Configuration management with

Ansible

Roles

- Playbooks best practices (docs.ansible.com)
- Master playbook
  - Call specific playbooks from master playbook for specific types of hosts
  - Consider using different inventory files to differentiate between production and staging phases
  - Use group_vars/ and host_vars/ to set host related variables
  - Use roles to standardize common tasks

Roles

- Role is a collection of tasks, variables, files, templates and other resources in a fixed directory structure that, as such, can easily be included from a playbook
- Roles should be written in a generic way, such that play specifics can be defined as variables in the play, and overwrite the default variables that should be set in the role
- Using roles makes working with large projects more manageable

Roles

- **defaults** contains default values of role variables. If varialbes are set at the play level as well, these default values are overwritten
- **files** may contain static files that are needed from the role tasks
- **handlers** has a main.yml that defines handlers used in the role
- **meta** has a main.yml that may be used to include role metadata, such as information about author, license, dependencies and more
- **tasks** contains a main.yml defines the role task definitions
- **templates** is used to store Jinja2 templates
- **tests** may contain an optional inventory file, as well as a test.yml playbook that can be used to test the role
- **vars** may contain a main.yml with standard variables for the role (which are not meant to be overwritten by playbook variables)

Role variables

- Variables can be defined at different levels in a role
- **vars/main.yml** has the role default variables, which are used in default role functioning. They are not intended to be overwritten
- **defaults/main.yml** can contain default variables. These have a low precedence, and can be overwritten by variables with the same name that are set in the playbook and which have higher precedence
- Playbook variables will always overwrite the variables as set in the role. Specific variables such as secrets and vault encrypted data should always be managed from the playbook, as role variables are intended to generic

Role location

Roles can be obtained in many ways:

- Your own roles
- Community provides roles through the Ansible Galaxy website
- rhel-system-roles

- Roles can be stored at default location, and from there can easily be used from playbooks
  - ./roles has highest precedence
  - ~/.ansible/roles is checked after that
  - /etc/ansible/roles
  - /usr/share/ansible/roles is checked last

Ansible Galaxy

- galaxy.ansible.com
- `ansible-galaxy install geerlingguy.nginx`
- `ansible-galaxy search ‘wordpress’ –platform EL`

Requirements file

- A requirements file is a yml file that defines a list of required roles that are specified using the **src** keyword – it can contain the name of a role from Galaxy, or a URL to a custom location pointing to your own roles
- Create roles/requirements.yml in the project directory to use it
- Always specify the optional **version** attribute, to avoid getting surprises when a newer version of a role has become available

Creating roles

- **ansible-galaxy init** to create directory structure
- Use local playbooks or Ansible Vault for sensitive data
- Don’t forget to edit the README.md and the meta/main.yml to contain documentation about your role
- Roles should be dedicated to one task/function. Use multiple roles to manage multiple tasks/functions.
- Have a look at existing roles before starting to write your own

Conditional roles

- Conditional roles call a role dynamically, using the **include_role** module
  - This makes it so conditional roles are treated more as tasks
- Conditional roles can be combined with conditional statements
  - Run role only if conditional statement is true
- Use **include_role** in task statement to do so

Order of execution

- Role tasks are always executed before playbook tasks
- Next, playbook tasks are executed
- After that, handlers are executed
- Use **pre_tasks** to define playbook tasks that are to be executed before the tasks in role
  - Handler can be executed before as well
- **post_tasks** can be used to define playbook tasks that are executed after playbook tasks and roles

Inventory

- A static inventory file can be used as a list of managed hosts
- Dynamic inventory can automatically discover hosts, by talking to an external host management system, such as Active Directory, Spacewalk or cloud providers
- Also, multiple inventories can be used, for instance by putting multiple inventory files in a directory and use that as the source of inventory

Dynamic inventory

- Dynamic inventory scripts are available for different environments
- They are used like static inventory files, through ansible.cfg, or using the -i option to the ansible[-playbook] command
- Instead of using community dynamic inventory scripts, you can also write your own

Dynamic inventory scripts

- The only requirement is that the script returns the inventory information in JSON format
- To see the correct output format, use **ansible-inventory --list** on any inventory
- Scripts can be written in any language, but Python is common

Example

- **Tutorial: Configure dynamic inventories of your Azure resources using Ansible**
  - https://docs.microsoft.com/en-us/azure/ansible/ansible-manage-azure-dynamic-inventories

Multiple inventory files

- If the inventory specified is a directory, all inventory files in that directory are considered
- This includes static as well as dynamic inventory
- Inventory files cannot be created with dependencies to other inventory files

