Ansible Workshop - Exercises

Basics

Get to know Ansible and learn to write your first Ansible Playbooks.

3 - Writing your first Playbook

Objective

This exercise covers using Ansible to build two Apache web servers on Red Hat Enterprise Linux. This exercise covers the following Ansible fundamentals:

- Understanding Ansible module parameters
- Understanding and using the following modules
  - package module
  - service module
  - copy module
- Understanding Idempotence and how Ansible modules can be idempotent

Guide

Playbooks are files which describe the desired configurations or steps to implement on managed hosts. Playbooks can change lengthy, complex administrative tasks into easily repeatable routines with predictable and successful outcomes.

A playbook can have multiple plays and a play can have one or multiple tasks. In a task a *module* is called, like the modules in the previous chapter. The goal of a *play* is to map a group of hosts. The goal of a *task* is to implement modules against those hosts.

> **Tip**
>
> Here is a nice analogy: When Ansible modules are the tools in your workshop, the inventory is the materials and the Playbooks are the instructions.

Step 1 - Playbook Basics

Playbooks are text files written in YAML format and therefore need:

- to start with three dashes (`---`)
- proper indentation using spaces and **not** tabs!

There are some important concepts:

- **hosts**: the managed hosts to perform the tasks on
- **tasks**: the operations to be performed by invoking Ansible modules and passing them the necessary options
- **become**: privilege escalation in playbooks

> **Warning**
>
> The ordering of the contents within a Playbook is important, because Ansible executes plays and tasks in the order they are presented.

A Playbook should be **idempotent**, so if a Playbook is run once to put the hosts in the correct state, it should be safe to run it a second time and it should make no further changes to the hosts.

> **Tip**
>
> Most Ansible modules are idempotent, so it is relatively easy to ensure this is true.

Step 2 - Directory Structure and files for your Playbook

Enough theory, it's time to create your first Ansible playbook. In this lab you create a playbook to set up an Apache web server in three steps:

1. Install httpd package
2. Enable/start httpd service
3. Copy over an web.html file to each web host

This Playbook makes sure the package containing the Apache web server is installed on `node1`.

There is a best practice Guide on the preferred directory structures for playbooks. We strongly encourage you to read and understand these practices as you develop your Ansible skills.
That said, our playbook today is very basic and creating a complex structure will just confuse things.

Instead, we are going to create a very simple directory structure for our playbook, and add just a couple of files to it.

If you haven't done this already, on your control host **ansible-1**, create a directory called `ansible-files` in your home directory and change directories into it:

```
[student@ansible-1 ~]$ mkdir ansible-files
[student@ansible-1 ~]$ cd ansible-files/
```

Add a file called `apache.yml` with the following content. As discussed in the previous exercises, use `vi` / `vim` or, if you are new to editors on the command line, check out the editor alternatives again.

```
---
- name: Apache server installed
  hosts: node1
  become: true
```

This shows one of Ansible's strengths: The Playbook syntax is easy to read and understand. In this Playbook:

- A name is given for the play via `name:`.
- The host to run the playbook against is defined via `hosts:`.

- We enable user privilege escalation with `become:`.

> **Tip**
>
> You obviously need to use privilege escalation to install a package or run any other task that requires root permissions. This is done in the Playbook by `become: true`.

Now that we've defined the play, let's add a task to get something done. We will add a task in which the RHEL package manager will ensure that the Apache package is installed in the latest version. Modify the file so that it looks like the following listing:

```
---
- name: Apache server installed
  hosts: node1
  become: true
  tasks:
    - name: Install Apache package
      ansible.builtin.package:
        name: httpd
        state: present
```

> **Tip**
>
> Since playbooks are written in YAML, alignment of the lines and keywords is crucial. Make sure to vertically align the *t* in `task` with the *b* in `become`. Once you are more familiar with Ansible, make sure to take some time and study a bit the YAML Syntax.

In the added lines:

- We started the tasks part with the keyword `tasks:`.
- A task is named and the module for the task is referenced. Here it uses the `package` module.
- Parameters for the module are added:
  - `name:` to identify the package name
  - `state:` to define the wanted state of the package

> **Tip**
>
> The module parameters are individual to each module. If in doubt, look them up again with `ansible-doc`.

Save your playbook and exit your editor.

Step 3 - Running the Playbook

To run your playbook, use the `ansible-playbook <playbook>` command as follows:

Ansible

```
[student@ansible-1 ansible-files]$ ansible-playbook apache.yml
```

What does `Invalid callback for stdout specified` mean?

