Automation using Ansible

RED HAT CERTIFIED SPECIALIST Red Hat
Red Hat Certified SPECIALIST in Ansible Automation
Anand Nevase

Introduction

Ansible is an open source IT Configuration Management, Deployment & Orchestration tool. It aims to provide large productivity gains to a wide variety of automation challenges. This tool is very simple to use yet powerful enough to automate complex multi-tier IT application environments.

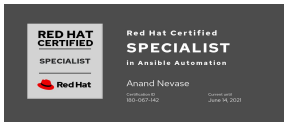
Why Do We Need Ansible?

Well before I tell you what is Ansible, it is of utmost importance to understand the problems that were faced before Ansible.

Let us take a little flashback to the beginning of networked computing when deploying and managing servers reliably and efficiently has been a challenge. Previously, system administrators managed servers by hand, installing software, changing configurations, and administering services on individual servers.

As data centers grew, and hosted applications became more complex, administrators realized they couldn't scale their manual systems management as fast as the applications they were enabling. It also hampered the velocity of the work of the developers since the development team was agile and releasing software frequently, but IT operations were spending more time configuring the systems. That's why server provisioning and configuration management tools came to flourish.

Consider the tedious routine of administering a server fleet. We always need to keep updating, pushing changes, copying files on them etc. These tasks make things very complicated and time consuming.



But let me tell you that there is a solution to the above stated problem. The solution is – *Ansible.*

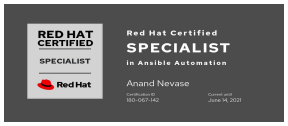
But before I go ahead to explain you all about Ansible, let me get you familiarized with few Ansible terminologies:

Ansible Terms:

- **Controller Machine:** The machine where Ansible is installed, responsible for running the provisioning on the servers you are managing.
- **Inventory:** An initialization file that contains information about the servers you are managing.
- **Playbook:** The entry point for Ansible provisioning, where the automation is defined through tasks using YAML format.
- **Task:** A block that defines a single procedure to be executed, e.g. Install a package.
- **Module:** A module typically abstracts a system task, like dealing with packages or creating and changing files. Ansible has a multitude of built-in modules, but you can also create custom ones.
- **Role:** A pre-defined way for organizing playbooks and other files in order to facilitate sharing and reusing portions of a provisioning.
- **Play:** A provisioning executed from start to finish is called a play. In simple words, execution of a playbook is called a play.
- **Facts:** Global variables containing information about the system, like network interfaces or operating system.
- **Handlers:** Used to trigger service status changes, like restarting or stopping a service.

Ansible is a helpful tool that allows you to create groups of machines, describe how these machines should be configured or what actions should be taken on them. Ansible issues all commands from a central location to perform these tasks.

No other client software is installed on the node machines. It uses SSH to connect to the nodes. Ansible only needs to be installed on the control machine (the machine from



which you will be running commands) which can even be your laptop. It is a simple solution to a complicated problem.

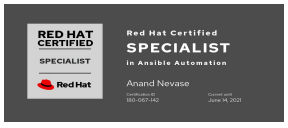
I am not boasting off when I say that Ansible has filled up all the holes in Configuration Management and IT Orchestration world. You will know it too, when you take a look at the benefits of Ansible mentioned below:

Advantages Of Using Ansible

Simple: Ansible uses a simple syntax written in YAML called *playbooks*. YAML is a human-readable data serialization language. It is extraordinarily simple. So, no special coding skills are required and even people in your IT organization, who do not know what is Ansible can likely read a playbook and understand what is happening. Ansible always executes tasks in order. It is simple to install too. Altogether the simplicity ensures that you can get started quickly.

Agentless: Finally, Ansible is completely agentless. There are no agents/software or additional firewall ports that you need to install on the client systems or hosts which you want to automate. You do not have to separately set up a management infrastructure which includes managing your entire systems, network and storage. Ansible further reduces the effort required for your team to start automating right away.

