

Ansible





Table of Contents

Learning the Fundamentals of Ansible

1. Getting Started with Ansible: 6
2. Understanding the Fundamentals of Ansible: 63
3. Defining Your Inventory: 146
4. Playbooks and Roles: 206

Expanding the Capabilities of Ansible

5. Consuming and Creating Modules: 286
6. Consuming and Creating Plugins: 352

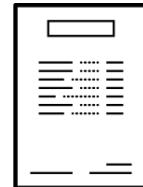
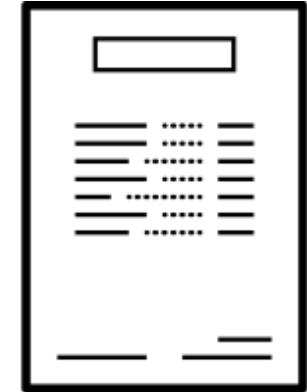




Table of Contents

7. Coding Best Practices: 395
8. Advanced Ansible Topics: 446



Using Ansible in an Enterprise

9. Troubleshooting and Testing Strategies: 529

Learning the Fundamentals of Ansible

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or educational setting.

Learning the Fundamentals of Ansible

This section contains the following lessons:

- lesson 1, Getting Started with Ansible
- lesson 2, Understanding the Fundamentals of Ansible
- lesson 3, Defining Your Inventory
- lesson 4, Playbooks and Roles



ANSIBLE

1: Getting Started with Ansible

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or work-related environment.

Getting Started with Ansible

In this lesson, we will cover the following topics:

- Installing and configuring Ansible
- Understanding your Ansible installation
- Running from source versus pre-built RPMs



Technical requirements

- Ansible has a fairly minimal set of system requirements—as such, you should find that if you have a machine (either a laptop, a server, or a virtual machine) that is capable of running Python, then you will be able to run Ansible on it.
- Later in this lesson, we will demonstrate the installation methods for Ansible on a variety of operating systems—it is hence left to you to decide which operating systems are right for you.

Installing and configuring Ansible

- Ansible is written in Python and, as such, can be run on a wide range of systems.
- This includes most popular flavors of Linux, FreeBSD, and macOS.
- The one exception to this is Windows, where though native Python distributions exist, there is as yet no native Ansible build.

Installing Ansible on Linux and FreeBSD

- The release cycle for Ansible is usually about four months, and during this short release cycle, there are normally many changes, from minor bug fixes to major ones, to new features and even sometimes fundamental changes to the language.
- The simplest way to not only get up and running with Ansible but to keep yourself up to date is to use the native packages built for your operating system where they are available.

Installing Ansible on Linux and FreeBSD

- **Installing Ansible on Ubuntu:** To install the latest version of the Ansible control machine on Ubuntu, the apt packaging tool makes it easy using the following commands:

\$ sudo apt-get update

```
$ sudo apt-get install software-properties-common
```

```
$ sudo apt-add-repository --yes --update
```

ppa:ansible/ansible

```
$ sudo apt-get install ansible
```



Installing Ansible on Linux and FreeBSD

- **Installing Ansible on Debian:** You should add the following line into your /etc/apt/sources.list file:

```
deb http://ppa.launchpad.net/ansible/ansible/ubuntu  
trusty main
```

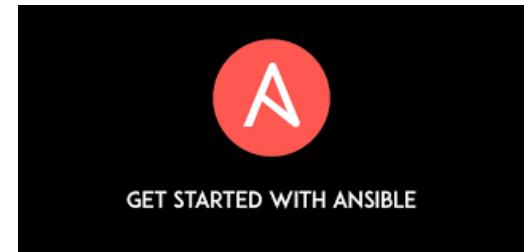


A screenshot of a terminal window displaying the Ansible command-line interface. The output shows various configuration and status messages related to Ansible's internal operations, such as loading hosts, connecting to hosts, and managing inventory files.

Installing Ansible on Linux and FreeBSD

- Once previous is done, you can install Ansible on Debian as follows:

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 93C4A3FD7BB9C367  
$ sudo apt-get update  
$ sudo apt-get install ansible
```



Installing Ansible on Linux and FreeBSD

- **Installing Ansible on Gentoo:** To install the latest version of the Ansible control machine on Gentoo, the portage package manager makes it easy with the following commands:

```
$ echo 'app-admin/ansible' >> /etc/portage/package.accept_keywords  
$ emerge -av app-admin/ansible
```



Installing Ansible on Linux and FreeBSD

- **Installing Ansible on FreeBSD:** To install the latest version of the Ansible control machine on FreeBSD, the PKG manager makes it easy with the following commands:

```
$ sudo pkg install py36-ansible
```

```
$ sudo make -C /usr/ports/sysutils/ansible install
```

- **Installing Ansible on Fedora:** To install the latest version of the Ansible control machine on Fedora, the dnf package manager makes it easy with the following commands:

```
$ sudo dnf -y install ansible
```

Installing Ansible on Linux and FreeBSD

- **Installing Ansible on CentOS:** To install the latest version of the Ansible control machine on CentOS or RHEL, the yum package manager makes it easy with the following commands:

```
$ sudo yum install epel-release  
$ sudo yum -y install ansible
```



Installing Ansible on Linux and FreeBSD

- If it's not, you need to enable the relevant repository with the following commands:

```
sudo subscription-manager repos --enable rhel-7-
server-ansible-2.9-rpms
```

- **Installing Ansible on Arch Linux:** To install the latest version of the Ansible control machine on Arch Linux, the pacman package manager makes it easy with the following commands:

```
$ pacman -S ansible
```

Installing Ansible on Linux and FreeBSD

- Once you have installed Ansible on the specific Linux distribution that you use, you can begin to explore.
- Let's start with a simple example—when you run the ansible command, you will see output similar to the following:

Refer to the file 1_1.txt



Installing Ansible on Linux and FreeBSD

- To do this, you will need to clone source code from the GitHub repository and build the RPM package as follows:

```
$ git clone https://github.com/ansible/ansible.git  
$ cd ./ansible  
$ make rpm  
$ sudo rpm -Uvh ./rpm-build/ansible-* .noarch.rpm
```

Installing Ansible on macOS

- **Installing Homebrew:** Normally the two commands shown here are all that is required to install Homebrew on macOS:

```
$ xcode-select --install  
$ ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Installing Ansible on macOS

- If you have already installed the Xcode command-line tools for another purpose, you might see the following error message:

xcode-select: error: command line tools are already installed, use "Software Update" to update



Installing Ansible on macOS

```
$ brew doctor
```

Please note that these warnings are just used to help the Homebrew maintainers with debugging if you file an issue. If everything you use Homebrew for is working fine: please don't worry or file an issue; just ignore this. Thanks!

Warning: Homebrew's sbin was not found in your PATH but you have installed formulae that put executables in /usr/local/sbin.

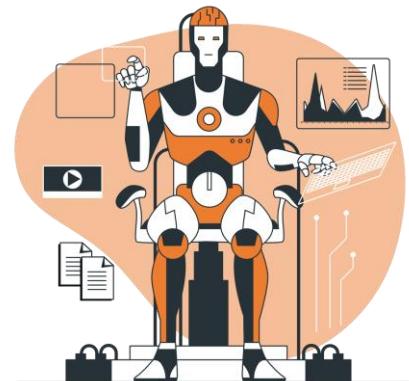
Consider setting the PATH for example like so

```
echo 'export PATH="/usr/local/sbin:$PATH"' >> ~/.bash_profile
```

Installing Ansible on macOS

- **Installing the Python package manager (pip):** If you don't wish to use Homebrew to install Ansible, you can instead install pip using with the following simple commands:

```
$ sudo easy_install pip
```



Installing Ansible on macOS

- Also check that your Python version is at least 2.7, as Ansible won't run on anything older (this should be the case with almost all modern installations of macOS):

```
$ python --version  
Python 2.7.16
```



Installing Ansible on macOS

- **Installing Ansible via Homebrew:** To install Ansible via Homebrew, run the following command:

```
$ brew install ansible
```

- **Installing Ansible via the Python package manager (pip):** To install Ansible via pip, use the following command:

```
$ sudo pip install ansible
```

Installing Ansible on macOS

- You might be interested in running the latest development version of Ansible direct from GitHub, and if so, you can achieve this by running the following command:

```
$ pip install git+https://github.com/ansible/ansible.git@devel
```

Installing Ansible on macOS

- Now that you have installed Ansible using your preferred method, you can run the ansible command as before, and if all has gone according to plan, you will see output similar to the following:

Refer to the file 1_2.txt



Installing Ansible on macOS

- If you are running macOS 10.9, you may experience issues when installing Ansible using pip.
- The following is a workaround that should resolve the issue:

```
$ sudo CFLAGS=-Qunused-arguments CPPFLAGS=-Qunused-arguments pip install ansible
```

Installing Ansible on macOS

- If you want to update your Ansible version, pip makes it easy via the following command:

```
$ sudo pip install ansible –upgrade
```

- Similarly, you can upgrade it using the brew command if that was your install method:

```
$ brew upgrade ansible
```



Configuring Windows hosts for Ansible

Let's look at what Ansible is capable of when automating Windows hosts:

- Gather facts about remote hosts.
- Install and uninstall Windows features.
- Manage and query Windows services.
- Manage user accounts and a list of users.
- Manage packages using Chocolatey (a software repository and accompanying management tool for Windows) etc.



Configuring Windows hosts for Ansible

- Ansible allows you to automate tasks on Windows machines by connecting with either a local user or a domain user.
- You can run actions as an administrator using the Windows runas support, just as with the sudo command on Linux distributions.

Configuring Windows hosts for Ansible

- With regard to prerequisites, you have to make sure PowerShell 3.0 and .NET Framework 4.0 are installed on Windows machines.
- If you're still using the older version of PowerShell or .NET Framework, you need to upgrade them.
- You are free to perform this manually, or the following PowerShell script can handle it automatically for you:

Refer to the file 1_3.txt

Configuring Windows hosts for Ansible

- When PowerShell has been upgraded to at least version 3.0, the next step will be to configure the WinRM service so that Ansible can connect to it.
- WinRM service configuration defines how Ansible can interface with the Windows hosts, including the listener port and protocol.
- you can create a listener with a specific configuration by running the following PowerShell commands:

Refer to the file 1_4.txt

Configuring Windows hosts for Ansible

- If you are running in PowerShell v3.0, you might face an issue with the WinRM service that limits the amount of memory available.
- This is a known bug, and a hotfix is available to resolve it.
- An example process (written in PowerShell) to apply this hotfix is given here:

Refer to the file 1_5.txt



Configuring Windows hosts for Ansible

- Configuring the WinRM listeners can be a complex task, so it is important to be able to check the results of your configuration process.
- The following command (which can be run from Command Prompt) will display the current WinRM listener configuration:

`winrm enumerate winrm/config/Listener`

- If all goes well, you should have output similar to this:
Refer to the file 1_6.txt

Configuring Windows hosts for Ansible

- If you need to debug any connection issues after setting up your WinRM listener, you may find the following commands valuable as they perform WinRM-based connections between Windows hosts without Ansible
- You can use them to distinguish whether an issue you might be experiencing is related to your Ansible host or whether there is an issue with the WinRM listener itself:
Refer to the file 1_7.txt

Configuring Windows hosts for Ansible

- First, you will need to install the winrm Python module, which, depending on your control hosts' configuration, may or may not have been installed before.
- The installation method will vary from one operating system to another, but it can generally be installed on most platforms with pip as follows:

```
$ pip install winrm
```

Configuring Windows hosts for Ansible

- The following example is just for reference:

```
[windows]
```

```
192.168.1.52
```

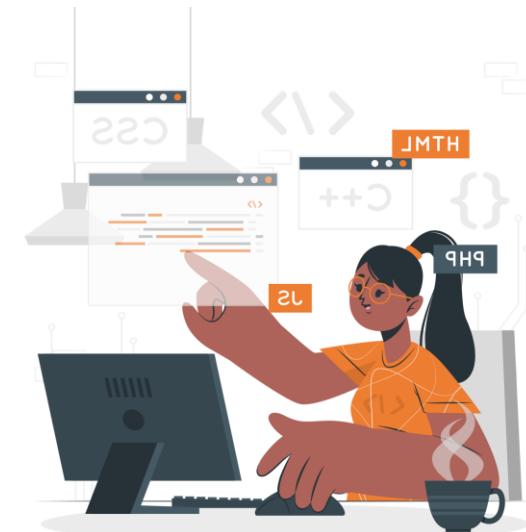
```
[windows:vars]
```

```
ansible_user=administrator
```

```
ansible_password=password
```

```
ansible_connection=winrm
```

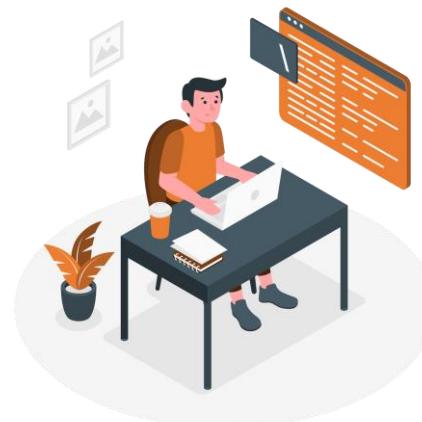
```
ansible_winrm_server_cert_validation=ignore
```



Configuring Windows hosts for Ansible

- Finally, you should be able to run the Ansible ping module to perform an end-to-end connectivity test with a command like the following (adjust for your inventory):

```
$ ansible -i inventory -m ping windows  
192.168.1.52 | SUCCESS => {  
    "changed": false,  
    "ping": "pong"  
}
```



Understanding your Ansible installation

- By this stage in this lesson, regardless of your operating system choice for your Ansible control machine, you should have a working installation of Ansible with which to begin exploring the world of automation.
- In this section, we will carry out a practical exploration of the fundamentals of Ansible to help you to understand how to work with it.

Understanding how Ansible connects to hosts

- With exception of Windows hosts (as discussed at the end of the previous section), Ansible uses the SSH protocol to communicate with hosts.
- The reasons for this choice in the Ansible design are many, not least that just about every Linux/FreeBSD/macOS host has it built in, as do many network devices such as switches and routers.

Understanding how Ansible connects to hosts

- For now, let's focus on the INI formatted inventory. An example is shown here with four servers, each split into two groups.
- Ansible commands and playbooks can be run against an entire inventory (that is, all four servers), one or more groups (for example, web servers), or even down to a single server:

Refer to the file 1_8.txt

Understanding how Ansible connects to hosts

- Let's use this inventory file along with the Ansible ping module, which is used to test whether Ansible can successfully perform automation tasks on the inventory host in question.
- The following example assumes you have installed the inventory in the default location, which is normally /etc/ansible/hosts.
- When you run the following ansible command, you see a similar output to this:

Refer to the file 1_9.txt

Verifying the Ansible installation

- To ensure Ansible can authenticate with your private key, you could make use of ssh-agent—the commands show a simple example of how to start ssh-agent and add your private key to it.
- Naturally, you should replace the path with that to your own private key:

```
$ ssh-agent bash  
$ ssh-add ~/.ssh/id_rsa
```

Verifying the Ansible installation

- As we discussed in the previous section, we must also define an inventory for Ansible to run against.
- Another simple example is shown here:

[frontends]

frt01.example.com
frt02.example.com



Verifying the Ansible installation

- **Ping hosts:** You can perform an Ansible "ping" on your inventory hosts using the following command:

```
$ ansible frontends -i hosts -m ping
```

- **Display gathered facts:** You can display gathered facts about your inventory hosts using the following command:

```
$ ansible frontends -i hosts -m setup | less
```

- **Filter gathered facts:** You can filter gathered facts using the following command:

```
$ ansible frontends -i hosts -m setup -a "filter=ansible_distribution"
```

Verifying the Ansible installation

- For every ad hoc command you run, you will get a response in JSON format—the following example output results from running the ping module successfully:

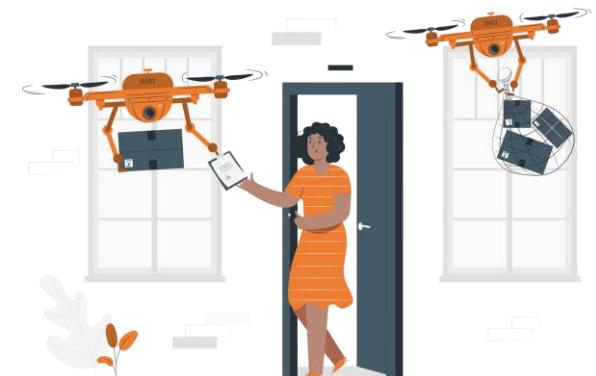
```
$ ansible frontends -m ping  
frontend01.example.com | SUCCESS => {  
    "changed": false,  
    "ping": "pong"  
}  
frontend02.example.com | SUCCESS => {  
    "changed": false,  
    "ping": "pong"  
}
```



Verifying the Ansible installation

- The following is an example of the filtered facts from a macOS-based host:

```
$ ansible frontend01.example.com -m setup -a  
"filter=ansible_distribution"  
frontend01.example.com | SUCCESS => {  
    ansible_facts": {  
        "ansible_distribution": "macOS",  
        "ansible_distribution_major_version": "10",  
        "ansible_distribution_release": "18.5.0",  
        "ansible_distribution_version": "10.14.4"  
    },  
    "changed": false
```



Verifying the Ansible installation

- Copy a file from the Ansible control host to all hosts in the frontends group with the following command:

```
$ ansible frontends -m copy -a "src=/etc/yum.conf  
dest=/tmp/yum.conf"
```

- Create a new directory on all hosts in the frontends inventory group, and create it with specific ownership and permissions:

```
$ ansible frontends -m file -a "dest=/path/user1/new  
mode=777 owner=user1 group=user1 state=directory"
```

Verifying the Ansible installation

- Delete a specific directory from all hosts in the frontends group with the following command:

```
$ ansible frontends -m file -a "dest=/path/user1/new state=absent"
```

- Install the httpd package with yum if it is not already present—if it is present, do not update it.
- Again, this applies to all hosts in the frontends inventory group:

```
$ ansible frontends -m yum -a "name=httpd state=present"
```

Verifying the Ansible installation

- The following command is similar to the previous one, except that changing state=present to state=latest causes Ansible to install the (latest version of the) package if it is not present, and update it to the latest version if it is present:

```
$ ansible frontends -m yum -a "name=demo-tomcat-1 state=latest"
```

- Display all facts about all the hosts in your inventory (warning—this will produce a lot of JSON!):

```
$ ansible all -m setup
```

Managed node requirements

- To install Python using yum (on older releases of Fedora and CentOS/RHEL 7 and below), use the following:

```
$ sudo yum -y install python
```

- On RHEL and CentOS version 8 and newer versions of Fedora, you would use the dnf package manager instead:

```
$ sudo dnf install python
```

Managed node requirements

- You might also elect to install a specific version to suit your needs, as in this example:

```
$ sudo dnf install python37
```

- On Debian and Ubuntu systems, you would use the apt package manager to install Python, again specifying a version if required (the example given here is to install Python 3.6 and would work on Ubuntu 18.04):

```
$ sudo apt-get update
```

```
$ sudo apt-get install python3.6
```



Managed node requirements

- The following are some examples of tasks in an Ansible playbook that you might use to bootstrap a managed node and prepare it for Ansible management:

Refer to the file 1_10.txt



Running from source versus pre-built RPMs

- You must clone the sources from the git repository first, and then change to the directory containing the checked-out code:

```
$ git clone https://github.com/ansible/ansible.git --  
recursive  
$ cd ./ansible
```



Running from source versus pre-built RPMs

- For example, if you are running the venerable Bash shell, you would set up your environment with the following command:

```
$ source ./hacking/env-setup
```

- Conversely, if you are running the Fish shell, you would set up your environment as follows:

```
$ source ./hacking/env-setup.fish
```

Running from source versus pre-built RPMs

- Once you have set up your environment, you must install the pip Python package manager, and then use this to install all of the required Python packages (note: you can skip the first command if you already have pip on your system):

```
$ sudo easy_install pip  
$ sudo pip install -r ./requirements.txt
```



Running from source versus pre-built RPMs

- When you run the env-setup script, Ansible runs from the source code checkout, and the default inventory file is /etc/ansible/hosts; however, you can optionally specify an inventory file wherever you want on your machine.
- The following command provides an example of how you might do this, but obviously, your filename and contents are almost certainly going to vary:

```
$ echo "ap1.example.com" > ~/my_ansible_inventory  
$ export ANSIBLE_INVENTORY=~/my_ansible_inventory
```

Running from source versus pre-built RPMs

- For example, if you set up your inventory as in the preceding code and clone the Ansible source into your home directory, you could run the ad hoc ping command that we are now familiar with, as follows:

```
$ ~/ansible/bin/ansible all -m ping  
ap1.example.com | SUCCESS => {  
    "changed": false,  
    "ping": "pong"  
}
```

Running from source versus pre-built RPMs

- When the time comes to update it, you don't need to clone a new copy; you can simply update your existing working copy using the following commands (again, assuming that you initially cloned the source tree into your home directory):

```
$ git pull --rebase
```

```
$ git submodule update --init --recursive
```

