

# Exercise 1

## 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	IP
Windows Jump Host	jumphost	192.168.0.5
Ansible Control Host	ansible	192.168.0.188
Managed Host 1	centos1	192.168.0.61
Managed Host 2	Centos2	192.168.0.62
ONTAP Cluster 1	cluster1	192.168.0.101
ONTAP Cluster 2	cluster2	192.168.0.102

## Step 1.1 - Access the Environment

Your instructor will have the login information for your lab environment

Once the lab is started click the "Connect" button on the right-hand side.

Some prerequisite tasks have already been done for you:

- SSH connection and keys are configured (for Managed Hosts)
- sudo has been configured on the managed hosts to run commands that require root privileges.

Right click on the putty icon and select 'Ansible'



- **Tip**
- **The username is ansible**
- **The password is Netapp1!**

Install Python 3.x using YUM and Ansible via pip (an update to pip will also be required)

```
sudo yum install -y python3
sudo pip3 install pip --upgrade
sudo pip3 install ansible
```

Also install the netapp-lib, six, and requests python modules for api connections to ONTAP.

```
sudo pip3 install netapp-lib six requests
```

Finally, add netapp-lib also to the python2 environment for a later step.

```
sudo pip install netapp-lib
```

#### **Note**

*Ansible is keeping configuration management simple. Ansible requires no database or running daemons and can run easily on a laptop. On the managed hosts it needs no running agent.*

Check Ansible has been installed correctly

```
ansible --version
```

With Ansible installed it is now time to add the NetApp ONTAP Collection that will be used in these exercises.

```
sudo ansible-galaxy collection install netapp.ontap -p \
/usr/share/ansible/collections
```

And we have to make this location readable and executable by all users

```
sudo chmod -R a+rx /usr/share/ansible/collections
```

Now that Ansible is installed, clone the workshop's Git repository and use Ansible to prepare the lab environment.

```
git clone https://www.github.com/schmots1/ansible_workshop.git
cd ansible_workshop
ansible-playbook setup.yml
```

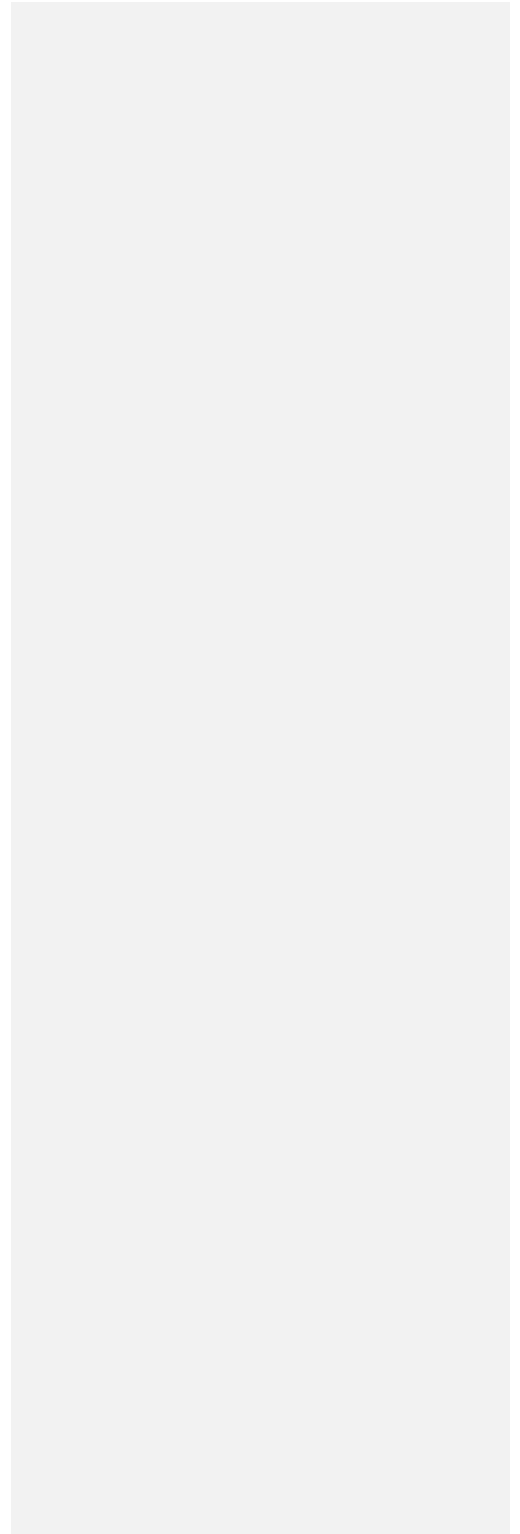
## Step 1.2 - Working the Labs

This lab is pretty command line centric.

- Don't type everything manually, use copy & paste from this document when appropriate. But stop to think and understand.
- All labs were prepared using **vim**, but we understand not everybody loves it. Feel free to use alternative editors. In the lab environment we provide **Midnight Commander** (just run **mc**, function keys can be reached via Esc-<n> or simply clicked with the mouse) or **Nano** (run **nano**). Here is a short [editor intro](#).

**Tip**

*In the lab guide commands you are supposed to run are shown with or without the expected output, whatever makes more sense in the context.*



# Exercise 2

## Step 2.1 - Work with your Inventory

Right click on the putty icon and select 'Ansible'  
*Tip*



**The username is ansible**  
**The password is Netapp1!**

To use the ansible 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 for you. The inventory is an ini formatted file listing your hosts, sorted in groups, additionally providing some variables. It looks like:

```
[all:vars]
ansible_user=ansible
ansible_ssh_pass=Netapp1!
ansible_port=22

[nodes]
centos1 ansible_host=192.168.0.61
centos2 ansible_host=192.168.0.62

[control]
ansible ansible_host=192.168.0.188

[ontap]
192.168.0.101 shortname=cluster1
192.168.0.102 shortname=cluster2
```

To reference inventory hosts, you supply a host pattern to the ansible command. Ansible has a `--list-hosts` option which can be useful for clarifying which managed hosts are referenced by the host pattern in an ansible command.

The most basic host pattern is the name for a single managed host listed in the inventory file. This specifies that the host will be the only one in the inventory file that will be acted upon by the ansible command. Run:

```
ansible centos1 --list-hosts
```

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:

```
ansible ontap --list-hosts
ansible ontap,ansible --list-hosts
ansible 'node*' --list-hosts
ansible all --list-hosts
```

It is OK to put systems in more than one group. For instance, a server could be both a web server and a database server. Note that in Ansible the groups are not necessarily hierarchical.

**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 2.2 - The Ansible Configuration Files

The behavior of Ansible can be customized by modifying settings in Ansible's ini-style configuration file. Ansible will select its configuration file from one of several possible locations on the control node, please refer to the [documentation](#).

**Tip**

*The recommended practice is to create an `ansible.cfg` file in the directory from which you run Ansible commands. This directory would also contain any files used by your Ansible project, such as the inventory and playbooks. Another recommended practice is to create a file `.ansible.cfg` in your home directory.*

In the lab environment provided to you an `.ansible.cfg` file has already been created and filled with the necessary details in the home directory of your `> user` on the control node:

```
ls -la ~/.ansible.cfg
```

Output the content of the file:

```
cat ~/.ansible.cfg
```

```
[defaults]
stdout_callback = yaml
connection = smart
timeout = 60
deprecation_warnings = False
```

```
host_key_checking = False
retry_files_enabled = False
inventory = /home/ansible/inventory
```

There are multiple configuration flags provided. Most of them are not of interest here, but make sure to note the last line: there the location of the inventory is provided. That is the way Ansible knew in the previous commands what machines to connect to.