Powerful & Flexible: Ansible has powerful features that can enable you to model even the most complex IT workflows. In this aspect, Ansible's *batteries included approach* (This philosophy means that something is self-sufficient, comes out-of-the-box ready to use, with everything that is needed) can manage the infrastructure, networks, operating systems



and services that you are already using, as Ansible provides you with hundreds of modules to manage them. Together Ansible's capabilities allow you to orchestrate the entire application environment regardless of where it is deployed.

Efficient: No extra software on your servers means more resources for your applications. Also, since Ansible modules work via JSON, Ansible is extensible with modules written in a programming language you already know. Ansible introduces modules as basic building blocks for your software. So, you can even customize it as per your needs. For e.g. If you have an existing message sending module which sends messages in plain-text, and you want to send images too, you can add image sending features on top of it.

What Ansible Can Do?

Ansible is usually grouped along with other Configuration Management tools like Puppet, Chef, SaltStack etc. Well, let me tell you, Ansible is not just limited to Configuration Management. It can be used in many different ways too. I have mentioned some of them below:

Provisioning: Your apps have to live somewhere. If you're PXE (Preboot eXecution Environment) booting and kick starting bare-metal servers or Virtual Machines, or creating virtual or cloud instances from templates, Ansible & Ansible Tower helps to streamline this process. For example, if I want to test the debug version of an application that is built with Visual C++, I ought to meet some prerequisite requirements like having Visual C++ library DLLs (msvcr100d.dll). I will also need Visual Studio installed in your

computer. This is when Ansible makes sure that the required packages are downloaded and installed in order to provision my application.

Configuration Management: It establishes and maintains consistency of the product performance by recording and updating detailed information which describes an enterprise's hardware and software. Such information typically includes the versions and updates that have been applied to installed software packages and the locations and network addresses of hardware devices. For e.g. If you want to install the new version of Tomcat on all of the machines present in your enterprise, it is not feasible for you to manually go and update each and every machine. You can install Tomcat in one go on all of your machines with Ansible playbooks and inventory written in the most simple way. All you have to do is list out the IP addresses of your nodes in the inventory and write a playbook to install Tomcat. Run the playbook from your control machine & it will be installed on all your nodes.

Application Deployment: When you define your application with Ansible, and manage the deployment with Ansible Tower, teams are able to effectively manage the entire application life cycle from development to production. For example, let's say I want to deploy the Default Servlet Engine. There are a number of steps that needs to be undergone to deploy the engine.

- Move a .war application from dropins directory to apps directory
- Add server.xml file
- Navigate to the webpage to see your application.

But why worry about performing these steps one by one when we have a tool like Ansible. All you need to do is list these tasks in your Ansible playbook and sit back watching Ansible executing these tasks in order.

Security and Compliance: When you define your security policy in Ansible, scanning and remediation of site-wide security policy can be integrated into other automated processes. And it'll be integral in everything that is deployed. It means that, you need to configure your security details once in your control machine and it will be embedded in all other nodes automatically. Moreover, all the credentials (admin users id's & passwords) that are stored within Ansible are not retrievable in plain-text by any user.

Orchestration: Configurations alone don't define your environment. You need to define how multiple configurations interact and ensure the disparate pieces can be managed as a whole. Out of complexity and chaos, Ansible brings order. Ansible provides Orchestration in the sense of aligning the business request with the applications, data, and infrastructure. It defines the policies and service levels through automated workflows, provisioning, and change management. This creates an application-aligned infrastructure that can be scaled up or down based on the needs of each application.

For example, Consider the situation where I want to deploy a new website in place of my existing one. For that, we will remove the existing website, and deploy our new website, and restart the load balancer or the web cluster if needed. Now, if we just did something like this, users would notice downtime because we have not removed live traffic going to these machines via the load balancer. So, we need some type of pre-task, where we tell the load balancer to put this web server into maintenance mode,