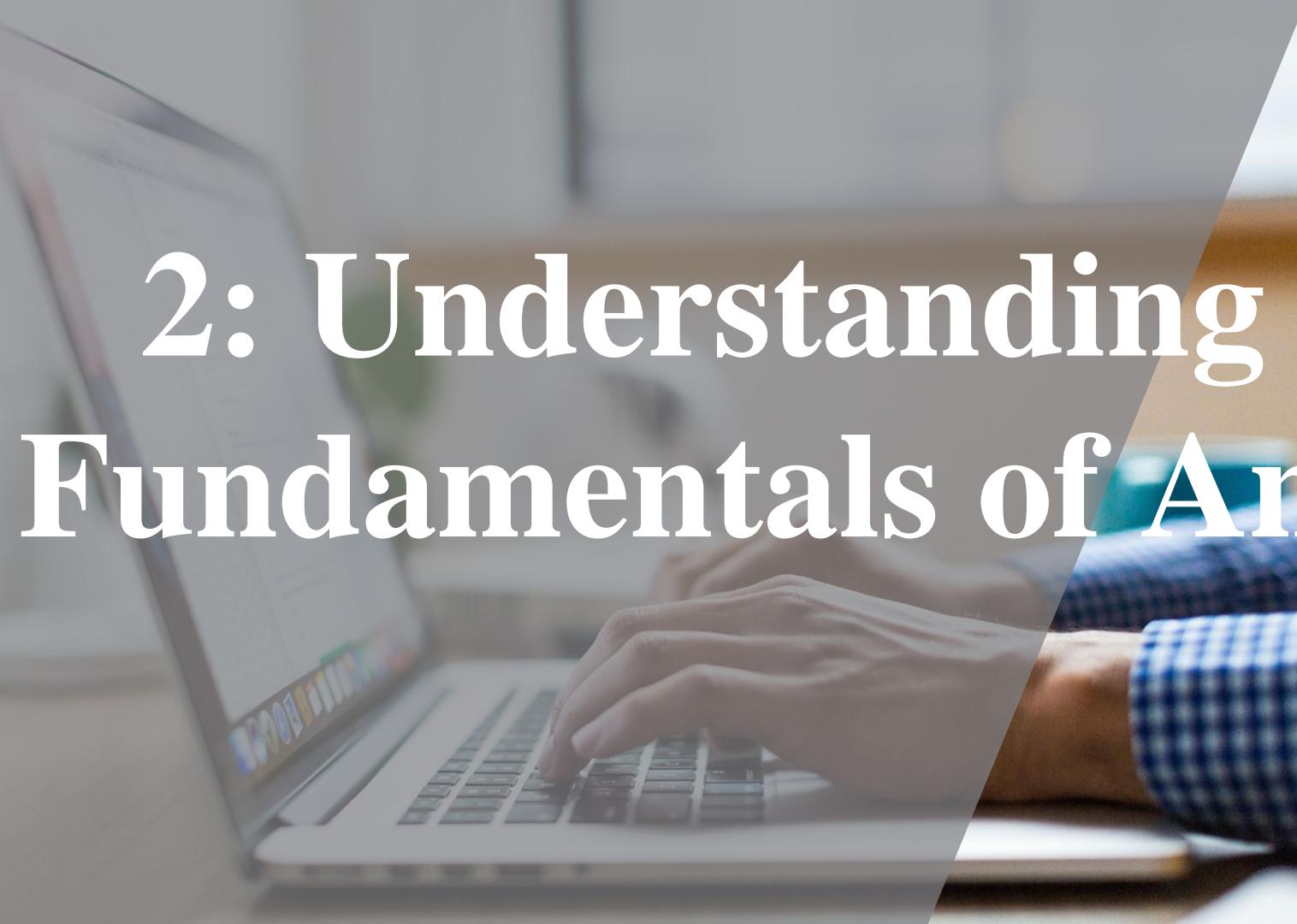


Summary

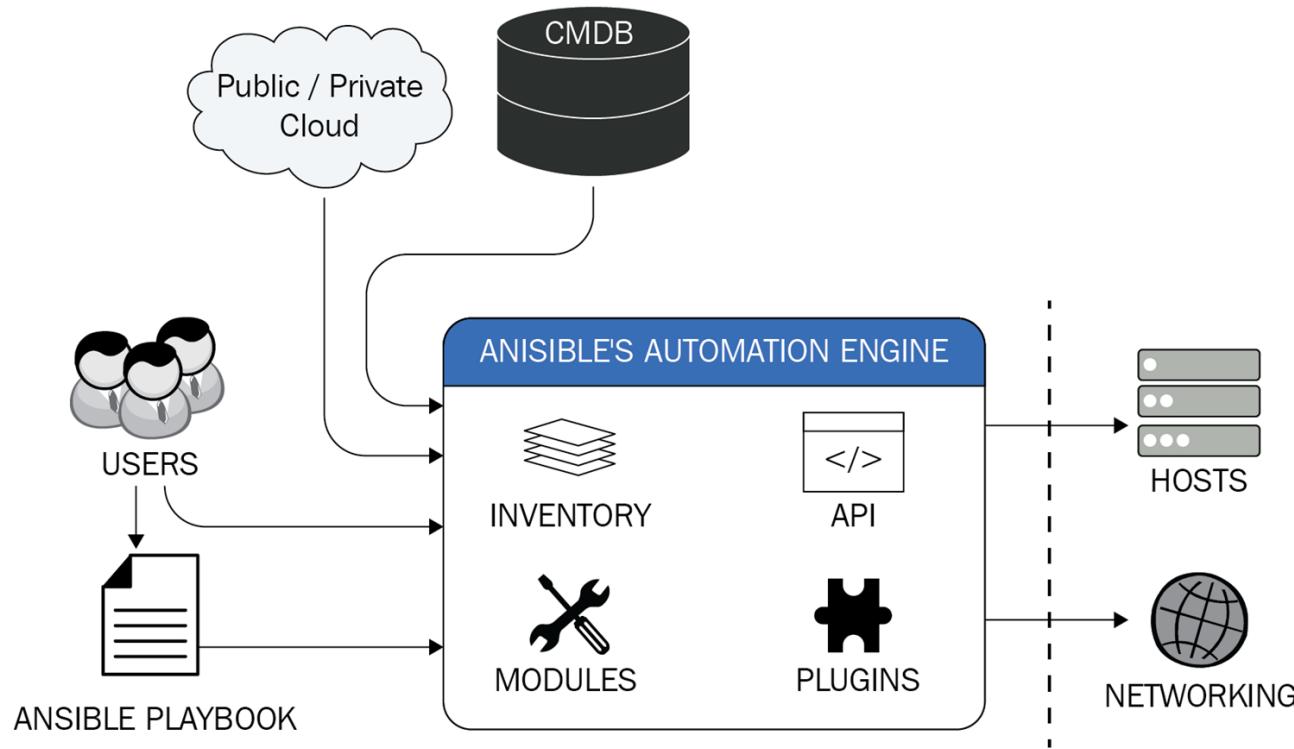
- Ansible is a powerful and versatile yet simple automation tool, of which the key benefits are its agentless architecture and its simple installation process.
- Ansible was designed to get you from zero to automation rapidly and with minimal effort, and we have demonstrated the simplicity with which you can get up and running with Ansible in this lesson.

"Complete Lab 1"

2: Understanding the Fundamentals of Ansible



Understanding the Fundamentals of Ansible



Understanding the Fundamentals of Ansible

In this lesson, we will cover the following topics:

- Getting familiar with the Ansible framework
- Exploring the configuration file
- Command-line arguments
- Defining variables
- Understanding Jinja2 filters

Technical requirements

- This lesson assumes that you have successfully installed the latest version of Ansible onto a Linux node, as discussed in lesson 1, Getting Started with Ansible.
- It also assumes that you have at least one other Linux host to test automation code on; the more hosts you have available, the more you will be able to develop the examples in this lesson and learn about Ansible.

Getting familiar with the Ansible framework

- In order to run Ansible's ad hoc commands via an SSH connection from your Ansible control machine to multiple remote hosts, you need to ensure you have the latest Ansible version installed on the control host.
- Use the following command to confirm the latest Ansible version:

Refer to the file 2_1.txt

Getting familiar with the Ansible framework

- You can use a simple, manual SSH connection on each of your remote hosts to test the connectivity, as Ansible will make use of SSH during all remote Linux-based automation tasks:

```
$ ssh <username>@frontend.example.com
```

The authenticity of host 'frontend.example.com (192.168.1.52)'
can't be established.

ED25519 key fingerprint is

SHA256:hU+saFERGFDERW453tasdFPAkpVws.

Are you sure you want to continue connecting (yes/no)? yes

password:<Input_Your_Password>

Getting familiar with the Ansible framework

- Add some example hosts to your inventory—these must be the IP addresses or hostnames of real machines for Ansible to test against.
- The following are examples from my network, but you need to substitute these for your own devices. Add one hostname (or IP address) per line:

frontend.example.com

backend1.example.com

backend2.example.com

Getting familiar with the Ansible framework

- Whatever format you choose for your inventory addresses, you should be able to successfully ping each host.
- See the following output as an example:

```
$ ping frontend.example.com
PING frontend.example.com (192.168.1.52): 56 data bytes
64 bytes from 192.168.1.52: icmp_seq=0 ttl=64 time=0.040 ms
64 bytes from 192.168.1.52: icmp_seq=1 ttl=64 time=0.115 ms
64 bytes from 192.168.1.52: icmp_seq=2 ttl=64 time=0.097 ms
64 bytes from 192.168.1.52: icmp_seq=3 ttl=64 time=0.130 ms
```

Getting familiar with the Ansible framework

- To make the automation process seamless, we'll generate an SSH authentication key pair so that we don't have to type in a password every time we want to run a playbook.
- If you do not already have an SSH key pair, you can generate one using the following command:

```
$ ssh-keygen
```

Getting familiar with the Ansible framework

- When you run the ssh-keygen tool, you will see an output similar to the following.
- Note that you should leave the passphrase variable blank when prompted; otherwise, you will need to enter a passphrase every time you want to run an Ansible task, which removes the convenience of authenticating with SSH keys:

Refer to the file 2_2.txt

Getting familiar with the Ansible framework

- Start ssh-agent and add your new authentication key, as follows (note that you will need to do this for every shell that you open):

```
$ ssh-agent bash  
$ ssh-add ~/.ssh/id_rsa
```

Getting familiar with the Ansible framework

- Before you can perform key-based authentication with your target hosts, you need to apply the public key from the key pair you just generated to each host.
- You can copy the key to each host, in turn, using the following command:

Refer to the file 2_3.txt

Getting familiar with the Ansible framework

- You will find that you are not prompted for a password at any point as the SSH connections to all the hosts in your inventory are authenticated with your SSH key pair.
- So, you should see an output similar to the following:

Refer to the file 2_4.txt

Getting familiar with the Ansible framework

- For example, assume that a certain host (such as backend2.example.com) can't be connected to and you receive an error similar to the following:

backend2.example.com | FAILED => SSH encountered an unknown error during the connection. We recommend you re-run the command using -vvvv, which will enable SSH debugging output to help diagnose the issue

Breaking down the Ansible components

- Let's create an example one, ideally with multiple hosts in it—this could be the same as the one you created in the previous section.
- As discussed in that section, you should populate the inventory with the hostnames or IP addresses of the hosts that you can reach from the control host itself:

remote1.example.com

remote2.example.com

remote3.example.com

Breaking down the Ansible components

- Specify the play name and inventory hosts to run your tasks against at the very top of your playbook.
- Also, note the use of ---, which denotes the beginning of a YAML file (Ansible playbooks that are written in YAML):

```
- name: My first Ansible playbook
hosts: all
```

Breaking down the Ansible components

- After this, we will tell Ansible that we want to perform all the tasks in this playbook as a superuser (usually root).
- We do this with the following statement (to aid your memory, think of become as shorthand for become superuser):

become: yes

Breaking down the Ansible components

- In this example, it restarts the web server, but only if it changes, preventing unnecessary restarts if the playbook is run several times and there are no updates for Apache.
- The following code performs these functions exactly and should form the basis of your first playbook:

Refer to the file 2_5.txt

Breaking down the Ansible components

- Congratulations, you now have your very first Ansible playbook! If you run this now, you should see it iterate through all the hosts in your inventory, as well as on each update in the Apache package, and then restart the service where the package was updated.
- Your output should look something as follows:

Refer to the file 2_6.txt

Breaking down the Ansible components

- If we run the playbook a second time, we know that it is very unlikely that the Apache package will need upgrading again.
- Notice how the playbook output differs this time:

Refer to the file 2_7.txt

Learning the YAML syntax

- Lists are an important construct in the YAML language—in fact, although it might not be obvious, the tasks: block of the playbook is actually a YAML list. A list in YAML lists all of its items at the same indentation level, with each line starting with -.
- For example, we updated the httpd package from the preceding playbook using the following code:

```
- name: Update the latest of an Apache Web Server
  yum:
    name: httpd
    state: latest
```

Learning the YAML syntax

- However, we could have specified a list of packages to be upgraded as follows:

```
- name: Update the latest of an Apache Web Server
  yum:
    name:
      - httpd
      - mod_ssl
    state: latest
```

Learning the YAML syntax

- Dictionaries are another important concept in YAML—they are represented by a key: value format, as we have already extensively seen, but all of the items in the dictionary are indented by one more level.
- This is easiest explained by an example, so consider the following code from our example playbook:

`service:`

```
  name: httpd  
  state: restarted
```

Learning the YAML syntax

- As you become more advanced at playbook design (we will see examples of this later on in this course), you may very well start to produce quite complicated variable structures that you will put into their own separate files to keep your playbook code readable.
- The following is an example of a variables file that provides the details of two employees of a company:

Refer to the file 2_8.txt



Learning the YAML syntax

- What if you actually need to add a block of text (for example, to a variable)? In this case, you can use a literal block scalar, |, to write multiple lines and YAML will faithfully preserve the new lines, carriage returns, and all the whitespace that follows each line (note, however, that the indentation at the beginning of each line is part of the YAML syntax):

Specialty: |

Agile methodology

Cloud-native app development practices

Advanced enterprise DevOps practices

Learning the YAML syntax

- if we were to get Ansible to print the preceding contents to the screen, it would display as follows (note that the preceding two spaces have gone—they were interpreted correctly as part of the YAML language and not printed):

Agile methodology

Cloud-native app development practices

Advanced enterprise DevOps practices

Learning the YAML syntax

- This is useful for very long strings that you want to print on a single line, but also want to wrap across multiple lines in your code for the purpose of readability. Take the following variation on our example:

Specialty: >

Agile methodology

Cloud-native app development practices

Advanced enterprise DevOps practices

Learning the YAML syntax

- Now, if we were to print this, we would see the following:

Agile methodology
Cloud-native app development
practices
Advanced enterprise DevOps practices

Learning the YAML syntax

- One additional example is provided in the following code block of a variables file for you to consider, which shows the various examples we have covered all in one place:

Refer to the file 2_9.txt

Learning the YAML syntax

- You can also express both dictionaries and lists in an abbreviated form, known as flow collections.
- The following example shows the same data structure as our original employee's variable file:

```
employees: [{"fullname": "Daniel Oh", "level": "Expert", "name": "daniel", "role": "DevOps Evangelist", "skills": ["Kubernetes", "Microservices", "Ansible", "Linux Container"]}, {"fullname": "Michael Smiths", "level": "Advanced", "name": "michael", "role": "Enterprise Architect", "skills": ["Cloud", "Middleware", "Windows", "Storage"]}]]
```

Learning the YAML syntax

- YAML tries to make assumptions about variable types based on the data they contain, so if you want assign 1.0 to a variable, YAML will assume it is a floating-point number.
- If you need to express it as a string (perhaps because it is a version number), you need to put quotation marks around it, which causes the YAML parser to interpret it as a string instead, such as in the following example:

`version: "2.0"`

Organizing your automation code

- Our hypothetical example will contain a frontend server and application servers for a fictional application, located in two different geographic locations.
- Our inventory file will be called production-inventory and the example contents are as follows:

Refer to the file 2_10.txt

Organizing your automation code

- Create a playbook to run a connection test on a specific host group, such as frontends_na_zone.
- Put the following contents into the playbook:

```
- hosts: frontends_na_zone
  remote_user: danieloh
  tasks:
    - name: simple connection test
      ping:
```

Organizing your automation code

- Now, try running this playbook against the hosts (note that we have configured it to connect to a remote user on the inventory system, called danieloh, so you will either need to create this user and set up the appropriate SSH keys or change the user in the remote_user line of your playbook).
- When you run the playbook after setting up the authentication, you should see an output similar to the following:

Refer to the file 2_11.txt

Organizing your automation code

- Specify that this playbook is run against this host group from the appservers_emea_zone inventory.
- Add the following contents to the playbook:

```
- hosts: appservers_emea_zone
  remote_user: danieloh
  tasks:
    - name: simple connection test
      ping:
```

Organizing your automation code

- As before, you need to ensure you can access these servers, so either create the danieloh user and set up authentication to that account or change the remote_user line in the example playbook.
- Once you have done this, you should be able to run the playbook and you will see an output similar to the following:

Refer to the file 2_12.txt

Organizing your automation code

- Fortunately, we can do exactly that by taking advantage of the `import_playbook` directive in a top-level playbook that we will call `site.yml`:

- `import_playbook: frontend-na.yml`
- `import_playbook: appserver-emea.yml`



Organizing your automation code

- Now, when you run this single playbook using the (by now, familiar) ansible-playbook command, you will see that the effect is the same as if we had actually run both playbooks back to back.
- In this way, even before we explore the concept of roles, you can see that Ansible supports splitting up your code into manageable chunks without needing to run each chunk manually:

Refer to the file 2_13.txt

Exploring the configuration file

The first instance of the file is the configuration it will use; all of the others are ignored, even if they are present:

- `ANSIBLE_CONFIG`: The file location specified by the value of this environment variable, if set
- `ansible.cfg`: In the current working directory
- `~/.ansible.cfg`: In the home directory of the user
- `/etc/ansible/ansible.cfg`: The central configuration that we previously mentioned

Exploring the configuration file

- Obviously, if you don't have Ansible installed, there's little point in exploring its configuration, so let's just check whether you have Ansible installed and working by issuing a command such as the following (the output shown is from the latest version of Ansible at the time of writing, installed on macOS with Homebrew):

Refer to the file 2_14.txt

Exploring the configuration file

- Other valuable information, including the default configuration values and a description of the configuration, is given (see the default and description fields, respectively).
- All of the information is sourced from lib/constants.py. Run the following command to explore the output:

```
$ ansible-config list
```

Exploring the configuration file

- The following is an example of the kind of output you will see.
 - There are, of course, many pages to it, but a snippet is shown here as an example:

Refer to the file 2_15.txt



Exploring the configuration file

- If you want to see a straightforward display of all the possible configuration parameters, along with their current values (regardless of whether they are configured from environment variables or a configuration file in one of the previously listed locations), you can run the following command:

```
$ ansible-config dump
```

Exploring the configuration file

- The output shows all the configuration parameters (in an environment variable format), along with the current settings.
- If the parameter is configured with its default value, you are told so (see the (default) element after each parameter name):

Refer to the file 2_16.txt

Exploring the configuration file

- Let's see the effect on this output by editing one of the configuration parameters.
- Let's do this by setting an environment variable, as follows (this command has been tested in the bash shell, but may differ for other shells):

```
$ export ANSIBLE_FORCE_COLOR=True
```

Exploring the configuration file

- Now, let's re-run the ansible-config command, but this time get it to tell us only the parameters that have been changed from their default values:

```
$ ansible-config dump --only-change  
ANSIBLE_FORCE_COLOR(env:  
ANSIBLE_FORCE_COLOR) = True
```

Exploring the configuration file

- You only need to place the parameters you wish to change from their defaults in your configuration file, so if you wanted to create a simple configuration to change the location of your default inventory file, it might look as follows:

```
# Set my configuration variables  
[defaults]  
inventory = /Users/danieloh/ansible/hosts ; Here is the  
path of the inventory file
```

Command-line arguments

- We are already very familiar with one of these arguments, the --version switch, which we use to confirm that Ansible is installed (and which version is installed):

Refer to the file 2_17.txt

Command-line arguments

- You can view all the options and arguments when you execute the ansible command line.
- Use the following command:

```
$ ansible --help
```

Command-line arguments

- You will see a great deal of helpful output when you run the preceding command; an example of this is shown in the following code block (you might want to pipe this into a pager, such as `less`, so that you can read it all easily):

Refer to the file `2_18.txt`

Command-line arguments

- We could explore this using the production-inventory file we used earlier in this lesson:

```
$ ansible -i production-inventory --list-host appservers_emea_zone
```

Command-line arguments

- Although perhaps a little contrived, this example is incredibly valuable when you start working with dynamic inventory files and you can no longer just cat your inventory file to the terminal to view the contents:

```
$ ansible -i production-inventory --list-host  
appservers_emea_zone  
hosts (2):  
appserver1-emea.example.com  
appserver2-emea.example.com
```

Command-line arguments

- You can use one of the command-line arguments for ansible-playbook to specify the private SSH key file, instead, as follows:

```
$ ansible-playbook -i production-inventory site.yml --  
private-key ~/keys/id_rsa
```

Command-line arguments

- Similarly, in the preceding section, we specified the `remote_user` variable for Ansible to connect with in the playbook.
- However, command-line arguments can also set this parameter for the playbook; so, rather than editing the `remote_user` line in the playbook, we could remove it altogether and instead have run it using the following command-line string:

```
$ ansible-playbook -i production-inventory site.yml --user danieloh
```

Understanding ad hoc commands

- As with every Ansible example, we need an inventory to run against.
- Let's reuse our production-inventory file from before:

Refer to the file 2_19.txt

Understanding ad hoc commands

- Run the following ad hoc command to retrieve the current date and time from all of the frontends_emea_zone servers:

```
$ ansible -i production-inventory frontends_emea_zone  
-a /usr/bin/date
```

Understanding ad hoc commands

- You will see that Ansible faithfully logs in to each machine in turn and runs the date command, returning the current date and time.
- Your output will look something as follows:

```
$ ansible -i production-inventory frontends_emea_zone -a  
/usr/bin/date  
frontend1-emea.example.com | CHANGED | rc=0 >>  
Sun 5 Apr 18:55:30 BST 2020  
frontend2-emea.example.com | CHANGED | rc=0 >>  
Sun 5 Apr 18:55:30 BST 2020
```

Understanding ad hoc commands

- This command is run with the user account you are logged in to when the command is run.
- You can use a command-line argument (discussed in the previous section) to run as a different user:

Refer to the file 2_20.txt

Understanding ad hoc commands

- We can fix this by adding the --become command-line argument, which tells Ansible to become root on the remote systems:

```
$ ansible -i production-inventory frontends_emea_zone -a  
/usr/sbin/pvs -u danieloh --become  
frontend2-emea.example.com | FAILED | rc=-1 >>  
Missing sudo password  
frontend1-emea.example.com | FAILED | rc=-1 >>  
Missing sudo password
```

Understanding ad hoc commands

- We can see that the command still fails because although danieloh is in /etc/sudoers, it is not allowed to run commands as root without entering a sudo password.
- Luckily, there's a switch to get Ansible to prompt us for this at run time, meaning we don't need to edit our /etc/sudoers file:

Refer to the file 2_21.txt

Understanding ad hoc commands

- By default, if you don't specify a module using the -m command-line argument, Ansible assumes you want to use the command module (see https://docs.ansible.com/ansible/latest/modules/command_module.html).
- If you wish to use a specific module, you can add the -m switch to the command-line arguments and then specify the module arguments under the -a switch, as in the following example:

Refer to the file 2_22.txt

Understanding ad hoc commands

- For example, to execute sleep 2h asynchronously in the background with a timeout of 7,200 seconds (-B) and without polling (-P), use this command:

Refer to the file 2_23.txt

Understanding ad hoc commands

- Note that the output from this command gives a unique job ID for each task on each host.
- Let's now say that we want to see how this task proceeds on the second frontend server.
- Simply issue the following command from your Ansible control machine:

Refer to the file 2_24.txt

Understanding ad hoc commands

- We can see that the job has started but not finished.
- If we now kill the sleep command that we issued and check on the status again, we can see the following:

Refer to the file 2_25.txt



Defining variables

Let's get started with a practical look at defining variables in Ansible.

Variables in Ansible should have well-formatted names that adhere to the following rules:

- The name of the variable must only include letters, underscores, and numbers—spaces are not allowed.
- The name of the variable can only begin with a letter—they can contain numbers but cannot start with one.

Defining variables

For example, the following are good variable names:

- external_svc_port
- internal_hostname_ap1

Defining variables

- As discussed in the Learning the YAML syntax section, variables can be defined in a dictionary structure, such as the following.
- All values are declared in key-value pairs:

region:

 east: app

 west: frontend

 central: cache

Defining variables

- In order to retrieve a specific field from the preceding dictionary structure, you can use either one of the following notations:

```
# bracket notation  
region['east']
```

```
# dot notation  
region.east
```

Defining variables

- This dictionary structure is valuable when defining host variables; although earlier in this lesson we worked with a fictional set of employee records defined as an Ansible variables file, you could use this to specify something, such as some redis server parameters:

redis:

- server: cacheserver01.example.com
- port: 6379
- slaveof: cacheserver02.example.com

Defining variables

- For example, the following playbook code calls four hypothetical roles and each assigns a different value to the username variable for each one.
- These roles could be used to set up various administration roles on a server (or multiple servers), with each passing a changing list of usernames as people come and go from the company:

Refer to the file 2_26.txt

Defining variables

- To access variables from within a playbook, you simply place the variable name inside quoted pairs of curly braces.
- Consider the following example playbook (based loosely on our previous redis example):

Refer to the file 2_27.txt

Defining variables

- To access the contents of these variables, we use pairs of curly braces around them (as described previously) and the entire string is encased in quotation marks, which means we don't have to individually quote the variables.
- If you run the playbook on a local machine, you should see an output that looks as follows:

Refer to the file 2_28.txt

Understanding Jinja2 filters

- As Ansible is written in Python, it inherits an incredibly useful and powerful templating engine called Jinja2.
- We will look at the concept of templating later in this course, so for now, we will focus on one aspect of Jinja2 known as filtering.
- Jinja2 filters provide an incredibly powerful framework that you can use to manipulate and transform your data.

