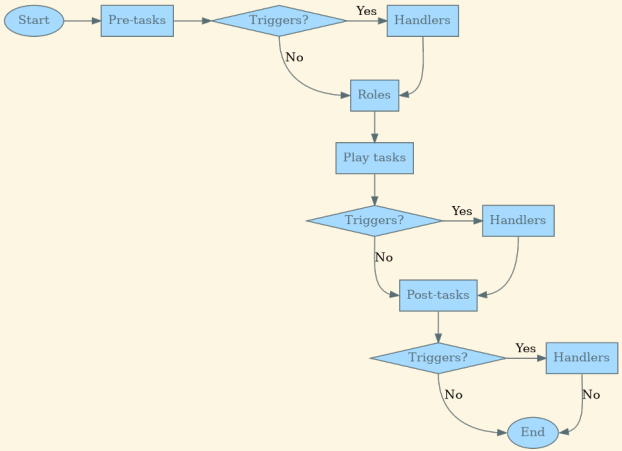

Figure 1: Workflow

Pre and post tasks

Pre tasks

- Tasks executed **before any** other tasks.
- Declared at **play level** with `pre_tasks:` keyword.
- Can be useful to:
  - Set up environment (tunnels, connections, …),
  - Setting facts with values obtained at runtime,
  - Logging,
  - Starting temporary services,
  - …

Pre-tasks example (1)

```yaml
---
- hosts: webservers
  pre_tasks:
    - name: Disable apache to renew letsencrypt certs
      ansible.builtin.service:
        name: "{{ apache_service }}"
        state: stopped
  roles:
    - renew_letsencrypt
    - start_apache
```

Pre-tasks example (2)

```yaml
# Example from https://www.redhat.com/sysadmin/ansible-pretasks-posttasks
---
- hosts: webservers
  pre_tasks:
    - name: Get latest software version from artifact server
      ansible.builtin.uri:
        url: http://artifact-server.example.com:8080/software/latest
        return_content: yes
      register: uri_output
      when: latest_app_version is not defined

    - name: Set latest_software_version fact
      ansible.builtin.set_fact:
        latest_app_version: "{{ uri_output.json.version }}"
      when: latest_app_version is not defined

  roles:
    - appserver
    - apache
    - monitored_host
```

Post tasks

- Tasks executed **after any** other tasks (exctep post tasks handlers).
- Declared at **play level** with `post_tasks:` keyword.
- Can be useful to:
  - Unset environment (set with `pre_tasks:`),
  - Sending notifications,
  - Logging,
  - Manage temporary services,
  - …

Post tasks example

```yaml
---
- hosts: webservers
  pre_tasks:
    - name: Disable apache to renew letsencrypt certs
      ansible.builtin.service:
        name: "{{ apache_service }}"
        state: stopped
  roles:
    - renew_letsencrypt
  post_tasks:
    - name: Start apache service
      ansible.builtin.service:
        name: "{{ apache_service }}"
        state: started
```

Handlers

What are handlers?

- Set of tasks executed when a task **performs a change** on the machine.
- Defined at play level with `handlers:` keyword.
- Needed to be **triggered** by tasks with `notify:` keyword.
- A single task may trigger multiple handlers.
- Only handlers **statically** included can be triggered.

Handlers example

```yaml
---
- name: Upgrade apache server
  hosts: webservers
  become: true
  tasks:
    - name: Install latest apache package
      package:
        name:
          - "{{ apache_pkg }}"
          - "{{ php_pkg }}"
        state: latest
      notify:
        - Restart apache service
  handlers:
    - name: Restart apache service
      ansible.builtin.service:
        name: "{{ apache_service }}"
        state: restarted
```

Handlers execution

- Executed only **once**, after the tasks of the play:
  - Avoid unnecessary tasks execution,
  - Reduce the number of error source.
- They are executed in the **order of definition** (not order of triggering).
- Handlers mechanisms are set up at playbook process (before execution):
  - Use of **variables in handlers name** may lead to **playbook failure** (if variable is not yet defined)
  - Need to used **unique handler names** to avoid conflict. If conflict arises, only the **last on defined** will be executed.
- Can ask to execute triggered handler before end of tasks using the `meta:` module.