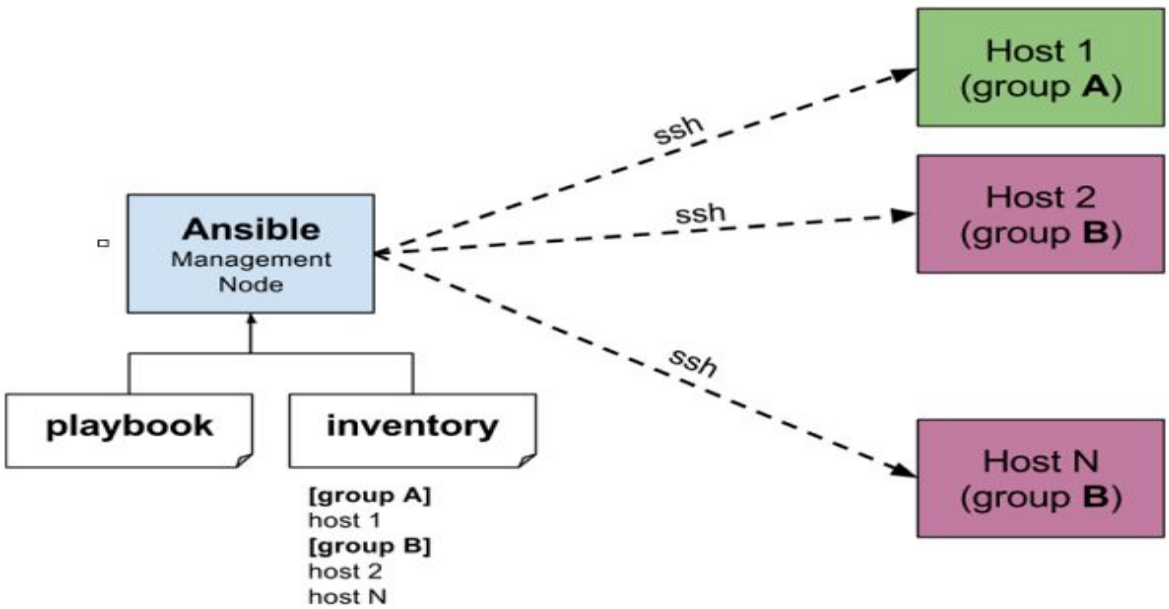
so that we can temporarily disable traffic from going to it, as it gets upgraded. Let's say, I added a block up here, that says a pre-task will be to disable web node in the load balancer.

So, this is our pre-task, where we disable traffic, then down here, we upgrade the node using these various tasks. Finally, we need some type of post-task, which will enable traffic to this web node again, by taking it out of maintenance mode. These tasks can be written in Ansible playbooks and hence it helps to orchestrate the environment.

How Ansible Works?

The picture given below shows the working of Ansible.

Ansible works by connecting to your nodes and pushing out small programs, called "Ansible modules" to them. Ansible then executes these modules (over SSH by default), and removes them when finished. Your library of modules can reside on any machine, and there are no servers, daemons, or databases required.