Understanding Jinja2 filters

- Our YAML data file might look as follows:

```
tags:  
  - key: job  
    value: developer  
  - key: language  
    value: java
```

Understanding Jinja2 filters

- Now, we could create a playbook to read this file and register the result, but how can we turn it into a variable structure that Ansible can understand and work with?
- Let's consider the following playbook:

Refer to the file 2_29.txt

Understanding Jinja2 filters

- The `from_yaml_all` filter parses the source document lines as YAML and then the `list` filter converts the parsed data into a valid Ansible list.
- If we run the playbook, we should see Ansible's representation of the data structure from within our original file:

Refer to the file `2_30.txt`

Understanding Jinja2 filters

- If this data structure was already stored in our playbook, we could take this one step further and use the items2dict filter to turn the list into true key: value pairs, removing the key and value items from the data structure.
- For example, consider this second playbook:

Refer to the file 2_31.txt

Understanding Jinja2 filters

- Now, if we run this, we can see that our data is converted into a nice neat set of key: value pairs:

Refer to the file 2_32.txt



Understanding Jinja2 filters

- Earlier in this section, we used the shell module to read a file and used register to store the result in a variable.
- This is perfectly fine, if a little inelegant. Jinja2 contains a series of lookup filters that, among other things, can read the contents of a given file.
- Let's examine the behavior of this following playbook::

Refer to the file 2_33.txt

Understanding Jinja2 filters

- When we run this, we can see that Ansible has captured the contents of the /etc/hosts file for us, without us needing to resort to the copy and shell modules as we did earlier:

Refer to the file 2_34.txt

Understanding Jinja2 filters

- The following are a handful of other examples that will give you an idea of the kinds of things that Jinja2 filters can achieve for you, from quoting strings to concatenating lists to obtaining useful path information for a file:

Refer to the file 2_35.txt



Summary

- Ansible is a very powerful and versatile automation engine that can be used for a wide variety of tasks.
- Understanding the basics of how to work it is of paramount importance, before addressing the more complex challenges of playbook creation and large-scale automation.

"Complete Lab 2"

3: Defining Your Inventory

A photograph of a person's hands typing on a silver laptop keyboard. The background is slightly blurred, showing a blue and white checkered shirt and a notebook. The overall image has a professional, tech-oriented feel.

Defining Your Inventory

In this lesson, we will cover the following topics:

- Creating an inventory file and adding hosts
- Generating a dynamic inventory file
- Special host management using patterns

Technical requirements

- This lesson assumes that you have set up your control host with Ansible, as detailed in lesson 1, Getting Started with Ansible, and you are using the most recent version available—the examples in this lesson were tested with Ansible 2.9.
- This lesson also assumes that you have at least one additional host to test against, and ideally this should be Linux-based.

Creating an inventory file and adding hosts

- You will have seen in the earlier lessons of this course, most Ansible commands use the -i flag to specify the location of the inventory file if not using the default.
- Hypothetically, this might look like the following example:

```
$ ansible -i /home/cloud-user/inventory all -m ping
```

Creating an inventory file and adding hosts

- Create an inventory file in /etc/ansible/my_inventory using the following INI-formatted code:

```
target1.example.com ansible_host=192.168.81.142 ansible_port=3333
```

```
target2.example.com ansible_port=3333 ansible_user=danieloh
```

```
target3.example.com ansible_host=192.168.81.143 ansible_port=5555
```

- Now, if you wanted to create exactly the same inventory as the preceding, but this time, format it as YAML, you would specify it as follows:

```
ungrouped:  
hosts:  
  target1.example.com:  
    ansible_host: 192.168.81.142  
    ansible_port: 3333  
  target2.example.com:  
    ansible_port: 3333  
    ansible_user: danieloh  
  target3.example.com:  
    ansible_host: 192.168.81.143  
    ansible_port: 5555
```

Creating an inventory file and adding hosts

- Now if you were to run the preceding inventory within Ansible, using a simple shell command, the result would appear as follows:

```
$ ansible -i /etc/ansible/my_inventory.yaml all -m shell -a 'echo hello-yaml' -f 5
target1.example.com | CHANGED | rc=0 >>
hello-yaml
target2.example.com | CHANGED | rc=0 >>
hello-yaml
target3.example.com | CHANGED | rc=0 >>
hello-yaml
```

Using host groups

Let's assume you have a simple three-tier web architecture, with multiple hosts in each tier for high availability and/or load balancing. The three tiers in this architecture might be the following:

- Frontend servers
- Application servers
- Database servers

Using host groups

- Let's, first of all, create the inventory for the three-tier frontend using the INI format.
- We will call this file hostsgroups.ini, and the contents of this file should look something like this:

Refer to the file 3_1.txt

Using host groups

- We could, of course, handle this case using facts gathered from each host as these will contain the operating system details.
- We could also create a new version of the inventory, as follows:

Refer to the file 3_2.txt



Using host groups

- The code listed next shows the YAML version of the preceding inventory—the two are identical as far as Ansible is concerned, but it is left to you to decide which format you prefer working with:

Refer to the file 3_3.txt



Using host groups

- When you want to work with any of the groups from the preceding inventory, you would simply reference it either in your playbook or on the command line.
- For example, in the last section we ran, we can use the following:

```
$ ansible -i /etc/ansible/my_inventory.yaml all -m shell -  
a 'echo hello-yaml' -f 5
```

Using host groups

- If we wanted to run the same command, but this time on just the centos group hosts from the previous YAML inventory, we would run this variation of the command:

```
$ ansible -i hostgroups-yml centos -m shell -a 'echo hello-yaml' -f 5  
app01.example.com | CHANGED | rc=0 >>  
hello-yaml  
app02.example.com | CHANGED | rc=0 >>  
hello-yaml  
dbms01.example.com | CHANGED | rc=0 >>  
hello-yaml  
dbms02.example.com | CHANGED | rc=0 >>  
hello-yaml
```



Using host groups

- Let's assume you have 100 app servers, all named sequentially, as follows:

[apps]

app01.example.com
app02.example.com

...

app99.example.com
app100.example.com

Using host groups

- Luckily, Ansible provides a quick shorthand notation to achieve this, and the following inventory snippet actually produces an inventory with the same 100 app servers that we could create manually:

```
[apps]
app[01:100].prod.com
```

Using host groups

- It is also possible to use alphabetic ranges as well as numeric ones—extending our example to add some cache servers, you might have the following:

[caches]

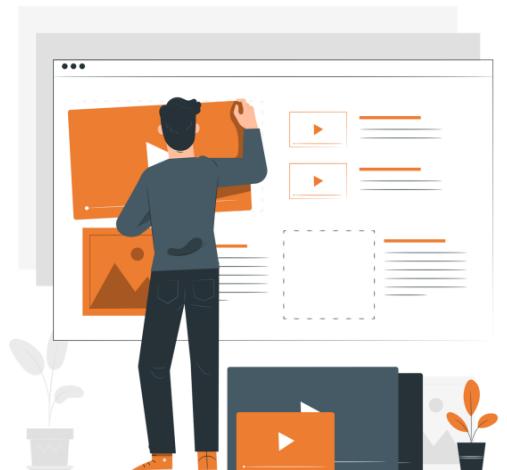
cache-[a:e].prod.com

Using host groups

- This is the same as manually creating the following:

[caches]

cache-a.prod.com
cache-b.prod.com
cache-c.prod.com
cache-d.prod.com
cache-e.prod.com



Adding host and group variables to your inventory

These are not special Ansible variables, but instead are variables entirely of our own choosing, which we will use later in the playbooks that run against this server.

Suppose that these variables are as follows:

- https_port, which defines the port that the frontend proxy should listen on
- lb_vip, which defines the FQDN of the load-balancer in front of the frontend servers

Adding host and group variables to your inventory

- We could simply add these to each of the hosts in the frontends part of our inventory file, just as we did before with the Ansible connection variables.
- In this case, a portion of our INI-formatted inventory might look like this:

```
[frontends]
```

```
frt01.example.com https_port=8443 lb_vip=lb.example.com  
frt02.example.com https_port=8443 lb_vip=lb.example.com
```

Adding host and group variables to your inventory

- If we run an ad hoc command against this inventory, we can see the contents of both of these variables:

```
$ ansible -i hostvars1-hostgroups-ini frontends -m debug -a  
"msg=\"Connecting to {{ lb_vip }}, listening on {{ https_port }}\""  
frt01.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8443"  
}  
frt02.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8443"  
}
```

Adding host and group variables to your inventory

- Luckily, you can assign variables to a host group as well as to hosts individually.
- If we edited the preceding inventory to achieve this, the frontends section would now look like this:

```
[frontends]
frt01.example.com
frt02.example.com
[frontends:vars]
https_port=8443
lb_vip=lb.example.com
```

Adding host and group variables to your inventory

- Notice how much more readable that is? Yet, if we run the same command as before against our newly organized inventory, we see that the result is the same:

```
$ ansible -i groupvars1-hostgroups-ini frontends -m debug -a  
"msg=\"Connecting to {{ lb_vip }}, listening on {{ https_port }}\""  
frt01.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8443"  
}  
frt02.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8443"  
}
```

Adding host and group variables to your inventory

- It is also worth noting that host variables override group variables, so if we need to change the connection port to 8444 on the frt01.example.com one, we could do this as follows:

```
[frontends]
```

```
frt01.example.com https_port=8444
```

```
frt02.example.com
```

```
[frontends:vars]
```

```
https_port=8443
```

```
lb_vip=lb.example.com
```

Adding host and group variables to your inventory

- Now if we run our ad hoc command again with the new inventory, we can see that we have overridden the variable on one host:

```
$ ansible -i hostvars2-hostgroups-ini frontends -m debug -a  
"msg=\"Connecting to {{ lb_vip }}, listening on {{ https_port }}\""  
frt01.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8444"  
}  
frt02.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8443"  
}
```

Adding host and group variables to your inventory

- Just for completeness, if we were to add the host variables we defined previously to our YAML version of the inventory, the frontends section would appear as follows (the rest of the inventory has been removed to save space):

```
frontends:
```

```
    hosts:
```

```
        frt01.example.com:
```

```
            https_port: 8444
```

```
        frt02.example.com:
```

```
    vars:
```

```
        https_port: 8443
```

```
        lb_vip: lb.example.com
```

Adding host and group variables to your inventory

- Running the same ad hoc command as before, you can see that the result is the same as for our INI-formatted inventory:

```
$ ansible -i hostvars2-hostgroups-yml frontends -m debug -a  
"msg=\"Connecting to {{ lb_vip }}, listening on {{ https_port }}\""  
frt01.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8444"  
}  
frt02.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8443"  
}
```

Adding host and group variables to your inventory

- Let's start by creating a new directory structure for this purpose:

```
$ mkdir vartree
```

```
$ cd vartree
```

- Now, under this directory, we'll create two more directories for the variables:

```
$ mkdir host_vars group_vars
```

Adding host and group variables to your inventory

- Now, under the `host_vars` directory, we'll create a file with the name of our host that needs the proxy setting, with `.yml` appended to it (that is, `frt01.example.com.yml`).
- This file should contain the following:

`https_port: 8444`

Adding host and group variables to your inventory

- Similarly, under the group_vars directory, create a YAML file named after the group to which we want to assign variables (that is, frontends.yml) with the following contents:

```
---
```

```
https_port: 8443
```

```
lb_vip: lb.example.com
```



Adding host and group variables to your inventory

- Finally, we will create our inventory file as before, except that it contains no variables:

```
loadbalancer.example.com
```

```
[frontends]
```

```
frt01.example.com
```

```
frt02.example.com
```

```
[apps]
```

```
app01.example.com
```

```
app02.example.com
```

```
[databases]
```

```
dbms01.example.com
```

```
dbms02.example.com
```



Adding host and group variables to your inventory

- Just for clarity, your final directory structure should look like this:

```
$ tree
```



2 directories, 3 files

Adding host and group variables to your inventory

- Now, let's try running our familiar ad hoc command and see what happens:

```
$ ansible -i inventory frontends -m debug -a  
"msg=\"Connecting to {{ lb_vip }}, listening on {{ https_port }}\""  
frt02.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8443"  
}  
frt01.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8444"  
}
```

Adding host and group variables to your inventory

- Let's recreate the directory structure but now with directories instead:

```
$ tree
```

```
.
├── group_vars
│   └── frontends
│       ├── https_port.yml
│       └── lb_vip.yml
└── host_vars
    └── frt01.example.com
        └── main.yml

```



Adding host and group variables to your inventory

- The files themselves are simply an adaptation of our previous ones:

```
$ cat host_vars/frt01.example.com/main.yml
```

```
---
```

```
https_port: 8444
```

```
$ cat group_vars/frontends/https_port.yml
```

```
---
```

```
https_port: 8443
```

```
$ cat group_vars/frontends/lb_vip.yml
```

```
---
```

```
lb_vip: lb.example.com
```



Adding host and group variables to your inventory

- Even with this more finely divided directory structure, the result of running the ad hoc command is still the same:

```
$ ansible -i inventory frontends -m debug -a "msg=\"Connecting to  
{{ lb_vip }}, listening on {{ https_port }}\""  
frt01.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8444"  
}  
frt02.example.com | SUCCESS => {  
    "msg": "Connecting to lb.example.com, listening on 8443"  
}
```

Adding host and group variables to your inventory

- Consider our earlier inventory where we used child groups to differentiate between CentOS and Ubuntu hosts—if we add a variable with the same name to both the ubuntu child group and the frontends group (which is a child of the ubuntu group) as follows, what will the outcome be? The inventory would look like this:

Refer to the file 3_4.txt

Adding host and group variables to your inventory

- Now, let's run an ad hoc command to see what value of testvar is set:

```
$ ansible -i hostgroups-children-vars-ini ubuntu -m  
debug -a "var=testvar"  
frt01.example.com | SUCCESS => {  
    "testvar": "childgroup"  
}  
frt02.example.com | SUCCESS => {  
    "testvar": "childgroup"  
}
```

Generating a dynamic inventory file

- In these days of cloud computing and infrastructure-as-code, the hosts you may wish to automate could change on a daily if not hourly basis!
- Keeping a static Ansible inventory up to date could become a full-time job, and hence, in many large-scale scenarios, it becomes unrealistic to attempt to use a static inventory on an ongoing basis.
- This is where Ansible's dynamic inventory support comes in.

Generating a dynamic inventory file

- Your first task is to install the relevant Cobbler packages using yum.
- Note that, at the time of writing, the SELinux policy provided with CentOS 7 did not support Cobbler's functionality and blocks some aspects from working.
- Although this is not something you should do in a production environment, your simplest path to getting this demo up and running is to simply disable SELinux:

```
$ yum install -y cobbler cobbler-web  
$ setenforce 0
```



Generating a dynamic inventory file

- Next, ensure that the `cobblerd` service is configured to listen on the loopback address by checking the settings in `/etc/cobbler/settings`—the relevant snippet of the file is shown here and should appear as follows:

```
# default, localhost  
server: 127.0.0.1
```

Generating a dynamic inventory file

- With this step complete, you can start the `cobblerd` service using `systemctl`:

```
$ systemctl start cobblerd.service  
$ systemctl enable cobblerd.service  
$ systemctl status cobblerd.service
```



Generating a dynamic inventory file

- On the test system used for this demo, the following commands were used—however, you must replace the version number in the vmlinuz and initramfs filenames with the appropriate version numbers from your system's /boot directory:

```
$ cobbler distro add --name=CentOS --kernel=/boot/vmlinuz-  
3.10.0-957.el7.x86_64 --initrd=/boot/initramfs-3.10.0-  
957.el7.x86_64.img  
$ cobbler profile add --name=webservers --distro=CentOS
```

Generating a dynamic inventory file

- Let's now add those systems to Cobbler.
- The following two commands will add two hosts called frontend01 and frontend02 to our Cobbler system, using the webservers profile we created previously:

```
$ cobbler system add --name=frontend01 --profile=webservers --  
dns-name=frontend01.example.com --interface=eth0
```

```
$ cobbler system add --name=frontend02 --profile=webservers --  
dns-name=frontend02.example.com --interface=eth0
```

Generating a dynamic inventory file

- For our simple example, we will download these files into our current working directory:



```
$ wget  
https://raw.githubusercontent.com/ansible/ansible-devel/contrib/inventory/cobbler.py  
$ wget  
https://raw.githubusercontent.com/ansible/ansible-devel/contrib/inventory/cobbler.ini  
$ chmod +x cobbler.py
```

Generating a dynamic inventory file

- Edit the cobbler.ini file and ensure that it points to the localhost as, for this example, we are going to run Ansible and Cobbler on the same system.
- In real life, you would point it at the remote URL of your Cobbler system.
- A snippet of the configuration file is shown here to give you an idea of what to configure:

Refer to the file 3_5.txt

Generating a dynamic inventory file

- You can now run an Ansible ad hoc command in the manner you are used to—the only difference this time is that you will specify the filename of the dynamic inventory script rather than the name of the static inventory file.
- Assuming you have set up hosts at the two addresses we entered Cobbler earlier, your output should look something like that shown here:

Refer to the file 3_6.txt

Using multiple inventory sources in inventory directories

- So far in this course, we have been specifying our inventory file (either static or dynamic) using the -i switch in our Ansible commands.
- What might not be apparent is that you can specify the -i switch more than once and so use multiple inventories at the same time.
- This enables you to perform tasks such as running a playbook (or ad hoc command) across hosts from both static and dynamic inventories at the same time.

Using static groups with dynamic groups

- Of course, the possibility of mixing inventories brings with it an interesting question—what happens to the groups from a dynamic inventory and a static inventory if you define both?
- The answer is that Ansible combines both, and this leads to an interesting possibility.
- As you will have observed, our Cobbler inventory script produced an Ansible group called webservers from a Cobbler profile that we called webservers

Using static groups with dynamic groups

- The second group definition should state that webservers is a child group of the centos group.
- The resulting file should look something like this:

```
[webservers]
```

```
[centos:children]  
webservers
```

Using static groups with dynamic groups

- We know that Cobbler has no centos group because we never created one, and we know that any hosts in this group must come via the webservers group when you combine the two inventories, as our static inventory has no hosts in it.
- The results will look something like this:

Refer to the file 3_7.txt

Special host management using patterns

- As a starting point, let's consider again an inventory that we defined earlier in this lesson for the purposes of exploring host groups and child groups.
- For your convenience, the inventory contents are provided again here:

Refer to the file 3_8.txt

Special host management using patterns

- We have already mentioned the special all group to specify all hosts in the inventory:

```
$ ansible -i hostgroups-children-ini all --list-hosts
```

hosts (7):

loadbalancer.example.com

frt01.example.com

frt02.example.com

app01.example.com

app02.example.com

dbms01.example.com

dbms02.example.com

Special host management using patterns

- The asterisk character has the same effect as all, but needs to be quoted in single quotes for the shell to interpret the command properly:

```
$ ansible -i hostgroups-children-ini '*' --list-hosts
```

hosts (7):

loadbalancer.example.com

frt01.example.com

frt02.example.com

app01.example.com

app02.example.com

dbms01.example.com

dbms02.example.com

Special host management using patterns

- Use : to specify a logical OR, meaning "apply to hosts either in this group or that group," as in this example:

```
$ ansible -i hostgroups-children-ini frontends:apps --list-hosts  
hosts (4):
```

frt01.example.com

frt02.example.com

app01.example.com

app02.example.com

Special host management using patterns

- Again, ! is a special character in the shell and so you must quote your pattern string in single quotes for it to work, as in this example:

```
$ ansible -i hostgroups-children-ini 'all:!apps' --list-hosts  
hosts (5):
```

loadbalancer.example.com

frt01.example.com

frt02.example.com

dbms01.example.com

dbms02.example.com

Special host management using patterns

- Use `:&` to specify a logical AND between two groups, for example, if we want all hosts that are in the `centos` group and the `apps` group (again, you must use single quotes in the shell):

```
$ ansible -i hostgroups-children-ini 'centos:&apps' --list-hosts
```

hosts (2):

app01.example.com

app02.example.com

Special host management using patterns

- Use * wildcards in a similar manner to what you would use in the shell, as in this example:

```
$ ansible -i hostgroups-children-ini 'db*.example.com' --  
list-hosts  
hosts (2):  
dbms02.example.com  
dbms01.example.com
```

Special host management using patterns

- Another way you can limit which hosts a command is run on is to use the --limit switch with Ansible.
- This uses the same syntax and pattern notation as in the preceding but has the advantage that you can use it with the ansible-playbook command, where specifying a host pattern on the command line is only supported for the ansible command itself.
- Hence, for example, you could run the following:
Refer to the file 3_9.txt

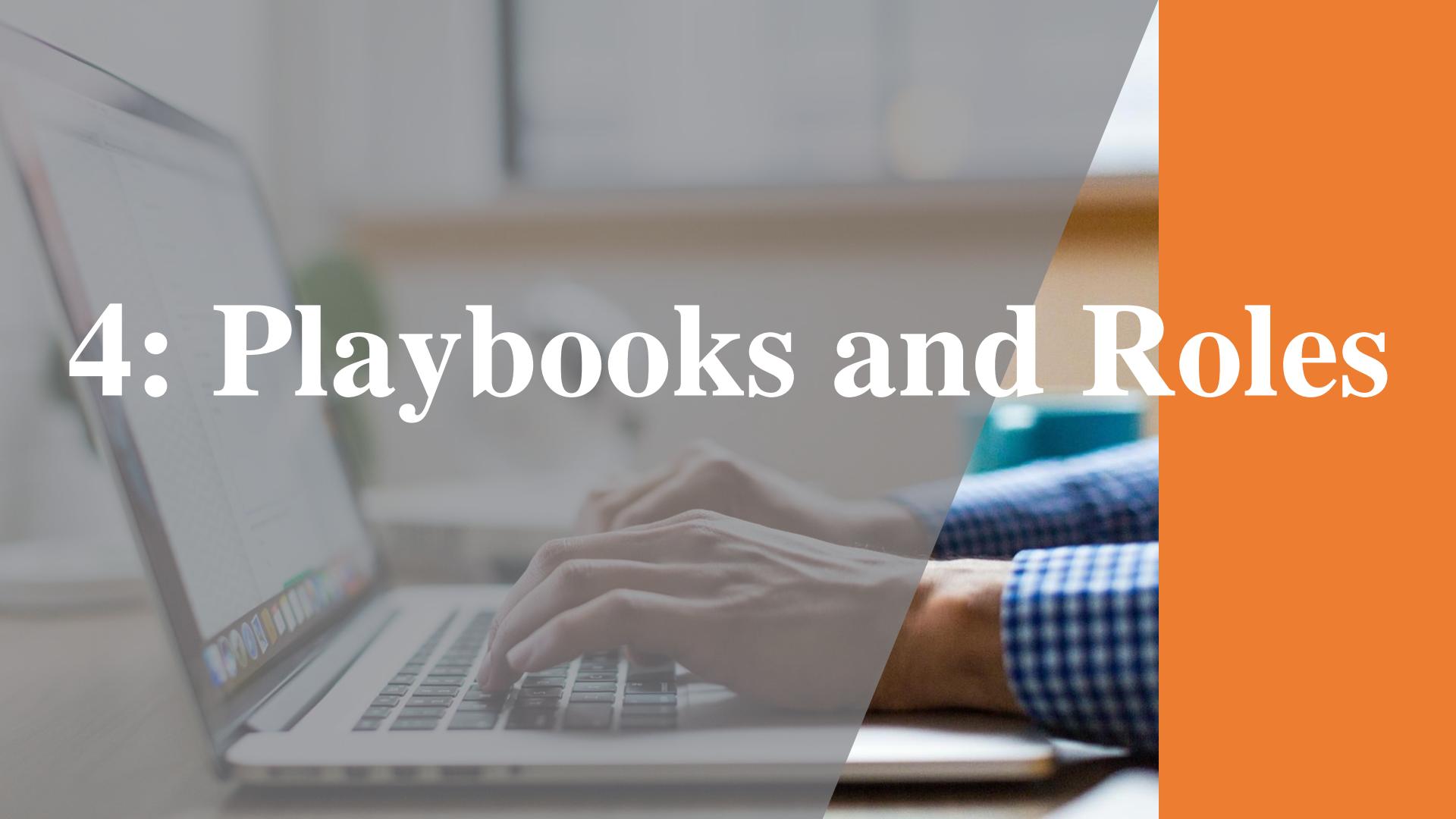


Summary

- Creating and managing Ansible inventories is a crucial part of your work with Ansible, and hence we have covered this fundamental concept early in this course.
- They are vital as without them Ansible would have no knowledge of what hosts it is to run automation tasks against, yet they provide so much more than this.

