IBM TechExchange EMEA 2024 - lab 2078

Fabio Alessandro “Fale” Locati

v2024-01-23

Contents

Ansible

Check the Prerequisites

Objective

- Understand the lab topology and how to access the environment.
- Understand how to work the workshop exercises
- Understand challenge labs

These first few lab exercises will be exploring the command-line utilities of the Ansible Automation Platform. This includes:

- ansible-navigator - a command line utility and text-based user interface (TUI) for running and developing Ansible automation content.
- ansible-core - the base executable that provides the framework, language and functions that underpin the Ansible Automation Platform. It also includes various cli tools like `ansible`, `ansible-playbook` and `ansible-doc`. Ansible Core acts as the bridge between the upstream community with the free and open source Ansible and connects it to the downstream enterprise automation offering from Red Hat, the Ansible Automation Platform.
- Execution Environments - not specifically covered in this workshop because the built-in Ansible Execution Environments already included all the Red Hat supported collections which includes all the collections we use for this workshop. Execution Environments are container images that can be utilized as Ansible execution.
- ansible-builder - not specifically covered in this workshop, `ansible-builder` is a command line utility to automate the process of building Execution Environments.

If you need more information on new Ansible Automation Platform components bookmark this landing page https://red.ht/AAP-20

Guide

Your Lab Environment

In this lab you work in a pre-configured lab environment. You will have access to the following hosts:

Role	Inventory name
Ansible Control Host	10.3.48.100
Managed Host	10.3.48.[100+PARTICIPANT_ID]

Step 1 - Access the Environment

You can access the environment, by connecting via SSH to the Ansible Control Host:

```
ssh USER@10.3.48.100
```

Step 2 - Using the Terminal

Create and navigate to the `rhel-workshop` directory on the Ansible control node terminal.

```
[student@controller ~]$ mkdir ~/rhel-workshop/
[student@controller ~]$ cd ~/rhel-workshop/
[student@controller rhel-workshop]$ pwd
/home/student/rhel-workshop
[student@controller rhel-workshop]$
```

- `~` - the tilde in this context is a shortcut for the home directory, i.e. `/home/student`
- `mkdir` - Linux command to create a directory
- `cd` - Linux command to change directory
- `pwd` - Linux command for print working directory. This will show the full path to the current working directory.

Step 3 - Examining Execution Environments

Run the `ansible-navigator` command with the `images` argument to look at execution environments configured on the control node:

```
$ ansible-navigator images
```

Figure 1: ansible-navigator images

Note: The output you see might differ from the above output

This command gives you information about all currently installed Execution Environments or EEs for short. Investigate an EE by pressing the corresponding number. For example pressing **2** with the above example will open the `ee-supported-rhel8` execution environment:

Figure 2: ee main menu

Selecting `2` for `Ansible version and collections` will show us all Ansible Collections installed on that particular EE, and the version of `ansible-core`:

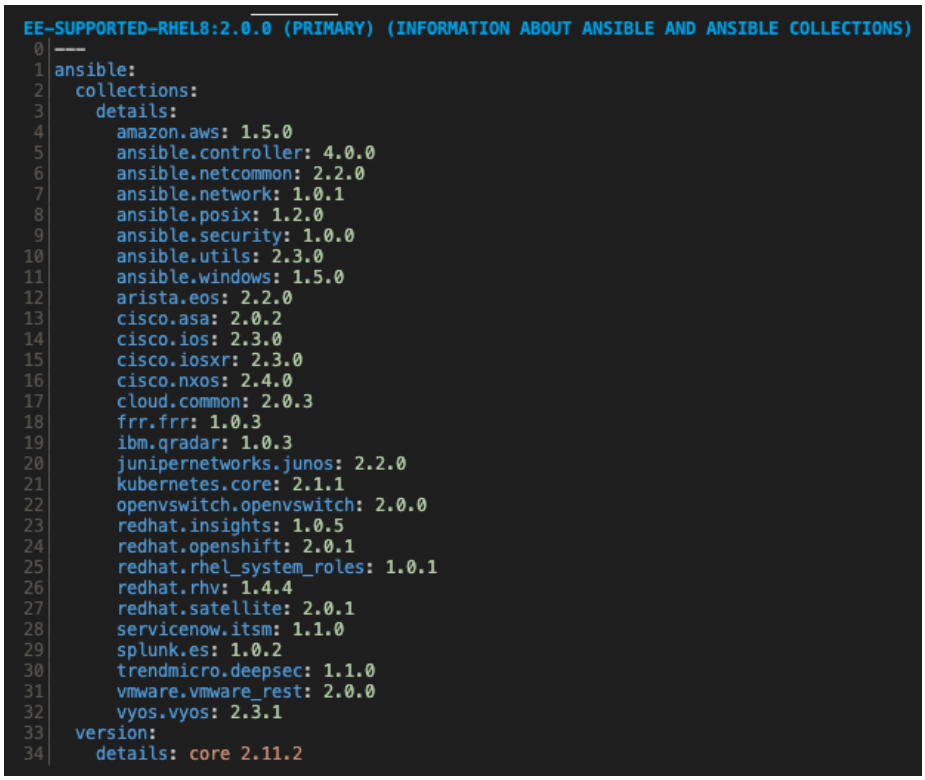


Figure 3: ee info

Step 4 - Examining the ansible-navigator configuration

Either use Visual Studio Code to open or use the `cat` command to view the contents of the `ansible-navigator.yml` file. The file is located in the home directory:

```
$ cat ~/.ansible-navigator.yml
---
ansible-navigator:
  ansible:
    inventory:
      entries:
      - ~/hosts
  execution-environment:
    image: registry.redhat.io/ansible-automation-platform-24/ee-supported-rhel8:latest
    enabled: true
    container-engine: podman
    pull:
      policy: missing
    volume-mounts:
    - src: /etc/ansible
      dest: /etc/ansible
```

Note the following parameters within the `.ansible-navigator.yml` file:

- `inventories`: shows the location of the ansible inventory being used
- `execution-environment`: where the default execution environment is set

For a full listing of every configurable knob checkout the documentation

Step 5 - Challenge Labs

You will soon discover that many chapters in this lab guide come with a "Challenge Lab" section. These labs are meant to give you a small task to solve using what you have learned so far. The solution of the task is shown underneath a warning sign.

The Ansible Basics

Objective

In this exercise, we are going to explore the latest Ansible command line utility `ansible-navigator` to learn how to work with inventory files and the listing of modules when needing assistance. The goal is to familarize yourself with how `ansible-navigator` works and how it can be used to enrich your Ansible experience.

This exercise will cover

- Working with inventory files
- Locating and understanding an `ini` formatted inventory file
- Listing modules and getting help when trying to use them

Guide

Step 1 - Work with your Inventory

An inventory file is a text file that specifies the nodes that will be managed by the control machine. The nodes to be managed may include a list of hostnames or

IP addresses of those nodes. The inventory file allows for nodes to be organized into groups by declaring a host group name within square brackets ([]).

To use the `ansible-navigator` command for host management, you need to provide an inventory file which defines a list of hosts to be managed from the control node. In this lab, the inventory is provided by your instructor. The inventory file is an `ini` formatted file listing your hosts, sorted in groups, additionally providing some variables. It looks like:

```
[web]
node ansible_host=<10.3.48.[100+PARTICIPANT_ID]>

[control]
controller ansible_host=10.3.48.100
```

Ansible is already configured to use the inventory specific to your environment. We will show you in the next step how that is done. For now, we will execute some simple commands to work with the inventory.

To reference all the inventory hosts, you supply a pattern to the `ansible-navigator` command. `ansible-navigator inventory` has a `--list` option which can be useful for displaying all the hosts that are part of an inventory file including what groups they are associated with.

```
[student@controller rhel_workshop]$ cd /home/student
[student@controller ~]$ ansible-navigator inventory --list -m stdout
{
    "_meta": {
        "hostvars": {
            "controller": {
                "ansible_host": "10.3.48.100"
            },
            "node": {
                "ansible_host": "10.3.48.101"
            }
        }
    },
    "all": {
        "children": [
            "control",
            "ungrouped",
            "web"
        ]
    },
    "control": {
        "hosts": [
            "controller"
        ]
    },
    "web": {
        "hosts": [
            "node"
```

```
        ]
    }
}
```

NOTE: `-m` is short for `--mode` which allows for the mode to be switched to standard output instead of using the text-based user interface (TUI).

If the `--list` is too verbose, the option of `--graph` can be used to provide a more condensed version of `--list`.

```
[student@controller ~]$ ansible-navigator inventory --graph -m stdout
@all:
  |--@control:
  |  |--controller
  |--@ungrouped:
  |--@web:
  |  |--node
```

We can clearly see that nodes: `node` is part of the `web` group, while `controller` is part of the `control` group.