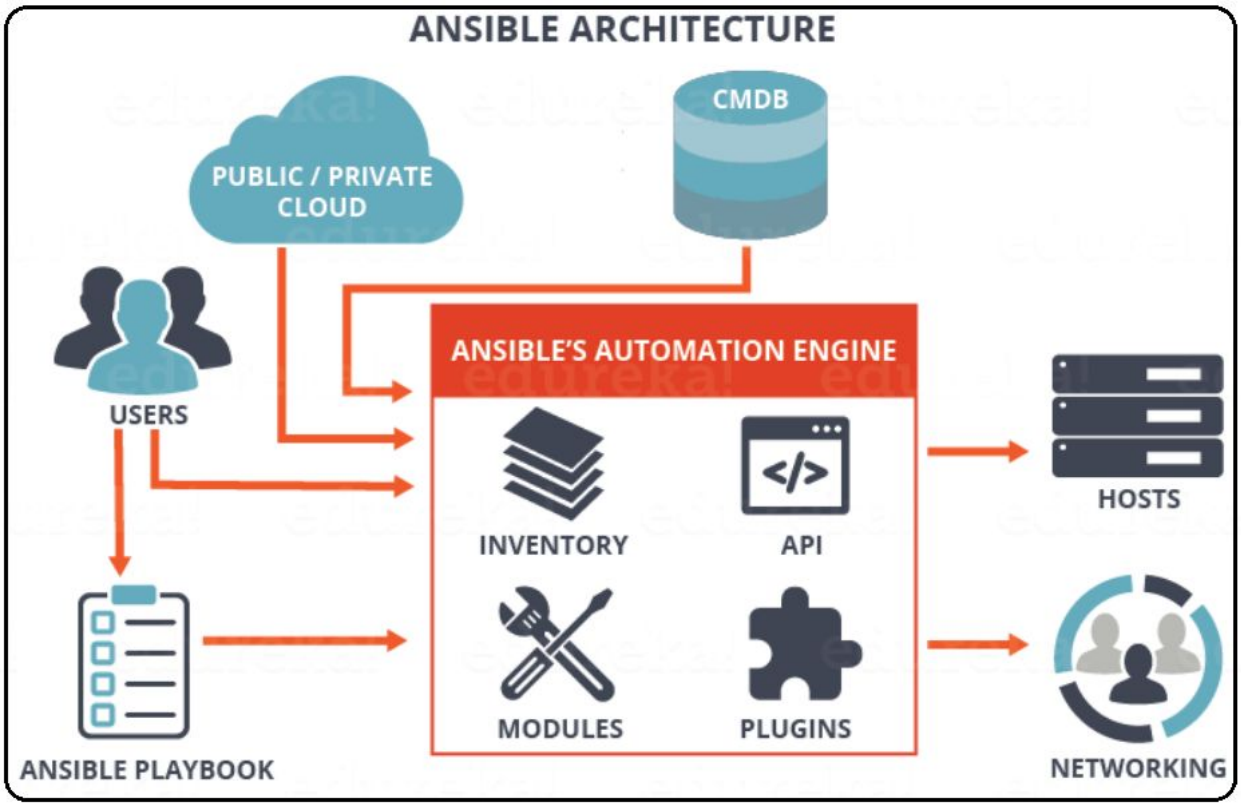


The management node in the above picture is the controlling node (managing node) which controls the entire execution of the playbook. It's the node from which you are running the installation. The inventory file provides the list of hosts where the Ansible modules needs to be run and the management node does a SSH connection and executes the small modules on the hosts machine and installs the product/software.

Beauty of Ansible is that it removes the modules once those are installed so effectively it connects to the host machine , executes the instructions and if it's successfully installed removes the code which was copied on the host machine which was executed.

What is Ansible & its Architecture?

Ansible architecture is fairly straightforward. Refer to the diagram below to understand the Ansible architecture:



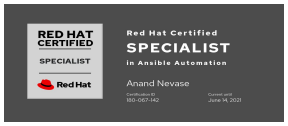
As you can see, in the diagram above, the Ansible automation engine has a direct interaction with the users who write playbooks to execute the Ansible Automation engine. It also interacts with cloud services and Configuration Management Database (CMDB).

The Ansible Automation engine consists of:

- **Inventories:** Ansible inventories are lists of hosts (nodes) along with their IP addresses, servers, databases etc. which needs to be managed. Ansible then takes action via a transport – SSH for UNIX, Linux or Networking devices and WinRM for Windows system.
- **APIs:** APIs in Ansible are used as transport for Cloud services, public or private.
- **Modules:** Modules are executed directly on remote hosts through playbooks. The modules can control system resources, like services, packages, or files (anything really), or execute system commands. Modules do it by acting on system files, installing packages or making API calls to the service network. There are over 450 Ansible-provided modules that automate nearly every part of your environment. For e.g.
    - Cloud Modules like *cloudformation* which creates or deletes an AWS cloud formation stack;
    - Database modules like *mssql_db* which removes MYSQL databases from remote hosts.
- **Plugins:** Plugins allows to execute Ansible tasks as a job build step. Plugins are pieces of code that augment Ansible's core functionality. Ansible ships with a number of handy plugins, and you can easily write your own. For example,
    - *Action* plugins are front ends to modules and can execute tasks on the controller before calling the modules themselves.
    - *Cache* plugins are used to keep a cache of 'facts' to avoid costly fact-gathering operations.
    - *Callback* plugins enable you to hook into Ansible events for display or logging purposes.