Output the content of your dedicated inventory:

```
cat ~/ansible_workshop/inventory
```

```
[all:vars]
ansible_user=ansible
ansible_ssh_pass=Netapp1!
ansible_port=22

[nodes]
centos1 ansible_host=192.168.0.61
centos2 ansible_host=192.168.0.62

[control]
ansible ansible_host=192.168.0.188

[ontap]
192.168.0.101 shortname=cluster1
192.168.0.102 shortname=cluster2
```

## Step 2.3 - Ping a host

Let's start with something really basic - pinging a host. To do that we use the Ansible `ping` module. The `ping` module makes sure our target hosts are responsive. Basically, it connects to the managed host, executes a small script there and collects the results. This ensures that the managed host is reachable and that Ansible is able to execute commands properly on it.

### *Tip*

*Think of a module as a tool which is designed to accomplish a specific task.*

Ansible needs to know that it should use the `ping` module: The `-m` option defines which Ansible module to use. Options can be passed to the specified module using the `-a` option.

```
ansible nodes -m ping
```

As you see each node reports the successful execution and the actual result - here "pong".

## Step 2.4 - Listing Modules and Getting Help

Ansible comes with a lot of modules by default. To list all modules run:

```
ansible-doc -l
```

### Tip

In *ansible-doc* leave by pressing the button *q*. Use the *up/down* arrows to scroll through the content.

To find a module try e.g.:

```
ansible-doc -l | grep -i ontap
```

Get help for a specific module including usage examples:

```
ansible-doc na_ontap_volume
```

### Tip

Mandatory options are marked by a "=" in *ansible-doc*.

## Step 2.5 - Use the command module:

Now let's see how we can run a Linux command and format the output using the `command` module. It simply executes the specified command on a managed host:

```
ansible centos1 -m command -a "id"
```

In this case the module is called `command` and the option passed with `-a` is the actual command to run. Try to run this ad hoc command on all managed hosts using the `all` host pattern. \*Don't do this though as it will fail on ONTAP systems.

Another example: Have a quick look at the kernel versions your hosts are running:

```
ansible ansible -m command -a 'uname -r'
```

Sometimes it's desirable to have the output for a host on one line:

```
ansible nodes -m command -a 'uname -r' -o
```

### Tip

Like many Linux commands, *ansible* allows for long-form options as well as short-form. For example *ansible web --module-name ping* is the same as running *ansible web -m ping*. We are going to be using the short-form options throughout this workshop.

## Step 2.6 - The copy module and permissions

Using the copy module, execute an ad hoc command on centos1 to change the contents of the /etc/motd file. **The content is handed to the module through an option in this case.**

Run the following, but **expect an error**:

```
ansible centos1 -m copy -a 'content="Managed by Ansible\n" \
dest=/etc/motd'
```

As mentioned, this produces an error:

```
centos1 | FAILED! => {
  "changed": false,
  "checksum": "a314620457effe3a1db7e02eacd2b3fe8a8badca",
  "failed": true,
  "msg": "Destination /etc not writable"
}
```

The output of the ad hoc command is screaming **FAILED** in red at you. Why? Because user **ansible** is not allowed to write the motd file.

Now this is a case for privilege escalation and the reason sudo has to be setup properly. We need to instruct Ansible to use sudo to run the command as root by using the parameter `-b` (think “become”).

### Tip

*Ansible will connect to the machines using your current user name (ansible in this case), just like SSH would. To override the remote user name, you could use the `-u` parameter.*

For us it's okay to connect as **ansible** because sudo is set up. Not just for the user in the sudoers file, but also the password to use in our inventory file. Change the command to use the `-b` parameter and run again:

```
ansible centos1 -m copy -a 'content="Managed by Ansible\n" \
dest=/etc/motd' -b
```

This time the command is a success:

```
centos1 | CHANGED => {
  "changed": true,
  "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
  "dest": "/etc/motd",
  "gid": 0,
  "group": "root",
  "md5sum": "65a4290ee5559756ad04e558b0e0c4e3",
  "mode": "0644",
  "owner": "root",
}
```



```

    "secontext": "system_u:object_r:etc_t:s0",
    "size": 19,
    "src": "/home/student1/.ansible/tmp/ansible-tmp-1557857641.21-120920996103312/source",
    "state": "file",
    "uid": 0

```

Use Ansible with the generic command module to check the content of the motd file:

```
ansible centos1 -m command -a 'cat /etc/motd'
```

Run the `ansible centos1 -m copy ...` command from above with `-b` again. Note:

- The different output color (proper terminal config provided).
- The change from `"changed": true`, to `"changed": false`.
- The first line says SUCCESS instead of CHANGED.

**Tip**

*This makes it a lot easier to spot changes and what Ansible actually did.*

## Step 2.7 - The na\_ontap\_command ad-hoc

Using the `na_ontap_command` module, execute an ad hoc command against `cluster1` to check the version of ONTAP.

```
ansible localhost -m na_ontap_command -a 'hostname=192.168.0.101 \
username=admin password=Netapp1! https=true validate_certs=false \
command="version"'
```

```
localhost | CHANGED => {
```

```
    "changed": true,
```

```
    "msg": "<results xmlns='http://www.netapp.com/filer/admin'>
status='passed'><cli-output>NetApp Release Voodoooranger__9.6.0: Thu
Jan 04 05:29:16 UTC 2018\n\n</cli-output><cli-result-value>1</cli-
result-value></results>"
```

```
}
```

Pay attention to the fact that the host is defined as `localhost`. This is because all communication to ONTAP happens from the host that Ansible is running on. Also, the `na_ontap_command` module is not idempotent and will always return a changed result.

## Exercise 3

While Ansible ad hoc commands are useful for simple operations, they are not suited for complex configuration management or orchestration scenarios. For such use cases *playbooks* are the way to go.

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 is where you can take some of those ad-hoc commands you just ran and put them into a repeatable set of *plays* and *tasks*.

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.

### Step 3.1 - 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
- **collections**: List of collections being used as modules or roles.
- **tasks**: the operations to be performed by invoking Ansible modules and passing them the necessary options.
- **become**: privilege escalation in Playbooks, same as using `-b` in the ad hoc command. \*Not used with NetApp modules

#### **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 3.2 - Creating a Directory Structure and File for your Playbook

In this lab you create a Playbook to set up a Volume for NFS export:

- First step: Create an Export Policy for the export
- Second step: Create an Export Policy Rule for the new Export Policy
- Third step: Create a Volume

This Playbook makes sure all the steps for getting a new NFS export created are completed.

There is a [best practice](#) on the preferred directory structures for playbooks. It is strongly encouraged to read and understand these practices as you develop your Ansible skills. That said, this playbook is very basic and creating a complex structure will just confuse things.

Instead, you are going to create a very simple directory structure for your 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:



Right click on the putty icon and select 'Ansible'

```
cd ~/ansible_workshop/ansible-files/
```

Add a file called `nfsexport.yml` with the following content. As discussed in the previous exercises, use `vi/vim` or, if you are new to editors on the command line, check out the [editor intro](#) again.

```
---
- hosts: localhost
  collections:
    - netapp.ontap
  gather_facts: false
  name: Setup NFS Export
```

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

- The host to run the playbook against is defined via `hosts:`.
- The NetApp ONTAP collection is defined as being used.
- Ansible's default host fact gathering is disabled as we don't need it for ONTAP and this saves time.
- A name is given for the play via `name:`.

