CEC 2025
Automating Power Virtual Servers with Ansible
part 2

Fabio Alessandro “Fale” Locati

v2025-06-01

Contents

Ansible **2**
Check the Prerequisites . . . 2
Objective . . . 2
Guide . . . 2
The Ansible Basics . . . 5
Objective . . . 5
Guide . . . 5
Writing Your First Playbook . . . 10
Objective . . . 10
Guide . . . 10
Using Variables . . . 18
Objective . . . 19
Guide . . . 19

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	150.239.19.229
Managed Host	[PARTICIPANT_MACHINE_IP]

Step 1 - Access the Environment

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

```
ssh USER@150.239.19.229
```

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`:

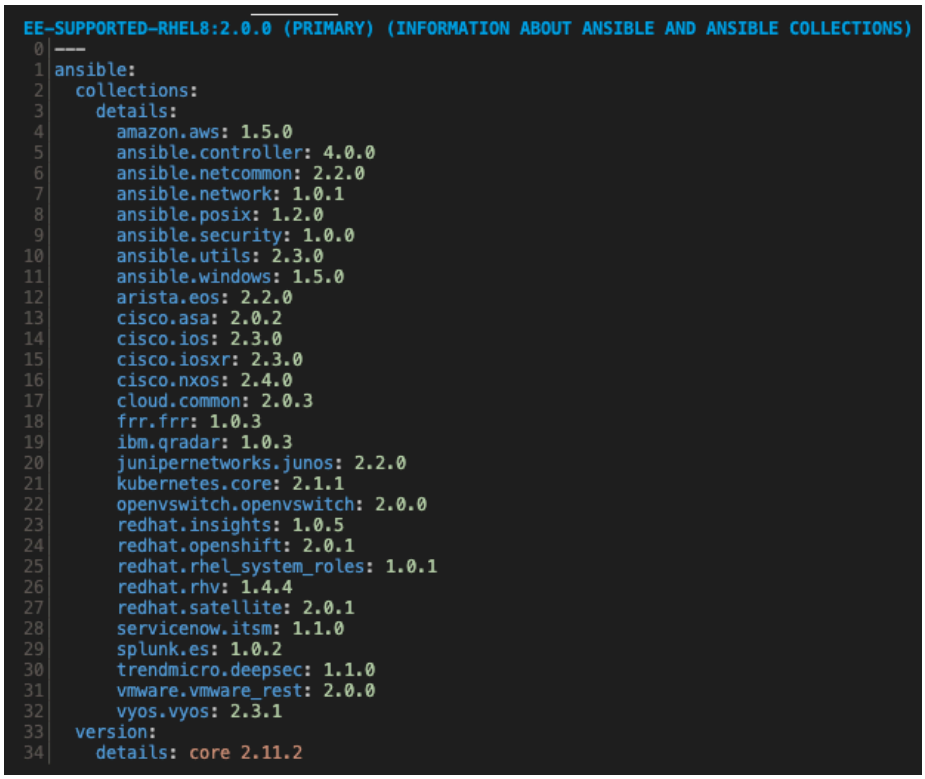

Figure 3: ee info

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=[PARTICIPANT__MACHINE_IP]  ansible_user=s[PARTICIPANT_ID]

[control]
controller ansible_host=150.239.19.229 ansible_user=s[PARTICIPANT_ID]
```

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.

```
$ ansible-navigator inventory --list -m stdout
```


```
{
    "_meta": {
        "hostvars": {
            "controller": {
                "ansible_host": "10.3.44.1",
                "ansible_user": "s22"
            },
            "node": {
                "ansible_host": "10.3.44.122",
                "ansible_user": "s22"
            }
        }
    },
    "all": {
        "children": [
            "ungrouped",
            "web",
            "control"
        ]
    },
    "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`.

```
$ ansible-navigator inventory --graph -m stdout
```

```
@all:
  |--@ungrouped:
  |--@web:
  |  |--node
  |--@control:
  |  |--controller
```

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.

```
$ ansible-navigator inventory --graph web -m stdout
$ ansible-navigator inventory --graph control -m stdout
$ 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
```