There are a few more components in Ansible Architecture which are explained below:

Networking: Ansible can also be used to automate different networks. Ansible uses the same simple, powerful, and the agentless automation framework IT operations and



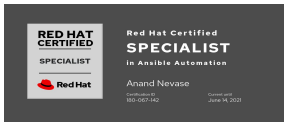
development are already using. It uses a data model (a playbook or role) that is separate from the Ansible automation engine that easily spans different network hardware.

Hosts: The hosts in the Ansible architecture are just node systems which are getting automated by Ansible. It can be any kind of machine – Windows, Linux, RedHat etc.

Playbooks: Playbooks are simple files written in YAML format which describes the tasks to be executed by Ansible. Playbooks can declare configurations, but they can also orchestrate the steps of any manual ordered process, even if it contains jump statements. They can launch tasks synchronously or asynchronously.

CMDB : It is a repository that acts as a data warehouse for IT installations. It holds data relating to a collection of IT assets (commonly referred to as configuration items (CI)), as well as to describe relationships between such assets.

Cloud: It is a network of remote servers hosted on the Internet to store, manage, and process data, rather than a local server. You can launch your resources and instances on cloud and connect to your servers.



Quiz: Ansible Architecture

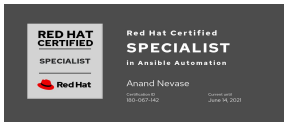
Choose the correct answer to the following Questions:

1. Which of the following terms best describes the Ansible Architecture
    a. Agentless
    b. Client/Server
    c. Event-driven
    d. Stateless

2. Which network protocol does Ansible Use, by default, to communicate with managed nodes?
    a. HTTP
    b. HTTPS
    c. SNMP
    d. SSH

3. Which of the following files define the actions Ansible performs on managed Nodes?
    a. Host Inventory
    b. Manifest
    c. Playbook
    d. Script

4. What syntax is used to define Ansible Playbook?
    a. Bash
    b. Perl
    c. Python
    d. YAML



Installing Ansible

This page describes how to install Ansible on Centos/RHEL platforms. Ansible is an agentless automation tool that by default manages machines over the SSH protocol. Once installed, Ansible does not add a database, and there will be no daemons to start or keep running. You only need to install it on one machine (which could easily be a laptop) and it can manage an entire fleet of remote machines from that central point. When Ansible manages remote machines, it does not leave software installed or running on them, so there's no real question about how to upgrade Ansible when moving to a new version.

On RHEL and CentOS:

```
$ sudo yum install ansible
```

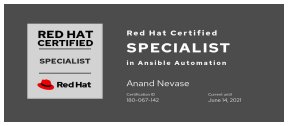
Installing Ansible with `pip`

Ansible can be installed with `pip`, the Python package manager. If `pip` isn't already available on your system of Python, run the following commands to install it:

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ sudo python get-pip.py
```

Then install Ansible

```
$ sudo pip install ansible
```

Working with Inventory

Static-Inventory

Ansible works against multiple systems in your infrastructure at the same time. It does this by selecting portions of systems listed in Ansible's inventory, which defaults to being saved in the location `/etc/ansible/hosts`. You can specify a different inventory file using the `-i <path>` option on the command line.

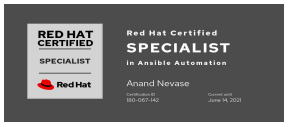
Not only is this inventory configurable, but you can also use multiple inventory files at the same time and pull inventory from dynamic or cloud sources or different formats (YAML, ini, etc), as described in Working With Dynamic Inventory. Introduced in version 2.4, Ansible has inventory plugins to make this flexible and customizable.

Hosts and Groups

The inventory file can be in one of many formats, depending on the inventory plugins you have. For this example, the format for `/etc/ansible/hosts` is an INI-like (one of Ansible's defaults) and looks like this:

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
Three.example.com
```

The headings in brackets are group names, which are used in classifying systems and deciding what systems you are controlling at what times and for what purpose.If you are



adding a lot of hosts following similar patterns, you can do this rather than listing each hostname:

```
[webservers]
www[01:50].example.com
```

For numeric patterns, leading zeros can be included or removed, as desired. Ranges are inclusive. You can also define alphabetic ranges:

```
[databases]
db-[a:f].example.com
```

You can also select the connection type and user on a per host basis:

```
[targets]
localhost               ansible_connection=local
other1.example.com      ansible_connection=ssh        ansible_user=mpdehaan
other2.example.com      ansible_connection=ssh        ansible_user=mdehaan
```

As mentioned above, setting these in the inventory file is only a shorthand, and we'll discuss how to store them in individual files in the 'host_vars' directory a bit later on.