An inventory file can contain a lot more information, it can organize your hosts in groups or define variables. In our example, the current inventory has the groups `web` and `control`. Run Ansible with these host patterns and observe the output:

Using the `ansible-navigator inventory` command, we can also run commands that provide information only for one host or group. For example, give the following commands a try to see their output.

```
[student@controller ~]$ ansible-navigator inventory --graph web -m stdout
[student@controller ~]$ ansible-navigator inventory --graph control -m stdout
[student@controller ~]$ ansible-navigator inventory --host node -m stdout
```

> **Tip**
>
> The inventory can contain more data. E.g. if you have hosts that run on non-standard SSH ports you can put the port number after the hostname with a colon. Or you could define names specific to Ansible and have them point to the "real" IP or hostname.

Step 2 - Listing Modules and Getting Help

Ansible Automation Platform comes with multiple supported Execution Environments (EEs). These EEs come with bundled supported collections that contain supported content, including modules.

> **Tip**
>
> In `ansible-navigator` exit by pressing the button `ESC`.

To browse your available modules first enter interactive mode:

```
$ ansible-navigator
```

Figure 4: picture of ansible-navigator

First browse a collection by typing `:collections`

```
:collections
```

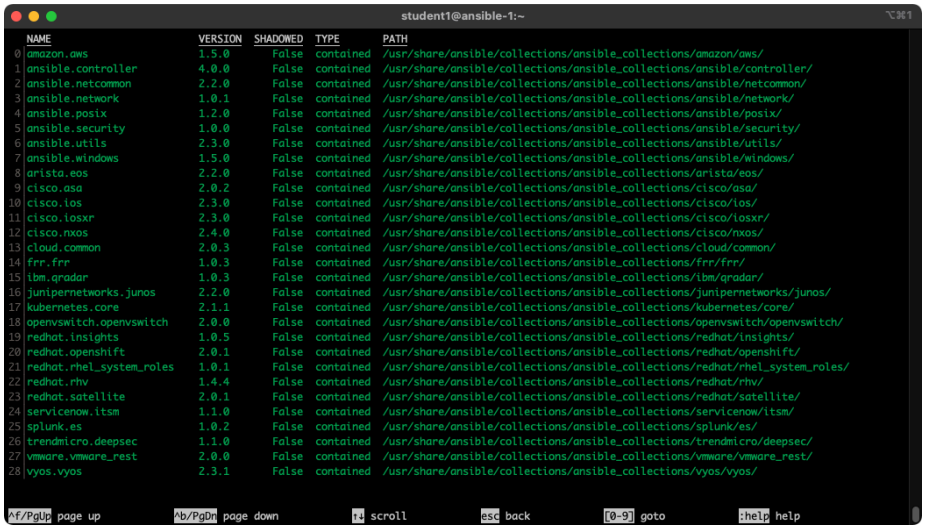


Figure 5: picture of ansible-navigator

To browse the content for a specific collections, type the corresponding number. For example in the example screenshot above the number `0` corresponds to `amazon.aws` collection. To zoom into collection type the number `0`.

```
0
```

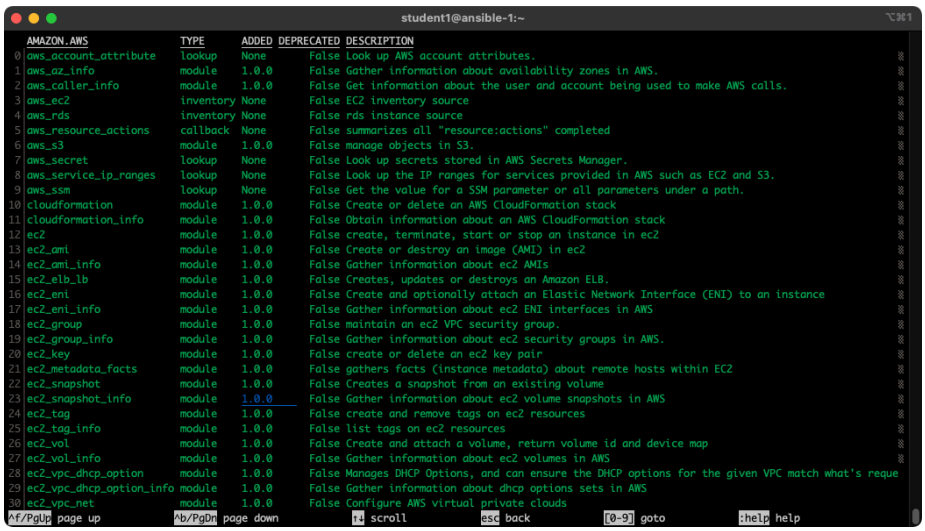


Figure 6: picture of ansible-navigator

Get help for a specific module including usage by zooming in further. For example the module `ec2_metadata_facts` corresponds to `3`.

```
:3
```

Scrolling down using the arrow keys or page-up and page-down can show us documentation and examples.

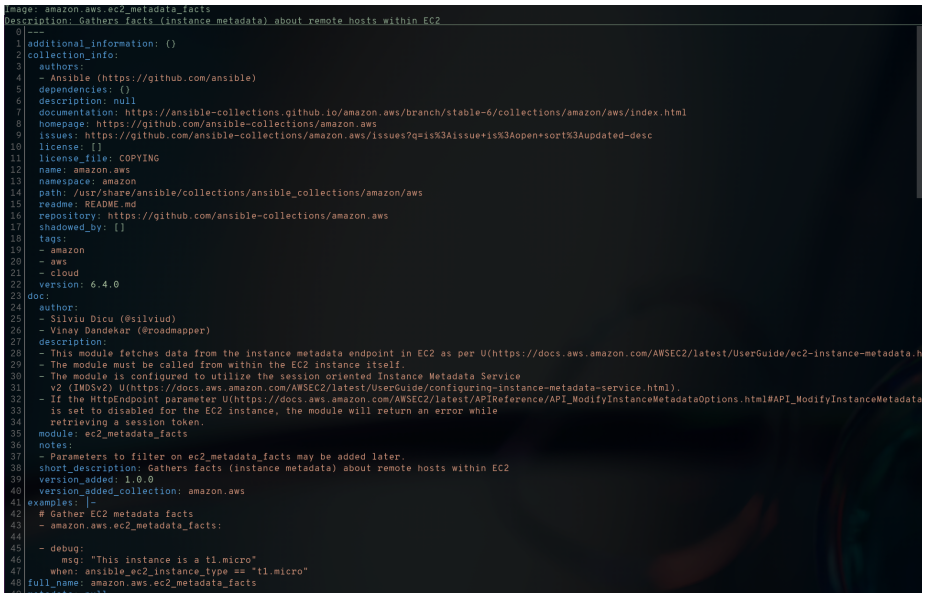


Figure 7: picture of ansible-navigator

You can also skip directly to a particular module by simply typing

`:doc namespace.collection.module-name`. For example typing `:doc amazon.aws.ec2_metadata_facts` would skip directly to the final page shown above.

> **Tip**
>
> Different execution environments can have access to different collections, and different versions of those collections. By using the built-in documentation you know that it will be accurate for that particular version of the collection.

Writing Your First Playbook

Objective

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

- Understanding Ansible module parameters
- Understanding and using the following modules
  - dnf 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 - Creating a Directory Structure and File 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 `node`.

There is a best practice on the preferred directory structures for playbooks. We strongly encourage you to read and understand these practices as you develop your Ansible ninja 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.

On your control host **ansible**, create a directory called `ansible-files` in your home directory and change directories into it:

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

Add a file called `apache.yml` with the following content. As discussed in the previous exercises, use `vi`/`vim` or `nano`.

```
---
- name: Apache server installed
  hosts: node
  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: yes`.

Now that we've defined the play, let's add a task to get something done. We will add a task in which dnf 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: node
  become: True
  tasks:
    - name: Install Apache
      ansible.builtin.dnf:
        name: httpd
```

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 `dnf` 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

With the introduction of Ansible Automation Platform 2, several new key components are being introduced as a part of the overall developer experience. Execution environments have been introduced to provide predictable environments to be used during automation runtime. All collection dependencies are contained within the execution environment to ensure that automation created in development environments runs the same as in production environments.

What do you find within an execution environment?

- RHEL UBI 8

- Ansible 2.9 or Ansible Core 2.11
- Python 3.8
- Any content Collections
- Collection python or binary dependencies.

Why use execution environments?

They provide a standardized way to define, build and distribute the environments that the automation runs in. In a nutshell, Automation execution environments are container images that allow for easier administration of Ansible by the platform administrator.

Considering the shift towards containerized execution of automation, automation development workflow and tooling that existed before Ansible Automation Platform 2 have had to be reimagined. In short, `ansible-navigator` replaces `ansible-playbook` and other `ansible-*` command line utilities.

With this change, Ansible playbooks are executed using the `ansible-navigator` command on the control node.

The prerequisites and best practices for using `ansible-navigator` have been done for you within this lab.

These include:

- Installing the `ansible-navigator` package
- Creating a default settings `~/.ansible-navigator.yml` for all your projects (optional)
- All execution environment (EE) logs are stored within the execution folder

For more information on the Ansible navigator settings

> **Tip**
>
> The parameters for ansible-navigator maybe modified for your specific environment. The current settings use a default `ansible-navigator.yml` for all projects, but a specific `ansible-navigator.yml` can be created for each project and is the recommended practice.