**Tip**

Now that you have defined the play, add a task to get something done. You will add a task in which ONTAP will ensure that an Export Policy called 'ansible\_volume\_policy'. . Modify the file so that it looks like the following listing:

```
---
- hosts: localhost
  collections:
    - netapp.ontap
  gather_facts: false
  name: Setup NFS Export
  tasks:
    - name: Create Policy
      na_ontap_export_policy:
        state: present
        name: ansible_volume_policy
        vserver: ansible_vserver
        hostname: 192.168.0.101
        username: admin
        password: Netapp1!
        https: true
        validate_certs: false
```

**Tip**

Since playbooks are written in YAML, alignment of the lines and keywords is crucial. Each section is two spaces in from the section it is a part of. Make sure to vertically align the - in the first run with the t in task. 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 prepare the area where our tasks will go:

- A task is named and the module for the task is referenced. Here it uses the `na_ontap_export_policy` module.
- Parameters for the module are added:
  - `state`: to define the wanted state of the policy
  - `name`: to identify the policy name
  - `vserver`: to identify the vserver this policy is part of
  - `hostname`: to identify ONTAP cluster to run this against
  - `username`: define the username to run this operation as
  - `password`: define the password for that username
  - `https`: configure the task to run over https instead of http
  - `validate_certs`: disable certification validation since we are using a self-signed certificate.

**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 3.3 - Running the Playbook

Playbooks are executed using the `ansible-playbook` command on the control node. Before you run a new Playbook it's a good idea to check for syntax errors:

```
ansible-playbook --syntax-check nfsexport.yml
```

Now you should be ready to run your Playbook:

```
ansible-playbook nfsexport.yml
```

The output should not report any errors but provide an overview of the tasks executed and a play recap summarizing what has been done. Use SSH to make sure the policy has been installed on `cluster1`.

```
ssh -l admin 192.168.0.101 export-policy show
```

Vserver	Policy Name
-----	-----
<code>ansible_vserver</code>	<code>ansible_volume_policy</code>
<code>ansible_vserver</code>	<code>default</code>
<code>[...]</code>	

Run the Playbook a second time and compare the output: The output changed from “changed” to “ok”, and the color changed from yellow to green. Also, the “PLAY RECAP” is different now. This make it easy to spot what Ansible actually did.

### Step 3.4 - Extend your Playbook: Create an Export Rule

The next part of the Playbook makes sure that a rule allowing access for the lab subnet is created with in the Export Policy from the first task.

Review the file `~/ansible_workshop/ansible-files/nfsexport_full.yml` to see the second and third tasks added to the play.

```
---
- hosts: localhost
  collections:
    - netapp.ontap
  gather_facts: false
  name: Setup NFS Export
  tasks:
    - name: Create Policy
      na_ontap_export_policy:
        state: present
        name: ansible_volume_policy
        vserver: ansible_vserver
        hostname: 192.168.0.101
        username: admin
        password: Netapp1!
        https: true
        validate_certs: false
    - name: Setup rules
      na_ontap_export_policy_rule:
        state: present
        policy_name: ansible_volume_policy
        client_match: 192.168.0.0/24
        ro_rule: sys
        rw_rule: sys
        super_user_security: sys
        vserver: ansible_vserver
        hostname: 192.168.0.101
        username: admin
        password: Netapp1!
        https: true
        validate_certs: false
    - name: Create volume
      na_ontap_volume:
        state: present
        name: ansible_volume
```

```

aggregate_name: aggr1
size: 10
size_unit: gb
policy: ansible_volume_policy
junction_path: /ansible_volume
space_guarantee: "none"
vserver: ansible_vserver
hostname: 192.168.0.101
username: admin
password: Netapp1!
https: true
validate_certs: false
volume_security_style: unix

```

You are getting used to the Playbook syntax, so what happens? The new task uses the `na_ontap_export_policy_rule` and `na_ontap_volume` module and defines the source and destination options for the volume operations as parameters.

Run the extended Playbook:

```
ansible-playbook nfsexport_full.yml
```

- Have a good look at the output
- Either using ssh or an ad-hoc command do a 'volume show' against the ONTAP system.

## Step 3.5 - Practice: Apply to Multiple Host

This was nice but the real power of Ansible is to apply the same set of tasks reliably to many hosts.

- Doing this with ONTAP is possible but takes a little bit of thought and the use of a variable.  
\*Variables will be covered more in the next exercise

```

[ontap]
192.168.0.101 shortname=cluster1
192.168.0.102 shortname=cluster2

```

Commented [FS1]: remove bullet

Look at the file `motd.yml` to see this playbook.

```

---
- name: Set the Message of the Day
  hosts: ontap
  collections:
    - netapp.ontap
  gather_facts: false

```

```
tasks:
- name: Verifying the MOTD
  na_ontap_motd:
    state: present
    motd_message: Set by Ansible
    vserver: "{{ shortname }}"
    hostname: "{{ inventory_hostname }}"
    username: admin
    password: Netapp1!
    https: true
    validate_certs: false
    connection: local
```

The vserver `{{ shortname }}` is coming from the inventory file and the `{{ inventory_hostname }}` is a special variable that is the inventory name for that run.

Now run the Playbook:

```
ansible-playbook motd.yml
```

Finally check if the login is set with ssh.

Setting your own variables will be covered in the next Exercise.



## Exercise 4

Previous exercises showed you the basics of Ansible Engine. In the next few exercises, you are going to learn some more advanced Ansible skills that will add flexibility and power to playbooks.

Ansible exists to make tasks simple and repeatable. However, not all systems are exactly alike and often require some slight change to the way an Ansible playbook is run. Enter variables.

Ansible supports variables to store values that can be used in 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.

Variables are referenced in Playbooks by placing the variable name in quoted 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` with the variable definitions is created.
- To define variables specifically for a host `centos1`, the file `host_vars/centos1` with the variable definitions is created.

### **Tip**

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

## Step 4.1 - Create Variables in Playbook

In the last exercise a playbook to create an NFS export and new Volume was used. The way the playbook was written makes it more difficult to update than needed, so you will be modifying it to contain variables.



Right click on the putty icon and select 'Ansible'

On the ansible control host, as the `ansible` user, change into the 'ansible-files' directory

```
cd ansible-files/
```

open the `nfsexport_vars.yml` file for editing

Now add a 'vars' section before the tasks section so that it looks like this.

```
---
- hosts: Localhost
  collections:
    - netapp.ontap
  gather_facts: false
  name: Setup NFS Export
  vars:
    hostname: 192.168.0.101
    username: admin
    password: Netapp1!
    vserver: ansible_vserver
    volname: ansible_volume
    size: 10
    client_match: 192.168.0.0/24
  tasks:
[...]
```

\* The [...] just represents that there is more to this file that is not being shown.

This allows you to use as variables, 'hostname', 'username', 'password', 'vserver', 'volname', 'size', 'client\_match'. Next step is to use those variables in the rest of the playbook.

The hard-coded entries for these options have been replaced in the respective task sections. Here is the `na_export_policy` section.

```
- name: Create Policy
  na_ontap_export_policy:
    state: present
    name: "{{ volname }}_policy"
    vserver: "{{ vserver }}"
    hostname: "{{ hostname }}"
    username: "{{ username }}"
    password: "{{ password }}"
    https: true
    validate_certs: false
```

Pay attention to the 'name:' section replacement. You can use one or more variables and plain text to create naming conventions. Each variable has to be in its own set of {}. For example "{{ username }}:{{ password }}".