Use of meta module

```yaml
---
- name: Upgrade apache server with test
  hosts: webservers
  become: true
  tasks:
    - name: Install latest apache package
      package:
        name: [ "{{ apache_pkg }}", "{{ php_pkg }}" ]
        state: latest
      notify:
        - Restart apache service
    - name: Force handler execution
      meta: flush_handlers
    - name: Test connection (Get request with status 200)
      ansible.builtin.url:
        url: "http://{{ ansible_facts['ansible_default_ipv4']['address'] }}/"
  handlers:
    - name: Restart apache service
      ansible.builtin.service:
        name: "{{ apache_service }}"
        state: restarted
```

Multiple triggering

- Can be done by giving a list of handlers name in `notify:` block.
- Can also be done by ask a handlers to **listen** to a generic **topic**.
- Defined at **task level** with `listen:` keyword inside a handler:
  - Easier to use,
  - Easier to share and reuse (no more based on handlers names).

Listening example

```yaml
# From https://docs.ansible.com/ansible/latest/user_guide/playbooks_handlers.html
---
- hosts: webservers
  handlers:
    - name: Restart memcached
      ansible.builtin.service:
        name: memcached
      state: restarted
      listen: "restart web services"

    - name: Restart apache
      ansible.builtin.service:
        name: apache
        state: restarted
        listen: "restart web services"

  tasks:
    - name: Restart everything
      ansible.builtin.command: echo "this task will restart the web services"
      notify: "restart web services"
```

Other way to detect changes

- Changes can be detected with **`register:`** keyword.
- **`register:`** `foo` keyword saves the output of the task execution into variable foo.
- Such variable can be used with ***`changed`*** mask in conditional to detect changes:

```
1  ---
2  - hosts: webservers
3    name: Handle changes with register
4    tasks:
5      - name: Install Apache
6        register: apache_install
7        package:
8          name:
9            - "{{ apache_pkg }}"
10           - "{{ php_pkg }}"
11         state: present
12
13     - name: Verbose output
14       debug:
15         msg: "Apache was not present"
16       when: apache_install|changed
```

Error Handling

Default behaviour

- When error is encountered on a `host`, the execution of playbooks keeps on going on other hosts.
- Here `host` stands for the set of machines defined by **`hosts:`**.
- Ansible keeps track of the unreachable hosts.
- Any return code different from 0 is interpreted as a failure.

Ignoring failing commands

- Failed tasks can be ignored by using `ignore_errors:` boolean.
- This applies to tasks that can be executed and send back a return code.
- It does not cover:
  - Undefined variables errors,
  - Unreachable hosts,
  - Interrupted/failed execution (segfault, missing dependency, …)

Ignoring unreachable hosts

- Unreachable hosts related failures can be ignored by usin `ignore_unreachable:` boolean.
- Can be defined at **play level** or **task level**.
- List of unreachable hosts can be reset with `meta:` module:

```
1  - name: Reset unreachable hosts
2    meta: clear_host_errors
```

Failures and handlers

- By default is a host encounters a failure, triggered handlers **are not** executed.
- This may lead to host being in unexpected state.
- The behaviour can be overridden with:
  - `force_handlers:` boolean.
  - `--force-handlers` option given to `ansible-playbook` command.
  - `force_handlers = True` in `ansible.cfg`.

Failure and Change conditions

- Failure condition (by default non-zero return code) can be changed with `failed_when:` keyword.
- Changed condition can be change with `changed_when:` keyword.
- Both take conditions as a parameter.
- As a reminder:
  - Changed conditions impacts handlers triggering.
  - Failed conditions determines runtime stop.

Dummy example

```yaml
---
- hosts: all
  name: Fake success
  tasks:
    - name: Ignore bad creation dir
      command: mkdir /root/a
      ignore_errors: yes
      ignore_unreachable: yes

    - name: Fake success
      command: mkdir /root/a
      register: mkdir_return
      notify: Bouilla
      failed_when: false
      changed_when: mkdir_return.rc == 1

  handlers:
    - name: Bouilla
      debug:
        msg: "Got you, sucker!"
```

Failure management

Failure management

- Playbook can be stopped for all hosts if a single failure occurs by using `any_errors_fatal:` boolean.
- Tasks error can be managed using `block:`, `rescue:` and `always:` sections.
- Similar to `try:`, `except:`, `finally:` structure in python:
  - if error occurs in a `block:`,
  - `rescue:` tasks are executed,
  - `always:` tasks are always executed.
- If `rescue:` succeeds, the failure is not triggered.

Exercise

Exercise

Modify your server installation playbook such that:

- Service management is performed through handlers,
- Ensure that upgrade tasks are performed before any other task,
- Upgrade task does not notifies a change,
- Changes are notified through IRC.