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

```
[student@controller ansible-files]$ ansible-navigator run apache.yml
```

> **Tip**
>
> The existing `ansible-navigator.yml` file provides the location of your inventory file. If this was not set within your `ansible-navigator.yml` file, the command to run the playbook would be: `ansible-navigator run apache.yml -i ~/hosts`

When running the playbook, you'll be displayed a text user interface (TUI) that displays the play name among other information about the playbook that is currently run.

```
  PLAY NAME                        OK  CHANGED    UNREACHABLE      FAILED    SKIPPED    IG
0 Apache server installed           2        1              0           0          0
```

If you notice, prior to the play name `Apache server installed`, you'll see a `0`. By pressing the `0` key on your keyboard, you will be provided a new window view displaying the different tasks that ran for the playbook completion. In this example, those tasks included the "Gathering Facts" and "Install Apache". The "Gathering Facts" is a built-in task that runs automatically at the beginning of each play. It collects information about the managed nodes. Exercises later on will cover this in more detail. The "Install Apache" was the task created within the `apache.yml` file that installed `httpd`.

The display should look something like this:

```
  RESULT      HOST  NUMBER      CHANGED       TASK
0 OK          node           0        False       Gathering Facts
1 OK          node           1         True       Install Apache                          dnf
```

Taking a closer look, you'll notice that each task is associated with a number. Task 1, "Install Apache", had a change and used the `dnf` module. In this case, the change is the installation of Apache (`httpd` package) on the host `node`.

By pressing `0` or `1` on your keyboard, you can see further details of the task being run. If a more traditional output view is desired, type `:st` within the text user interface.

Once you've completed, reviewing your Ansible playbook, you can exit out of the TUI via the Esc key on your keyboard.

> **Tip**
>
> The Esc key only takes you back to the previous screen. Once at the main overview screen an additional Esc key will take you back to the terminal window.

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

```
[student@controller ansible-files]$ ssh root@10.3.48.[100+PARTICIPANT_ID]
Last login: Thu Jan 11 10:35:34 2024 from 10.3.48.100
```

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

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

Log out of `node` with the command `exit` so that you are back on the control host and verify the installed package with an Ansible playbook labeled `package.yml`

```
---
- name: Check packages
  hosts: node
  become: True
  vars:
    p: httpd
  tasks:
    - name: Gather the packages facts
```

```
      ansible.builtin.package_facts:
        manager: auto
    - name: "Check whether {{ p }} is installed"
      ansible.builtin.debug:
        msg: "{{ p }} {{ ansible_facts.packages[p][0].version }} is installed!"
      when: p in ansible_facts.packages
```

You can now run it similarly to the previous one:

```
[student@controller ~]$ ansible-navigator run package.yml -m stdout


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

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

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

TASK [Check whether 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
```

Run the the `ansible-navigator run apache.yml` playbook for a second time, and compare the output. The output "CHANGED" now shows `0` instead of `1` and the color changed from yellow to green. This makes it easier to spot when changes have occured when running the Ansible playbook.

Step 4 - Extend your Playbook: Start & Enable Apache

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

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:

```
---
- name: Apache server installed
  hosts: node
  become: True
  tasks:
    - name: Install Apache
      ansible.builtin.dnf:
        name: httpd
    - name: Apache enabled and running
      ansible.builtin.service:
        name: httpd
```

```
        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:

```
[student@controller ~]$ ansible-navigator run apache.yml
```

Notice in the output, we see the play had `1` "CHANGED" shown in yellow and if we press `0` to enter the play output, you can see that task 2, "Apache enabled and running", was the task that incorporated the latest change by the "CHANGED" value being set to True and highlighted in yellow.

- Run the playbook a second time using `ansible-navigator` 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 `node`, e.g. with: `systemctl status httpd`.

```
---
- name: Check Status
  hosts: node
  become: True
  vars:
    package: httpd
  tasks:
    - name: "Check status of {{ package }} service"
      ansible.builtin.service_facts:
      register: service_state
    - ansible.builtin.debug:
        var: service_state.ansible_facts.services["{{ package }}.service"].state
```

```
[student@controller ~]$ ansible-navigator run service_state.yml
```

Step 5 - Extend your Playbook: Create an web.html

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 `node`.

```
---
- name: Check URL
  hosts: control
  vars:
    node: "[YOUR NODE IP ADDRESS]"
  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!

```
[student@controller ~]$ 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 `web.html` file to be served by Apache, it will throw an ugly "HTTP Error 403: Forbidden" status and Ansible will report an error. Also, you are not even seeing the 403 error, since the `node` port 80 is not reachable due to firewalld's configuration which does not allow connections to be allowed.

Let's start fixing this last issue. To do so, we'll alter the `apache.yml` file in the following way:

```
---
- name: Apache server installed
  hosts: node
  become: True
  tasks:
    - name: Install Apache
      ansible.builtin.dnf:
        name: httpd
    - name: Apache enabled and running
      ansible.builtin.service:
        name: httpd
        enabled: True
        state: started
    - name: Open firewall port
      ansible.posix.firewalld:
        service: http
        immediate: True
        permanent: True
        state: enabled
```

What does this new task do? The new task uses the `firewalld` module and defines the service option (`HTTP` standard port is `80/tcp`) and the state enabled. Due to how the `firewalld` utility works, we need to tell Ansible that we want the new port to be immediately available and configured in the same way even after reboot (with the `permanent` option).

Run your extended Playbook:

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

Now that we have opened the port, you can re-run the `check_httpd.yml` playbook and see that we now get the 403 error.

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

```
[student@controller 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>
```

In a previous example, you used Ansible's `copy` module to write text supplied on the command line into a file. Now you'll use the module in your playbook to copy a file.

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 installed
  hosts: node
  become: True
  tasks:
    - name: Install Apache
      ansible.builtin.dnf:
        name: httpd
    - name: Apache enabled and running
      ansible.builtin.service:
        name: httpd
        enabled: True
        state: started
    - name: Open firewall port
      ansible.posix.firewalld:
        service: http
        immediate: True
        permanent: True
        state: enabled
    - name: Copy index.html
      ansible.builtin.copy:
        src: web.html
        dest: /var/www/html/index.html
        mode: '644'
```

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.

Run your extended Playbook:

```
[student@controller 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.

Using Variables

Objective

Ansible supports variables to store values that can be used in Ansible playbooks. Variables can be defined in a variety of places and have a clear precedence. Ansible substitutes the variable with its value when a task is executed.

This exercise covers variables, specifically

- How to use variable delimiters `{{` and `}}`
- What `host_vars` and `group_vars` are and when to use them
- How to use `ansible_facts`
- How to use the `debug` module to print variables to the console window

Guide

Intro to Variables

Variables are referenced in Ansible Playbooks by placing the variable name in double curly braces:

```
Here comes a variable {{ variable1 }}
```

Variables and their values can be defined in various places: the inventory, additional files, on the command line, etc.

The recommended practice to provide variables in the inventory is to define them in files located in two directories named `host_vars` and `group_vars`:

- To define variables for a group "servers", a YAML file named `group_vars/servers.yml` with the variable definitions is created.
- To define variables specifically for a host `node`, the file `host_vars/node.yml` with the variable definitions is created.

  Tip

  Host variables take precedence over group variables (more about precedence can be found in the docs).

Step 1 - Create Variable Files

For understanding and practice let's do a lab. Following up on the theme "Let's build a web server. Or two. Or even more…", you will change the `index.html` to show the development environment (dev/prod) a server is deployed in.

On the ansible control host, as the `student` user, create the directories to hold the variable definitions in `~/ansible-files/`:

```
[student@controller ansible-files]$ mkdir host_vars group_vars
```

Now create two files containing variable definitions. We'll define a variable named `stage` which will point to different environments, `dev` or `prod`:

- Create the file `~/ansible-files/group_vars/web.yml` with this content:

```
---
stage: dev
```

- Create the file `~/ansible-files/host_vars/node.yml` with this content:

```
---
stage: prod
```

What is this about?

- For all servers in the `web` group the variable `stage` with value `dev` is defined. So as default we flag them as members of the dev environment.
- For server `node` this is overridden and the host is flagged as a production server. In our case, the `web` group only contains `node`, but if it contained multiple nodes the difference would be more obvious.

Step 2 - Create web.html Files

Now create two files in `~/ansible-files/files/`:

One called `prod_web.html` with the following content:

```
<body>
  <h1>This is a production webserver, take care!</h1>
</body>
```

And the other called `dev_web.html` with the following content:

```
<body>
  <h1>This is a development webserver, have fun!</h1>
</body>
```

Step 3 - Create the Playbook

Now you need a Playbook that copies the prod or dev `web.html` file - according to the "stage" variable.

Create a new Playbook called `deploy_index_html.yml` in the `~/ansible-files/` directory.

> **Tip**
>
> Note how the variable "stage" is used in the name of the file to copy.