"Complete Lab 3"

4: Playbooks and Roles

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a professional or technical environment.

Playbooks and Roles

Specifically, in this lesson, we will cover the following topics:

- Understanding the playbook framework
- Understanding roles—the playbook organizer
- Using conditions in your code
- Repeating tasks with loops
- Grouping tasks using blocks
- Configuring play execution via strategies
- Using ansible-pull

Technical requirements

- This lesson assumes that you have set up your control host with Ansible, as detailed in lesson 1, Getting Started with Ansible, and are using the most recent version available—the examples in this lesson were tested with Ansible 2.9.
- This lesson also assumes that you have at least one additional host to test against, and ideally this should be Linux based.

Understanding the playbook framework

- We recommend that you adopt an editor with YAML support to aid you in writing your playbooks if you are doing this for the first time, perhaps Vim, Visual Studio Code, or Eclipse, as these will help you to ensure that your indentation is correct.
- To test the playbooks, we develop in this lesson, we will reuse a variant of an inventory created in lesson 3, Defining Your Inventory (unless stated otherwise):

Refer to the file 4_1.txt

Understanding the playbook framework

- Create a simple playbook to run on the hosts in the frontends host group defined in our inventory file.

```
- hosts: frontends  
  remote_user: danieloh
```

```
tasks:
```

```
  - name: simple connection test  
    ping:  
    remote_user: danieloh
```

Understanding the playbook framework

- We'll also add the ignore_errors directive to this task to ensure that our playbook doesn't fail if the ls command fails (for example, if the directory we're trying to list doesn't exist).
- Be careful with the indentation and ensure it matches that of the first part of the file:
 - name: run a simple command
shell: /bin/ls -al /nonexistent
ignore_errors: True

Understanding the playbook framework

- Let's see how our newly created playbook behaves when we run it:

Refer to the file 4_2.txt

Understanding the playbook framework

- they can be notified more than once but will only be run once regardless, again preventing needless service restarts.
- Consider the following playbook:

Refer to the file 4_3.txt

Understanding the playbook framework

- To keep the output concise, I've turned off fact-gathering for this playbook (we won't use them in any of the tasks).
- I'm also running this on just one host again for conciseness, but you are welcome to expand the demo code as you wish.
- If we run this task a first time, we will see the following results:

Refer to the file 4_4.txt

Understanding the playbook framework

- Notice how the handler was run at the end, as the configuration file was updated.
- However, if we run this playbook a second time without making any changes to the template or configuration file, we will see something like this:

```
$ ansible-playbook -i hosts handlers1.yml
PLAY [Handler demo 1] ****
TASK [Update Apache configuration] ****
ok: [frt01.example.com]
PLAY RECAP ****
frt01.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
```

Understanding the playbook framework

- All handlers should have a globally unique name so that the notify action can call the correct handler.
- You could also call multiple handlers by setting a common name for using the listen directive—this way, you can call either the handler's name or the listen string—as demonstrated in the following example:

Refer to the file 4_5.txt

Understanding the playbook framework

- We only have one task in the playbook, but when we run it, both handlers are called.
- Also, remember that we said earlier that command was among a set of modules that were a special case because they can't detect whether a change has occurred—as a result, they always return the changed value, and so, in this demo playbook, the handlers will always be notified:

Refer to the file 4_6.txt

Comparing playbooks and ad hoc tasks

- If you were to perform the basic installation by hand, you would need to install the package, open up the firewall, and ensure the service is running (and runs at boot time).
- To perform these commands in the shell, you might do the following:

```
$ sudo yum install httpd  
$ sudo firewall-cmd --add-service=http --permanent  
$ sudo firewall-cmd --add-service=https --permanent  
$ sudo firewall-cmd --reload  
$ sudo systemctl enable httpd.service  
$ sudo systemctl restart httpd.service
```

Comparing playbooks and ad hoc tasks

- Now, for each of these commands, there is an equivalent ad hoc Ansible command that you could run.
- We won't go through all of them here in the interests of space; however, let's say you want to restart the Apache service—in this case, you could run an ad hoc command similar to the following (again, we will perform it only on one host for conciseness):

```
$ ansible -i hosts frt01* -m service -a "name=httpd  
state=restarted"
```

Comparing playbooks and ad hoc tasks

- A snippet of this is shown in the following for you to check yours against—the key thing being that the command resulted in the changed status, meaning that it ran successfully and that the service was indeed restarted:

```
frt01.example.com | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": true,
    "name": "httpd",
    "state": "started",
```

Comparing playbooks and ad hoc tasks

- A playbook is hence a far more valuable way to approach this—not only will it perform all of the steps in one go, but it will also give you a record of how it was done for you to refer to later on.
- There are multiple ways to do this, but consider the following as an example:

Refer to the file 4_7.txt

Comparing playbooks and ad hoc tasks

- Now, when you run this, you should see that all of our installation requirements have been completed by one fairly simple and easy to read playbook.
- There is a new concept here, loops, which we haven't covered yet, but don't worry, we will cover this later in this lesson:

Refer to the file 4_8.txt



Defining plays and tasks

- Add the first play to the playbook and define some simple tasks to set up the Apache server on the front end, as shown here:

```
- name: Play 1 - configure the frontend servers
hosts: frontends
become: yes
```

tasks:

```
- name: Install the Apache package
yum:
  name: httpd
  state: latest
- name: Start the Apache server
service:
  name: httpd
  state: started
```

Defining plays and tasks

- Immediately below this, in the same file, add the second play to configure the application tier servers:

```
- name: Play 2 - configure the application servers  
hosts: apps  
become: true
```

tasks:

```
- name: Install Tomcat  
yum:  
  name: tomcat  
  state: latest  
- name: Start the Tomcat server  
service:  
  name: tomcat  
  state: started
```

Defining plays and tasks

- When we run this playbook, we'll see the two plays performed sequentially, in the order they appear in the playbook.
- Note the presence of the PLAY keyword, which denotes the start of each play:

Refer to the file 4_9.txt

Understanding roles – the playbook organizer

Let's explore this in a little more detail.

- Consider the following directory structure:

```
site.yml  
frontends.yml  
dbservers.yml  
roles/  
  installapache/  
    tasks/  
    handlers/  
    templates/  
    vars/  
    defaults/  
  installtomcat/  
    tasks/  
    meta/
```

Understanding roles – the playbook organizer

- For the examples we will develop in this part of this lesson, we will need an inventory, so let's reuse the inventory we used in the previous section (included in the following for convenience):

Refer to the file 4_10.txt

Understanding roles – the playbook organizer

- Create the directory structure for the `installapache` role from within your chosen playbook directory—this is as simple as this:

```
$ mkdir -p roles/installapache/tasks
```

Understanding roles – the playbook organizer

- We can use this special variable, `ansible_distribution`, in a `when` condition to determine which of the tasks files to import:

```
- name: import a tasks based on OS platform
  import_tasks: centos.yml
  when: ansible_distribution == 'CentOS'
- import_tasks: ubuntu.yml
  when: ansible_distribution == 'Ubuntu'
```

Understanding roles – the playbook organizer

- Create centos.yml in roles/installapache/tasks to install the latest version of the Apache web server via the yum package manager. This should contain the following content:

```
---
```

```
- name: Install Apache using yum
  yum:
    name: "httpd"
    state: latest
- name: Start the Apache server
  service:
    name: httpd
    state: started
```

Understanding roles – the playbook organizer

- Create a file called ubuntu.yml in roles/installapache/tasks to install the latest version of the Apache web server via the apt package manager on Ubuntu.
- Notice how the content differs between CentOS and Ubuntu hosts:

```
- name: Install Apache using apt
  apt:
    name: "apache2"
    state: latest
- name: Start the Apache server
  service:
    name: apache2
    state: started
```

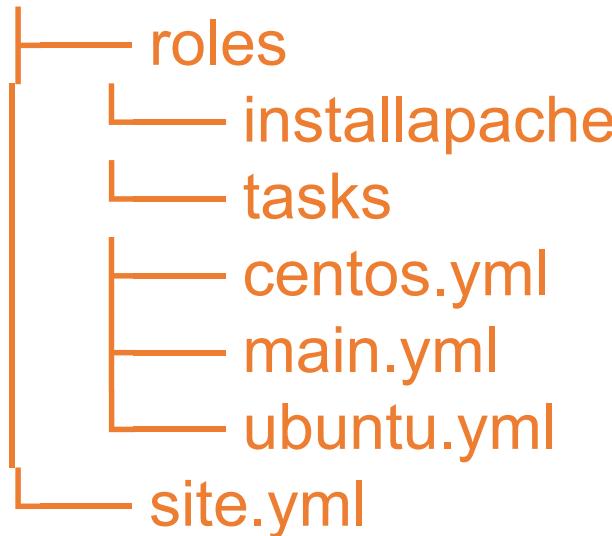
Understanding roles – the playbook organizer

- We have a roles: section where the roles are declared instead.
- Convention dictates that this file be called site.yml, but you are free to call it whatever you like:

```
- name: Install Apache using a role
hosts: frontends
become: true
roles:
  - installapache
```

Understanding roles – the playbook organizer

- For clarity, your final directory structure should look like this:



Understanding roles – the playbook organizer

- With this completed, you can now run your site.yml playbook using ansible-playbook in the normal way—you should see output similar to this:

Refer to the file 4_11.txt

Understanding roles – the playbook organizer

- The roles directory structure including both the common and approle roles would have been created in a similar manner as in the preceding example:

```
- name: Play to import and include a role  
hosts: frontends
```

tasks:

```
- import_role:  
  name: common  
- include_role:  
  name: approle
```

Setting up role-based variables and dependencies

Roles based variables can go in one of two locations:

- defaults/main.yml
- vars/main.yml

Setting up role-based variables and dependencies

- At the top level of our practical example, we will have a copy of the same inventory we have been using throughout this lesson.
- We will also create a simple playbook called site.yml, which contains the following code:

```
- name: Role variables and meta playbook
  hosts: frt01.example.com
  roles:
    - platform
```

Setting up role-based variables and dependencies

- Let's go ahead and create the platform role—unlike our previous role, this will not contain any tasks or even any variable data; instead, it will just contain a meta directory:

```
$ mkdir -p roles/platform/meta
```

Setting up role-based variables and dependencies

- Inside this directory, create a file called main.yml with the following contents:

```
---
```

```
dependencies:
```

```
- role: linuxtype
```

```
vars:
```

```
  type: centos
```

```
- role: linuxtype
```

```
vars:
```

```
  type: ubuntu
```

Setting up role-based variables and dependencies

- Let's now go ahead and create the `linuxtype` role—again, this will contain no tasks, but more dependency declarations:

```
$ mkdir -p roles/linuxtype/meta/
```

- Again, create a `main.yml` file in the `meta` directory, but this time containing the following:

```
---
```

```
dependencies:
```

- `role: version`
- `role: network`

Setting up role-based variables and dependencies

- Let's create the version role first—this will have both meta and tasks directories in it:

```
$ mkdir -p roles/version/meta
```

```
$ mkdir -p roles/version/tasks
```

- In the meta directory, we'll create a main.yml file with the following contents:

```
---
```

```
allow_duplicates: true
```

Setting up role-based variables and dependencies

- We'll also create a simple main.yml file in the tasks directory, which prints the value of the type variable that gets passed to the role:

```
---
```

```
- name: Print type variable
  debug:
    var: type
```

Setting up role-based variables and dependencies

- We will now repeat the process with the network role—to keep our example code simple, we'll define it with the same contents as the version role:

```
$ mkdir -p roles/network/meta
```

```
$ mkdir -p roles/network/tasks
```

- In the meta directory, we'll again create a main.yml file with the following contents:

```
---
```

```
allow_duplicates: true
```

Setting up role-based variables and dependencies

- Again, we'll create a simple main.yml file in the tasks directory, which prints the value of the type variable that gets passed to the role:

- name: Print type variable
debug:
var: type

- At the end of this process, your directory structure should look like this:

Refer to the file 4_12.txt

Setting up role-based variables and dependencies

- You could be forgiven for thinking that we'll see the network and version roles called twice, printing centos once and ubuntu the second time (as this is how we originally specified the dependencies in the platform role). However, when we run it, we actually see this:

Refer to the file 4_13.txt

Setting up role-based variables and dependencies

- Role parameters remain scoped at the role level even when the Ansible parser is run, hence we can declare our dependency twice without the variable getting overwritten.
- To do this, change the contents of the roles/platform/meta/main.yml file to the following:

```
---
```

```
dependencies:  
- role: linuxtype  
  type: centos  
- role: linuxtype  
  type: ubuntu
```

Setting up role-based variables and dependencies

- Do you notice the subtle change? The vars: keyword has gone, and the declaration of type is now at a lower indentation level, meaning it is a role parameter.
- Now, when we run the playbook, we get the results that we had hoped for:

Refer to the file 4_14.txt



Ansible Galaxy

- A quick search on the Ansible Galaxy website returns (at the time of writing) 106 roles for setting the MOTD.
- If we want to use one of these, we could download it into our role's directory using the following command:

```
$ ansible-galaxy role install -p roles/ arillso.motd
```

Ansible Galaxy

- Another neat trick is to use ansible-galaxy to create an empty role directory structure for you to create your own roles in—this saves all the manual directory and file creation we have been undertaking in this lesson, as in this example:

Refer to the file 4_15.txt



Using conditions in your code

- As ever, we'll need an inventory to get started, and we'll reuse the inventory we have used throughout this lesson:

Refer to the file 4_16.txt

- Instead, let's define the task that will perform our update but add a when clause containing a Jinja 2 expression to it in a simple example playbook:

```
- name: Play to patch only CentOS systems
hosts: all
become: true
tasks:
- name: Patch CentOS systems
  yum:
    name: httpd
    state: latest
    when: ansible_facts['distribution'] == "CentOS"
```

Using conditions in your code

- Now, when we run this task, if your test system(s) are CentOS-based (and mine are), you should see output like the following:

Refer to the file 4_17.txt

- In this case, we can expand the logic in our playbook to check both the distribution and major version, as follows:

```
- name: Play to patch only CentOS systems
hosts: all
become: true
tasks:
- name: Patch CentOS systems
  yum:
    name: httpd
    state: latest
    when: (ansible_facts['distribution'] == "CentOS" and
ansible_facts['distribution_major_version'] == "6")
```

Using conditions in your code

- Now, if we run our modified playbook, depending on the systems you have in your inventory, you might see output similar to the following.
- In this case, my app01.example.com server was based on CentOS 6 so had the patch applied.
- All other systems were skipped because they did not match my logical expression:

Refer to the file 4_18.txt

Using conditions in your code

- Consider the following playbook code. It contains two tasks, the first of which is to obtain the listing of the current directory and capture the output of the shell module in a variable called shellresult.
- When then print a simple debug message, but only on the condition that the hosts string is in the output of the shell command:

Refer to the file 4_19.txt

Using conditions in your code

- Now, when we run this in the current directory, which if you are working from the GitHub repository that accompanies this course will contain a file named hosts, then you should see output similar to the following:

Refer to the file 4_20.txt



Using conditions in your code

- Yet, if the file doesn't exist, then you'll see that the debug message gets skipped:

Refer to the file 4_21.txt



Using conditions in your code

- Ansible will by default treat it as a string, If you need to perform an integer comparison instead, you must first cast the variable to an integer type.
- For example, here is a fragment of a playbook that will run a task only on Fedora 25 and newer:

tasks:

```
- name: Only perform this task on Fedora 25 and later
  shell: echo "only on Fedora 25 and later"
  when: ansible_facts['distribution'] == "Fedora" and
        ansible_facts['distribution_major_version']|int >= 25
```

Repeating tasks with loops

- As ever, we must start with an inventory to work against, and we will use our by-now familiar inventory, which we have consistently used throughout this lesson:

Refer to the file 4_22.txt

- Consider the following code:

```
---
```

```
- name: Simple loop demo play
  hosts: frt01.example.com
```

```
tasks:
```

```
  - name: Echo a value from the loop
```

```
    command: echo "{{ item }}"
```

```
loop:
```

```
  - 1
```

```
  - 2
```

```
  - 3
```

```
  - 4
```

```
  - 5
```

```
  - 6
```

Repeating tasks with loops

- When working with the loop data, we use a special variable called item, which contains the current value from the loop iteration to be echoed.
- Hence, if we run this playbook, we should see output like the following:

Refer to the file 4_23.txt



Repeating tasks with loops

- You can combine the conditional logic we discussed in the preceding section with loops, to make the loop operate on just a subset of its data.
- For example, consider the following iteration of the playbook:

Refer to the file 4_24.txt

Repeating tasks with loops

- Now, when we run this, we see that the task is skipped until we reach the integer value of 4 and higher in the loop contents:

Refer to the file 4_25.txt

- let's see what happens if we further enhance the playbook, as follows:

Refer to the file 4_26.txt

- Now, when we run the playbook, you will see pages of output containing the dictionary with the contents of loop result.
- The following output is truncated in the interests of space but demonstrates the kind of results you should expect from running this playbook:

Refer to the file 4_27.txt

- We'll use the loop_var directive to change the name of the special loop contents variable from item to second_item:

```
- name: Play to demonstrate nested loops  
hosts: localhost
```

tasks:

```
- name: Outer loop  
include_tasks: loopsubtask.yml
```

loop:

- a
- b
- c

loop_control:

```
loop_var: second_item
```



Repeating tasks with loops

- Note that the structure of this file is very much like a tasks file in a role—it is not a complete playbook, but rather simply a list of tasks:

```
- name: Inner loop
  debug:
    msg: "second item={{ second_item }} first item={{ item }}"
  loop:
    - 100
    - 200
    - 300
```

Repeating tasks with loops

- Now you should be able to run the playbook, and you will see Ansible iterate over the outer loop first and then process the inner loop over the data defined by the outer loop.
- As the loop variable names do not clash, all works exactly as we would expect:

Refer to the file 4_28.txt

Grouping tasks using blocks

- Blocks are also valuable when it comes to error handling and especially when it comes to recovering from an error condition.
- We shall explore both of these through simple practical examples in this lesson to get you up to speed with blocks in Ansible.
- As ever, let's ensure we have an inventory to work from:
Refer to the file 4_29.txt

Grouping tasks using blocks

- We could achieve this with three individual tasks, all with a when clause associated with them, but blocks provide us with a better way.
- The following example playbook shows the three tasks discussed contained in a block (notice the additional level of indentation required to denote their presence in the block):

Refer to the file 4_30.txt

Grouping tasks using blocks

- When you run this playbook, you should find that the Apache-related tasks are only run on any Fedora hosts you might have in your inventory; you should see that either all three tasks are run or are skipped—depending on the makeup and contents of your inventory, it might look something like this:

Refer to the file 4_31.txt

Grouping tasks using blocks

- Let's create a new playbook, this time with the following contents:

Refer to the file 4_32.txt

Grouping tasks using blocks

- With this flow of execution in mind, you should see output like the following when you execute this playbook, noting that we have deliberately created two error conditions to demonstrate the flow:

Refer to the file 4_33.txt

Configuring play execution via strategies

- Let's demonstrate this by creating a playbook that has a deliberate error in it. Note the strategy: debug and debugger: on_failed statements in the play definition:

```
- name: Play to demonstrate the debug strategy
  hosts: frt01.example.com
  strategy: debug
  debugger: on_failed
  gather_facts: no
  vars:
    username: daniel
  tasks:
    - name: Generate an error by referencing an undefined variable
      ping: data={{ mobile }}
```

Configuring play execution via strategies

- Now if you execute this playbook, you should see that it starts to run, but then drops you into the integrated debugger when it encounters the deliberate error it contains.
- The start of the output should be similar to the following:

Refer to the file 4_34.txt



Configuring play execution via strategies

- To take you through a very simple, practical debugging example, however, enter the p task command at the prompt—this will cause the Ansible debugger to print the name of the failing task; this is very useful if you are in the midst of a large playbook:

```
[frt01.example.com] TASK: Generate an error by referencing  
an undefined variable (debug)> p task  
TASK: Generate an error by referencing an undefined variable
```

Configuring play execution via strategies

- Now we know where the play failed, so let's dig a little deeper by issuing the p task.args command, which will show us the arguments that were passed to the module in the task:

```
[frt01.example.com] TASK: Generate an error by  
referencing an undefined variable (debug)> p task.args  
{u'data': u'{{ mobile }}'}
```

Configuring play execution via strategies

```
[frt01.example.com] TASK: Generate an error by
referencing an undefined variable (debug)> p task_vars
{'ansible_check_mode': False,
'ansible_current_hosts': [u'frt01.example.com'],
'ansible_dependent_role_names': [],
'ansible_diff_mode': False,
'ansible_facts': {},
'ansible_failed_hosts': [],
'ansible_forks': 5,
```

Configuring play execution via strategies

- Hence, this should be enough information to fix your playbook. Enter q to quit the debugger:

```
[frt01.example.com] TASK: Generate an error by  
referencing an undefined variable (debug)> q
```

User interrupted execution

\$



Using ansible-pull

- The ansible-pull command is a special feature of Ansible that allows you to, all in one go, pull a playbook from a Git repository (for example, GitHub)
- Then execute it, hence saving the usual steps such as cloning (or updating the working copy of) the repository, then executing the playbook.

Using ansible-pull

- Let's use a simple playbook from GitHub that sets the message of the day based on variable content.
- To do this, we will run the following command (which we'll break down in a minute):

```
$ ansible-pull -d /var/ansible-set-motd -i ${HOSTNAME}, -  
U https://github.com/jamesfreeman959/ansible-set-  
motd.git site.yml -e "ag_motd_content='MOTD generated  
by ansible-pull'" >> /tmp/ansible-pull.log 2>&1
```

Using ansible-pull

- When you run this command, you should see some output like the following (note that log redirection has been removed to make it easier to see the output):

Refer to the file 4_35.txt



Summary

- Playbooks are the lifeblood of Ansible automation, providing a robust framework within which to define logical collections of tasks and handle error conditions cleanly and robustly.
- The addition of roles into this mix is valuable in terms of not only organizing your code but also in terms of supporting code reuse as your automation requirements grow.

"Complete Lab 4"

5: Consuming and Creating Modules



Consuming and Creating Modules

Specifically, in this lesson, you will cover the following topics:

- Executing multiple modules using the command line
- Reviewing the module index
- Accessing module documentation from the command line
- Module return values
- Developing custom modules

Technical requirements

- This lesson assumes that you have set up your control host with Ansible, as detailed in lesson 1, Getting Started with Ansible, and are using the most recent version available – the examples in this lesson were tested with Ansible 2.9.
- This lesson also assumes that you have at least one additional host to test against.