Go ahead and review the rest of the variable values. Below is what it should look like.

```

---
- hosts: Localhost
  collections:
    - netapp.ontap
  gather_facts: false
  name: Setup NFS Export
  vars:
    hostname: 192.168.0.101
    username: admin
    password: Netapp1!
    vservers: ansible_vserver
    volname: ansible_volume
    size: 10
    client_match: 192.168.0.0/24
  tasks:
    - name: Create Policy
      na_ontap_export_policy:
        state: present
        name: "{{ volname }}_policy"
        vservers: "{{ vservers }}"
        hostname: "{{ hostname }}"
        username: "{{ username }}"
        password: "{{ password }}"
        https: true
        validate_certs: false
    - name: Setup rules
      na_ontap_export_policy_rule:
        state: present
        policy_name: "{{ volname }}_policy"
        client_match: "{{ client_match }}"
        ro_rule: sys
        rw_rule: sys
        super_user_security: sys
        vservers: "{{ vservers }}"
        hostname: "{{ hostname }}"
        username: "{{ username }}"
        password: "{{ password }}"
        https: true
        validate_certs: false
    - name: Create volume
      na_ontap_volume:
        state: present
        name: "{{ volname }}"
        aggregate_name: aggr1
        size: "{{ size }}"
        size_unit: gb
        policy: "{{ volname }}_policy"
        junction_path: "/"{{ volname }}"
        space_guarantee: "none"
        vservers: "{{ vservers }}"

```

```
hostname: "{{ hostname }}"
username: "{{ username }}"
password: "{{ password }}"
https: true
validate_certs: false
volume_security_style: unix
```

Now you can save and exit, then run the playbook again and see that it doesn't change anything because you are using the same variables.

```
ansible-playbook nfsexport_vars.yml
```

To see how easy changing variables are, change the volume name variable to something else and rerun the playbook.

## Step 4.2 - Create Variable Files

In addition to declaring variables directly in a playbook, you can also import variable files to a playbook.

Now create a new file called `variables.yml` in `~/ansible-files/`:

Variable files are simple YAML pair lists. Add your variables to the file so it looks like this.

```
hostname: 192.168.0.101
username: admin
password: Netapp1!
vserver: ansible_vserver
volname: ansible_volume
size: 10
client_match: 192.168.0.0/24
```

With this file saved, you can replace the 'vars' section of your `nfsexport_vars.yml` playbook with a `vars_files` section so it looks like this.

```
---
- hosts: Localhost
  collections:
    - netapp.ontap
  name: Setup NFS Export
  vars_files:
    - variables.yml
  tasks:
[...]
```

Now run the playbook again to verify those variables work.

## Step 4.3 - Defining Variables and Variable Files at the Command Line

Variables can also be passed into a playbook from the command line, allowing for dynamic changing of variable values which can be used with automation setups.

This is accomplished using the `--extra-vars` switch for `ansible-playbook`. The format is the following

```
ansible-playbook nfsexport_vars.yml --extra-vars "volname=clivol1 size=11 @variables.yml"
```

The defined variables at the command line override even the variable file listed in the command line. The important part to see is that `<variable>=<value>` and `@<path_to_file>` is how you define variables or variable files at the command line.

Try resizing your volume by passing in a larger 'size' at the command line with extra-vars

```
ansible-playbook nfsexport_vars.yml --extra-vars "size=15"
```

## Step 4.4 - YAML Aliases

YAML aliases are a component of YAML not Ansible, but are very useful, especially with NetApp tasks. A YAML aliases allows any number of lines to be represented by a single line as many times as you wish.

Since the playbook that was created for NFSexport uses `vserver`, `hostname`, `username`, `password`, `https`, and `validate_certs` in each task, we can define this once at the top of the page and use it where it's needed.

**The YAML alias can be defined in the 'vars' section. You start a YAML alias section with '`<aliasname>: &<aliasname>`'. This exercise will use 'ontap' as the alias name. A YAML alias section is filled out like any other YAML section with a two space indentation. Edit the vars section of `nfsexport_alias.yml` to look like this.**

```
---
- hosts: Localhost
  collections:
    - netapp.ontap
  name: Setup NFS Export
  vars_files:
    variables.yml
  vars:
    ontap: &ontap
```

```
  vserver: "{{ vserver }}"
  hostname: "{{ hostname }}"
  username: "{{ username }}"
  password: "{{ password }}"
  https: true
  validate_certs: false
  tasks:
[...]
```

Since the variable file is still being loaded, all those variables will exist for creating the alias.

See the tasks section to see how the Alias is being called.

```
- name: Create Policy
  na_ontap_export_policy:
    state: present
    name: "{{ volname }}_policy"
    <<: *ontap
```

Rerun the playbook to see that everything still works. \*If you resized your volume but have the old size in the playbook, it will show as changed for the volume as it is resized again.

## Step 4.5 - 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:

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

For example, creating a single playbook that can do a new NFS export or a new CIFS share.

Now we edit `naexport_conditional.yml` and add a `when` statement for 'protocol' to the NFS specific sections.

```
[...]
- name: Create Policy
  na_ontap_export_policy:
    state: present
    name: "{{ volname }}_policy"
    <<: *ontap
  when: protocol.Lower() == 'nfs'
- name: Setup rules
  na_ontap_export_policy_rule:
    state: present
    policy_name: "{{ volname }}_policy"
    client_match: "{{ client_match }}"
    ro_rule: sys
    rw_rule: sys
    super_user_security: sys
    <<: *ontap
  when: protocol.Lower() == 'nfs'
```

### Tip

The `Lower()` converts the `protocol` variable into all lowercase to make sure matching is easier.

If you were to run the `nascreate.yml` playbook right now, you would get an error, because the protocol variable hasn't been defined. Add the variable to the `variables.yml` file and set it to 'nfs' for now.

```
hostname: 192.168.0.20
username: admin
password: Netapp1!
vserver: ansible_vserver
volname: ansible_volume
size: 10
client_match: 192.168.0.0/24
protocol: nfs
```

Now if you run the `nascreate.yml` playbook it should show all ok's as it will just verify what was create in the previous exercise.

Now a CIFS share section can be added to `nascreate.yml`. This will use the `'na_ontap_cifs'` module.

```
- name: Create CIFS share
  na_ontap_cifs:
    state: present
    share_name: "{{ volname }}"
    path: "/"
    <<: *ontap
  when: protocol.Lower() == 'cifs'
```

\*note. Because of procedural playbook ordering, the CIFS share needs to be placed after the volume has been created.

Change the `variables.yml` protocol entrance and set it to 'CIFS' and run the `nascreate.yml` playbook again.

You will get a failure because there isn't actually a CIFS server running by default in this lab, but you can see that the export policy steps are skipped and the CIFS share task was run.

\*The `nfsexport.yml` playbook will not be negatively affected by the extra variable 'protocol'. Errors are only caused if you are calling a variable that is not defined somewhere, not if you define more variables than you use.

## Step 4.6 - 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 an example, here is a playbook that:

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

```
---
- name: manage httpd.conf
  hosts: web
  become: yes
  tasks:
    - name: Copy Apache configuration file
      copy:
        src: httpd.conf
        dest: /etc/httpd/conf/
      notify:
        - restart_apache
  handlers:
    - name: restart_apache
      service:
        name: httpd
        state: restarted
```

- 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.

None of the NetApp modules really have a point for handlers, but it's important to understand how they can be used.