```
---
- name: Copy web.html
  hosts: web
  become: True
  tasks:
    - name: Copy web.html
      ansible.builtin.copy:
        src: "{{ stage }}_web.html"
        dest: /var/www/html/index.html
```

- Run the Playbook:

```
[student@controller ansible-files]$ ansible-navigator run deploy_index_html.yml
```

Step 4 - Test the Result

The Ansible Playbook copies different files as index.html to the hosts, use `curl` to test it.

For node:

```
[student@controller ansible-files]$ curl http://[10.3.48.[100+PARTICIPANT_ID]
<body>
  <h1>This is a production webserver, take care!</h1>
</body>
```

Tip

You can remove the `~/ansible-files/host_vars/node.yml` file and see that by re-running the Ansible playbook, the deployed page will change.

Tip

If by now you think: There has to be a smarter way to change content in files… you are absolutely right. This lab was done to introduce variables, you are about to learn about templates in one of the future labs.

Step 5 - Ansible Facts

Ansible facts are variables that are automatically discovered by Ansible from a managed host. Remember the "Gathering Facts" task listed in the output of each `ansible-navigator` execution? At that moment the facts are gathered for each managed nodes. Facts can also be pulled by the `setup` module. They contain useful information stored into variables that administrators can reuse.

To get an idea what facts Ansible collects by default, on your control node as your student user run the following playbook called `setup.yml` to get the setup details of `node`:

```
---
- name: Capture Setup
  hosts: node
  tasks:
    - name: Collect only facts returned by facter
      ansible.builtin.setup:
        gather_subset:
          - all
      register: setup
    - ansible.builtin.debug:
        var: setup
```

```
[student@controller ansible-files]$ cd ~
[student@controller ~]$ ansible-navigator run setup.yml -m stdout
```

This might be a bit too much, you can use filters to limit the output to certain facts, the expression is shell-style wildcard within your playbook. Create a playbook labeled `setup_filter.yml` as shown below. In this example, we filter to get the `eth0` facts as well as memory details of `node`.

```
---
- name: Capture Setup
  hosts: node
```

```
  tasks:
    - name: Collect only specific facts
      ansible.builtin.setup:
        filter:
          - 'ansible_eth0'
          - 'ansible_*_mb'
      register: setup
    - debug:
        var: setup
```

```
[student@controller ansible-files]$ ansible-navigator run setup_filter.yml -m stdout
```

Step 6 - Challenge Lab: Facts

- Try to find and print the distribution (Red Hat) of your managed hosts using a playbook.

  Tip

  Use the wildcard to find the fact within your filter, then apply a filter to only print this fact.

  Warning

  Solution below!

```
---
- name: Capture Setup
  hosts: node
  tasks:
    - name: Collect only specific facts
      ansible.builtin.setup:
        filter:
          - '*distribution'
      register: setup
    - ansible.builtin.debug:
        var: setup
```

With the wildcard in place, the output shows:

```
TASK [debug] *******************************************************************
ok: [ansible] => {
    "setup": {
        "ansible_facts": {
            "ansible_distribution": "RedHat"
        },
        "changed": false,
        "failed": false
    }
}
```

With this we can conclude the variable we are looking for is labeled `ansible_distribution`.

Then we can update the playbook to be explicit in its findings and change the following line:

```
filter:
- '*distribution'
```

to:

```
filter:
- 'ansible_distribution'
```

```
[student@controller ansible-files]$ ansible-navigator run setup_filter.yml -m stdout
```

Step 7 - Using Facts in Playbooks

Facts can be used in a Playbook like variables, using the proper naming, of course. Create this Playbook as `facts.yml` in the `~/ansible-files/` directory:

```
---
- name: Output facts within a playbook
  hosts: node
  tasks:
    - name: Prints Ansible facts
      ansible.builtin.debug:
        msg: The IPv4 address of {{ ansible_fqdn }} is {{ ansible_default_ipv4.address }}
```

> **Tip**
>
> The "debug" module is handy for e.g. debugging variables or expressions.

Execute it to see how the facts are printed:

```
[student@controller ansible-files]$ ansible-navigator run facts.yml
```

Within the text user interface (TUI) window, type `:st` to capture the following output:

```
PLAY [Output facts within a playbook] ***************************************

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

TASK [Prints Ansible facts] *************************************************
ok: [node] =>
  msg: The IPv4 address of node is 10.3.48.101

PLAY RECAP ******************************************************************
node                       : ok=2    changed=0    unreachable=0    failed=0
```

AAP

Introduction to Ansible automation controller

What's New in Ansible automation controller 4.0

Ansible Automation Platform 2 is the next evolution in automation from Red Hat's trusted enterprise technology experts. The Ansible Automation Platform 2 release includes automation controller 4.0, the improved and renamed Ansible Tower.

Controller continues to provide a standardized way to define, operate, and delegate automation across the enterprise. It introduces new technologies and an enhanced architecture that enables automation teams to scale and deliver automation rapidly.

Why was Ansible Tower renamed to automation controller?

As Ansible Automation Platform 2 continues to evolve, certain functionality has been decoupled (and will continue to be decoupled in future releases) from what was formerly known as Ansible Tower. It made sense to introduce the naming change that better reflects these enhancements and the overall position within the Ansible Automation Platform suite.

Who is automation controller for?

All automation team members interact with or rely on automation controller, either directly or indirectly.

- Automation creators develop Ansible playbooks, roles, and modules.
- Automation architects elevate automation across teams to align with IT processes and streamline adoption.
- Automation operators ensure the automation platform and framework are operational.

These roles are not necessarily dedicated to a person or team. Many organizations assign multiple roles to people or outsource specific automation tasks based on their needs.

Automation operators are typically the primary individuals who interact directly with the automation controller, based on their responsibilities.

Objective

The following exercise will provide an Ansible automation controller overview including going through features that are provided by the Red Hat Ansible Automation Platform. This will cover automation controller fundamentals such as:

- Job Templates
- Projects
- Inventories
- Credentials
- Workflows

Guide

Why Ansible automation controller?

Automation controller is a web-based UI that provides an enterprise solution for IT automation. It

- has a user-friendly dashboard.
- complements Ansible, adding automation, visual management, and monitoring capabilities.
- provides user access control to administrators.
- provides distinct *view* and *edit* perspectives for automation controller objects and components.
- graphically manages or synchronizes inventories with a wide variety of sources.
- has a RESTful API.
- And much more…

Your Ansible automation controller lab environment

In this lab you work in a pre-configured lab environment. You will have access to the following hosts:

Role	Inventory name
Ansible control host & automation controller	controller
Managed Host 1	node

The Ansible automation controller provided in this lab is individually setup for you. Make sure to access the right machine whenever you work with it. Automation controller has already been installed and licensed for you, the web UI will be reachable over HTTP/HTTPS.

Dashboard

Let's have a first look at the automation controller: Point your browser to `https://10.3.48.100`.

The web UI of automation controller greets you with a dashboard with a graph

showing:

- recent job activity
- the number of managed hosts
- quick pointers to lists of hosts with problems.

The dashboard also displays real time data about the execution of tasks completed in playbooks.

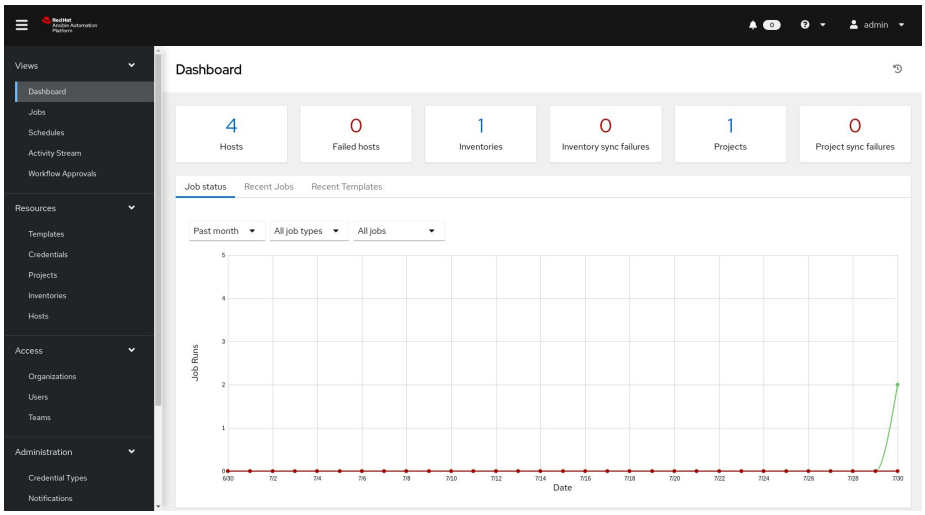


Figure 8: Ansible automation controller dashboard

Concepts

Before we dive further into using Ansible automation controller, you should get familiar with some concepts and naming conventions.

Projects Projects are logical collections of Ansible playbooks in Ansible automation controller. These playbooks either reside on the Ansible automation controller instance, or in a source code version control system supported by automation controller.

Inventories An Inventory is a collection of hosts against which jobs may be launched, the same as an Ansible inventory file. Inventories are divided into groups and these groups contain the actual hosts. Groups may be populated manually, by entering host names into automation controller, from one of Ansible Automation controller's supported cloud providers or through dynamic inventory scripts.

Credentials Credentials are utilized by automation controller for authentication when launching Jobs against machines, synchronizing with inventory sources, and importing project content from a version control system. Credential configuration can be found in the Settings.

automation controller credentials are imported and stored encrypted in automation controller, and are not retrievable in plain text on the command line by any user. You can grant users and teams the ability to use these credentials, without actually exposing the credential to the user.

Templates A job template is a definition and set of parameters for running an Ansible job. Job templates are useful to execute the same job many times. Job templates also encourage the reuse of Ansible playbook content and collaboration between teams. To execute a job, automation Controller requires that you first create a job template.

Jobs A job is basically an instance of automation controller launching an Ansible playbook against an inventory of hosts.

Workshop Exercise - Inventories, credentials and ad hoc commands

Objective

Explore and understand the lab environment. This exercise will cover

- Locating and understanding:
  - Ansible Automation Controller **Inventories**
  - Ansible Automation Controller **Credentials**
- Running ad hoc commands via the Ansible Automation Controller web UI

Guide

Examine an Inventory

The first thing we need is an inventory of your managed hosts. This is the equivalent of an inventory file in command-line Ansible. There is a lot more to it (like dynamic inventories) but let's start with the basics.

- You should already have the web UI open, if not: Point your browser to `https://10.3.48.100`. The password will be provided by the instructor.

There will be one inventory, the [**USER**] **Inventory**. Click the [**USER**] **Inventory** then click the **Hosts** button

The inventory information at `~/hosts` was pre-loaded into the Ansible Automation controller Inventory as part of the provisioning process.