If you see this error, this is not your fault, but a missing *plugin*.
In the demo environment, only the `ansible-core` package is installed. The missing plugin (a *callback* plugin formats the output Ansible is producing) is not part of the `ansible.builtin` collection, you need to install it.

```
ansible-galaxy collection install community.general
```

If you want to know where this configuration is stored, take a look at the following tip.

Navigator

```
[student@ansible-1 ansible-files]$ ansible-navigator run apache.yml -m stdout
```

Tip

The existing `ansible.cfg` file (which you created previously) provides the location of your inventory file. If this was not set within your `ansible.cfg` file, the command to run the playbook would be:

```
ansible-playbook -i /home/student1/lab_inventory/hosts apache.yml
```

If you want to know which config file is used, run `ansible --version` and observe the output.

Once the playbook has completed, connect to `node1` via SSH to make sure Apache has been installed:

```
[student@ansible-1 ansible-files]$ ssh node1
Last login: Wed May 15 14:03:45 2019 from 44.55.66.77
Managed by Ansible
```

Use the command `rpm -qi httpd` to verify httpd is installed:

```
[ec2-user@node1 ~]$ rpm -qi httpd
Name        : httpd
Version     : 2.4.37
[...]
```

Log out of `node1` with the command `exit` so that you are back on the control host and verify the installed package with an Ansible playbook named `package.yml`. Create the file and paste in the following content:

```
---
- name: Check packages
  hosts: node1
  become: true
  vars:
    package: "httpd"
  tasks:
    - name: Gather the package facts
      ansible.builtin.package_facts:
        manager: auto

    - name: Output message if package is installed
      ansible.builtin.debug:
        msg: "{{ package }} in Version {{ ansible_facts.packages[package][0].version }} is
installed!"
```

Note

The playbook (and some of the following playbooks) make use of *variables*, you will learn about them in the next chapter.

The playbook has two tasks, the first one uses the `package_facts` module, it does what it says, it gathers information about packages. These facts are not gathered by default with the "Gather facts" tasks (which uses the `setup` module) and must be collected separately.
The second task uses the `debug` module. The variable *ansible_facts* is extended with the *packages* key, which contains a dictionary with **all** packages installed on the managed node. The *httpd* package could be installed in multiple versions, therefore every *package* key, in our case *httpd*, is a list. We have installed only one version of *httpd* (thus, we have a list with only one element), we get the version of *httpd* with `[0].version`.

Ansible

```
[student@ansible-1 ansible-files]$ ansible-playbook package.yml
```

Navigator

```
[student@ansible-1 ansible-files]$ ansible-navigator run package.yml -m stdout
```

The output should look like this:

```
PLAY [Check packages] ***************************************************

TASK [Gathering Facts] **************************************************
ok: [ansible]

TASK [Gather the package facts] *****************************************
ok: [ansible]

TASK [Check whether a httpd  is installed] ******************************
ok: [ansible] => {
    "msg": "httpd 2.4.37 is installed!"
}

PLAY RECAP **************************************************************
ansible                    : ok=3    changed=0    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

Execute the command `ansible-playbook apache.yml` for a second time, and compare the output.

Step 4 - Add one more task

The next part of the Ansible playbook makes sure the Apache application is enabled and started on `node1`.

On the control host, as your student user, edit the file `~/ansible-files/apache.yml` to add a second task using the `service` module. The Playbook should now look like this:

```yaml
---
- name: Apache server installation
  hosts: node1
  become: true
  tasks:
    - name: Install Apache package
      ansible.builtin.package:
        name: httpd
        state: present

    - name: Ensure Apache is enabled and running
      ansible.builtin.service:
        name: httpd.service
        enabled: true
        state: started
```

What exactly did we do?

- a second task named "Apache enabled and running" is created
- a module is specified (`service`)
- The module `service` takes the name of the service (`httpd`), if it should be permanently set (`enabled`), and its current state (`started`)

Thus with the second task we make sure the Apache server is indeed running on the target machine. Run your extended Playbook:

Ansible

```
[student@ansible-1 ansible-files]$ ansible-playbook apache.yml
```

- Run the playbook a second time to get used to the change in the output.
- Use an Ansible playbook labeled `service_state.yml` to make sure the Apache (httpd) service is running on `node1`.