Executing multiple modules using the command line

- As ever, when working with Ansible commands, we need an inventory to run our commands against.
- For this lesson, as our focus is on the modules themselves, we will use a very simple and small inventory, as shown here:

[frontends]

frt01.example.com

[appservers]

app01.example.com

Executing multiple modules using the command line

- Now, for the first part of our recap, you can run a module very easily via an ad hoc command and use the -m switch to tell Ansible which module you want to run.
- Hence, one of the simplest commands you can run is the Ansible ping command, as shown here:

```
$ ansible -i hosts appservers -m ping
```

Executing multiple modules using the command line

- Now, one thing we have not previously looked at is the communication between Ansible and its modules; however, let's examine the output of the preceding command:

```
$ ansible -i hosts appservers -m ping  
app01.example.com | SUCCESS => {  
    "ansible_facts": {  
        "discovered_interpreter_python": "/usr/bin/python"  
    },  
    "changed": false,  
    "ping": "pong"  
}
```

Executing multiple modules using the command line

- Now, let's run another command that takes an argument and passes that data to the module:

```
$ ansible -i hosts appservers -m command -a "/bin/echo  
'hello modules'"
```

Executing multiple modules using the command line

- In this case, we provided a single string as an argument to the command module, which Ansible, in turn, converts into JSON and passes down to the command module when it's invoked.
- When you run this ad hoc command, you will see an output similar to the following:

```
$ ansible -i hosts appservers -m command -a  
"/bin/echo 'hello modules'"  
app01.example.com | CHANGED | rc=0 >>  
hello modules
```

Reviewing the module index

- Begin by opening the categorized module index in your web browser, as discussed previously:
https://docs.ansible.com/ansible/latest/modules/modules_by_category.html
- Now, we know that Amazon Web Services is almost certainly going to feature in the Cloud modules category, so let's open that in our browser.

Ansible
2.9
latest ▾

Search docs

INSTALLATION, UPGRADE & CONFIGURATION

Installation Guide

Ansible Porting Guides

USING ANSIBLE

User Guide

- Ansible Quickstart Guide
- Ansible concepts
- Getting Started
- How to build your inventory
- Working with dynamic inventory
- Patterns: targeting hosts and groups
- Introduction to ad-hoc commands
- Connection methods and details
- Working with command line tools
- Working With Playbooks
- Understanding privilege escalation: become
- Ansible Vault

Working With Modules

- [aws_direct_connect_virtual_interface](#) – Manage Direct Connect virtual interfaces
- [aws_eks_cluster](#) – Manage Elastic Kubernetes Service Clusters
- [aws_elasticbeanstalk_app](#) – create, update, and delete an elastic beanstalk application
- [aws_glue_connection](#) – Manage an AWS Glue connection
- [aws_glue_job](#) – Manage an AWS Glue job
- [aws_inspector_target](#) – Create, Update and Delete Amazon Inspector Assessment Targets
- [aws_kms](#) – Perform various KMS management tasks
- [aws_kms_info](#) – Gather information about AWS KMS keys
- [aws_netapp_cvs_active_directory](#) – NetApp AWS CloudVolumes Service Manage Active Directory
- [aws_netapp_cvs_FileSystems](#) – NetApp AWS Cloud Volumes Service Manage FileSystem
- [aws_netapp_cvs_pool](#) – NetApp AWS Cloud Volumes Service Manage Pools
- [aws_netapp_cvs_snapshots](#) – NetApp AWS Cloud Volumes Service Manage Snapshots
- [aws_region_info](#) – Gather information about AWS regions
- [aws_s3](#) – manage objects in S3
- [aws_s3_bucket_info](#) – Lists S3 buckets in AWS
- [aws_s3_cors](#) – Manage CORS for S3 buckets in AWS
- [aws_secret](#) – Manage secrets stored in AWS Secrets Manager
- [aws_ses_identity](#) – Manages SES email and domain identity
- [aws_ses_identity_policy](#) – Manages SES sending authorization policies
- [aws_ses_rule_set](#) – Manages SES inbound receipt rule sets
- [aws_sgw_info](#) – Fetch AWS Storage Gateway information
- [aws_ssm_parameter_store](#) – Manage key-value pairs in aws parameter store
- [aws_waf_condition](#) – create and delete WAF Conditions
- [aws_waf_info](#) – Retrieve information for WAF ACLs, Rule , Conditions and Filters
- [aws_waf_rule](#) – create and delete WAF Rules

Q Search this site

Ansible

2.9

latest ▾

 Search docs

INSTALLATION, UPGRADE & CONFIGURATION

[Installation Guide](#)[Ansible Porting Guides](#)

USING ANSIBLE

User Guide

[Ansible Quickstart Guide](#)[Ansible concepts](#)[Getting Started](#)[How to build your inventory](#)[Working with dynamic inventory](#)[Patterns: targeting hosts and groups](#)[Introduction to ad-hoc commands](#)[Connection methods and details](#)[Working with command line tools](#)[Working With Playbooks](#)[Understanding privilege escalation:
become](#)[Ansible Vault](#)

Working With Modules

- [route53_health_check](#) – add or delete health-checks in Amazon's Route53 DNS service
- [route53_info](#) – Retrieves route53 details using AWS methods
- [route53_zone](#) – add or delete Route53 zones
- [s3_bucket](#) – Manage S3 buckets in AWS, DigitalOcean, Ceph, Walrus and FakeS3
- [s3_bucket_notification](#) – Creates, updates or deletes S3 Bucket notification for lambda
- [s3_lifecycle](#) – Manage s3 bucket lifecycle rules in AWS
- [s3_logging](#) – Manage logging facility of an s3 bucket in AWS
- [s3_sync](#) – Efficiently upload multiple files to S3
- [s3_website](#) – Configure an s3 bucket as a website
- [sns](#) – Send Amazon Simple Notification Service messages
- [sns_topic](#) – Manages AWS SNS topics and subscriptions
- [sq_s_queue](#) – Creates or deletes AWS SQS queues
- [sts_assume_role](#) – Assume a role using AWS Security Token Service and obtain temporary credentials
- [sts_session_token](#) – Obtain a session token from the AWS Security Token Service

Atomic

- [atomic_container](#) – Manage the containers on the atomic host platform
- [atomic_host](#) – Manage the atomic host platform
- [atomic_image](#) – Manage the container images on the atomic host platform

Azure

- [azure_rm_acs](#) – Manage an Azure Container Service(ACS) instance
- [azure_rm_aks](#) – Manage a managed Azure Container Service (AKS) instance
- [azure_rm_aks_info](#) – Get Azure Kubernetes Service facts

 Search this site



A screenshot of the Ansible Documentation sidebar. It includes sections for Installation, Upgrades, Configuration, Installation Guide, Ansible Porting Guides, Using Ansible, User Guide, Contributing to Ansible, Ansible Community Guide, Extending Ansible, Developer Guide, Common Ansible Scenarios, Public Cloud Guides, Network Technology Guides, Virtualization and Containerization Guides, and Ansible for Network Automation.

aws_s3 – manage objects in S3

- Synopsis
- Requirements
- Parameters
- Notes
- Examples
- Return Values
- Status

Synopsis

- This module allows the user to manage S3 buckets and the objects within them. Includes support for creating and deleting both objects and buckets, retrieving objects as files or strings and generating download links. This module has a dependency on boto3 and botocore.

Requirements

The below requirements are needed on the host that executes this module.

- boto
- boto3
- botocore
- python >= 2.6

 Search this site

Accessing module documentation from the CL

The following are some examples to show you how to interact with the ansible-doc tool:

- You can list all of the modules that there's documentation for on your Ansible control machine by simply issuing the following command:

```
$ ansible-doc -l
```

Accessing module documentation from the CL

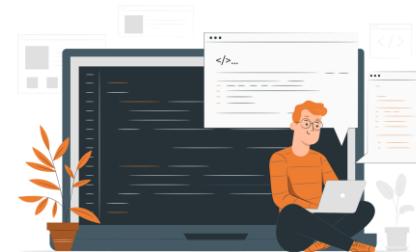
- You should see an output similar to the following:

fortios_router_community_list	Configure community lists in Fortinet's FortiOS ...
azure_rm_devtestlab_info	Get Azure DevTest Lab facts
ecs_taskdefinition	register a task definition in ecs
avi_alertscriptconfig	Module for setup of AlertScriptConfig Avi RESTfu...
tower_receive	Receive assets from Ansible Tower
netapp_e_iscsi_target	NetApp E-Series manage iSCSI target configuratio...
azure_rm_acs	Manage an Azure Container Service(ACS) instance
fortios_log_syslogd2_filter	Filters for remote system server in Fortinet's F...
junos_rpc	Runs an arbitrary RPC over NetConf on an Juniper...
na_elementsw_vlan	NetApp Element Software Manage VLAN
pn_ospf	CLI command to add/remove ospf protocol to a vRo...
pn_snmp_vacm	CLI command to create/modify/delete snmp-vacm
cp_mgmt_service_sctp	Manages service-sctp objects on Check Point over...
onyx_ospf	Manage OSPF protocol on Mellanox ONYX network de.

Accessing module documentation from the CL

- There are many pages of output, which just shows you how many modules there are! In fact, you can count them:

```
$ ansible-doc -l | wc -l  
3387
```



- As before, you can search for specific modules using your favorite shell tools to process the index; for example, you could grep for s3 to find all of the S3-related modules, as we did interactively in the web browser in the previous section:

```
$ ansible-doc -l | grep s3
s3_bucket_notification          Creates, upda...
purefb_s3user                   Create or del...
purefb_s3acc                    Create or del...
aws_s3_cors                      Manage CORS f...
s3_sync                          Efficiently u...
s3_logging                       Manage loggin...
s3_website                        Configure an ...
s3_bucket                         Manage S3 buc...
s3_lifecycle                     Manage s3 buc...
aws_s3_bucket_info               Lists S3 buck...
aws_s3                            manage object...
```

Accessing module documentation from the CL

- Now, we can easily look up the specific documentation for the module that interests us.
- Say we want to learn more about the aws_s3 module – just as we did on the website, simply run the following:

```
$ ansible-doc aws_s3
```

- This should produce an output similar to the following:

Refer to the file 5_1.txt

Module return values

- Without further ado, let's use the ansible-doc tool that we learned about in the previous section and see what this says about the return values for this module:

```
$ ansible-doc ping
```



Module return values

- If you scroll to the bottom of the output from the preceding command, you should see something like this:

```
$ ansible-doc ping
```

...

RETURN VALUES:

ping:

description: value provided with the data parameter

returned: success

type: str

sample: pong

- Let's put a very simple playbook together. We're going to run the ping module with no arguments, capture the return values using the register keyword, and then use the debug module to dump the return values onto the Terminal:

```
- name: Simple play to demonstrate a return value
  hosts: localhost
  tasks:
    - name: Perform a simple module based task
      ping:
        register: pingresult
    - name: Display the result
      debug:
        var: pingresult
```



Module return values

- Now, let's see what happens when we run this playbook:

Refer to the file 5_2.txt



Developing custom modules

- On Fedora, you would run the following command to install the required packages:

```
$ sudo dnf install python python-devel
```

- Similarly, on CentOS, you would run the following command to install the required packages:

```
$ sudo yum install python python-devel
```



Developing custom modules

- On Ubuntu, you would run the following commands to install the packages you need:

```
$ sudo apt-get update
```

```
$ sudo apt-get install python-pip python-dev build-essential
```

- If you are working on macOS and are using the Homebrew packaging system, the following command will install the packages you need:

```
$ sudo brew install python
```

Developing custom modules

- Once you have the required packages installed, you will need to clone the Ansible Git repository to your local machine as there are some valuable scripts in there that we will need later on in the module development process.
- Use the following command to clone the Ansible repository to your current directory on your development machine:

```
$ git clone https://github.com/ansible/ansible.git
```



Developing custom modules

- The following commands were tested on CentOS 7.7 with the default Python 2.7.5 to create a virtual environment called moduledev inside the Ansible source code directory you just cloned from GitHub:

```
$ cd ansible
```

```
$ python -m virtualenv moduledev
```

New python executable in

/home/james/ansible/moduledev/bin/python

Installing setuptools, pip, wheel...done.

Developing custom modules

- In your preferred editor, create a new file called (for example) `remote_filecopy.py`:

```
$ vi remote_filecopy.py
```

- Start with a shebang to indicate that this module should be executed with Python:

```
#!/usr/bin/python
```

Developing custom modules

- Text given here is merely an example; you should investigate the various appropriate licenses for yourself and determine which is the best for your module:

```
# Copyright: (c) 2018, Jesse Keating
<jesse.keating@example.org>
# GNU General Public License v3.0+ (see COPYING or
https://www.gnu.org/licenses/gpl-3.0.txt)
```

Developing custom modules

- The values suggested in the following code will be fine for just getting started, but if your module gets accepted into the official Ansible source code, they are likely to change:

```
ANSIBLE_METADATA = {'metadata_version': '1.1',
                    'status': ['preview'],
                    'supported_by': 'community'}
```

Developing custom modules

- Remember ansible-doc and that excellent documentation that is available on the Ansible documentation website? That all gets automatically generated from special sections you add to this file.
- Let's get started by adding the following code to our module:



Refer to the file 5_3.txt

Developing custom modules

- The examples that you will find in the documentation are also generated from this file – they have their own special documentation section immediately after DOCUMENTATION and should provide practical examples on how you might create a task using your module, as shown in the following example:

EXAMPLES = ""

```
# Example from Ansible Playbooks
- name: backup a config file
  remote_copy:
    source: /etc/herp/derp.conf
    dest: /root/herp-derp.conf.bak
```



Developing custom modules

- The data that's returned by your module to Ansible should also be documented in its own section.
- Our example module will return the following values:

Refer to the file 5_4.txt



Developing custom modules

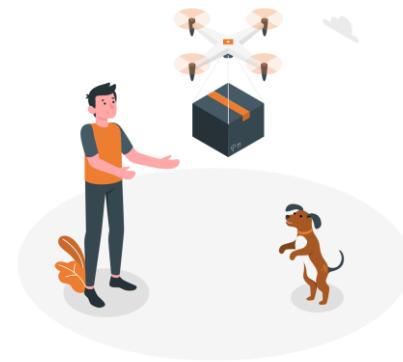
- Immediately after we have finished our documentation section, we should import any Python modules we're going to use.
- Here, we will include the `shutil` module, which will be used to perform our file copy:

```
import shutil
```

Developing custom modules

- Our module should start by defining a main function, in which we will create an object of the AnsibleModule type and use an argument_spec dictionary to obtain the options that the module was called with:

```
def main():
    module = AnsibleModule(
        argument_spec = dict(
            source=dict(required=True, type='str'),
            dest=dict(required=True, type='str')
        )
    )
```



Developing custom modules

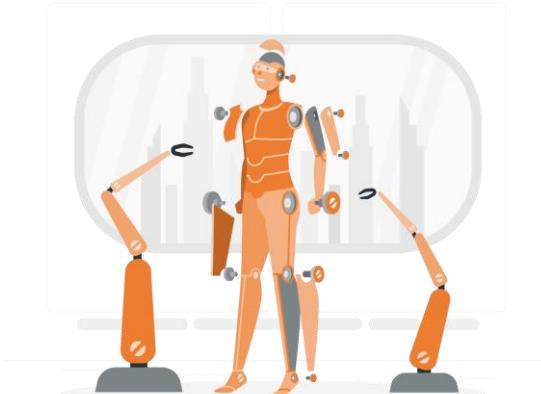
- At this stage, we have everything we need to write our module's functional code – even the options that it was called with.
- Hence, we can use the Python shutil module to perform the local file copy, based on the arguments provided:

```
shutil.copy(module.params['source'],  
           module.params['dest'])
```

Developing custom modules

- for simplicity, we'll simply exit with the changed status every time – expanding this logic and making the return status more meaningful is left as an exercise for you:

```
module.exit_json(changed=True)
```



Developing custom modules

- As we approach the end of our module code, we must now tell Python where it can import the AnsibleModule object from.
- This can be done with the following line of code:

```
from ansible.module_utils.basic import *
```

- Now for the final two lines of code for the module – this is where we tell the module that it should be running the main function when it starts:

```
if __name__ == '__main__':  
    main()
```

Developing custom modules

- Create a file with the following contents to provide the arguments:

```
{  
    "ANSIBLE_MODULE_ARGS": {  
        "source": "/tmp/foo",  
        "dest": "/tmp/bar"  
    }  
}
```



Developing custom modules

- Armed with this little snippet of JSON, you can execute your module directly with Python.
- If you haven't already done so, you'll need to set up your Ansible development environment as follows.
- Note that we also manually create the source file, /tmp/foo, so that our module can really perform the file copy:

Refer to the file 5_5.txt

Developing custom modules

- Now, you're finally ready to run your module for the first time. You can do this as follows:

```
(moduledev) $ python remote_filecopy.py args.json
{"invocation": {"module_args": {"dest": "/tmp/bar",
"source": "/tmp/foo"}}, "changed": true}
```

```
(moduledev) $ ls -l /tmp/bar
-rw-r--r-- 1 root root 0 Apr 16 16:24 /tmp/bar
```

Developing custom modules

- By default, Ansible will check the playbook directory for a subdirectory called library/ and will run referenced modules from here. Hence, we might create the following:

```
$ cd ~  
$ mkdir testplaybook  
$ cd testplaybook  
$ mkdir library  
$ cp ~/ansible/moduledev/remote_filecopy.py library/
```

Developing custom modules

- Now, create a simple inventory file in this playbook directory, just as we did previously, and add a playbook with the following contents:

Refer to the file 5_6.txt

Developing custom modules

- For the purposes of clarity, your final directory structure should look like this:

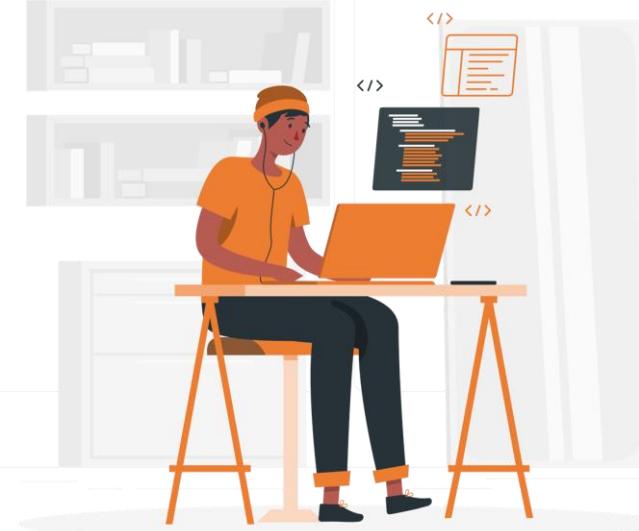
```
testplaybook
├── hosts
└── library
    └── remote_filecopy.py
    └── testplaybook.yml
```



Developing custom modules

- Now, try running the playbook in the usual manner and see what happens:

Refer to the file 5_7.txt



Avoiding common pitfalls

```
(moduledev) $ rm -f /tmp/foo
(moduledev) $ python remote_filecopy.py args.json
Traceback (most recent call last):
  File "remote_filecopy.py", line 99, in <module>
    main()
  File "remote_filecopy.py", line 93, in main
    module.params['dest'])
  File "/usr/lib64/python2.7/shutil.py", line 119, in copy
    copyfile(src, dst)
  File "/usr/lib64/python2.7/shutil.py", line 82, in copyfile
    with open(src, 'rb') as fsrc:
IOError: [Errno 2] No such file or directory: '/tmp/foo'
```

Avoiding common pitfalls

- We can, without a doubt, do better. Let's make a copy of our module and add a little code to it.
- First, replace the shutil.copy lines of code with the following:

```
try:  
    shutil.copy(module.params['source'],  
module.params['dest'])  
except:  
    module.fail_json(msg="Failed to copy file")
```



Avoiding common pitfalls

- Now, when we try and run the module with a non-existent source file, we will see the following cleanly formatted JSON output:

```
(moduledev) $ rm -f /tmp/foo
(moduledev) $ python better_remote_filecopy.py args.json
{"msg": "Failed to copy file", "failed": true, "invocation": {"module_args": {"dest": "/tmp/bar", "source": "/tmp/foo"}}}
```

Avoiding common pitfalls

- However, the module still works in the same manner as before if the copy succeeds:

```
(moduledev) $ touch /tmp/foo
```

```
(moduledev) $ python better_remote_filecopy.py args.json
```

```
{"invocation": {"module_args": {"dest": "/tmp/bar", "source": "/tmp/foo"}}, "changed": true}
```

Testing and documenting your module

- To run the sanity tests, assuming you have cloned the official repository, change into this directory and set up your environment.
- Note that if your standard Python binary isn't Python 3, the ansible-test tool will not run, so you should ensure Python 3 is installed and, if necessary, set up a virtual environment to ensure you are using Python 3.
- This can be done as follows:

Refer to the file 5_8.txt

Testing and documenting your module

- Next, use pip to install the Python requirements so that you can run the ansible-test tool:

```
(venv) $ pip install -r test/runner/requirements/sanity.txt
```

- Now, provided you have copied your module code into the appropriate location in the source tree (an example copy command is shown here), you can run the sanity tests as follows:

Refer to the file 5_9.txt



Testing and documenting your module

- Now, you can manually tell ansible-doc where to look for modules instead of the default path.
- This means that you could run the following:

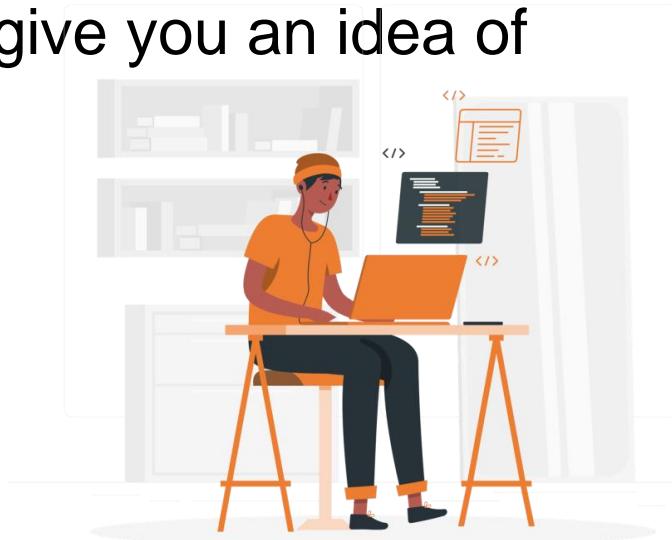
```
$ cd ~/ansible
```

```
$ ansible-doc -M moduledev/ remote_filecopy
```

Testing and documenting your module

- You should be presented with the textual rendering of the documentation we created earlier – an example of the first page is shown here to give you an idea of how it should look:

Refer to the file 5_10.txt



Testing and documenting your module

- Under `lib/ansible/modules/`, you will find a series of categorized directories that modules are placed under – ours fits best under the `files` category, so copy it to this location in preparation for the build process to come:

```
$ cp moduledev/remote_filecopy.py lib/ansible/modules/files/
```

- Change to the `docs/docsite/` directory as the next step in the documentation creation process:

```
$ cd docs/docsite/
```

Testing and documenting your module

- Build a documentation-based Python file.
- Use the following command to do so:

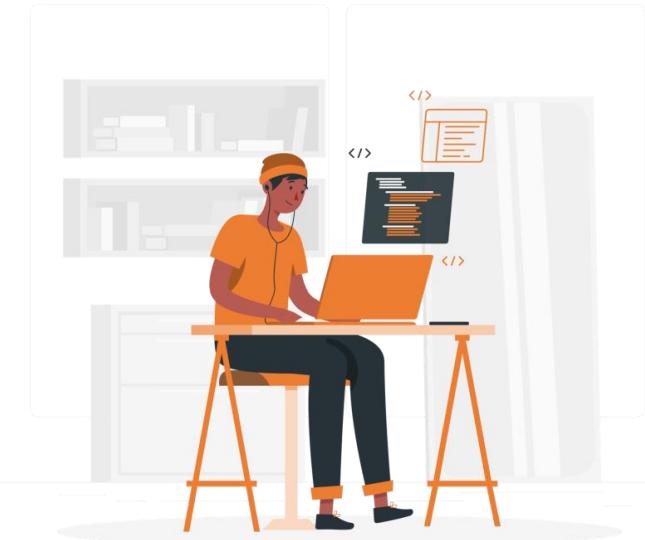
```
$ MODULES=hello_module make webdocs
```



Testing and documenting your module

- If you don't have this installed on your node, please do this before proceeding) and then ran the following commands:

```
$ pip3 uninstall sphinx  
$ pip3 install sphinx==2.4.4  
$ pip3 install sphinx-notfound-page
```



Testing and documenting your module

- With this in place, you will be able to successfully run make webdocs to build your documentation.
- You will see pages of output, A successful run should end with something like the output shown here:

Refer to the file 5_11.txt

Testing and documenting your module

- Now, notice how, at the end of this process, the make command tells us where to look for the compiled documentation.
- If you look in here, you will find the following:

```
$ find /home/james/ansible/docs/docsite -name remote_filecopy*
/home/james/ansible/docs/docsite/rst/modules/remote_filecopy_module.rst
/home/james/ansible/docs/docsite/_build/html/modules/remote_filecopy_mo
dule.html
/home/james/ansible/docs/docsite/_build/doctrees/modules/remote_filecopy
_module.doctree
```

The module checklist

- In addition to the pointers and good practices that we have covered so far, there are a few more things you should adhere to in your module code
- To produce something that will be considered of a high standard for potential inclusion with Ansible



Contributing upstream

- When you've worked hard on your module and thoroughly tested and documented it, you might feel that it is time to submit it to the Ansible project for inclusion.
- Doing this means creating a pull request on the official Ansible repository.