Host Variables

As described above, it is easy to assign variables to hosts that will be used later in playbooks:

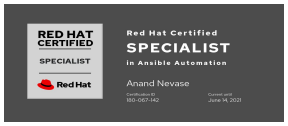
```
[atlanta]
host1 http_port=80 maxRequestsPerChild=808
host2 http_port=303 maxRequestsPerChild=909
```

Group Variables

Variables can also be applied to an entire group at once:

The INI way:

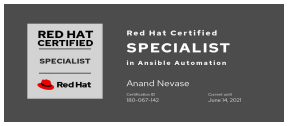
```
[atlanta]
host1
host2
[atlanta:vars]
ntp_server=ntp.atlanta.example.com
proxy=proxy.atlanta.example.com
```

Groups of Groups, and Group Variables

It is also possible to make groups of groups using the `:children` suffix in INI or the `children:` entry in YAML. You can apply variables using `:vars` or `vars:`:

```
[atlanta]
host1
host2

[raleigh]
host2
host3

[southeast:children]
atlanta
raleigh

[southeast:vars]
some_server=foo.southeast.example.com
halon_system_timeout=30
self_destruct_countdown=60
escape_pods=2

[usa:children]
southeast
northeast
southwest
northwest
```

Guided Exercises : Static inventory

1. Create `inventory` directory in `/root/ansible-training` workspace:

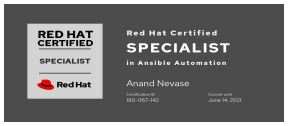
   ```
   $ mkdir -p /root/ansible-training/inventory
   ```

   Create file `myinventory` and specify following inventory details:

   ```
   server1

   server2

   [web]

   Web1

   web2


   [db]
   db1 db_user=dbuser db_password=password

   [app:children]
   web
   db

   [app:vars]
   app_username=admin
   app_password=admin
   ```

2. Execute following commands

   ```
   $ ansible-inventory -i myinventory --list      # display all inventory details
   $ ansible-inventory -i myinventory --graph     # display hierarchy
   $ ansible-inventory -i myinventory app --list  # display app inventory details
   $ ansible-inventory -i myinventory db --host db1 # display db1 host vars
   $ ansible-inventory -i myinventory app --graph --vars # display group vars
   $ ansible all -i myinventory --list-hosts      # list all hosts
   $ ansible ungrouped -i myinventory --list-hosts # list ungrouped hosts
   ```

Dynamic-Inventory

In a configuration – especially a cloud setup such as AWS where the inventory file keeps constantly changing as you add or decommission servers, keeping tabs on the hosts defined in the inventory file becomes a real challenge. It becomes inconvenient going back to the host file and updating the list of hosts with their IP addresses.

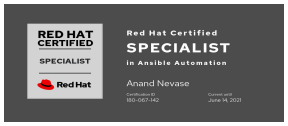
And this is where a dynamic inventory comes to play. So what is a dynamic inventory? A dynamic inventory is a shell script written in Python, PHP or any other programming language. It comes in handy in cloud environments such as AWS where IP addresses change once a virtual server is stopped and started again.

Ansible already has developed inventory scripts for public cloud platforms such as Google Compute Engine, Amazon EC2 instance, OpenStack, RackSpace, cobbler, among others.

What are the advantages of a dynamic inventory over a static inventory?

- Dynamic inventories do a perfect job of reducing human error as information is gathered using scripts.
- Minimal effort is required in managing inventories.

You can write your own customize dynamic inventory in a programming language of your choice. The inventory should return a format in JSON when appropriate options are passed.