```
---
- name: Check Service status
  hosts: node1
  become: true
  vars:
    service: "httpd.service"
  tasks:
    - name: Get state of all service
      ansible.builtin.service_facts:

    - name: Output service state of {{ service }}
      ansible.builtin.debug:
        msg: "{{ ansible_facts['services'][service]['state'] }}"
```

Ansible

```
[student@ansible-1 ansible-files]$ ansible-playbook service_state.yml
```

Navigator

```
[student@ansible-1 ansible-files]$ ansible-navigator run service_state.yml -m stdout
```

This would be the same as checking the service state manually on `node1` with: `systemctl status httpd`.

Step 5 - Extend your Playbook

Check that the tasks were executed correctly and Apache is accepting connections: Make an HTTP request using Ansible's `uri` module in a playbook named `check_httpd.yml` from the control node to `node1`.

```
---
- name: Check URL
  hosts: control
  vars:
    node: "node1"
  tasks:
    - name: Check that you can connect (GET) to a page and it returns a status 200
      ansible.builtin.uri:
        url: "http://{{ node }}"
```

Warning

Expect a lot of red lines and a 403 status!
If you are using the local development environment, remember, you are using containers instead of actual VMs! You need to **append the correct port** (e.g. `node: "node1:8002"`).
Take a look at the table with the ports overview or execute `podman ps` and check the output.

Ansible

```
[student@ansible-1 ansible-files]$ ansible-playbook check_httpd.yml
```

Navigator

```
[student@ansible-1 ansible-files]$ ansible-navigator run check_httpd.yml -m stdout
```

There are a lot of red lines and an error: As long as there is not at least an `index.html` file to be served by Apache, it will throw an ugly "HTTP Error 403: Forbidden" status and Ansible will report an error.

So why not use Ansible to deploy a simple `index.html` file? On the ansible control host, as the `student` user, create the directory `files` to hold file resources in `~/ansible-files/`:

```
[student@ansible-1 ansible-files]$ mkdir files
```

Then create the file `~/ansible-files/files/web.html` on the control node:

```
<body>
<h1>Apache is running fine</h1>
</body>
```

Now, you'll use Ansible's `copy` module in your playbook to *copy* a file from your controller to the managed node(s).

On the control node, as your student user, edit the file `~/ansible-files/apache.yml` and add a new task utilizing the `copy` module. It should now look like this:

```
---
- name: Apache server installation
  hosts: node1
  become: true
  tasks:
    - name: Install Apache package
      ansible.builtin.package:
        name: httpd
        state: present

    - name: Ensure Apache is enabled and running
      ansible.builtin.service:
        name: httpd.service
        enabled: true
        state: started

    - name: Copy file for webserver index
      ansible.builtin.copy:
        src: web.html
        dest: /var/www/html/index.html
        mode: "0644"
        owner: apache
        group: apache
```

What does this new copy task do? The new task uses the `copy` module and defines the source and destination options for the copy operation as parameters, as well as setting permissions and owner of the resulting file.

Run your extended Playbook:

Ansible

```
[student@ansible-1 ansible-files]$ ansible-playbook apache.yml
```

Navigator

```
[student@ansible-1 ansible-files]$ ansible-navigator run apache.yml -m stdout
```

- Have a good look at the output, notice the changes of "CHANGED" and the tasks associated with that change.
- Run the Ansible playbook `check_httpd.yml` using the "uri" module from above again to test Apache. The command should now return a friendly green "status: 200" line, amongst other information.

Step 6 - Practice: Apply to Multiple Host

While the above, shows the simplicity of applying changes to a particular host. What about if you want to set changes to many hosts? This is where you'll notice the real power of Ansible as it applies the same set of tasks reliably to many hosts.

> All right, what about changing the `apache.yml` playbook to run on `node1` **and** `node2` **and** `node3` ?

As you might remember, the inventory lists all nodes as members of the group `web` :

```
[web]
node1 ansible_host=node1.example.com
node2 ansible_host=node2.example.com
node3 ansible_host=node3.example.com
```

Change the playbook `hosts` parameter to point to `web` instead of `node1` :

```
---
- name: Apache server installation
  hosts: web
  become: true
  tasks:
    - name: Install Apache package
      ansible.builtin.package:
        name: httpd
        state: present

    - name: Ensure Apache is enabled and running
      ansible.builtin.service:
        name: httpd.service
        enabled: true
        state: started

    - name: Copy file for webserver index
      ansible.builtin.copy:
        src: web.html
        dest: /var/www/html/index.html
        mode: "0644"
        owner: apache
        group: apache
```

Now run the playbook:

Ansible

```
[student@ansible-1 ansible-files]$ ansible-playbook apache.yml
```

Navigator

```
[student@ansible-1 ansible-files]$ ansible-navigator run apache.yml -m stdout
```

Verify if Apache is now running on all web servers (node1, node2, node3). All output should be green.