Code

Issues 3,931

Pull requests 1,670

Projects 24

Insights

Ansible is a radically simple IT automation platform that makes your applications and systems easier to deploy. Avoid writing scripts or custom code to deploy and update your applications — automate in a language that approaches plain English, using SSH, with no agents to install on remote systems. <https://docs.ansible.com/ansible/> <https://www.ansible.com/>

[python](#) [ansible](#)

42,409 commits

43 branches

254 releases

4,137 contributors

GPL-3.0

Branch: [devel](#) ▾[New pull request](#)[Create new file](#)[Upload files](#)[Find file](#)[Clone or download ▾](#)

WojciechowskiPiotr and ansibot	docker_host_facts: Get system-wide information about docker host (#51373)	...	Latest commit e633b93 5 hours ago
.github	Added new AIX and Gitlab members		23 hours ago
bin	Save the command line arguments into a global context		a month ago
changelogs	hashi_vault: add support for userpass authentication (#51538)		7 hours ago
contrib	inventory: vagrant: rename deprecated ansible_ssh_* (#50694)		24 days ago
docs	doc: Correct path of unit tests directory (#51631)		18 hours ago

Contributing upstream

- Clone the devel branch that you've just forked to your local machine.
- Use a command similar to the following, but be sure to replace the URL with the one that matches your own GitHub account:

```
$ git clone https://github.com/danieloh30/ansible.git
```

Contributing upstream

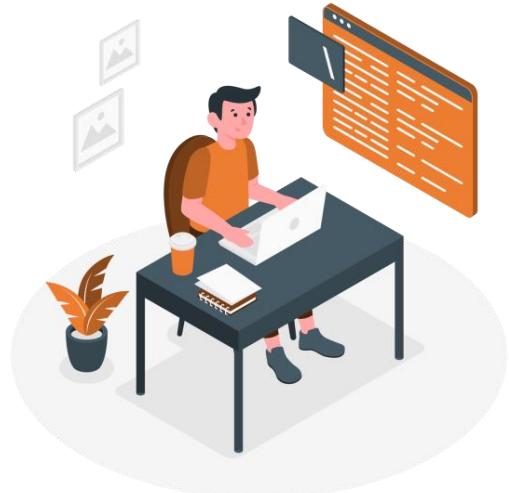
- Once you've added your Python file, perform git add to make Git aware of the new file, and then commit it with a meaningful commit message. Some example commands are as follows:

```
$ cd ansible  
$ cp ~/ansible-development/moduledev/remote_filecopy.py  
./lib/ansible/modules/files/  
$ git add lib/ansible/modules/files/remote_filecopy.py  
$ git commit -m 'Added tested version of remote_filecopy.py  
for pull request creation'
```

Contributing upstream

- Now, be sure to push the code to your forked repository using the following command:

```
$ git push
```



Contributing upstream

The screenshot shows a GitHub interface for a repository named "forked from ansible/ansible". The top navigation bar includes links for "Code", "Pull requests 0", "Projects 0", "Insights", and "Settings". Below the navigation is a search bar with the query "is:pr is:open", a "Filters" dropdown, and buttons for "Labels" and "Milestones". A prominent green button labeled "New pull request" is located on the right. The main content area features a large "Welcome to Pull Requests!" message with a gear icon above it. Below this, a paragraph explains the purpose of pull requests and provides a link to "create a pull request". At the bottom, a "ProTip!" section offers a keyboard shortcut for navigating back to the pull request listing.

forked from [ansible/ansible](#)

Code Pull requests 0 Projects 0 Insights Settings

Filters ▾ is:pr is:open Labels Milestones

New pull request

Welcome to Pull Requests!

Pull requests help you collaborate on code with other people. As pull requests are created, they'll appear here in a searchable and filterable list. To get started, you should [create a pull request](#).

💡 ProTip! Type **g** **p** on any issue or pull request to go back to the pull request listing page.

[Code](#)[Issues 3,871](#)[Pull requests 1,687](#)[Projects 25](#)[Insights](#)**First time contributing to ansible/ansible?**[Dismiss](#) ...

If you know how to fix an [issue](#), consider opening a pull request for it.
You can read this repository's [contributing guidelines](#) to learn how to
open a good pull request.

[Filters](#) ▾ is:pr is:open[Labels 338](#)[Milestones 1](#)[New pull request](#)[1,687 Open](#)[Author](#) ▾[Labels](#) ▾[Projects](#) ▾[Milestones](#) ▾[Reviews](#) ▾[Assignee](#) ▾[Sort](#) ▾

 **The module fails on switchport. Check added to fix.** ✓ [affects_2.8](#) [bug](#) [core_review](#) [module](#) [needs_triage](#) [networking](#) [support:community](#) [support:core](#) [test](#)

1

#54970 opened 8 minutes ago by amuraleedhar

 **filters: Add additional Truth values to bool filter** ✗ [affects_2.8](#) [feature](#) [needs_revision](#) [needs_triage](#) [small_patch](#) [support:community](#) [support:core](#)

#54969 opened 38 minutes ago by Akasurde

 **Fix handling of inventory and credential options for tower_job_launch** ✓ [affects_2.8](#) [bug](#) [core_review](#) [module](#) [needs_triage](#) [support:community](#) [support:core](#) [test](#) [tower](#) [web_infrastructure](#)

1

#54967 opened 2 hours ago by saito-hideki

 **WIP: Add encoding and codepage params to win_command/win_shell (#54896)** ✗ [WIP](#) [affects_2.8](#) [feature](#) [module](#) [needs_revision](#) [needs_triage](#) [support:community](#) [support:core](#) [windows](#)

2

#54966 opened 3 hours ago by h-hirokawa

 **WIP - Add unit testing with Pester for PowerShell modules** ✗ [WIP](#) [affects_2.8](#) [feature](#) [module](#) [needs_triage](#) [support:community](#) [support:core](#) [test](#) [windows](#)

1

#54965 opened 4 hours ago by jborean93

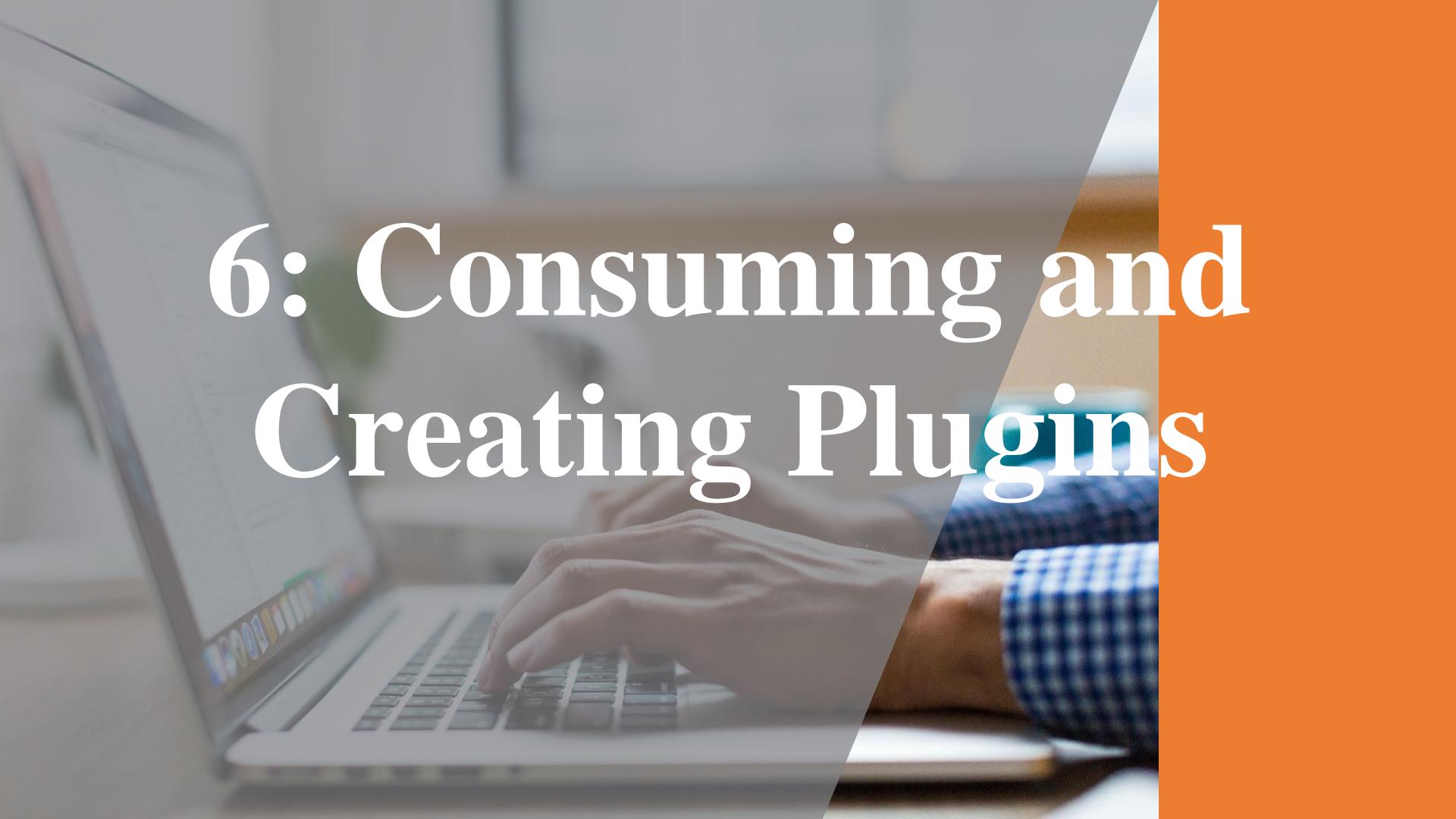


Summary

- Modules are the very lifeblood of Ansible – without them, Ansible could not perform all of the complex and varied tasks it performs so well across a wide variety of systems.
- By virtue of being an open-source project, it is incredibly easy to extend the functionality of Ansible by yourself, and in this lesson, we explored how you can, with a little Python knowledge, write your own custom module from scratch.

"Complete Lab 5"

6: Consuming and Creating Plugins

A photograph showing a person's hands typing on a silver laptop keyboard. The laptop screen is visible on the left, displaying some code or interface. In the background, there's a blurred view of an office environment with a monitor and other office equipment. The overall image has a professional and technical feel.

Consuming and Creating Plugins

In this lesson, we will cover the following topics:

- Discovering the plugin types
- Finding the included plugins
- Creating custom plugins

Technical requirements

- This lesson assumes that you have set up your control host with Ansible, as detailed in lesson 1, Getting Started with Ansible, and that you are using the most recent version available.
- The examples in this lesson are tested with Ansible 2.9.



Discovering the plugin types

- As ever, let's validate the presence of a suitably installed version of Ansible on your test machine before proceeding further:

```
$ ansible-doc --version
ansible-doc 2.9.6
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/root/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible-doc
  python version = 2.7.5 (default, Aug 7 2019, 00:51:29) [GCC 4.8.5
20150623 (Red Hat 4.8.5-39)]
```

Discovering the plugin types

- To use the ansible-doc command to list all the plugins available in a given category, you can run the following command:

```
$ ansible-doc -t connection -l
```



Discovering the plugin types

- This will return a textual index of the connection plugins, like what we saw when we were looking at the module documentation.
- The first few lines of the index output are shown here:

Refer to the file 6_1.txt



Discovering the plugin types

- You can then explore the documentation for a given plugin.
- For example, if we want to learn about the paramiko_ssh plugin, we can issue the following command:

```
$ ansible-doc -t connection paramiko_ssh
```

- You will find that the plugin documentation takes on a very familiar format, similar to what we saw for the modules in lesson 5, Consuming and Creating Modules:

Refer to the file 6_2.txt

Finding included plugins

- If you installed Ansible on a Linux system using a package manager (that is, via an RPM or DEB package), then the location of your plugins will depend on your OS.
- For example, on my test CentOS 7 system where I installed Ansible from the official RPM package, I can see the plugins installed here:

Refer to the file 6_3.txt



Finding included plugins

- If we want to look up the paramiko_ssh plugin that we reviewed the documentation of in the preceding section, we can look in the connection/ subdirectory:

```
$ ls -l /usr/lib/python2.7/site-
packages/ansible/plugins/connection/paramiko_ssh.py
-rw-r--r-- 1 root root 23544 Mar 5 05:39
/usr/lib/python2.7/site-
packages/ansible/plugins/connection/paramiko_ssh.py
```

Finding included plugins

- Clone the official Ansible repository from GitHub, as we did previously, and change the directory to the location of your clone:

```
$ git clone https://github.com/ansible/ansible.git
```

```
$ cd ansible
```

- Within the official source code directory structure, you will find that the plugins are all contained (again, in categorized subdirectories) under lib/ansible/plugins/:

```
$ cd lib/ansible/plugins
```

Finding included plugins

- We can explore the connection-based plugins by looking in the connection directory:

```
$ ls -al connection/
```

- The exact contents of this directory will depend on the version of Ansible source code that you have cloned.
- At the time of writing, it looks as follows, with one Python file for each plugin

Refer to the file 6_4.txt



Finding included plugins

- You can review the contents of each plugin to learn more about how they work, which is again part of the beauty of open source software:

```
$ less connection/paramiko_ssh.py
```

- An example of the beginning of this file is shown in the following code block to give you an idea of the kind of output you should be seeing if this command runs correctly:

Refer to the file 6_5.txt

Creating custom plugins

- On Fedora, you can run the following command to install the required packages:

```
$ sudo dnf install python python-devel
```

- Similarly, on CentOS, you can run the following command to install the required packages:

```
$ sudo yum install python python-devel
```

Creating custom plugins

- On Ubuntu, you can run the following commands to install the packages you will need:

```
$ sudo apt-get update
```

```
$ sudo apt-get install python-pip python-dev build-essential
```

- If you are working on macOS and using the Homebrew packaging system, the following command will install the packages you need:

```
$ sudo brew install python
```

Creating custom plugins

- Once you have installed the required packages, you will need to clone the Ansible Git repository to your local machine, as there are some valuable scripts in there that we will need later in the module development process.
- Use the following command to clone the Ansible repository to your current directory on your development machine:

```
$ git clone https://github.com/ansible/ansible.git  
$ cd ansible
```

Creating custom plugins

- Start your plugin file with a header so that people will know who wrote the plugin and what license it is released under.
- Naturally, you should update both the copyright and license fields with values appropriate to your plugin, but the following text is given as an example for you to get started with:

```
# (c) 2020, James Freeman <james.freeman@example.com>
# GNU General Public License v3.0+ (see COPYING or
https://www.gnu.org/licenses/gpl-3.0.txt)
```

Creating custom plugins

- Next, we'll add a very simple Python function—yours can be as complex as you want it to be, but for ours, we will simply use the Python `.replace` function to replace one string with another inside a string variable.
- The following example looks for instances of Puppet and replaces them with Ansible:

```
def improve_automation(a):
    return a.replace("Puppet", "Ansible")
```

Creating custom plugins

- Within this object, we can create a filters definition and return the value of our previously defined filter function to Ansible:

```
class FilterModule(object):
    "improve_automation filters"
    def filters(self):
        return {'improve_automation':
improve_automation}
```



Creating custom plugins

```
- name: Play to demonstrate our custom filter
  hosts: frontends
  gather_facts: false
  vars:
    statement: "Puppet is an excellent automation tool!"
```

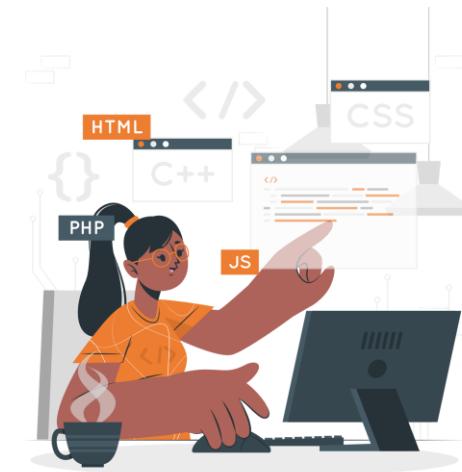
tasks:

```
  - name: make a statement
    debug:
      msg: "{{ statement | improve_automation }}"
```

Creating custom plugins

- Your directory tree structure, when you have finished coding the various file details in the preceding code block, should look something like this:

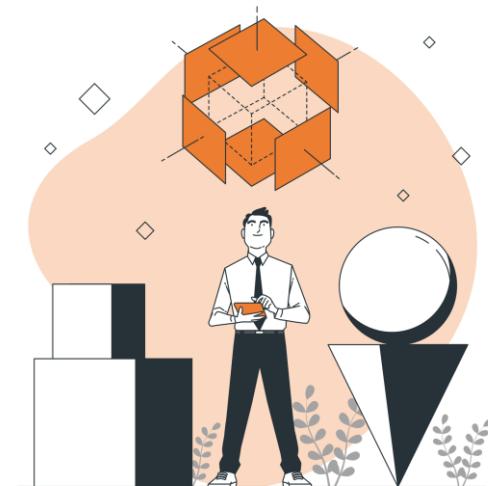
```
filter_plugins
├── custom_filter.py
hosts
myplugin.yml
```



Creating custom plugins

- Let's now run our little test playbook and see what output we get. If all goes well, it should look something like the following:

Refer to the file 6_6.txt



Creating custom plugins

- Start by adding a header to the plugin file, as before, so that the maintainer and copyright details are clear.
- We are borrowing a large chunk of the original file.py lookup plugin code for our example, so it is important we include the relevant credit:

Refer to the file 6_7.txt



Creating custom plugins

- Next, add in the Python 3 headers—these are an absolute requirement if you intend to submit your plugin via a Pull Request (PR) to the Ansible project:

```
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type
```

- Next, add a DOCUMENTATION block to your plugin so that other users can understand how to interact with it:

Refer to the file 6_8.txt

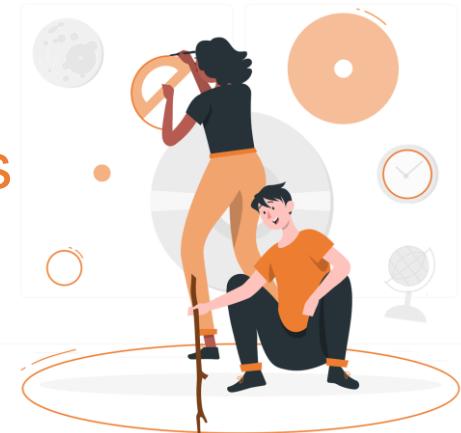
Creating custom plugins

- Add the relevant EXAMPLES blocks to show how to use your plugin, just as we did for modules:

```
EXAMPLES = """
```

```
- debug: msg="the first character in foo.txt is  
{{lookup('firstchar', '/etc/foo.txt') }}"
```

```
"""
```



Creating custom plugins

- Also, make sure you document the RETURN values from your plugin:

RETURN = """

_raw:

description:

- first character of content of file(s)

"""



Creating custom plugins

- We'll also set up the display object, which is used in verbose output and debugging.
- This should be used in place of the print statements in your plugin code if you need to display the debug output:

```
from ansible.errors import AnsibleError,  
AnsibleParserError  
from ansible.plugins.lookup import LookupBase  
from ansible.utils.display import Display  
display = Display()
```

Creating custom plugins

- We will now create an object of the `LookupModule` class.
- Define a default function within this called `run` (this is expected for the Ansible lookup plugin framework) and initialize an empty array for our return data:

```
class LookupModule(LookupBase):  
    def run(self, terms, variables=None, **kwargs):  
        ret = []
```



Creating custom plugins

- Within loop, we display valuable debugging information and, most importantly, define an object with the details of each of the files we will open, called `lookupfile`:

for term in terms:

```
    display.debug("File lookup term: %s" % term)
```

```
lookupfile = self.find_file_in_search_path(variables, 'files', term)
```

```
display.vvvv(u"File lookup using %s as file" % lookupfile)
```

Creating custom plugins

```
try:  
    if lookupfile:  
        contents, show_data =  
            self._loader._get_file_contents(lookupfile)  
            ret.append(contents.rstrip()[0])  
    else:  
        raise AnsibleParserError()  
except AnsibleParserError:  
    raise AnsibleError("could not locate file in lookup:  
%s" % term)
```

Creating custom plugins

- Finally, we return the character we gathered from the file to Ansible with a return statement:

```
return ret
```

- Once again, we can create a simple test playbook to test out our newly created plugin:

Refer to the file 6_9.txt

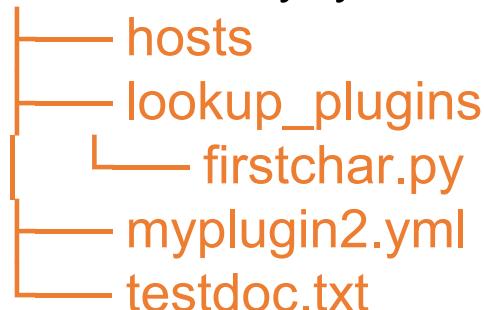


Creating custom plugins

- Create the text file referenced in the previous code block, called `testdoc.txt`.
- This can contain anything you like—mine contains the following simple text:

Hello

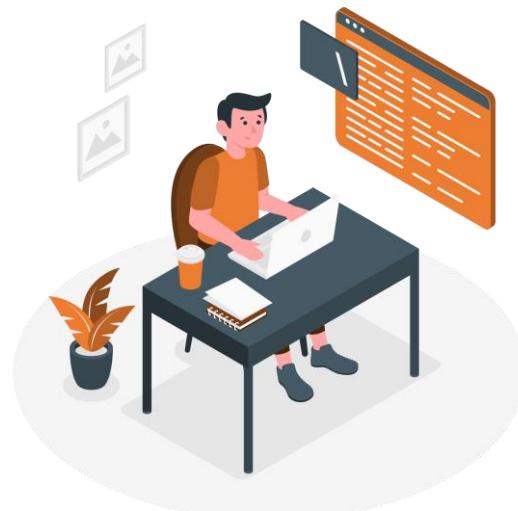
- For clarity, your final directory structure should look as follows:



Creating custom plugins

- Now, when we run our new playbook, we should see an output similar to the following:

Refer to the file 6_10.txt



Learning to integrate custom plugins

- Your first task will be to obtain a copy of the official Ansible project source code—for example, by cloning the GitHub repository to your local machine:

```
$ git clone https://github.com/ansible/ansible.git  
$ cd ansible
```



Learning to integrate custom plugins

- For example, our example filter would be copied to the following directory in the source code you just cloned:

```
$ cp ~/custom_filter.py ./lib/ansible/plugins/filter/
```

- Similarly, our custom lookup plugin would go in the lookup plugin's directory, using a command such as the following:

```
$ cp ~/firstchar.py ./lib/ansible/plugins/lookup/
```

Learning to integrate custom plugins

- With your code copied into place, you need to test the documentation (that is, whether your plugin includes it) as before.
- You can build the webdocs documentation in exactly the same way as we did in lesson 5, Consuming and Creating Modules, so we will not recap this here.
- However, as a refresher, we can quickly check whether the documentation renders correctly using the ansible-doc command, as follows:

Refer to the file 6_11.txt

Sharing plugins with the community

 [ansible / ansible](#)

Watch ▾ 2,047 Star 35,149 Fork 14,227

Code Issues 3,931 Pull requests 1,670 Projects 24 Insights

Ansible is a radically simple IT automation platform that makes your applications and systems easier to deploy. Avoid writing scripts or custom code to deploy and update your applications — automate in a language that approaches plain English, using SSH, with no agents to install on remote systems. <https://docs.ansible.com/ansible/> <https://www.ansible.com/>

[python](#) [ansible](#)

42,409 commits 43 branches 254 releases 4,137 contributors GPL-3.0

Branch: [devel](#) ▾ [New pull request](#) [Create new file](#) [Upload files](#) [Find file](#) [Clone or download](#) ▾

Author	Commit Message	Time Ago
 WojciechowskiPiotr and ansibot	docker_host_facts: Get system-wide information about docker host (#51373)	Latest commit e633b93 5 hours ago
	.github: Added new AIX and Gitlab members	23 hours ago
	bin: Save the command line arguments into a global context	a month ago
	changelogs: hashi_vault: add support for userpass authentication (#51538)	7 hours ago
	contrib: inventory: vagrant: rename deprecated ansible_ssh_* (#50694)	24 days ago
	docs: doc: Correct path of unit tests directory (#51631)	18 hours ago

Sharing plugins with the community

- Clone the devel branch that you just forked to your local machine.
- Use a command similar to the following, but be sure to replace the URL with one that matches your own GitHub account:

```
$ git clone https://github.com/<your GitHub account>/ansible.git
```



Sharing plugins with the community

- Once you've added your Python file, perform a git add command to make Git aware of the new file, and then commit it with a meaningful commit message. Some example commands are shown here:

```
$ cd ansible
```

```
$ cp ~/ansible-development/plugindev/firstchar.py  
./lib/ansible/plugins/lookup
```

```
$ git add lib/ansible/plugins/lookup/firstchar.py
```

```
$ git commit -m 'Added tested version of firstchar.py for  
pull request creation'
```

Sharing plugins with the community

- Now, be sure to push the code to your forked repository using the following command:

```
$ git push
```



forked from [ansible/ansible](#)

[Code](#)

[Pull requests 0](#)

[Projects 0](#)

[Insights](#)

[Settings](#)

[Filters ▾](#)

is:pr is:open

[Labels](#)

[Milestones](#)

[New pull request](#)



Welcome to Pull Requests!

Pull requests help you collaborate on code with other people. As pull requests are created, they'll appear here in a

[Code](#)[Issues 3,871](#)[Pull requests 1,687](#)[Projects 25](#)[Insights](#)**First time contributing to ansible/ansible?**[Dismiss](#) ...

If you know how to fix an [issue](#), consider opening a pull request for it.
You can read this repository's [contributing guidelines](#) to learn how to
open a good pull request.

[Filters](#) ▾ is:pr is:open[Labels 338](#)[Milestones 1](#)[New pull request](#)[1,687 Open](#) ✓ 30,980 Closed[Author](#) ▾[Labels](#) ▾[Projects](#) ▾[Milestones](#) ▾[Reviews](#) ▾[Assignee](#) ▾[Sort](#) ▾

The module fails on switchport. Check added to fix. ✓ [affects_2.8](#) [bug](#) [core_review](#) [module](#)
[needs_triage](#) [networking](#) [support:community](#) [support:core](#) [test](#)

#54970 opened 8 minutes ago by amuraleedhar

[1](#)

filters: Add additional Truth values to bool filter ✗ [affects_2.8](#) [feature](#) [needs_revision](#) [needs_triage](#)
[small_patch](#) [support:community](#) [support:core](#)

#54969 opened 38 minutes ago by Akasurde

[1](#)

Fix handling of inventory and credential options for tower_job_launch ✓ [affects_2.8](#) [bug](#)
[core_review](#) [module](#) [needs_triage](#) [support:community](#) [support:core](#) [test](#) [tower](#) [web_infrastructure](#)

#54967 opened 2 hours ago by saito-hideki

[1](#)

WIP: Add encoding and codepage params to win_command/win_shell (#54896) ✗ [WIP](#)
[affects_2.8](#) [feature](#) [module](#) [needs_revision](#) [needs_triage](#) [support:community](#) [support:core](#) [windows](#)

#54966 opened 3 hours ago by h-hirokawa

[2](#)

WIP - Add unit testing with Pester for PowerShell modules ✗ [WIP](#) [affects_2.8](#) [feature](#) [module](#)
[needs_triage](#) [support:community](#) [support:core](#) [test](#) [windows](#)

#54965 opened 4 hours ago by jborean93

[1](#)



Summary

- Ansible plugins are a core part of Ansible's functionality, and we discovered, in this lesson, that we have been working with them throughout this course without even realizing it!
- Ansible's modular design makes it easy to extend and add functionality to, regardless of whether you are working with modules or the various types of plugins that are currently supported.

"Complete Lab 6"

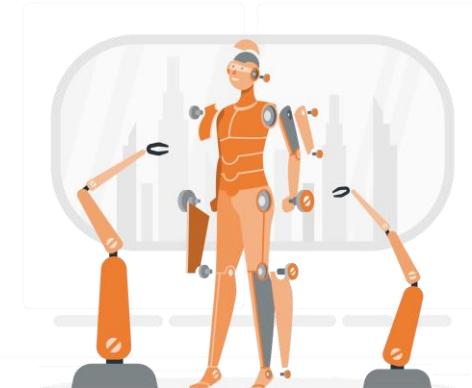
7: Coding Best Practices

A photograph of a person's hands typing on a silver laptop keyboard. The background is blurred, showing a desk with a computer monitor and some papers. The overall image has a professional, tech-oriented feel.

Coding Best Practices

In this lesson, we will cover the following topics:

- The preferred directory layout
- The best approach to cloud inventories
- Differentiating between different environment types
- The proper approach to defining group and host variables
- Using top-level playbooks
- Leveraging version control tools
- Setting OS and distribution variances
- Porting between Ansible versions



Technical requirements

- This lesson assumes that you have set up your control host with Ansible, as in lesson 1, Getting Started with Ansible, and that you are using the most recent version available; the examples in this lesson were tested on Ansible 2.9.
- This lesson also assumes that you have at least one additional host to test against; ideally, this should be Linux-based.

The preferred directory layout

Let's get started by building the directory structure:

- Create a directory tree for your development inventory with the following commands:

```
$ mkdir -p inventories/development/group_vars  
$ mkdir -p inventories/development/host_vars
```



- Next, we'll define an INI-formatted inventory file for our development inventory—in our example, we'll keep this simple with just two servers.
- The file to create is inventories/development/hosts:

[app]

app01.dev.example.com

app02.dev.example.com

- To further our example, we'll add a group variable to our app group, As discussed in lesson 3, Defining Your Inventory, create a file called app.yml in the group_vars directory we created in the previous step:

http_port: 8080

The preferred directory layout

- Next, create a production directory structure using the same method:

```
$ mkdir -p inventories/production/group_vars
```

```
$ mkdir -p inventories/production/host_vars
```

- Create an inventory file called hosts in the newly created production directory with the following contents:

```
[app]
```

```
app01.prod.example.com
```

```
app02.prod.example.com
```

The preferred directory layout

- Now, we'll define a different value to the http_port group variable for our production inventory.
- Add the following contents to inventories/production/group_vars/app.yml:

```
---
```

```
http_port: 80
```



The preferred directory layout

- Suppose we want to use the `remote_filecopy.py` module we created in lesson 5, Consuming and Creating Modules.
- Just as we discussed in this lesson, we first create the directory for this module:

```
$ mkdir library
```



The preferred directory layout

- The same can be done for the plugins; if we also want to use our filter plugin that we created in lesson 6, Consuming and Creating Plugins
- We would create an appropriately named directory:

```
$ mkdir filter_plugins
```



The preferred directory layout

- We'll call our role `installapp` and use the `ansible-galaxy` command (covered in lesson 4, Playbooks and Roles) to create the directory structure for us:

```
$ mkdir roles
```

```
$ ansible-galaxy role init --init-path roles/ installapp
```

```
- Role installapp was created successfully
```

The preferred directory layout

- Then, in our roles/installapp/tasks/main.yml file, we'll add the following contents:

Refer to the file 7_1.txt



The preferred directory layout

- The final stage in creating our best practice directory structure is to add a top-level playbook to run.
- By convention, this will be called site.yml and it will have the following simple contents :

```
- name: Play using best practise directory structure
  hosts: all
  roles:
    - installapp
```

The preferred directory layout

- For the purpose of clarity, your resulting directory structure should look as follows:

Refer to the file 7_2.txt



The preferred directory layout

- Now, we can simply run our playbook in the normal manner.

Refer to the file 7_3.txt

- For example, to run it on the development inventory, execute the following:

Refer to the file 7_4.txt



Differentiating between different environment types

We won't repeat the examples, but it's important to note that when working with multiple environments, your goals should be as follows:

- Try and reuse the same playbooks for all of your environments that run the same code.
- For example, if you deploy a web app in your development environment, you should be confident that your playbooks will deploy the same app in the production environment.

Differentiating between different environment types

- This means that not only are you testing your application deployments and code, but you are also testing your Ansible playbooks and roles as part of your overall testing process.
- Your inventories for each environment should be kept in separate directory trees, but all roles, playbooks, plugins, and modules (if used) should be in the same directory structure (this should be the case for both environments).

Proper approach to defining group and host variables

- Host variables are always of a higher order of precedence than group variables; so, you can override any group variable with a host variable.
- This behavior is useful if you take advantage of it in a controlled manner but can yield unexpected results if you are not aware of it.
- There is a special group variables definition called all, which is applied to all inventory groups.
- This has a lower order of precedence than specifically defined group variables.

Proper approach to defining group and host variables

- Create an inventory directory structure with the following commands:

```
$ mkdir -p inventories/development/group_vars
```

```
$ mkdir -p inventories/development/host_vars
```

- Create a simple inventory file with two hosts in a single group in the inventories/development/hosts file; the contents should be as follows:

```
[app]
app01.dev.example.com
app02.dev.example.com
```



Proper approach to defining group and host variables

- Now, let's create a special group variable file for all the groups in the inventory; this file will be called inventories/development/group_vars/all.yml and should contain the following content:

```
http_port: 8080
```

Proper approach to defining group and host variables

- Finally, let's create a simple playbook called site.yml to query and print the value of the variable we just created:

```
- name: Play using best practise directory structure  
hosts: all
```

tasks:

```
- name: Display the value of our inventory variable  
  debug:  
    var: http_port
```

Proper approach to defining group and host variables

- Now, if we run this playbook, we'll see that the variable (which we only defined in one place) takes the value we would expect:

Refer to the file 7_7.txt



Proper approach to defining group and host variables

- Now, let's add a new file to our inventory directory structure, with the all.yml file remaining unchanged.
- Let's also create a new file located in inventories/development/group_vars/app.yml, which will contain the following content:

http_port: 8081

Proper approach to defining group and host variables

- We have now defined the same variable twice—once in a special group called all and once in the app group (which both servers in our development inventory belong to).
- What happens if we now run our playbook? The output should appear as follows:

Refer to the file 7_8.txt

Proper approach to defining group and host variables

- To complete this example, we'll create a child group, called centos, and another group that could notionally contain hosts built to a new build standard, called newcentos, which both application servers will be a member of. This means modifying inventories/development/hosts so that it now looks as follows:

```
[app]
```

```
app01.dev.example.com
```

```
app02.dev.example.com
```

```
[centos:children]
```

```
app
```

```
[newcentos:children]
```

```
app
```



Proper approach to defining group and host variables

- Now, let's redefine the http_port variable for the centos group by creating a file called inventories/development/group_vars/centos.yml, which contains the following content:

```
http_port: 8082
```

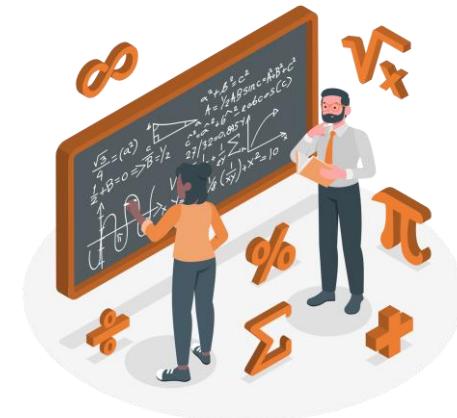
- Just to add to the confusion, let's also define this variable for the newcentos group in inventories/development/group_vars/newcentos.yml, which will contain the following content:

```
http_port: 8083
```

Proper approach to defining group and host variables

- We've now defined the same variable four times at the group level!
- Let's rerun our playbook and see which value comes through:

Refer to the file 7_9.txt



Proper approach to defining group and host variables

- Just for completeness, we can override all of this by leaving the group_vars directory untouched, but adding a file called inventories/development/host_vars/app01.dev.example.com.yml, which will contain the following content:

```
---
```

```
http_port: 9090
```



Proper approach to defining group and host variables

- Now, if we run our playbook one final time, we will see that the value we defined at the host level completely overrides any value that we set at the group level for app01.dev.example.com.
- app02.dev.example.com is unaffected as we did not define a host variable for it, so the next highest level of precedence—the group variable from the newcentos group—won:

Refer to the file 7_10.txt

Using top-level playbooks

- In all of the examples so far, we have built out using the best practice directory structure recommended by Ansible and continually referred to a top-level playbook, typically called site.yml.
- The idea behind this playbook, and, indeed, its common name across all of our directory structures, is so that it can be used across your entire server estate—that is to say, your site.

Leveraging version control tools

- In addition to this, you will need to install the command-line Git tools on your Linux host.
- On CentOS, you would install these as follows:

```
$ sudo yum install git
```

- On Ubuntu, the process is similarly straightforward:

```
$ sudo apt-get update
```

```
$ sudo apt-get install git
```



Leveraging version control tools

- Clone your git repository to your local machine to create a working copy using a command such as the following:

```
$ git clone
```

```
https://github.com/<YOUR_GIT_ACCOUNT>/<GIT_REPO>.git  
Cloning into '<GIT_REPO>'...
```

```
remote: Enumerating objects: 7, done.
```

```
remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 7
```

```
Unpacking objects: 100% (7/7), done.
```

Leveraging version control tools

- Change to the directory of the code you cloned (the working copy) and make any code changes you need to make:

```
$ cd <GIT_REPO>
```

```
$ vim myplaybook.yml
```

- Be sure to test your code and, when you are happy with it, add the changed files that are ready for committing a new version using a command such as the following:

```
$ git add myplaybook.yml
```

Leveraging version control tools

- The next step is to commit the changes you have made.
- A commit is basically a new version of code within the repository, so it should be accompanied by a meaningful commit message (specified in quotes after the -m switch), as follows:

Refer to the file 7_11.txt

Leveraging version control tools

- Right now, all these changes live solely in the working copy on your local machine.
- This is good by itself, but it would be better if the code was available to everyone who needs to view it on the version control system.
- To push your updated commits back to (for example) GitHub, run the following command:

Refer to the file 7_12.txt

Leveraging version control tools

- Now, other collaborators can clone your code just as we did in step 1.
- Alternatively, if they already have a working copy of your repository, they can update their working copy using the following command (you can also do this if you want to update your working copy to see changes made by someone else):

\$ git pull

Setting OS and distribution variances

- Assume that we are using the following simple inventory file for this example, which has two hosts in a single group called app:

```
[app]  
app01.dev.example.com  
app02.dev.example.com
```



Setting OS and distribution variances

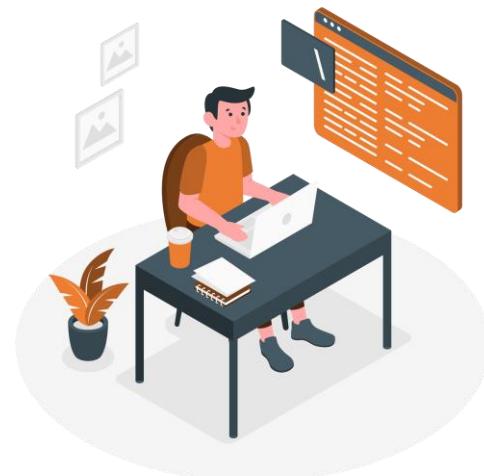
- Start by creating a new playbook—we'll call it `osvariants.yml`—with the following Play definition.
- It will also contain a single task, as shown:

```
- name: Play to demonstrate group_by module
hosts: all
tasks:
  - name: Create inventory groups based on host facts
    group_by:
      key: os_{{ ansible_facts['distribution'] }}
```

Setting OS and distribution variances

- Armed with this information, we can go ahead and create additional plays based on the newly created groups.
- Let's add the following Play definition to the same playbook file to install Apache on CentOS:

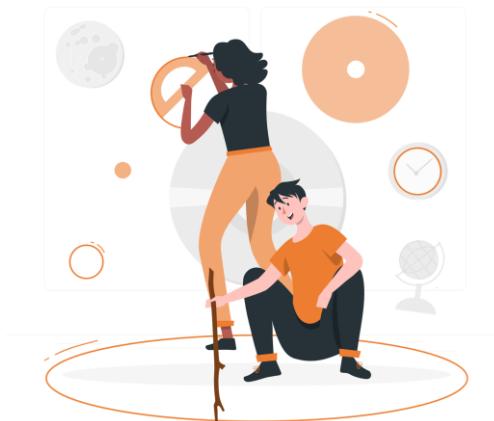
```
- name: Play to install Apache on CentOS
hosts: os_CentOS
become: true
tasks:
  - name: Install Apache on CentOS
    yum:
      name: httpd
      state: present
```



Setting OS and distribution variances

- We can, similarly, add a third Play definition, this time for installing the apache2 package on Ubuntu using the apt module:

```
- name: Play to install Apache on Ubuntu
hosts: os_Ubuntu
become: true
tasks:
  - name: Install Apache on Ubuntu
    apt:
      name: apache2
      state: present
```



Setting OS and distribution variances

- If our environment is based on CentOS servers and we run this playbook, the results are as follows:

Refer to the file 7_13.txt



Porting between Ansible versions

- Ansible is a fast-moving project, and with releases and new features added, new modules (and module enhancements) are released and the inevitable bugs that come with the software are fixed.
- There is no doubt that you will end up writing your code against one version of Ansible only to need to run it on a newer version again at some point.

Porting between Ansible versions

- The first part of the answer is to establish which version of Ansible you are starting from.
- For example, let's say you are preparing for the release of Ansible 2.10. If you query the version of Ansible you already have installed and see something like the following, then you know you are starting from Ansible release 2.9:



Refer to the file 7_14.txt

Porting between Ansible versions

- As you can see in the preceding link, there are a great number of changes between the 2.9 and 2.10 releases of Ansible.
- To that end, it's also important to note that the porting guides are written from the perspective of an upgrade from the previous major release.
- If you query your Ansible version and it returns the following, you are porting from Ansible 2.8:

Refer to the file 7_15.txt

Summary

- Ansible automation projects often start out small, but as people come to realize the power and simplicity of Ansible, both the code and the inventories tend to grow at an exponential pace (at least in my experience).
- It is important that in the push for greater automation, the Ansible automation code and infrastructure itself doesn't become another headache.



"Complete Lab 7"

8: Advanced Ansible Topics

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or professional environment.

Advanced Ansible Topics

In this lesson, we will cover the following topics:

- Asynchronous versus synchronous actions
- Controlling play execution for rolling updates
- Configuring the maximum failure percentage
- Setting task execution delegation
- Using the run_once option
- Running playbooks locally
- Working with proxies and jump hosts etc.

Technical requirements

- This lesson assumes that you have set up your control host with Ansible, as detailed in lesson 1, Getting Started with Ansible, and are using the most recent version available.
- The examples in this lesson are tested with Ansible 2.9 & This lesson also assumes that you have at least one additional host to test against and ideally, this should be Linux-based.

Asynchronous versus synchronous actions

- let's explore this through a practical example.
Suppose we have two servers in a simple INI-formatted inventory:

[frontends]

frt01.example.com

frt02.example.com



Asynchronous versus synchronous actions

- Run with the SSH connection blocked for the duration of the sleep command, we'll add two special parameters to the task, as shown:

```
- name: Play to demonstrate asynchronous tasks
  hosts: frontends
  become: true
  tasks:
    - name: A simulated long running task
      shell: "sleep 20"
      async: 30
      poll: 5
```

Asynchronous versus synchronous actions

- When you run this playbook, you will find that it runs just like any other playbook; from the terminal output, you won't be able to see any difference.
- But behind the scenes, Ansible checks the task every 5 seconds until it succeeds or reaches the async timeout value of 30 seconds:

Refer to the file 8_1.txt

Asynchronous versus synchronous actions

- If you want to check on the task later (that is, if poll is set to 0), you could add a second task to your playbook so that it looks as follows:

Refer to the file 8_2.txt



Asynchronous versus synchronous actions

- When using these in a playbook, you almost certainly wouldn't add the two tasks back-to-back like this—usually, you would perform additional tasks in between them—but to keep this example simple, we will run the two tasks sequentially.
- Running this playbook should yield an output similar to the following:

Refer to the file 8_3.txt

Control play execution for rolling updates

- If you run the date command using the command module, you will be able to see the time that each task is run, as well as if you specify -v to increase the verbosity when you run the play:

```
- name: Simple serial demonstration play
  hosts: frontends
  gather_facts: false
  tasks:
    - name: First task
      command: date
    - name: Second task
      command: date
```

Control play execution for rolling updates

- Now, if you run this play, you will see that it performs all the operations on each host simultaneously, as we have fewer hosts than the default number of forks—5.
- This behavior is normal for Ansible, but not really what we want as our users will experience service outage:

Refer to the file 8_4.txt

Control play execution for rolling updates

- Now, let's modify the play definition, as shown.
- We'll leave the tasks sections exactly as they were in step 1:

```
- name: Simple serial demonstration play
  hosts: frontends
  serial: 1
  gather_facts: false
```

Control play execution for rolling updates

- Notice the presence of the serial: 1 line. This tells Ansible to complete the play on 1 host at a time before moving on to the next. If we run the play again, we can see this in action:

Refer to the file 8_5.txt

Control play execution for rolling updates

- You can even build on this by passing a list to the serial directive.
- Consider the following code:

serial:

- 1
- 3
- 5

Configuring the maximum failure percentage

- For our practical example, let's consider an expanded inventory with 10 hosts in it. We'll define this as follows:

[frontends]

frt[01:10].example.com

Configuring the maximum failure percentage

- Create the following play definition to demonstrate the use of the max_fail_percentage directive:

```
- name: A simple play to demonstrate use of
max_fail_percentage
hosts: frontends
gather_facts: no
serial: 5
max_fail_percentage: 50
```

Configuring the maximum failure percentage

- It should deliberately fail this task regardless of the result; otherwise, it should allow the task to run as normal:

tasks:

```
- name: A task that will sometimes fail  
  debug:  
    msg: This might fail
```

```
  failed_when: inventory_hostname in  
ansible_play_batch[0:3]
```



Configuring the maximum failure percentage

- Finally, we'll add a second task that will always succeed.
- This is run if the play can continue, but not if it is aborted:
 - name: A task that will succeed
 debug:
 msg: Success!