Addressing hosts

- By default, hosts are addressed with their host name as specified in inventory
- IP addresses can also be used
- Host groups are common and are defined in inventory
  - Group **all** is implicit and doesn't have to be defined
  - Group **ungrouped** is also implicit and addresses all hosts that are not members of a group
- Wildcards can also be used, **- hosts: '*'** is equivalent to **– hosts: all**
- If special characters are used, always put them between quotes to avoid special characters being interpreted by the shell

Parallelism – Processing order

- Plays are executed in order on all hosts referred to, and normally Ansible will start the next task if this task successfully completed on all managed hosts
- Ansible can run on multiple managed hosts simultaneously, but by default the maximum number of simultaneous hosts is limited to five
- Set **forks = n** in ansible.cfg to change the maximum number of simultaneous hosts
- Alternatively, use **–f nn** to specify the max number of forks as argument to the **ansible[-playbook]** command
- The default of 5 is very limited, so you can set this parameter much higher, in particular if most of the work is done on the managed hosts and not on the control node
- Use the **serial** keyword in the playbook to run hosts through the entire play in batches

Asynchronous Tasks

- Normally, Ansible waits for completion of tasks before starting the next task
- Use the **async** keyword in a task to run the task in the background:
  - **async: 3600** tells Ansible to give the task an hour to complete, note that this will be the maximum amount of time permitted for the job to run
  - **poll: 10** indicates that Ansible will poll every 10 seconds to see if the command has completed
- Using **async** allows the next task to be started so it will make playbooks more efficient
  - Recommended for backup jobs, package updates, large file downloads, etc.
- **async-status**

wait_for

- **wait_for** module can be used in a task to check if a certain condition was met
- Using this module may be useful to verify successful restart of servers, etc.

Rolling updates

- The default behavior of running one task on all hosts, and next proceed to the next task means that in cluster environments you may have all hosts temporarily being unavailable
- Use the **serial** keyword in a playbook to reduce the number of parallel tasks to a value that is lower than what is specified with the **forks** option

Inclusion

- If playbooks grow larger, it is common to use modularity by using includes and imports
- Includes and imports can happen for playbooks as well as tasks
- An **include** is a dynamic process; Ansible processes the contents of the included files at the moment that this import is reached
- An **import** is a static process; Ansible preprocesses the imported file contents before the actual play is started
  - Playbook imports must be defined at the beginning of the playbook, using **import_playbook**

Task files

- A task file is a flat list of tasks
- Use **import_tasks** to statically import a task file in the playbook, it will be included at the location where it is imported
- Use **include_tasks** to dynamically include a task file
- Dynamically including tasks means that some features are not available
  - **ansible-playbook –list-tasks** will not show the tasks
  - **Ansible-playbook –start-at-task** doesn't work
  - You cannot trigger a handler in an imported task file from the main task file
- Best practice: store task files in a dedicated directory to make management easier

When to include task files

- When modularity is required
- When different groups of engineers are responsible for different setup tasks
- If a task needs to be executed only in specific cases

Variables for external plays and tasks

- It is recommended to keep include files as generic as possible
- Define variables independently from the playbook
  - In separate include files
  - Using group_vars and host_vars
  - Or using local facts
- This allows you to process different values on different groups of hosts, while still using the same playbook

Ansible logging

- By default Ansible is not configured to log its output anywhere
- Set **log_path** in ansible.cfg to write logs to a specific destination
  - Create this file in the project directory, /var/log is not writable by the Ansible user and will only work when running the playbook with sudo
- Log rotation

debug module

- The **debug** module is used to show values of variables in playbooks
- It can also be used to show messages in specific error situations
- ansible-doc debug

Check mode

- **ansible-playbook --check**
  - Modules in playbook must support check mode
  - Check mode doesn’t always work well in coditionals
- Set **check_mode: yes** within a task to always run that specific task in check mode
  - This is useful for checking individual tasks
  - **check_mode: no** – this task will never run in check mode

Check mode on templates

- Add **--diff** to an Ansible playbook run to see differences that would be made by template files on a managed hosts
  - ansible-playbook --check --diff playbook.yml

Modules to check

- **uri:** checks content that is returned from a specific URL
- **script:** runs a script from the control node on the managed hosts
- **stat:** checks status of file
- **assert:** this module will fail with an error if a specific condition is not met

Playbook errors

- Start by reading output messages
- -v
  - -v: the output data is displayed
  - -vv: output as well as input data is shown
  - -vvv: adds connection information
  - -vvvv: adds additional information, for instance, about scripts that are executed and the user who’s running tasks