```
$ cat ~/hosts
[web]
node ansible_host=10.3.48.101 ansible_user=s1

[control]
controller ansible_host=10.3.48.100 ansible_user=s1
```

Warning

In your inventory the IP addresses will be different.

Examine Machine Credentials

Now we will examine the credentials to access our managed hosts from Automation controller. As part of the provisioning process for this Ansible Workshop the **[USER] Credential** has already been setup.

In the **Resources** menu choose **Credentials**. Now click on the **[USER] Credential**.

Note the following information:

- **Credential Type**: Machine- Machine credentials define ssh and user-level privilege escalation access for playbooks. They are used when submitting jobs to run playbooks on a remote host.
- **Username**: matches our command-line Ansible inventory username for the other Linux nodes
- **SSH Private Key**: Encrypted - take note that you can't actually examine the SSH private key once someone hands it over to Ansible Automation controller

Run Ad Hoc commands

It is possible to run run ad hoc commands from Ansible Automation controller as well.

- In the webUI go to **Resources → Inventories → [USER] Inventory**
- Click the **Hosts** tab to change into the hosts view and select the `node` host by ticking the box to the left of the host entry.
- Click **Run Command** button. In the next screen you have to specify the ad hoc command.

Within the **Details** window, select **Module** `ping` and click **Next**.

Within the **Execution Environment** window, select **Default execution environment** and click **Next**.

Within the **Machine Credential** window, select **[USER] Credentials** and click **Launch**.

Tip

The output of the results is displayed once the command has completed.

The simple **ping** module doesn't need options. For other modules you need to supply the command to run as an argument. Try the **command** module to find the userid of the executing user using an ad hoc command.

- In the web UI go to **Resources → Inventories → [USER] Inventory**
- Click the **Hosts** tab to change into the hosts view and select the `node` host by ticking the box to the left of the host entry.

- Click **Run Command** button. In the next screen you have to specify the ad hoc command.

Within the **Details** window, select **Module** `command`, in **Arguments** type `id` and click **Next**.

Within the **Execution Environment** window, select **Default execution environment** and click **Next**.

Within the **Machine Credential** window, select **[USER] Credentials** and click **Launch**.

> **Tip**
>
> After choosing the module to run, Ansible Automation Controller will provide a link to the docs page for the module when clicking the question mark next to "Arguments". This is handy, give it a try.

How about trying to get some secret information from the system? Try to print out */etc/shadow*.

- In the web UI go to **Resources → Inventories → [USER] Inventory**
- Click the **Hosts** tab to change into the hosts view and select the `node` host by ticking the box to the left of the host entry.
- Click **Run Command** button. In the next screen you have to specify the ad hoc command.

Within the **Details** window, select **Module** `command`, in **Arguments** type `cat /etc/shadow` and click **Next**.

Within the **Execution Environment** window, select **Default execution environment** and click **Next**.

Within the **Machine Credential** window, select **[USER] Credentials** and click **Launch**.

> **Warning**
>
> **Expect an error!**

Oops, the last one didn't went well, all red.

Re-run the last ad hoc command but this time check the checkbox labeled **Enable privilege escalation**.

As you see, this time it worked. For tasks that have to run as `root` you need to escalate the privileges. This is the same as the **become: True** used in your Ansible Playbooks.

Challenge Lab: Ad Hoc Commands

Okay, a small challenge: Run an ad hoc to make sure the package "tmux" is installed on all hosts. If unsure, consult the documentation either via the web UI as shown above or by running `[ansible@controller ~]$ ansible-doc yum` on your Automation controller control host.

Warning

Solution below!

- In the web UI go to **Resources → Inventories → [USER] Inventory**
- Click the **Hosts** tab to change into the hosts view and select the `node` host by ticking the box to the left of the host entry.
- Click **Run Command** button. In the next screen you have to specify the ad hoc command.

Within the **Details** window, select **Module** `yum`, in **Arguments** type `name=tmux`, check **Enable privilege escalation** and click **Next**.

Within the **Execution Environment** window, select **Default execution environment** and click **Next**.

Within the **Machine Credential** window, select **[USER] Credentials** and click **Launch**.

Tip

Notice how the package was instaled via the "CHANGED" output. If you run the ad hoc command a second time, the output will mention "SUCCESS" and inform you via the message parameter that there is nothing to do.

Workshop Exercise - Projects & Job Templates

Objective

An Ansible automation controller **Project** is a logical collection of Ansible playbooks. You can manage your playbooks by placing them into a source code management (SCM) system supported by automation controller such as Git or Subversion.

This exercise covers:

- Understanding and using an Ansible automation controller Project
- Using Ansible playbooks kept in a Git repository.
- Creating and using an Ansible Job Template

Guide

Setup Git Repository

For this demonstration we will use playbooks stored in a Git repository:

https://github.com/ansible/workshop-examples

A playbook to install the Apache web server has already been committed to the directory **rhel/apache**, `apache_install.yml`:

```
---
- name: Apache server installed
  hosts: web
```

```yaml
tasks:
- name: latest Apache version installed
  ansible.builtin.yum:
    name: httpd
    state: latest

- name: latest firewalld version installed
  ansible.builtin.yum:
    name: firewalld
    state: latest

- name: firewalld enabled and running
  ansible.builtin.service:
    name: firewalld
    enabled: true
    state: started

- name: firewalld permits http service
  ansible.builtin.firewalld:
    service: http
    permanent: true
    state: enabled
    immediate: yes

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

Tip

Note the difference to other playbooks you might have written! Most importantly there is no `become` and `hosts` is set to `all`.

To configure and use this repository as a **Source Control Management (SCM)** system in automation controller you have to create a **Project** that uses the repository

Create the Project

- Go to **Resources → Projects** click the **Add** button. Fill in the form:
  - **Name**: Workshop Project
  - **Organization**: [USER]
  - **Default Execution Environment**: Default execution environment
  - **Source Control Credential Type**: Git
  - Enter the URL into the **Project configuration**
  - **Source Control URL**: https://github.com/ansible/workshop-examples.git
  - **Options**: Select Clean, Delete, Update Revision on Launch to re-

quest a fresh copy of the repository and to update the repository when launching a job.

- Click **SAVE**

The new project will be synced automatically after creation. But you can also do this manually: Sync the Project again with the Git repository by going to the **Projects** view and clicking the circular arrow **Sync Project** icon to the right of the Project.

After starting the sync job, go to the **Jobs** view: there is a new job for the update of the Git repository.

Create a Job Template and Run a Job

A job template is a definition and set of parameters for running an Ansible job. Job templates are useful to execute the same job many times. So before running an Ansible **Job** from automation controller you must create a **Job Template** that pulls together:

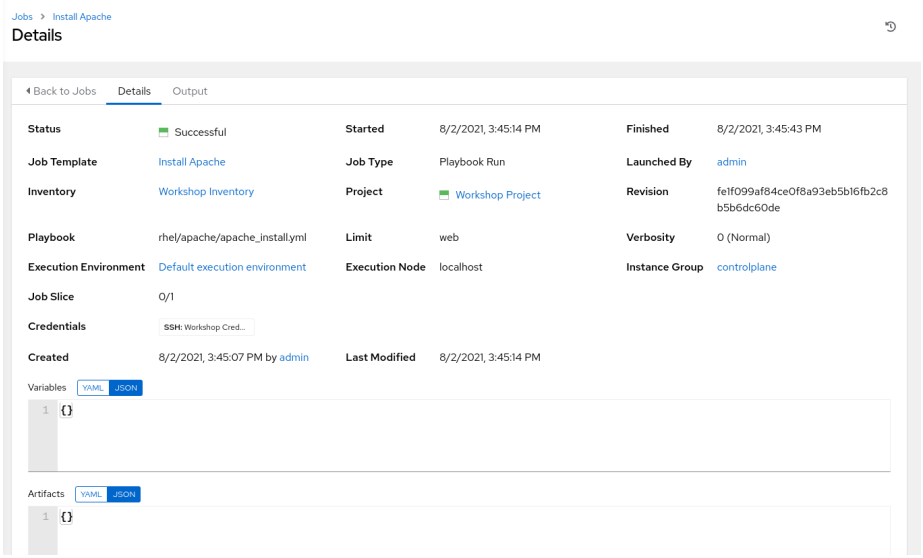
- **Inventory**: On what hosts should the job run?
- **Credentials** What credentials are needed to log into the hosts?
- **Project**: Where is the playbook?
- **What** playbook to use?

