<epam>

Ansible

Configuration management with

ANSIBLE

Inventory

- Default – /etc/ansible/hosts
- ansible.cfg
- **`-i inventory`** option
- Project inventory
- Multiple groups membership
- Nested groups are also available
- Ranges: server[1:5].example.com

Configuration file

- /etc/ansible/ansible.cfg
- ~/.ansible.cfg
- ./ansible.cfg
- $ANSIBLE_CONFIG

Modules

- `ansible-doc`
- `Ad-hoc commands`

Playbooks

- Plays and tasks
- `ansible-playbook --syntax-check`
- `ansible-playbook -C`
- `ansible-playbook --step`

Variables

- `{{ variable }}` and `“{{ variable }}”`
- Where we can define variables?
    - Playbook
    - Inventory
    - Include from variable file
    - Local fact

Ansible Vault

- Used to encrypt and decrypt files
- **`ansible-vault create/view/edit playbook.yml`**
- **`ansible-vault encrypt/decrypt/rekey playbook.yml`**
- **`ansible-playbook --ask-vault-pass`**
- **`--vault-password-file=VAULT-FILE`**
- Store encrypted and non-encrypted variables in separate files

Facts

- Variables that are automatically set and discovered by Ansible on managed hosts
- Contains information about hosts
- By default, all playbook perform fact gathering before running the actual plays
- Ad-hoc **Setup** module
- To show facts, use the debug module to print the value of the **ansible_facts** variable
- You can use a dotted format to refer to a specific fact

Facts

- Disabling fact gathering may seriously speed up playbooks
- Use **gather_facts: no** in the play header to disable
- Even if fact gathering is disabled, it can be enabled again by running the **setup** module in task

Custom Facts

- Custom facts allow administrators to dynamically generate variables which are stored as facts
- ini or json in /etc/ansible/facts.d on managed nodes (*.fact)
- Custom facts are stored in the ansible_facts.ansible_local variable

Loops

- loop
- The list that loop is using can be defined by a variable
- Each item in a loop can be a hash/dictionary with multiple keys in each hash/dictionary

register

- A **register** is used to store the output of a command and address it as a variable
- You can next use the result of the command in a conditional or in a loop

Conditions

- **when** statement
- A condition can be used to run a task only if specific conditions are true
- Playbook variables, registered variables, and facts can be used in conditions and make sure that tasks only run if specific conditions are true

Multiple Conditions

- **When** can be used to test multiple conditions as well
- Use **and** or **or** and group the conditions with parentheses
- Loops and conditionals can be combined

Handlers

- Handlers allow you to configure playbooks in a way that one task will only run if another task has been running successfully
- In order to run the handler, a **notify** statement is used from the main task to trigger the handler
- Handlers typically are used to restart services or reboot hosts
- Handlers are executed after running all tasks in a play
- Only run if CHANGED something, OK – handler will not run
- If one of tasks fails, the handler will not run, but this may be overwritten using **force_handlers: True**

Ansible Blocks

- A block is a logical group of tasks
- It can be used to control how tasks are executed
- One block can, for instance, be enabled using a single **when**
- Items cannot be used on blocks
- Blocks can also be used in error condition handling
    - Use **block** to define the main tasks to run
    - Use **rescue** to define tasks that run if tasks defined in the **block** fail
    - Use **always** to define tasks that will run, regardless of the success or failure of the **block** and **rescue** tasks

Failure Handling

- Ansible looks at the exit status of a task to determine whether it has failed
- When any task fails, Ansible aborts the rest of the play on that host and continues with the next host
- Different solutions can be used to change that behavior
- Use ignore_errors in a task to ignore failures
- Use force_handlers to force a handler that has been triggered to run, even if (another) task fails

Defining Failure States

- Use **failed_when** to specify what to look for in command output to recognize a failure
- The **fail** module can be used to print a message that informs why a task has failed
- To use **failed_when** or **fail**, the result of the command must be registered, and the registered variable output must be analyzed
- When using the **fail** module, the failing task must have **ignore_errors** set to **yes**