## Step 4.7 - Simple Loops

Loops enable us to repeat the same task over and over again without having to call the same module more than once in a playbook. For example, let's say you want to create multiple volumes. 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 volumes with their corresponding 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 volumes. Put together all you have done so far and make a playbook from scratch. Create the file `loop_volumes.yml` in `~/ansible_workshop/ansible-files` on your ansible node as the ansible user. We will use the `na_ontap_volume` module to generate the volumes.

```
---
- hosts: localhost
  collections:
    - netapp.ontap
  gather_facts: false
  name: Setup NFS Export
  vars_files:
    - variables.yml
  vars:
    ontap: &ontap
    vserver: "{{ vserver }}"
    hostname: "{{ hostname }}"
    username: "{{ username }}"
    password: "{{ password }}"
    https: true
    validate_certs: false
  tasks:
    - name: Create volume
      na_ontap_volume:
        state: present
        name: "{{ item }}"
        aggregate_name: aggr1
        size: "{{ size }}"
        size_unit: gb
        junction_path: "/" + "{{ item }}"
        space_guarantee: "none"
        <<: *ontap
      loop:
        - vol1
        - vol2
        - vol3
```

Run the playbook.

Understand the playbook and the output:

- The names are not provided to the `na_ontap_volume` module directly. Instead, there is only a variable called `{{ item }}` for the parameter name.

- The `loop` keyword lists the actual volume 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 4.8 - Loops over hashes

As mentioned, loops can also be over lists of hashes. We can use this process to create the different volumes each with a different size.

```
tasks:
- name: Create volume
  na_ontap_volume:
    state: present
    name: "{{ item.name }}"
    aggregate_name: aggr1
    size: "{{ item.size }}"
    size_unit: gb
    junction_path: "/" + "{{ item.name }}"
    space_guarantee: "none"
  <<: *ontap
  loop:
    - { name: 'vol1', size: '10' }
    - { name: 'vol2', size: '20' }
    - { name: 'vol3', size: '15' }
```

This time the 'item' variable also has the additional variable defined in the hash of the loop. This loop has 'name' and 'size' as variables in it.

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/>).

## Step 4.9 - 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 in the `~/ansible_workshop/ansible-files/` 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_workshop/ansible-files/` directory create the Playbook `motd-facts.yml`:

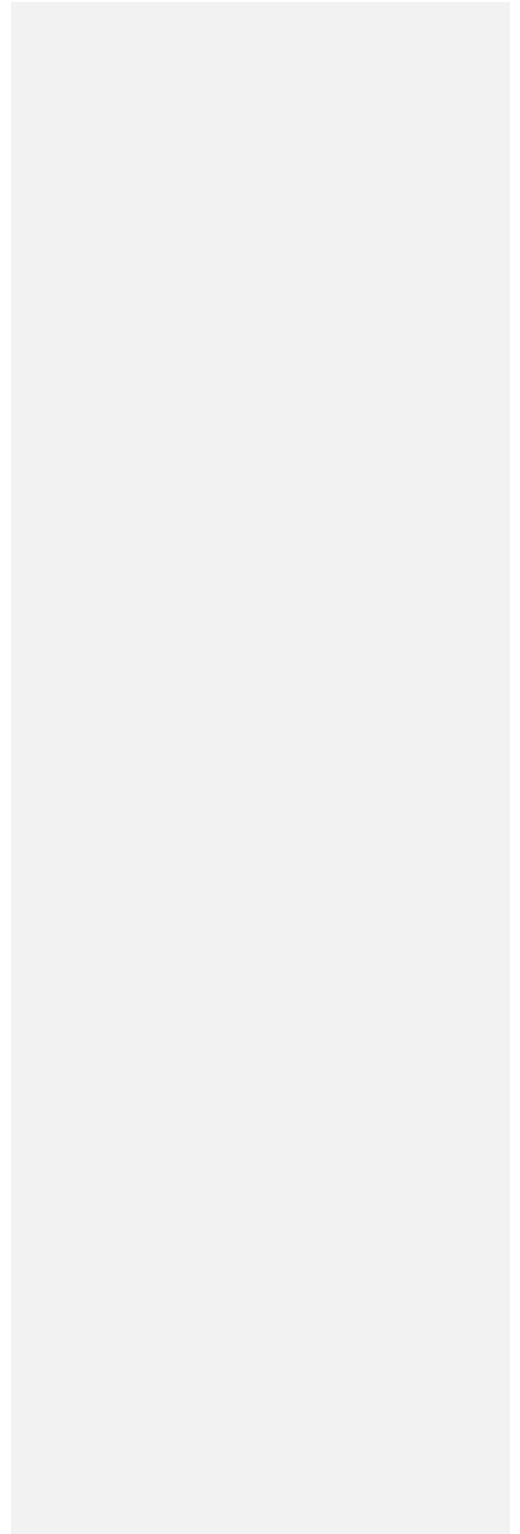
```
---
- name: Fill motd file with host data
  hosts: centos1
  become: yes
  tasks:
    - template:
        src: motd-facts.j2
        dest: /etc/motd
        owner: root
        group: root
        mode: 0644
```

You have done this a couple of times by now:

- Understand what the Playbook does.
- Execute the Playbook `motd-facts.yml`.
- Login to `centos1` via SSH or Putty and check the message of the day content.

- Log out of centos1.

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



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 detail in the [best practice](#) already mentioned in exercise 3.

## Step 4.10 - 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 "nas":

```
nas/
├── 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.yml` 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 `defaults`: Default variables 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 can be set in either `vars/main.yml` or `defaults/main.yml`, but not in both places.*

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 4.11 - 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.

Here is how to a role template. Run these commands in your `~/ansible_workshop/ansible-files` directory:

```
ansible-galaxy init --offline roles/roleexample
```

Have a look at the role directories and their content:

```
sudo yum install -y tree
tree roles
```

## Step 4.12 - Create the Tasks File

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

- Create an export policy and export policy rule if nfs is specified
- Create a volume
- Create a CIFS share if cifs is specified

### **WARNING**

*The `main.yml` (and other files possibly included by `main.yml`) can only contain tasks, not complete Playbooks! Also YAML alias don't work in roles files.*

Commented [FS2]: fix the indent level

Copy the nascreate.yml file into roles/nascreate/tasks/main.yml and edit it for the role. Remove the hosts, gather\_facts, vars\_files, vars, tasks, headers and sections, and change the task calls to fully filled out, not using aliases. Finally don't forget to delete the two extra spaces that are now in front of each line.

```
cp nascreate.yml roles/nascreate/tasks/main.yml
```

```
---
- name: Create Policy
  na_ontap_export_policy:
    state: present
    name: "{{{ volname }}}_policy"
    vservers: "{{{ vservers }}}"
    hostname: "{{{ hostname }}}"
    username: "{{{ username }}}"
    password: "{{{ password }}}"
    https: true
    validate_certs: false
  when: protocol.lower() == 'nfs'
- name: Setup rules
  na_ontap_export_policy_rule:
    state: present
    policy_name: "{{{ volname }}}_policy"
    client_match: "{{{ client_match }}}"
    ro_rule: sys
    rw_rule: sys
    super_user_security: sys
    vservers: "{{{ vservers }}}"
    hostname: "{{{ hostname }}}"
    username: "{{{ username }}}"
    password: "{{{ password }}}"
    https: true
    validate_certs: false
  when: protocol.lower() == 'nfs'
- name: Create volume
  na_ontap_volume:
    state: present
    name: "{{{ volname }}}"
    aggregate_name: aggr1
    size: "{{{ size }}}"
    size_unit: gb
    policy: "{{{ volname }}}_policy"
    junction_path: "{{{ volname }}}"
    space_guarantee: "none"
    vservers: "{{{ vservers }}}"
    hostname: "{{{ hostname }}}"
    username: "{{{ username }}}"
    password: "{{{ password }}}"
    https: true
    validate_certs: false
    volume_security_style: unix
```



```
- name: Create CIFS share
  na_ontap_cifs:
    state: present
    share_name: "{{ volname }}"
    path: "{{ volname }}"
    vserver: "{{ vserver }}"
    hostname: "{{ hostname }}"
    username: "{{ username }}"
    password: "{{ password }}"
    https: true
    validate_certs: false
    when: protocol.lower() == 'cifs'
```

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

Since we will still be using variables, we need to have the role know what variables it will accept. Copy the variable.yml file to roles/nascreate/defaults/main.yml.