Okay, let's just do that: Go to the **Resources -> Templates** view, click the **Add** button and choose **Add job template**.

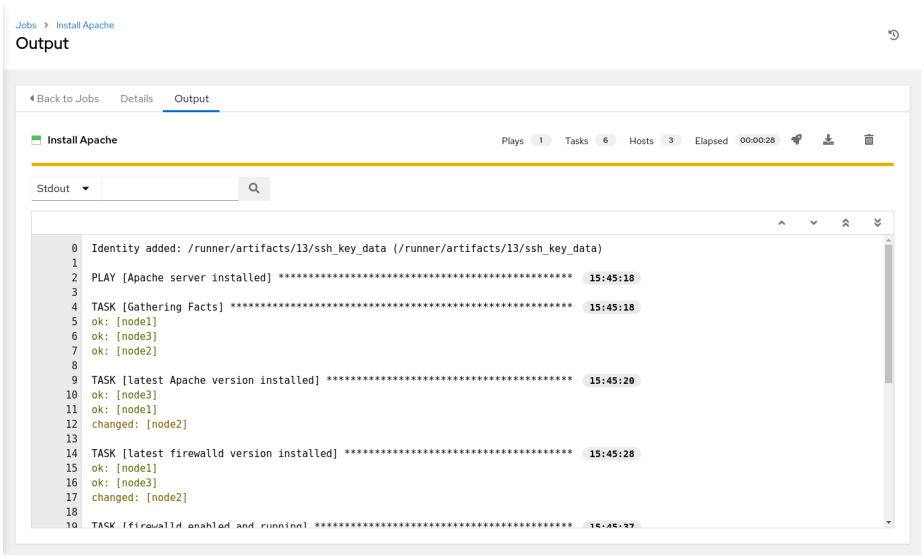
> **Tip**
>
> Remember that you can often click on magnfying glasses to get an overview of options to pick to fill in fields.

- **Name**: Install Apache
- **Job Type**: Run
- **Inventory**: [USER] Inventory
- **Project**: Workshop Project
- **Execution Environment**: Default execution environment
- **Playbook**: rhel/apache/apache_install.yml
- **Credentials**: [USER] Credential
- **Limit**: web
- tasks need to run as root so check **Privilege Escalation**
- Click **Save**

You can start the job by directly clicking the blue **Launch** button, or by clicking on the rocket in the Job Templates overview. After launching the Job Template, you are automatically brought to the job overview where you can follow the playbook execution in real time:



Job Details



Job Run

Since this might take some time, have a closer look at all the details provided:

- All details of the job template like inventory, project, credentials and playbook are shown.
- Additionally, the actual revision of the playbook is recorded here - this makes it easier to analyse job runs later on.
- Also the time of execution with start and end time is recorded, giving you an idea of how long a job execution actually was.
- Selecting **Output** shows the output of the running playbook. Click on a node underneath a task and see that detailed information are provided for each task of each node.

After the Job has finished go to the main **Jobs** view: All jobs are listed here,

you should see directly before the Playbook run an Source Control Update was started. This is the Git update we configured for the **Project** on launch!

Challenge Lab: Check the Result

Time for a little challenge:

- Use an ad hoc command on both hosts to make sure Apache has been installed and is running.

You have already been through all the steps needed, so try this for yourself.

> **Tip**
>
> What about `systemctl status httpd`?
>
> **Warning**
>
> **Solution Below**

- Go to **Resources** → **Inventories** → **[USER] Inventory**
- In the **Hosts** view select `node` and click **Run Command**

Within the **Details** window, select **Module** `command`, in **Arguments** type `systemctl status httpd` and click **Next**.

Within the **Execution Environment** window, select **Default execution environment** and click **Next**.

Within the **Machine Credential** window, select **[USER] Credential** and click **Launch**.

> **Tip**
>
> The output of the results is displayed once the command has completed.

Workshop Exercise - Role-based access control

Objective

You have already learned how Ansible automation controller separates credentials from users. Another advantage of Ansible automation controller is the user and group rights management. This exercise demonstrates Role Based Access Control (RBAC)

Guide

Ansible automation controller users

There are three types of automation controller users:

- **Normal User**: Have read and write access limited to the inventory and projects for which that user has been granted the appropriate roles and privileges.

- **System Auditor**: Auditors implicitly inherit the read-only capability for all objects within the automation controller environment.
- **System Administrator**: Has admin, read, and write privileges over the entire automation controller installation.

Let's create a user:

- In the automation controller menu under **Access** click **Users**
- Click the **Add** button
- Fill in the values for the new user:
  - **Username**: [USER]-wweb
  - **Password**: ansible
  - **Confirm Password**: ansible
  - **Organization**: [USER]
  - **User Type**: Normal User
- Click **Save**

Ansible automation controller teams

A Team is a subdivision of an organization with associated users, projects, credentials, and permissions. Teams provide a means to implement role-based access control schemes and delegate responsibilities across organizations. For instance, permissions may be granted to a whole Team rather than each user on the Team.

Create a Team:

- In the menu go to **Access → Teams**
- Click the **Add** button and create a team named `[USER] Web Content` within the `[USER]` Organization.
- Click **Save**

Add a user to the team:

- Click on the team `[USER] Web Content` and click the **Access** tab and click **Add**.
- Within the **Select a Resource Type** window, click on the **Users** resource type and click **Next**.
- Within the **Select Items from List**, select the checkbox next to the `[USER]-wweb` user and click **Next**.
- Within the **Select Roles to Apply**, select **Member** as the role to apply to the `[USER]-wweb` user.

Click **Save**.

Permissions allow to read, modify, and administer projects, inventories, and other automation controller elements. Permissions can be set for different resources.

Granting permissions

To allow users or teams to actually do something, you have to set permissions. The user **[USER]-wweb** should only be allowed to modify content of the assigned webservers.

Add the permission to use the `Create index.html` template:

- Within **Resources** -> **Templates**, select `Create index.html`.
- Select **Access** tab from the menu and click **Add**.
- Within the **Select a Resource Type** window, click on the **Users** resource type and click **Next**.
- Within the **Select Items from List**, select the checkbox next to the `[USER]-wweb` user and click **Next**.
- Within the **Select Roles to Apply**, select **Read** and **Execute** as the roles to apply to the `[USER]-wweb` user.
- Click **Save**

Test permissions

Now log out of automation controller's web UI and in again as the **[USER]-wweb** user.

- Go to the **Templates** view, you should notice for wweb only the `Create index.html` template is listed. He is allowed to view and launch, but not to edit the Template (no Edit button available).
- Run the Job Template by clicking the rocket icon. Enter the values for the survey questions and launch the job.
- In the following **Jobs** view have a good look around, note that there where changes to the host (as expected).

Check the result: execute `curl` again on the control host to pull the content of the webserver on `node`:

```
#> curl http://10.3.48.[100+PARTICIPANT_ID]
```

Just recall what you have just done: You enabled a restricted user to run an Ansible playbook

- Without having access to the credentials
- Without being able to change the playbook itself
- But with the ability to change variables you predefined!

Effectively you provided the power to execute automation to another user without handing out your credentials or giving the user the ability to change the

automation code. And yet, at the same time the user can still modify things based on the surveys you created.

This capability is one of the main strengths of Ansible automation controller!

Ansible - advanced

Conditionals, Handlers and Loops

Objective

Three foundational Ansible features are:

- Conditionals
- Handlers
- Loops

Guide

Step 1 - Conditionals

Ansible can use conditionals to execute tasks or plays when certain conditions are met.

To implement a conditional, the `when` statement must be used, followed by the condition to test. The condition is expressed using one of the available operators like e.g. for comparison:

==	Compares two objects for equality.
!=	Compares two objects for inequality.
>	true if the left hand side is greater than the right hand side.
>=	true if the left hand side is greater or equal to the right hand side.
<	true if the left hand side is lower than the right hand side.
<=	true if the left hand side is lower or equal to the right hand side.

For more on this, please refer to the documentation: http://jinja.pocoo.org/docs/2.10/templates/

As an example you would like to install an FTP server, but only on hosts that are in the "ftpserver" inventory group.

To do that, first edit the inventory to add another group, and place `node` in it. The section to add looks like this:

```
[ftpserver]
node
```

Edit the inventory `~/hosts` to add those lines. When you are done, it will look similar to the following listing:

> **Tip**
>
> The ansible_host variable only needs to be specified once for a node. When adding a node to other groups, you do not need to specify the variable again.