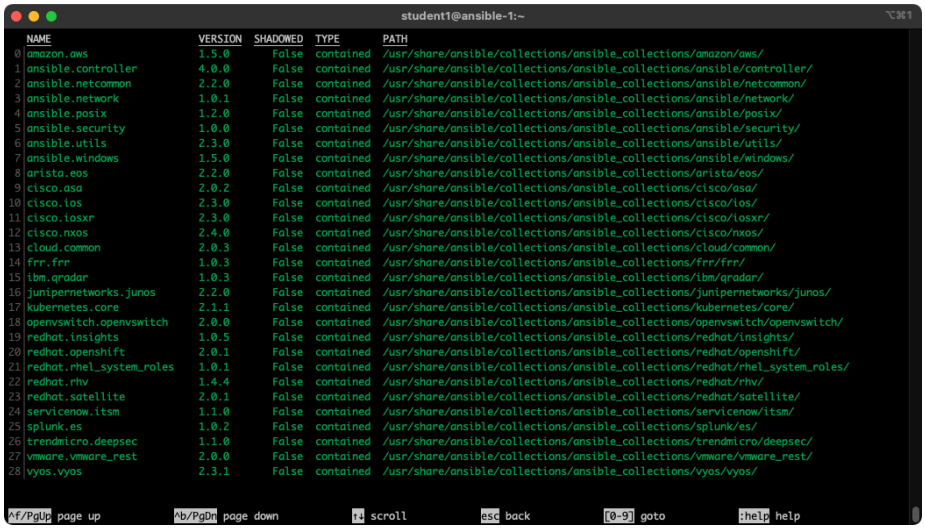

Figure 5: picture of ansible-navigator

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
```

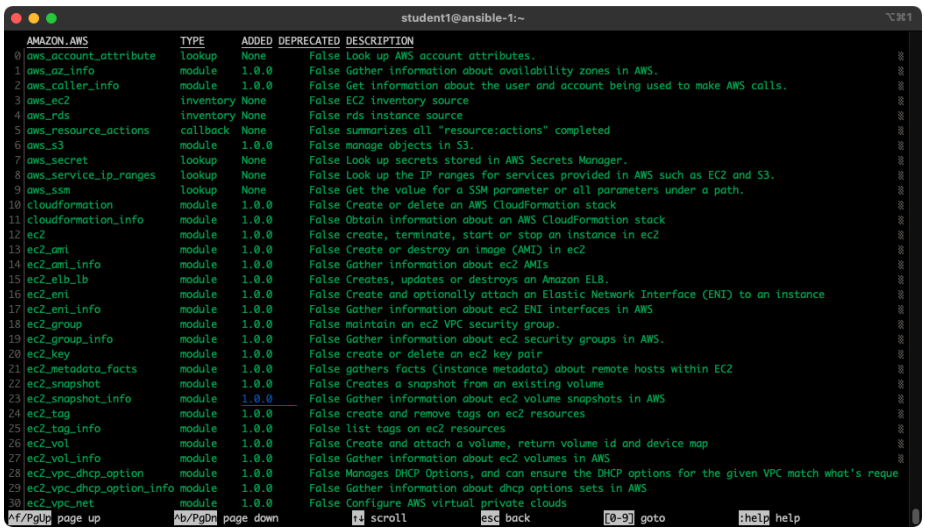

Figure 6: picture of ansible-navigator

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.

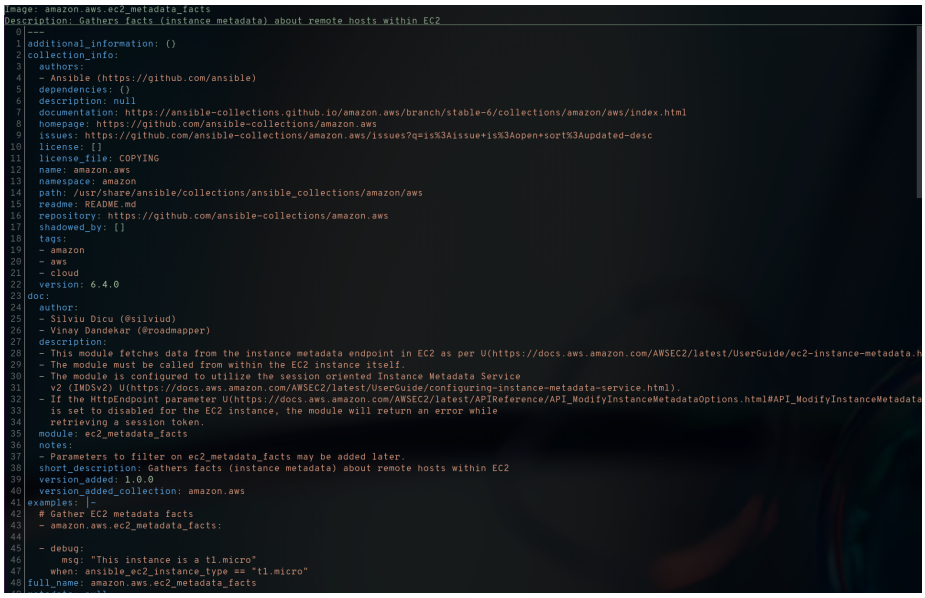

Figure 7: picture of ansible-navigator

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:

```
$ mkdir ansible-files
$ 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:

```
$ 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^^I  Failed     Skipped
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
```

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:

```
$ ssh root@10.3.44.[100+PARTICIPANT_ID]
```

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

```
Name        : httpd
Version     : 2.4.62
Release     : 4.el9
Architecture: ppc64le
[...]
```

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 fact
```

```
      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:

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

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

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

TASK [Gather the packages fact] *****************************************************
ok: [node]

TASK [Check whether httpd is installed] *********************************************
ok: [node] => {
    "msg": "httpd 2.4.62 is installed!"
}

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

```
$ 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
```

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

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: localhost
  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!

```
$ 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:

```
$ 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/`:

```
$ 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:

```
$ 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/`:

```
$ 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:

```
$ 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:

```
 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
```

```
$ cd ~
$ ansible-navigator run setup.yml -m stdou
\begin{Shaded}
```

```
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 \texttt{setup\_filter.yml} as shown
below. In this example, we filter to get the \texttt{eth0} facts as well
as memory details of \texttt{node}.
```

```
\begin{minted}{yaml}
---
- name: Capture Setup
  hosts: node
  tasks:
    - name: Collect only specific facts
      ansible.builtin.setup:
        filter:
          - 'ansible_eth0'
          - 'ansible_*_mb'
      register: setup
    - debug:
        var: setup
```

```
$ 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'
```

```
$ 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:

```
$ ansible-navigator run facts.ym
```

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
```