```
cp ~/ansible-files/variables.yml roles/nascreate/defaults/main.yml
```

This makes sure the variables exist in the role. You can remove the variable values from main.yml or leave them. You will be overriding them from the variables.yml file still.

## Step 4.13 - Test the role

You are ready to test the role. A playbook still needs to be created to call the role. Create the file nas\_role.yml in the directory ~/ansible-files: \* The role we are using isn't in a collection, but its modules are, so we still need to define a collection to use in the playbook.

```
---
- hosts: localhost
  collections:
    - netapp.ontap
  gather_facts: false
  vars_files:
    - variables.yml
  roles:
    - nascreate
```

That's the whole playbook to call the role.

Now you are ready to run your playbook:

```
ansible-playbook nas_role.yml
```

Experiment by changing some of the values `variable.yml` and run the `nas_role.yml` playbook again. Remember that with `'protocol: cifs'` it will always error on that step.

# Exercise 5

## Exercise 5.1 - Introduction to AWX

### Why AWX?

AWX 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.
- graphically manages or synchronizes inventories with a wide variety of sources.
- has a RESTful API
- And much more...

### Your AWX 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 & AWX	ansible
Managed Host 1	centos1
Managed Host 2	Centos2
ONTAP Cluster 1	cluster1
ONTAP Cluster 2	cluster2

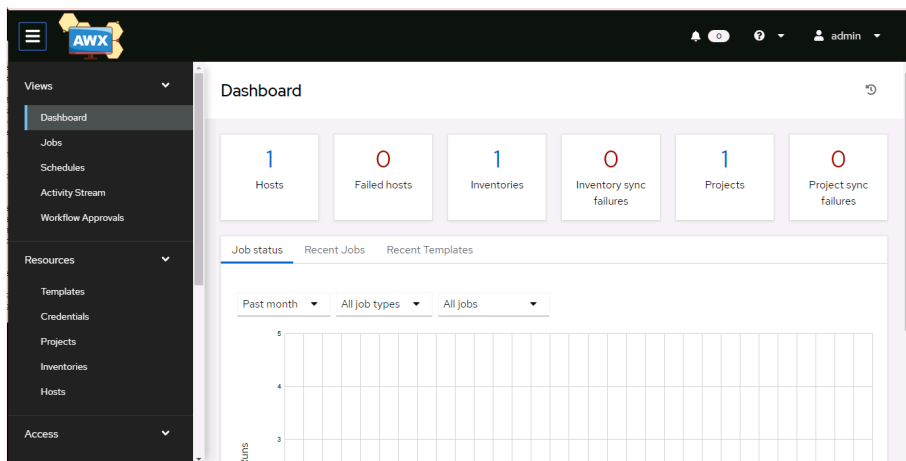
## Dashboard

Let's have a first look at AWX: Point your browser to the URL you were given, similar to `https://awx.lab.com` and log in as `admin`. The password is 'password'

The web UI of AWX 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.



Commented [FS3]: Check all images to make sure they are properly aligned with the left hand indent of the text

## Concepts

Before we dive further into using AWX for your automation, you should get familiar with some concepts and naming conventions.

### Projects

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

### 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 AWX, from one of AWX's supported cloud providers or through dynamic inventory scripts.

### Credentials

Credentials are utilized by AWX 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.

AWX credentials are imported and stored encrypted in AWX 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, AWX requires that you first create a job template.


### Jobs

A job is basically an instance of AWX launching an Ansible playbook against an inventory of hosts.

## Exercise 5.2 - Inventories, credentials and ad hoc commands

### Create an Inventory

This is the equivalent of an inventory file in Ansible Engine. There is a lot more to it (like dynamic inventories) but these are the basics.

- You should already have the web UI open, if not: Point your browser to the URL you were given, <http://aws.lab.com> and log in as `admin`. The password is 'password'  
Create the inventory:
- In the web UI menu on the left side, go to **RESOURCES** → **Inventories**, click the  button **Add Inventory**.
- **NAME:** Workshop Inventory

- **ORGANIZATION:** Default
- Click **SAVE**

Now there will be two inventories, the **Demo Inventory** and the **Workshop Inventory**. In the **Workshop Inventory** click the **Hosts** button, it will be empty since no hosts have been added there.


Add some hosts. The Section 1 exercise inventory will be used. Here it is as a reminder.

```
[all:vars]
ansible_user=ansible
ansible_ssh_pass=Netapp1!
ansible_port=22

[nodes]
centos1 ansible_host=192.168.0.61
centos2 ansible_host=192.168.0.62

[control]
ansible ansible_host=192.168.0.188

[ontap]
192.168.0.101 shortname=cluster1
192.168.0.102 shortname=cluster1
```

- In the inventory view of AWX click on your **Workshop Inventory**
- Click on the **HOSTS** button
- Click the  button.
- **HOST NAME:** centos1
- **Variables:** Under the three dashes ---, enter `ansible_host: 192.168.0.61` in a new line. Note that the variable definition has a colon : and a space between the values, not an equal sign = like in the inventory file.
- Click **SAVE**

## Machine Credentials

One of the great features of AWX is to make credentials usable to users without making them visible. To allow AWX to execute jobs on remote hosts, you must configure connection credentials.

### Note

This is one of the most important features of AWX: **Credential Separation!** Credentials are defined separately and not with the hosts or inventory settings.

## Configure Machine Credentials

Now we will configure the credentials to access our managed host from AWX. In the **RESOURCES** menu choose **Credentials**. Now:

Click the  button to add new credentials

- **NAME:** Workshop Credentials
- **ORGANIZATION:** Default
- **CREDENTIAL TYPE:** Click on the dropdown list and pick **Machine**
- **USERNAME:** ansible
- **PASSWORD:** Netapp1!
- **PRIVILEGE ESCALATION METHOD:** sudo
- Click **SAVE**
- Go back to the **RESOURCES** → **Credentials** → **Workshop Credentials** and note that the password is not visible.

### Tip

Whenever you see a magnifying glass icon next to an input field, clicking it will open a list to choose from.

You have now setup credentials to use later for your inventory hosts.

## Run Ad Hoc Commands

As you've probably done with Ansible before you can run ad hoc commands from AWX as well.

- In the web UI go to **RESOURCES** → **Inventories** → **Workshop Inventory**
- Click the **HOSTS** button to change into the hosts view and select the centos1 host by ticking the box to the left of the host entry.

- Click **RUN COMMAND**. In the next screen you have to specify the ad hoc command:
- As **MODULE** choose **ping** and click **Next**
- For the **Execution Environment** select **AWX EE 0.5.0** and click **Next**
- For **MACHINE CREDENTIAL** select **Workshop Credentials**.
- Click **LAUNCH** and watch the output.

The simple **ping** module doesn't need options. For other modules you need to supply the command to run as an argument.

## Create and Configure ONTAP Credentials

Credentials pass in a particular variable. You can create custom credentials and define the variables as anything you want. Make one now for ONTAP.

- In the web UI go to **ADMINISTRATION → Credential Types**
- Click the **Add ▼** button to add new credentials
- For the name enter ONTAP
- The INPUT CONFIGURATION defines how and what prompts are given when creating a credential with this type. Enter the following starting a line after ---

*fields:*