Important Do not copy/paste the example below. Just edit the file to add the above lines.

```
[web]
node ansible_host=xx.xx.xx.xx ansible_user=[USER]

[ftpserver]
node

[control]
controller ansible_host=xx.xx.xx.xx ansible_user=[USER]
```

Next create the file `ftpserver.yml` on your control host in the `~/ansible-files/` directory:

```
---
- name: Install vsftpd on ftpservers
  hosts: all
  become: True
  tasks:
    - name: Install FTP server when host in ftpserver group
      ansible.builtin.yum:
        name: vsftpd
        state: latest
      when: inventory_hostname in groups["ftpserver"]
```

> **Tip**
>
> By now you should know how to run Ansible Playbooks, we'll start to be less verbose in this guide. Go create and run it. :-)

Run it and examine the output. The expected outcome: The task is skipped on the ansible host (your control host) because they are not in the ftpserver group in your inventory file.

```
TASK [Install FTP server when host in ftpserver group] **************
skipping: [controller]
changed: [node]
```

Step 2 - Handlers

Sometimes when a task does make a change to the system, an additional task or tasks may need to be run. For example, a change to a service's configuration file may then require that the service be restarted so that the changed configuration takes effect.

Here Ansible's handlers come into play. Handlers can be seen as inactive tasks that only get triggered when explicitly invoked using the "notify" statement. Read more about them in the Ansible Handlers documentation.

As a an example, let's write a playbook that:

- manages Apache's configuration file `/etc/httpd/conf/httpd.conf` on all hosts in the `web` group
- restarts Apache when the file has changed

First we need the file Ansible will deploy, let's just take the one from node1. Remember to replace the IP address shown in the listing below with the IP address from your individual `node1`.

```
[student@controller ansible-files]$ scp 10.3.48.[100+PARTICIPANT_ID]:/etc/httpd/conf/httpd
~/ansible-files/files/httpd.conf
```

Next, create the Playbook `httpd_conf.yml`. Make sure that you are in the directory `~/ansible-files`.

```
---
- name: Manage httpd.conf
  hosts: web
  become: True
  tasks:
    - name: Copy Apache configuration file
      ansible.builtin.copy:
        src: httpd.conf
        dest: /etc/httpd/conf/
        mode: '644'
      notify:
        - Restart_apache
    - name: Open firewall port
      ansible.posix.firewalld:
        port: 8080/tcp
        immediate: True
        permanent: True
        state: enabled
  handlers:
    - name: Restart_apache
      ansible.builtin.service:
        name: httpd
        state: restarted
```

So what's new here?

- The "notify" section calls the handler only when the copy task actually changes the file. That way the service is only restarted if needed - and not each time the playbook is run.
- The "handlers" section defines a task that is only run on notification.

Run the playbook. We didn't change anything in the file yet so there should not be any `changed` lines in the output and of course the handler shouldn't have fired.

- Now change the `Listen 80` line in `~/ansible-files/files/httpd.conf` to:

```
Listen 8080
```

- Run the playbook again. Now the Ansible's output should be a lot more interesting:
  - httpd.conf should have been copied over
  - The handler should have restarted Apache

Apache should now listen on port 8080. Easy enough to verify:

```
[student@controller ansible-files]$ curl http://10.3.48.[100+PARTICIPANT_ID]
curl: (7) Failed to connect to 10.3.48.101 port 80: Connection refused
[student@controller ansible-files]$ curl http://10.3.48.[100+PARTICIPANT_ID]:8080
<body>
  <h1>Apache is running fine</h1>
</body>
```

Leave the setting for listen on port 8080. We'll use this in a later exercise.

Step 3 - Simple Loops

Loops enable us to repeat the same task over and over again. For example, lets say you want to create multiple users. By using an Ansible loop, you can do that in a single task. Loops can also iterate over more than just basic lists. For example, if you have a list of users with their coresponding group, loop can iterate over them as well. Find out more about loops in the Ansible Loops documentation.

To show the loops feature we will generate three new users on `node`. For that, create the file `loop_users.yml` in `~/ansible-files` on your control node as your student user. We will use the `user` module to generate the user accounts.

```
---
- name: Ensure users
  hosts: node
  become: True
  tasks:
    - name: Ensure three users are present
      ansible.builtin.user:
        name: "{{ item }}"
        state: present
      loop:
         - dev_user
         - qa_user
         - prod_user
```

Understand the playbook and the output:

- The names are not provided to the user module directly. Instead, there is only a variable called `{{ item }}` for the parameter `name`.
- The `loop` keyword lists the actual user names. Those replace the `{{ item }}` during the actual execution of the playbook.

- During execution the task is only listed once, but there are three changes listed underneath it.

Step 4 - Loops over hashes

As mentioned loops can also be over lists of hashes. Imagine that the users should be assigned to different additional groups:

```
- username: dev_user
  groups: ftp
- username: qa_user
  groups: ftp
- username: prod_user
  groups: apache
```

The `user` module has the optional parameter `groups` to list additional users. To reference items in a hash, the `{{ item }}` keyword needs to reference the subkey: `{{ item.groups }}` for example.

Let's rewrite the playbook to create the users with additional user rights:

```
---
- name: Ensure users
  hosts: node
  become: True
  tasks:
    - name: Ensure three users are present
      ansible.builtin.user:
        name: "{{ item.username }}"
        state: present
        groups: "{{ item.groups }}"
      loop:
        - { username: 'dev_user', groups: 'ftp' }
        - { username: 'qa_user', groups: 'ftp' }
        - { username: 'prod_user', groups: 'apache' }
```

Check the output:

- Again the task is listed once, but three changes are listed. Each loop with its content is shown.

Verify that the user `dev_user` was indeed created on `node` using the following playbook:

```
---
- name: Get user ID
  hosts: node
  vars:
    myuser: "dev_user"
  tasks:
    - name: Get {{ myuser }} info
      ansible.builtin.getent:
        database: passwd
        key: "{{ myuser }}"
```

```
    - ansible.builtin.debug:
        msg:
          - "{{ myuser }} uid: {{ getent_passwd[myuser].1 }}"
```

```
$ ansible-navigator run user_id.yml -m stdout

PLAY [Get user ID] *************************************************************

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

TASK [Get dev_user info] *******************************************************
ok: [node]

TASK [debug] *******************************************************************
ok: [node] => {
    "msg": [
        "dev_user uid: 1002"
    ]
}

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

Templates

Objective

This exercise will cover Jinja2 templating. Ansible uses Jinja2 templating to modify files before they are distributed to managed hosts. Jinja2 is one of the most used template engines for Python (http://jinja.pocoo.org/).

Guide

Step 1 - Using Templates in Playbooks

When a template for a file has been created, it can be deployed to the managed hosts using the `template` module, which supports the transfer of a local file from the control node to the managed hosts.

As an example of using templates you will change the motd file to contain host-specific data.

First create the directory `templates` to hold template resources in `~/ansible-files/`:

```
[student@controller ansible-files]$ mkdir templates
```

Then in the `~/ansible-files/templates/` directory create the template file `motd-facts.j2`:

```
Welcome to {{ ansible_hostname }}.
{{ ansible_distribution }} {{ ansible_distribution_version}}
```

`deployed on {{ ansible_architecture }} architecture.`

The template file contains the basic text that will later be copied over. It also contains variables which will be replaced on the target machines individually.

Next we need a playbook to use this template. In the `~/ansible-files/` directory create the Playbook `motd-facts.yml`:

```
---
- name: Fill motd file with host data
  hosts: node
  become: True
  tasks:
    - ansible.builtin.template:
        src: motd-facts.j2
        dest: /etc/motd
        owner: root
        group: root
        mode: 0644
```

You have done this a couple of times by now:

- Understand what the Playbook does.
- Execute the Playbook `motd-facts.yml`.
- Login to node via SSH and check the message of the day content.
- Log out of node.

You should see how Ansible replaces the variables with the facts it discovered from the system.

Step 2 - Challenge Lab

Add a line to the template to list the current kernel of the managed node.

- Find a fact that contains the kernel version using the commands you learned in the "Ansible Facts" chapter.

  Tip

  filter for kernel

  Run the newly created playbook to find the fact name.

- Change the template to use the fact you found.
- Run the motd playbook again.
- Check motd by logging in to node

  Warning

  Solution below!

- Find the fact:

```
---
- name: Capture Kernel Version
  hosts: node
```

```
  tasks:
    - name: Collect only kernel facts
      ansible.builtin.setup:
        filter:
          - '*kernel'
      register: setup
    - ansible.builtin.debug:
        var: setup
```

With the wildcard in place, the output shows:

```
TASK [debug] *******************************************************************
ok: [node1] => {
    "setup": {
        "ansible_facts": {
            "ansible_kernel": "4.18.0-513.11.1.el8_9.ppc64le"
        },
        "changed": false,
        "failed": false
    }
}
```

With this we can conclude the variable we are looking for is labeled `ansible_kernel`.

Then we can update the motd-facts.j2 template to include `ansible_kernel` as part of its message.

- Modify the template `motd-facts.j2`:

```
Welcome to {{ ansible_hostname }}.
{{ ansible_distribution }} {{ ansible_distribution_version}}
deployed on {{ ansible_architecture }} architecture
running kernel {{ ansible_kernel }}.
```

- Run the playbook.