Configuring the maximum failure percentage

- Run the playbook and let's observe what happens:

Refer to the file 8_6.txt



Setting task execution delegation

- We still want to run our play across our entire inventory, we certainly don't want to run the load balancer commands from those hosts.
- Let's once again explain this in more detail with a practical example.
- We'll reuse the two simple host inventories that we used earlier in this lesson:

[frontends]

frt01.example.com

frt02.example.com



Setting task execution delegation

- For our example, let's create a script called `remove_from_loadbalancer.sh`, which will contain the following:

```
#!/bin/sh  
echo Removing $1 from load balancer...
```

- We will also create a script called `add_to_loadbalancer.sh`, which will contain the following:

```
#!/bin/sh  
echo Adding $1 to load balancer...
```

- We'll first create a very simple play definition (you are free to experiment with the serial and max_fail_percentage directives as you wish) and an initial task:

```
- name: Play to demonstrate task delegation
hosts: frontends
```

tasks:

```
  - name: Remove host from the load balancer
    command: ./remove_from_loadbalancer.sh {{
```

```
inventory_hostname }}
```

args:

```
  chdir: "{{ playbook_dir }}"
  delegate_to: localhost
```

Setting task execution delegation

- We add a task where the upgrade work is carried out. This task has no delegate_to directive, and so it is actually run on the remote host from the inventory (as desired):

```
- name: Deploy code to host
  debug:
    msg: Deployment code would go here....
```

Setting task execution delegation

- Finally, we add the host back to the load balancer using the second script we created earlier.
- This task is almost identical to the first:

```
- name: Add host back to the load balancer
  command: ./add_to_loadbalancer.sh {{ inventory_hostname }}
  args:
    chdir: "{{ playbook_dir }}"
  delegate_to: localhost
```

Setting task execution delegation

- Let's see this playbook in action:

Refer to the file 8_7.txt

Setting task execution delegation

- Wrapping this all up into a second example, our playbook will look as follows:

```
- name: Second task delegation example
hosts: frontends
```

tasks:

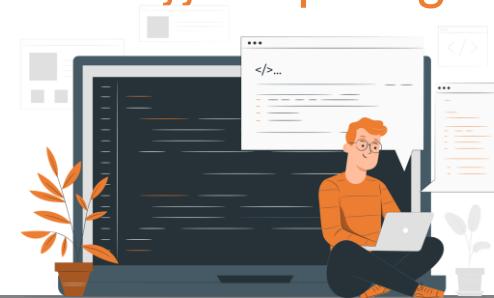
```
- name: Perform an rsync from localhost to inventory hosts
  local_action: command rsync -a /tmp/ {{
  inventory_hostname }}:/tmp/target/
```

Setting task execution delegation

- The preceding shorthand notation is equivalent to the following:

tasks:

```
- name: Perform an rsync from localhost to inventory hosts  
  command: rsync -a /tmp/ {{ inventory_hostname }}:/tmp/target/  
  delegate_to: localhost
```



Setting task execution delegation

- If we run this playbook, we can see that local_action does indeed run rsync from localhost, enabling us to efficiently copy whole directory trees across to remote servers in the inventory:

Refer to the file 8_8.txt

Using the run_once option

- Create the simple playbook as in the following code block.
- We're using a debug statement to display some output, but in real life, you would insert your script or command that performs your one-off cluster function here (for example, upgrading a database schema):

Refer to the file 8_9.txt

Using the run_once option

- Now, let's run this playbook and see what happens:

Refer to the file 8_10.txt



- It's important to note that the run_once option applies per batch of servers, so if we add serial: 5 to our play definition (running our play in two batches of 5 on our inventory of 10 servers), the schema upgrade task actually runs twice! It runs once as requested, but once per batch of servers, not once for the entire inventory.
- Be careful of this nuance when working with this directive in a clustered environment.
- Add serial: 5 to your play definition and rerun the playbook. The output should appear as follows:

Refer to the file 8_11.txt

Running playbooks locally

- Indeed, we can try creating a local inventory file with the following contents:

[local]

localhost

- Now, if we attempt to run the ping module in an ad hoc command against this inventory, we see the following:

Refer to the file 8_12.txt

Running playbooks locally

- We can now modify our inventory so that it looks as follows:

[local]

localhost ansible_connection=local



Running playbooks locally

- Ansible will not even attempt to connect to the remote host called frt01.example.com—it will connect locally to the machine running the playbook (without SSH):

[local]

frt01.example.com ansible_connection=local

- We can demonstrate this very simply. Let's first check for the absence of a test file in our local /tmp directory:

ls -l /tmp/foo

ls: cannot access /tmp/foo: No such file or directory

Running playbooks locally

- Now, let's run an ad hoc command to touch this file on all hosts in the new inventory we just defined:
Refer to the file 8_13.txt
- The command ran successfully, so let's see whether the test file is present on the local machine:

```
$ ls -l /tmp/foo  
-rw-r--r-- 1 root root 0 Apr 24 16:28 /tmp/fo
```

Working with proxies and jump hosts

- The configuration to support our bastion host is performed in the inventory, rather than in the playbook.
- We begin by defining an inventory group with the switches in, in the normal manner:

[switches]

cmcls01.example.com
cmcls02.example.com



Working with proxies and jump hosts

- We can now start to get clever by adding some special SSH arguments into the inventory variables for this group.
- Add the following code to your inventory file:

```
[switches:vars]
```

```
ansible_ssh_common_args='-o ProxyCommand="ssh -W %h:%p -q bastion.example.com"'
```

Working with proxies and jump hosts

- Now, if we attempt to run the Ansible ping module against this inventory, we can see whether it works:

Refer to the file 8_14.txt

Configuring playbook prompts

- Create a simple play definition in the usual manner, as follows:

```
- name: A simple play to demonstrate prompting in a
playbook
  hosts: frontends
```

Configuring playbook prompts

- One will be echoed to the screen, while the other won't be, by setting private: yes:

```
vars_prompt:
```

```
  - name: loginid
```

```
    prompt: "Enter your username"
```

```
    private: no
```

```
  - name: password
```

```
    prompt: "Enter your password"
```

```
    private: yes
```



Configuring playbook prompts

- We'll now add a single task to our playbook to demonstrate this prompting process of setting the variables:

tasks:

```
- name: Proceed with login
  debug:
    msg: "Logging in as {{ loginid }}..."
```

- Now, let's run the playbook and see how it behaves:
Refer to the file 8_15.txt

Placing tags in the plays and tasks

- Create the following simple playbook to perform two tasks—one to install the nginx package and the other to deploy a configuration file from a template:

Refer to the file 8_16.txt

Placing tags in the plays and tasks

- Now, let's run the playbook in the usual manner, but with one difference—this time, we'll add the --tags switch to the command line.
- This switch tells Ansible to only run the tasks that have tags matching the ones that are specified.
- So, for example, run the following command:

Refer to the file 8_17.txt

Placing tags in the plays and tasks

- There is also a --skip-tags switch that does the reverse of the previous switch—it tells Ansible to skip the tags listed.
- So, if we run the playbook again but skip the customize tag, we should see an output similar to the following:

Refer to the file 8_18.txt



Placing tags in the plays and tasks

- Some examples are provided for you in the following code block, based on the example playbook we just created:

Refer to the file 8_19.txt



Securing data with Ansible Vault

- Start by creating a new vault to store sensitive data in; we'll call this file secret.yml.
- You can create this using the following command:

```
$ ansible-vault create secret.yml
```

New Vault password:

Confirm New Vault password:

Securing data with Ansible Vault

- When you have entered the password, you will be set to your normal editor (defined by the EDITOR shell variable).
- On my test system, this is vi. Within this editor, you should create a vars file, in the normal manner, containing your sensitive data:

```
secretdata: "Ansible is cool!"
```

Securing data with Ansible Vault

- Save and exit the editor (press Esc, then :wq in vi). You will exit to the shell.
- Now, if you look at the contents of your file, you will see that they are encrypted and are safe from anyone who shouldn't be able to read the file:

```
$ cat secret.yml
$ANSIBLE_VAULT;1.1;AES256
6333373462376463386563323733166333634353334373862346334643631303163653931306138
6334356465396463643936323163323132373836336461370a343236386266313331653964326334
62363737663165336539633262366636383364343663396335643635623463626336643732613830
6139363035373736370a646661396464386364653935636366633663623261633538626230616630
35346465346430636463323838613037386636333334356265623964633763333532366561323266
3664613662643263383464643734633632383138363663323730
```

Securing data with Ansible Vault

- The great thing about Ansible Vault is that you can use this encrypted file in a playbook as if it were a normal variables file (although, obviously, you have to tell Ansible your vault password).
- Let's create a simple playbook as follows:

Refer to the file 8_20.txt

Securing data with Ansible Vault

- Try running the playbook without telling Ansible what the vault password is—in this instance, you should receive an error such as this:

```
$ ansible-playbook -i hosts vaultplaybook.yml  
ERROR! Attempting to decrypt but no vault secrets found
```

Securing data with Ansible Vault

- Ansible correctly understands that we are trying to load a variables file that is encrypted with ansible-vault, but we must manually tell it the password for it to proceed.
- There are a number of ways of specifying passwords for vaults (more on this in a minute), but for simplicity, try running the following command and enter your vault password when prompted:

Refer to the file 8_21.txt

Securing data with Ansible Vault

```
$ ansible-vault encrypt_string 'Ansible is cool!' --name secretdata
New Vault password:
Confirm New Vault password:
secretdata: !vault |
    $ANSIBLE_VAULT;1.1;AES256
    34393431303339353735656236656130336664666337363732376262343837663738393465623930
    336662306130636464396666565316235313136633264310a623736643362663035373861343435
    62346264313638656363323835323833633264636561366339326332356430383734653030306637
    3736336533656230380a316364313831666463643534633530393337346164356634613065396434
    33316338336266636666353334643865363830346566666331303763643564323065
Encryption successful
```

```
---
```

```
- name: A play that makes use of an Ansible Vault
  hosts: frontends
```

```
vars:
```

```
  secretdata: !vault |
    $ANSIBLE_VAULT;1.1;AES256
    34393431303339353735656236656130336664666337363732376262343837663738393465623930
    336662306130636464396666565316235313136633264310a623736643362663035373861343435
    62346264313638656363323835323833633264636561366339326332356430383734653030306637
    3736336533656230380a316364313831666463643534633530393337346164356634613065396434
    33316338336266636666353334643865363830346566666331303763643564323065
```

```
tasks:
```

```
  - name: Tell me a secret
    debug:
      msg: "Your secret data is: {{ secretdata }}"
```

Securing data with Ansible Vault

- Now, when you run this playbook in exactly the same manner as we did before (specifying the vault password using a user prompt), you should see that it runs just as when we used an external encrypted variables file:

Refer to the file 8_22.txt



Summary

- Ansible has many advanced features that allow you to run your playbooks in a variety of scenarios, whether that is upgrading a cluster of servers in a controlled manner; working with devices on a secure, isolated network; or controlling your playbook flow with prompts and tags.
- Ansible has been adopted by a large and ever-growing user base and, as such, is designed and evolved around solving real-world problems.



"Complete Lab 8"

9: Troubleshooting and Testing Strategies



Troubleshooting and Testing Strategies

There are a bunch of different ways to prevent or mitigate a bug in Ansible playbooks. In this lesson, we will cover the following topics:

- Digging into playbook execution problems
- Using host facts to diagnose failures
- Testing with a playbook
- Using check mode
- Solving host connection issues
- Passing working variables via the CLI etc.

Technical requirements

- This lesson assumes that you have set up your control host with Ansible, as detailed in lesson 1, Getting Started with Ansible, and are using the most recent version available – the examples in this lesson were tested with Ansible 2.9.
- Although we will give specific examples of hostnames in this lesson, you are free to substitute them with your own hostname and/or IP addresses.

Digging into playbook execution problems

- Sometimes, and this is particularly true for some modules, such as shell or command, the return code is non-zero, even though the execution was successful.
- In those cases, you can ignore the error by using the following line in your module:

```
ignore_errors: yes
```



Digging into playbook execution problems

- For instance, if you run the /bin/false command, it will always return 1.
- To execute this in a playbook so that you can avoid it blocking there, you can write something like the following:
 - name: Run a command that will return 1
command: /bin/false
ignore_errors: yes

Using host facts to diagnose failures

- To do so, we need to create a simple playbook, called `print_facts.yaml`, which contains the following content:

```
---
- hosts: target_host
  tasks:
    - name: Display all variables/facts known for a host
      debug:
        var: hostvars[inventory_hostname]
```

Testing with a playbook

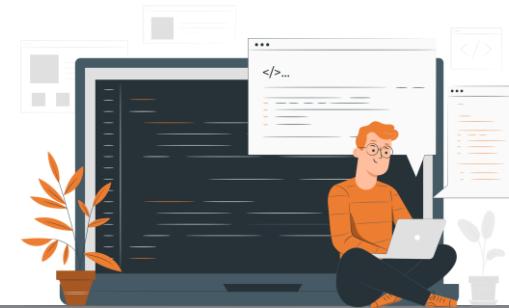
- First of all, we need a playbook called debug.yaml with the following content:

```
- hosts: localhost
  tasks:
    - shell: /usr/bin/uptime
      register: result
    - debug:
        var: result
```



Testing with a playbook

- Run it with the following command:
`$ ansible-playbook debug.yaml`
- You will receive an output like the following:
Refer to the file 10_1.txt



Testing with a playbook

- The debug module also provides the verbosity option.
- Let's say you change the playbook in the following way:

```
- hosts: localhost
  tasks:
    - shell: /usr/bin/uptime
      register: result
    - debug:
        var: result
        verbosity: 2
```



Testing with a playbook

- Now, if you try to execute it in the same way you did previously, you will notice that the debug step won't be executed and that the following line will appear in the output instead:

```
TASK [debug]
```

```
*****
```

```
*****
```

```
skipping: [localhost]
```

Testing with a playbook

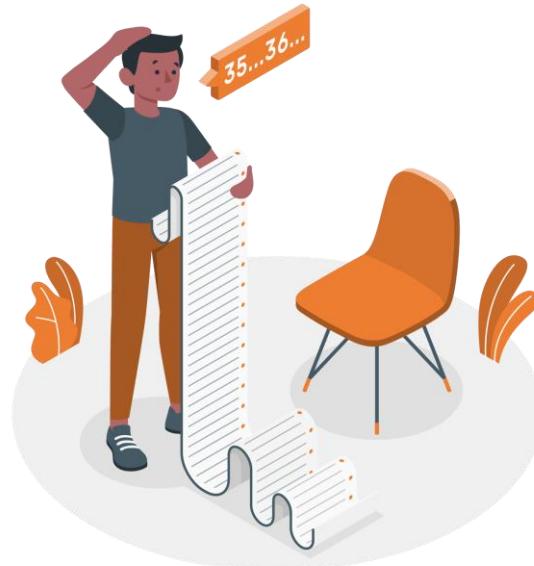
- To see the result of using the debug module with this new playbook, we will need to run a slightly different command:

```
$ ansible-playbook debug2.yaml -vv
```

Using check mode

- First, we need to create an easy playbook to test this feature.
- Let's create a playbook called check-mode.yaml that contains the following content:

```
- hosts: localhost
  tasks:
    - name: Touch a file
      file:
        path: /tmp/myfile
        state: touch
```



Using check mode

- Now, we can run the playbook in the check mode by specifying the --check option in the invocation:

```
$ ansible-playbook check-mode.yaml –check
```

- This will output everything as if it was really performing the operation, as follows:

Refer to the file 10_2.txt

Using check mode

- A similar feature to check mode is the --diff flag.
- What this flag allows us to do is track what exactly changed during an Ansible execution.
- So, let's say we run the same playbook with the following command:

```
$ ansible-playbook check-mode.yaml –diff
```

- This will return something like the following:
Refer to the file 10_3.txt

Solving host connection issues

- Let's create a playbook called `remote.yaml` with the following content:

```
---
```

```
- hosts: all
  tasks:
    - name: Touch a file
      file:
        path: /tmp/myfile
        state: touch
```



Solving host connection issues

- We can try to run the `remote.yaml` playbook against a non-existent FQDN, as follows:

```
$ ansible-playbook -i host.example.com, remote.yaml
```

- In this case, the output will clearly inform us that the SSH service did not reply in time:

Refer to the file `10_4.txt`

Solving host connection issues

- There is also the possibility that we'll receive a different error:

Refer to the file 10_5.txt



Solving host connection issues

- To investigate this further, you can try performing an SSH connection from the same machine to check if there are problems.
- For instance, I would do this like so:

```
$ ssh host.example.com -vvv
```

Solving host connection issues

- Another common case is that the username is wrong.
- To debug it, you can take the user@host address that is shown in the error (in my case, fale@host.example.com) and use the same command you used previously:

```
$ ssh fale@host.example.com -vvv
```

Passing working variables via the CLI

- The first thing we want to have is a simple playbook that prints the content of a variable.
- Let's create a playbook called printvar.yaml that contains the following content:

```
- hosts: localhost
  tasks:
    - debug:
        var: variable
```



Passing working variables via the CLI

- Now that we have an Ansible playbook that allows us to see if a variable has been set to what we were expecting, let's run it with variable declared in the execution statement:

```
$ ansible-playbook printvar.yaml --extra-vars='{"variable": "Hello, World!"}'
```

- By running this, we will receive an output similar to the following:

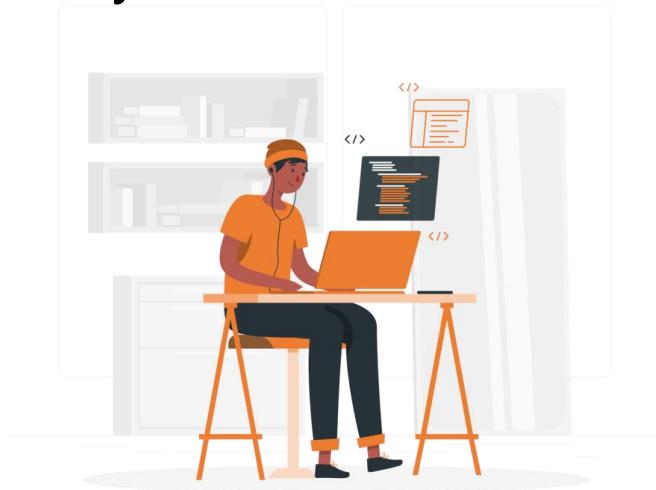
Refer to the file 10_6.txt

Limiting the host's execution

- To use the limitation of target hosts on Ansible, we will need a playbook.
- Create a playbook called helloworld.yaml that contains the following content:

```
---
```

```
- hosts: all
  tasks:
    - debug:
        msg: "Hello, World!"
```

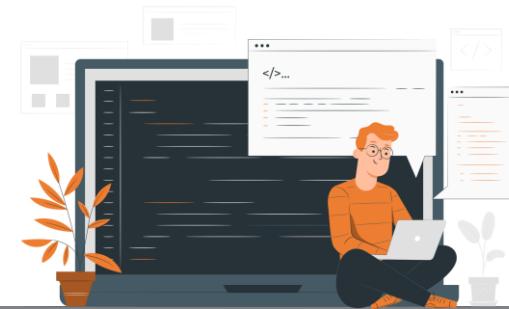


Limiting the host's execution

- We also need to create an inventory with at least two hosts.
- In my case, I created a file called inventory that contains the following content:

```
[hosts]
```

```
host1.example.com
host2.example.com
host3.example.com
```



Limiting the host's execution

- Let's run the playbook in the usual way with the following command:

```
$ ansible-playbook -i inventory helloworld.yaml
```

- By doing this, we will receive the following output:

Refer to the file 10_7.txt

Limiting the host's execution

- If we just want to run it against host3.example.com, we will need to specify this on the command line, as follows:

```
$ ansible-playbook -i inventory helloworld.yaml --  
limit=host3.example.com
```

- To prove that this works as expected, we can run it. By doing this, we will receive the following output:
Refer to the file 10_8.txt

Limiting the host's execution

- It's possible to specify multiple hosts as a list or with patterns, so both of the following commands will execute the playbook against host2.example.com and host3.example.com:

```
$ ansible-playbook -i inventory helloworld.yaml --  
limit=host2.example.com,host3.example.com
```

```
$ ansible-playbook -i inventory helloworld.yaml --  
limit=host[2-3].example.com
```

Limiting the host's execution

- let's say we limit to a host that is not part of the inventory, as follows:

```
$ ansible-playbook -i inventory helloworld.yaml --  
limit=host4.example.com
```

- Here, we will receive the following error, and nothing will be done:

```
[WARNING]: Could not match supplied host pattern, ignoring:  
host4.example.com  
ERROR! Specified hosts and/or --limit does not match any hosts
```

Flushing the code cache

- Flushing caches in Ansible is very straightforward, and it's enough to run `ansible-playbook`, which we are already running, with the addition of the `--flush-cache` option, as follows:

```
ansible-playbook -i inventory helloworld.yaml --flush-cache
```

Checking for bad syntax

- Let's create a `syntaxcheck.yaml` file with the following content:

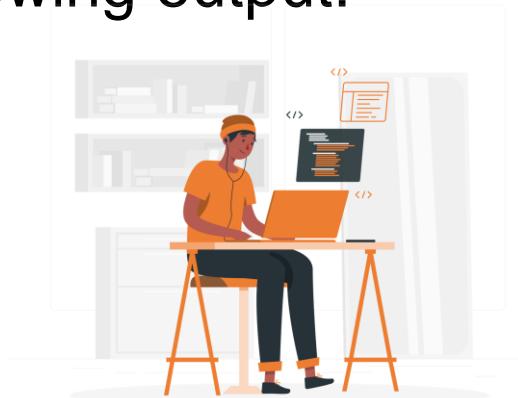
```
---
```

```
- hosts: all
  tasks:
    - debug:
        msg: "Hello, World!"
```



Checking for bad syntax

- Now, we can use the --syntax-check command:
`$ ansible-playbook syntaxcheck.yaml --syntax-check`
- By doing this, we will receive the following output:
Refer to the file 10_9.txt



Checking for bad syntax

- We can now proceed to fix the indentation problem on line 4:

```
- hosts: all
  tasks:
    - debug:
        msg: "Hello, World!"
```



Checking for bad syntax

- When we recheck the syntax, we will see that it now returns no errors:

```
$ ansible-playbook syntaxcheck-fixed.yaml --syntax-check
```

playbook: syntaxcheck.yaml

Summary

- In this lesson, you learned about the various options that Ansible provides so that you can look for problems in your Ansible code.
- More specifically, you learned how to use host facts to diagnose failures, how to include testing within a playbook, how to use check mode, how to solve host connection issues, how to pass variables from the CLI, how to limit the execution to a subset of hosts.

"Complete Lab 9"

Thankyou