```
- id: username
  type: string
  label: Username
- id: password
  type: string
  label: Password
  secret: true
```

The INJECTOR CONFIGURATION defines what variables will be passed in by this Credential Type. Since this is for ONTAP `ontap_username` and `ontap_password` are good unique variables. Enter the following starting a line after ---

*extra\_vars:*


```
netapp_password: '{{ password }}'
netapp_username: '{{ username }}'
```



Click **SAVE**.

Now you need to create a credential entry using the created type.

In the **RESOURCES** menu choose **Credentials**. Now:

Click the  button to add new credentials

- **NAME:** ONTAP Credentials
- **CREDENTIAL TYPE:** Click on the dropdown menu, pick **ONTAP**
- **USERNAME:** admin
- **PASSWORD:** Netapp1!
- Click **SAVE**

## • Exercise 5.3 – Execution Environment

Execution Environments are containerized installations of Ansible engine allowing you to have multiple different versions of a Ansible, or different sets of collections without having to update or upgrade AWX.

The default Execution Environment that AWX installs though does not have the needed python libraries to run all NetApp modules. To fix this a custom Execution Environment will need to be added to be used.

### Add the Execution Environment

In the **Administration** menu choose **Credentials**. Now:

Click the  button to add new credentials

- **Name:** NetApp EE
- **Image:** schmots1/na\_ansible\_ee
- Click **SAVE**

## • Exercise 5.4 - Projects & job templates

An AWX **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 AWX, including Git, Subversion, and Mercurial.

You should definitely keep your Playbooks under version control. In this lab you will use Playbooks kept in a GitHub repo.

### Setup Git Repository

The git repo being used viewable at:

[https://www.github.com/schmots1/ansible\\_workshop](https://www.github.com/schmots1/ansible_workshop)

There are playbooks in this repo that we will work with in AWX.

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

## Create the Project

- Go to **RESOURCES** → **Projects** in the side menu view click the **Add ▾** button. Fill in the form:
- **NAME:** Ansible Workshop Examples
- **ORGANIZATION:** Default
- **Default Execution Environment:** NetApp EE
- **Source Control Credential Type:** Git
- **Source Control URL:** `https://www.github.com/schmots1/ansible_workshop.git`
- 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 AWX 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 **Templates** view, click the **Add ▾** button and choose Add **job template**.

### *Tip*

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

- **NAME:** Create Volume
- **JOB TYPE:** Run
- **INVENTORY:** Workshop Inventory (though this playbook still runs against local host so this is mainly a required checkbox)
- **PROJECT:** Ansible Workshop Examples
- **PLAYBOOK:** volume.yml
- **CREDENTIAL:** ONTAP Credentials (hint: change the credential type dropdown)
- Click **SAVE**

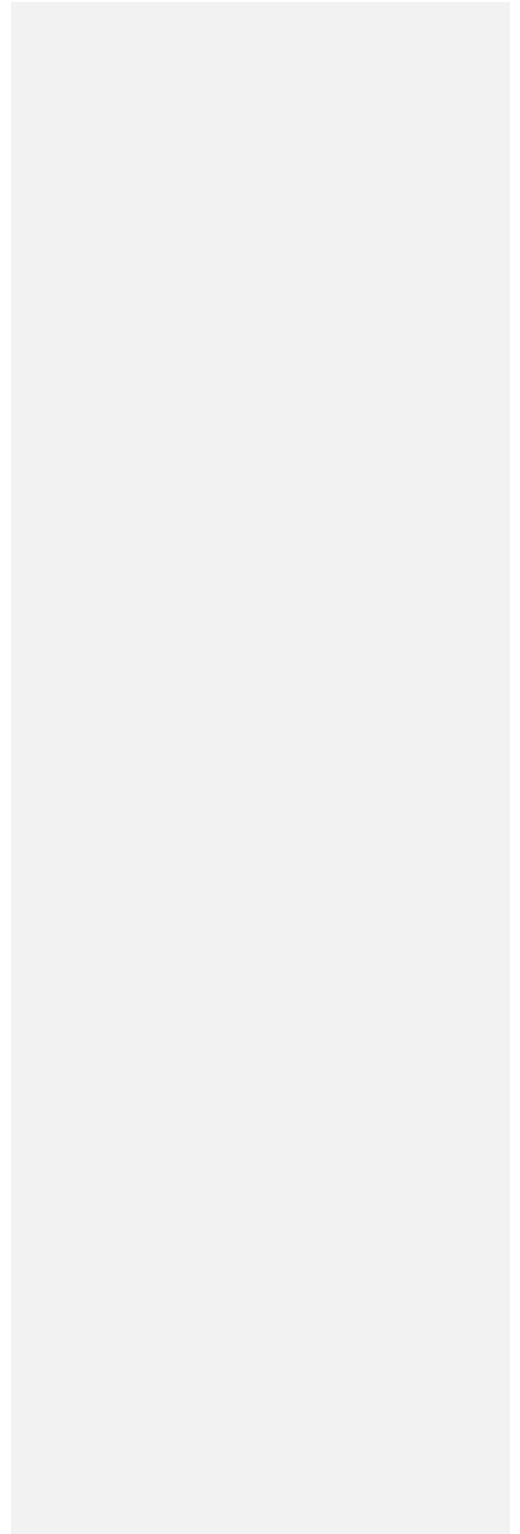
You can start the job by directly clicking the **LAUNCH** button, or by clicking on the rocket in the Job Templates overview. After launching the Job Template, you are automatically brought to the job overview where you can follow the playbook execution in real time:

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 analyze 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.

- The user who ran the job is recorded making it easy to identify who initiated what changes.
- On the right side, the output of the playbook run is shown.

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 SCM update was started. This is the Git update we configured for the **Project** on launch!



## Exercise 5.5 - Surveys

You might have noticed the **ADD SURVEY** button in the **Template** configuration view. A survey is a way to create a simple form to ask for parameters that get used as variables when a **Template** is launched as a **Job**.

You have a job template that will create a volume, but only with predefined options.

The survey feature only provides a simple query for data - it does not support four-eye principles, queries based on dynamic data or nested menus.

### Create a Survey

Go to the **Templates** view, click the name of the template that was just created "Create Volume". This will let you review its settings. To update them you would scroll down and click the 'Edit' button. Since we are adding a survey that isn't necessary. Instead select the 'Survey' tap from the Template view

Click on 'Add'

You will create two prompts. One for Volume Name one for Size.

If you look at the playbook in the git repo you will see that there is a vars section that has a 'volname' and 'size' entry. The survey will override those variables.

- **Question:** Volume Name
- **Ansible variable name:** volname
- **Answer type:** Text
- Click **Save**

Repeat the same steps with these values.

- **Question:** Size (in Gigs)
- **Answer variable name:** size
- **Answer type:** Text
- Click **Save**

## Launch the Template

Now launch **Volume Create** job template.

Before the actual launch the survey will ask for **Volume Name** and **Size (in Gigs)**. Fill in some text and click **Next**. The next window shows the values, if all is good run the Job by clicking **Launch**.

### *Tip*

*Note how the two survey lines are shown to the left of the Job view as **Extra Variables**.*

## Exercise 5.6 - Role-based access control

You have already learned how AWX separates credentials from users. Another advantage of AWX is the user and group rights management.

## Ansible AWX Users

There are three types of AWX 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 AWX environment.
- **System Administrator:** Has admin, read, and write privileges over the entire AWX installation.

Let's create a user:

- In the AWX menu under **ACCESS** click **Users**
- Click the blue 'Add' button
- Fill in the values for the new user:
- **Username:** netapp
- **Email:** netapp@local.com
- **Password:** Netapp1!
- Confirm password
- **First Name:** Netapp

- **Last Name:** User
- **User Type:** Normal User
- Click **SAVE**

## AWX 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 blue 'Add' button and create a team named Users.
- Click **SAVE**

Now you can add a user to the Team:

- Switch to the Access tab of the Users Team by clicking the **Access** tab button.
- Click the blue 'Add' button, select 'Users' and click 'Next'. Tick the box next to the netapp user and click 'Next'. Select 'Member' for type and click **SAVE**.

Now click the **Roles** tab in the **TEAMS** view, you will be greeted with "No Team Roles Found"

Permissions allow to read, modify, and administer projects, inventories, and other AWX 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 **netapp** should only be allowed to run the templates you define.

Add the permission to use the template:

- In the Roles view of the Team Users click the blue 'Add' button to add permissions.
- A new window opens. You can choose to set permissions for a number of resources.



- Select the resource type **JOB TEMPLATES** and click 'Next'
- Choose the Create Volume Template by ticking the box next to it and clicking 'Next'.
- Here you assign roles to the selected resource.
- Choose **EXECUTE**
- Click **SAVE**

## Test Permissions

Now log out of AWX's web UI and in again as the **netapp** user.

- Go to the **Templates** view, you should notice the Create Volume template is listed. This user is allowed to view and launch, but not to edit the Template. Just open the template and try to change it.
- Run the Job Template by clicking the rocket icon. Enter the survey content to your liking and launch the job.

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 AWX!

## Exercise 5.7 - Workflows

### AWX Workflows

Workflows were introduced as a major new feature in AWX 3.1. The basic idea of a workflow is to link multiple Job Templates together. They may or may not share inventory, Playbooks or even permissions. The links can be conditional:

- if job template A succeeds, job template B is automatically executed afterwards
- but in case of failure, job template C will be run.

And the workflows are not even limited to Job Templates but can also include project or inventory updates.

This enables new applications for AWX: different Job Templates can build upon each other. E.g. the networking team creates playbooks with their own content, in their own Git repository and even targeting their own inventory, while the operations team also has their own repos, playbooks and inventory.

In this lab you'll learn how to setup a workflow.

- Three new job templates will be added to use as building blocks for the workflow.

#### **Warning**

*If you are still logged in as user **netapp**, log out of and log in as user **admin** again. \*\**

### Set up Job Templates

Now you have to create Job Templates like you would for “normal” Jobs.

- Go to the **Templates** view, click the blue add button **Add ▾** and choose **Add Job Template**:
- **NAME:** Export Policy
- **JOB TYPE:** Run
- **INVENTORY:** Workshop Inventory
- **PROJECT:** Ansible Workshop Examples

- **PLAYBOOK:** policy.yml
- **CREDENTIAL:** ONTAP Credentials
- Click **SAVE**
- Go to the **Templates** view, click the green plus button and choose **Job Template:**
- **NAME:** Policy Rule
- **JOB TYPE:** Run
- **INVENTORY:** Workshop Inventory
- **PROJECT:** Ansible Workshop Examples
- **PLAYBOOK:** rule.yml
- **CREDENTIALS:** ONTAP Credentials
- Click **SAVE**
- Go to the **Templates** view, click the green plus button and choose **Job Template:**
- **NAME:** Volume
- **JOB TYPE:** Run
- **INVENTORY:** Workshop Inventory
- **PROJECT:** Ansible Workshop Examples
- **PLAYBOOK:** volume.yml
- **CREDENTIALS:** ONTAP Credentials

Default variables can be added to workflows to override any variables set in the playbooks or to allow setting variables. These are entered in the “EXTRA VARIABLES” section in the same way you add them to a variables file. Since this template job will always be used for NFS add a variable for the export\_policy