```
[student@controller ~]$ ansible-navigator run motd-facts.yml -m stdout
```

- Verify the new message via SSH login to `node`.

```
[student@controller ~]$ ssh 10.3.48.[100+PARTICIPANT_ID]
Welcome to node.
RedHat 8.9
deployed on ppc64le architecture
running kernel 4.18.0-513.11.1.el8_9.ppc64le.
```

Roles - Making your playbooks reusable

Objective

While it is possible to write a playbook in one file as we've done throughout this workshop, eventually you'll want to reuse files and start to organize things.

Ansible Roles are the way we do this. When you create a role, you deconstruct your playbook into parts and those parts sit in a directory structure. This is explained in more details in the Tips and tricks and the Sample Ansible setup.

This exercise will cover:

- the folder structure of an Ansible Role
- how to build an Ansible Role
- creating an Ansible Play to use and execute a role
- using Ansible to create a Apache VirtualHost on node

Guide

Step 1 - Understanding the Ansible Role Structure

Roles follow a defined directory structure; a role is named by the top level directory. Some of the subdirectories contain YAML files, named `main.yml`. The files and templates subdirectories can contain objects referenced by the YAML files.

An example project structure could look like this, the name of the role would be "apache":

```
apache/
   defaults
       main.yml
   files
   handlers
       main.yml
   meta
       main.yml
   README.md
   tasks
       main.yml
   templates
   tests
       inventory
       test.yml
   vars
        main.yml
```

The various `main.yml` files contain content depending on their location in the directory structure shown above. For instance, `vars/main.yml` references variables, `handlers/main.yaml` describes handlers, and so on. Note that in contrast to playbooks, the `main.yml` files only contain the specific content and not additional playbook information like hosts, `become` or other keywords.

> **Tip**
>
> There are actually two directories for variables: `vars` and `default`. Default variables, `defaults/main.yml`, have the lowest precedence and usually contain default values set by the role authors and are often used when it is intended that their values will be overridden.

> Variables set in `vars/main.yml` are for variables not intended to be modified.

Using roles in a Playbook is straight forward:

```
---
- name: launch roles
  hosts: web
  roles:
    - role1
    - role2
```

For each role, the tasks, handlers and variables of that role will be included in the Playbook, in that order. Any copy, script, template, or include tasks in the role can reference the relevant files, templates, or tasks *without absolute or relative path names*. Ansible will look for them in the role's files, templates, or tasks respectively, based on their use.

Step 2 - Create a Basic Role Directory Structure

Ansible looks for roles in a subdirectory called `roles` in the project directory. This can be overridden in the Ansible configuration. Each role has its own directory. To ease creation of a new role the tool `ansible-galaxy` can be used.

> **Tip**
>
> Ansible Galaxy is your hub for finding, reusing and sharing the best Ansible content. `ansible-galaxy` helps to interact with Ansible Galaxy. For now we'll just using it as a helper to build the directory structure.

Okay, lets start to build a role. We'll build a role that installs and configures Apache to serve a virtual host. Run these commands in your `~/ansible-files` directory:

```
[student@controller ansible-files]$ mkdir roles
[student@controller ansible-files]$ ansible-galaxy init --offline roles/apache_vhost
```

Have a look at the role directories and their content:

```
[student@controller ansible-files]$ tree roles
```

```
roles/
   apache_vhost
       defaults
           main.yml
       files
       handlers
           main.yml
       meta
           main.yml
       README.md
       tasks
           main.yml
       templates
```

```
tests
    inventory
    test.yml
vars
    main.yml
```

Step 3 - Create the Tasks File

The `main.yml` file in the tasks subdirectory of the role should do the following:

- Make sure httpd is installed
- Make sure httpd is started and enabled
- Put HTML content into the Apache document root
- Install the template provided to configure the vhost

WARNING

The `main.yml` (and other files possibly included by main.yml) can only contain tasks, *not* complete playbooks!

Edit the `roles/apache_vhost/tasks/main.yml` file:

```yaml
---
- name: install httpd
  ansible.builtin.yum:
    name: httpd
    state: latest

- name: start and enable httpd service
  ansible.builtin.service:
    name: httpd
    state: started
    enabled: true
```

Note that here just tasks were added. The details of a playbook are not present.

The tasks added so far do:

- Install the httpd package using the yum module
- Use the service module to enable and start httpd

Next we add two more tasks to ensure a vhost directory structure and copy html content:

```yaml
- name: ensure vhost directory is present
  ansible.builtin.file:
    path: "/var/www/vhosts/{{ ansible_hostname }}"
    state: directory

- name: deliver html content
  ansible.builtin.copy:
    src: web.html
    dest: "/var/www/vhosts/{{ ansible_hostname }}/index.html"
```

Note that the vhost directory is created/ensured using the `file` module.

The last task we add uses the template module to create the vhost configuration file from a j2-template:

```
- name: template vhost file
  ansible.builtin.template:
    src: vhost.conf.j2
    dest: /etc/httpd/conf.d/vhost.conf
    owner: root
    group: root
    mode: 0644
  notify:
    - restart_httpd
```

Note it is using a handler to restart httpd after an configuration update.

The full `tasks/main.yml` file is:

```
---
- name: install httpd
  ansible.builtin.yum:
    name: httpd
    state: latest

- name: start and enable httpd service
  ansible.builtin.service:
    name: httpd
    state: started
    enabled: true

- name: ensure vhost directory is present
  ansible.builtin.file:
    path: "/var/www/vhosts/{{ ansible_hostname }}"
    state: directory

- name: deliver html content
  ansible.builtin.copy:
    src: web.html
    dest: "/var/www/vhosts/{{ ansible_hostname }}/index.html"

- name: template vhost file
  ansible.builtin.template:
    src: vhost.conf.j2
    dest: /etc/httpd/conf.d/vhost.conf
    owner: root
    group: root
    mode: 0644
  notify:
    - restart_httpd
```

Step 4 - Create the handler

Create the handler in the file `roles/apache_vhost/handlers/main.yml` to restart httpd when notified by the template task:

```
---
# handlers file for roles/apache_vhost
- name: restart_httpd
  ansible.builtin.service:
    name: httpd
    state: restarted
```

Step 5 - Create the web.html and vhost configuration file template

Create the HTML content that will be served by the webserver.

- Create an web.html file in the "src" directory of the role, `files`:

```
#> echo 'simple vhost index' > ~/ansible-files/roles/apache_vhost/files/web.html
```

- Create the `vhost.conf.j2` template file in the role's `templates` subdirectory.

The contents of the `vhost.conf.j2` template file are found below.

```
# {{ ansible_managed }}

<VirtualHost *:8080>
    ServerAdmin webmaster@{{ ansible_fqdn }}
    ServerName {{ ansible_fqdn }}
    ErrorLog logs/{{ ansible_hostname }}-error.log
    CustomLog logs/{{ ansible_hostname }}-common.log common
    DocumentRoot /var/www/vhosts/{{ ansible_hostname }}/

    <Directory /var/www/vhosts/{{ ansible_hostname }}/>
  Options +Indexes +FollowSymlinks +Includes
  Order allow,deny
  Allow from all
    </Directory>
</VirtualHost>
```

Step 6 - Test the role

You are ready to test the role against `node`. But since a role cannot be assigned to a node directly, first create a playbook which connects the role and the host. Create the file `test_apache_role.yml` in the directory `~/ansible-files`:

```
---
- name: use apache_vhost role playbook
  hosts: node
  become: True
  pre_tasks:
    - ansible.builtin.debug:
        msg: 'Beginning web server configuration.'
```

```
  roles:
    - apache_vhost
  post_tasks:
    - ansible.builtin.debug:
        msg: 'Web server has been configured.'
```

Note the `pre_tasks` and `post_tasks` keywords. Normally, the tasks of roles execute before the tasks of a playbook. To control order of execution `pre_tasks` are performed before any roles are applied. The `post_tasks` are performed after all the roles have completed. Here we just use them to better highlight when the actual role is executed.

Now you are ready to run your playbook:

```
[student@controller ansible-files]$ ansible-navigator run test_apache_role.yml
```

Run a curl command against `node` to confirm that the role worked:

```
[student@controller ansible-files]$ curl -s http://10.3.48.[100+PARTICIPANT_ID]:8080
simple vhost index
```

Congratulations! You have successfully completed this exercise!

Troubleshooting problems

Did the final curl work? You can see what ports the web server is running by using the ss command:

```
#> sudo ss -tulpn | grep httpd
```

There should be a line like this:

```
tcp   LISTEN 0      511                *:8080             *:*    users:(("httpd",pid=182
```

Pay close attention to the fifth column of the above output. It should be `*:8080`. If it is `*:80` instead or if it is not working, then make sure that the `/etc/httpd/conf/httpd.conf` file has `Listen 8080` in it. This should have been changed by Exercise 1.5