```
1 ---
2 export_policy: "{{ volname }}"
```

- Click **SAVE**


A second Volume template is created because the 'Create Volume' template has a survey and we don't want our individual building blocks to have surveys, we want the Workflow to have a survey

**Tip**

*If you want to know what the Playbooks look like, check out the Github URL.*

## Set up the Workflow

And now you finally set up the workflow. Workflows are configured in the **Templates** view, you might have noticed you can choose between **Job Template** and **Workflow Template** when adding a template so this is finally making sense.

- Go to the **Templates** view and click the blue 'Add' button . This time choose **Add Workflow Template**
- **NAME:** NFS Export
- **ORGANIZATION:** Default
- Click **SAVE**
- After saving the template the **Workflow Visualizer** opens to allow you to build a workflow. You can later open the **Workflow Visualizer** again by using the button on the template details page.
- Click on the **START** button and the add new node window opens. Here you can assign an action to the node, you can choose between **Job Template**, **Project Sync**, **Approval**, **Workflow Job Template**, and **INVENTORY SYNC**.
- In this lab we'll link Jobs together, so select **Job Template** and the **Export Policy** job and click **SAVE**.
- The node gets annotated with the name of the job. Hover the mouse pointer over the node, you'll see buttons for deleting reordering, editing, information and adding nodes.
- Click the + sign
- This should only run if the first job succeeds so select 'On Success' and click 'Next'.

**Tip**

*The type allows for more complex workflows. You could lay out different execution paths for successful and for failed Playbook runs.*

- Still using a 'Job Template' Choose **Policy Rule** as the next Job (you might have to switch to the next page)
- Click **SAVE**
- Click the + sign on the Policy Rule task
- Choose 'On Success'
- Choose **volume** as the next Job (you might have to switch to the next page)
- Click **SAVE**
- Click **SAVE** in the **WORKFLOW VISUALIZER** view

**Tip**

*The **Workflow Visualizer** has options for setting up more advanced workflows, please refer to the documentation*

We are also going to add some extra variables to this Workflow so that its unique to the lab environment. Add some defaults for this environment. Select 'Edit' and add the following to the 'Variables' section

```
1 ---
2 hostname: 192.168.0.101
3 vserver: ansible_vserver
4 ro_rule: sys
5 rw_rule: sys
6 su_rule: sys
```

Click SAVE

The final step is to create a survey for the **NFS Export** Template Workflow. Refer back to exercise 2.4 if you need a refresher on how to do this. The following variables will need to be defined.

- volname
- client\_match
- size

## And Action

Your workflow is ready to go, launch it.

- Click the blue **LAUNCH** button directly or go to the **Templates** view and launch the **NFS Export** workflow by clicking the rocket icon.

The screenshot displays two panels from a workflow management system. The left panel, titled 'DETAILS', shows the execution status as 'Successful'. It lists the start time as 7/10/2019 11:45:24 PM and the finish time as 7/10/2019 11:46:16 PM. The template used is 'Deploy Webapp Server', launched by 'admin'. Below this, there is a section for 'EXTRA VARIABLES' with a 'YAML' button and an 'EXPAND' link. The right panel, titled 'Deploy Webapp Server', shows a visual representation of the workflow steps. It includes a progress bar at the top with 'TOTAL NODES' and 'ELAPSED' time. The workflow consists of two main steps: 'Tomcat Deploy' and 'Web App Deploy', each with a 'DETAILS' link below it. A blue square icon is visible at the start of the workflow line.

Note how the workflow run is shown in the job view. In contrast to a normal job template job execution this time there is no playbook output on the right, but a visual representation of the different workflow steps. If you want to look at the actual playbooks behind that, click **DETAILS** in each step. If you want to get back from a details view to the corresponding workflow, click the back button in your browser.

## EXTRA

Try adding the Workflow to the team permissions and then switch to the netapp user account and see if only two templates are available.