Guided Exercises: Dynamic inventory

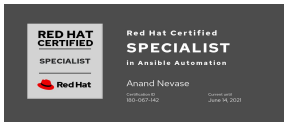
1. Create `dynamic-inventory` file in `/root/ansible-training/inventory` workspace and add file bash code:

```
$ vi /root/ansible-training/inventory/dynamic-inventory
```

```
#!/bin/bash
if [ "$1" == "--list" ] ; then
        curl  -X GET 'https://ansible.free.beeceptor.com/inventory'
fi
```

2. Execute following commands

```
$ ansible-inventory -i dynamic-inventory --list
$ ansible-inventory -i dynamic-inventory --graph
$ ansible all -i dynamic-inventory --list-hosts
```

Quiz: Building an Ansible Inventory

Choose the correct answers to the following questions:

1. Refer below inventory:

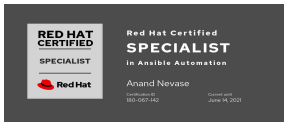
   [linux-dev]
   linux-server1
   linux -server2
   [windows-dev]
   windows-server1
   [dev:children]
   linux-dev
   windows-dev

   Given the Ansible inventory in the above exhibit, which host group, or groups, include the **linux-server2** host?

   a. linux-dev
   b. windows-dev
   c. dev
   d. Both linux-dev and dev

2. Which of the following expressions can be used in an Ansible inventory file to match hosts in the 10.1.0.0/16 address range?
   a. 10.1.0.0/16
   b. 10.1.[0:255].[0:255]
   c. 10.1.[0-255].[0:255]
   d. 10.1.*

3. In the inventory, a managed host
   a. Must be in no more than one group other than **all**
   b. May not be in a group that has child groups
   c. May be in more than one group other than all
   d. Can not be listed by an IP address



Ansible Configuration Settings

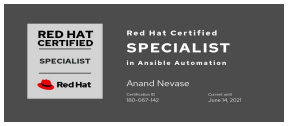
Ansible supports several sources for configuring its behavior, including an ini file named `ansible.cfg`, environment variables, command-line options, playbook keywords, and variables.

The `ansible-config` utility allows users to see all the configuration settings available, their defaults, how to set them and where their current value comes from.

Configuration settings include both values from the `ansible.cfg` file and environment variables. Within this category, values set in configuration files have lower precedence. Ansible uses the first `ansible.cfg` file it finds, ignoring all others. Ansible searches for `ansible.cfg` in these locations in order:

- `ANSIBLE_CONFIG` (environment variable if set)
- `ansible.cfg` (in the current directory)
- `~/.ansible.cfg` (in the home directory)
- `/etc/ansible/ansible.cfg`

> Environment variables have a higher precedence than entries in `ansible.cfg`. If you have environment variables set on your control node, they override the settings in whichever `ansible.cfg` file Ansible loads. The value of any given environment variable follows normal shell precedence: the last value defined overwrites previous values.



Guided Exercises : Managing Ansible Configuration files

In this exercise, you will customize your Ansible Environment.

1. Create `manage-config` directory in `/root/ansible-training` workspace. Change to this newly created directory.
2. In your `/root/ansible-training/manage-config` directory, create your own **ansible.cfg** file:

   Create a [defaults] section in that file. In that section, add a line which uses the **inventory** directive to specify the **./inventory** file as the default directory

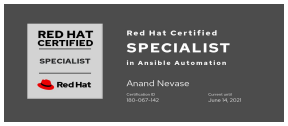
```
[defaults]
inventory = ./inventory
```

3. Add following hosts is file `/root/ansible-training/manage-config/inventory` for testing:

```
server1
server2
[web]
Web1
Web2
```

4. Execute ansible-inventory command without -i option to list hosts :

```
$ ansible-inventory --list
$ ansible-inventory --graph
$ ansible all --list-hosts
```

Introduction to ad-hoc commands

An Ansible ad-hoc command uses the /usr/bin/ansible command-line tool to automate a single task on one or more managed nodes. Ad-hoc commands are quick and easy, but they are not re-usable. So why learn about ad-hoc commands first? Ad-hoc commands demonstrate the simplicity and power of Ansible.

Why use ad-hoc commands?

Ad-hoc commands are great for tasks you repeat rarely. For example, if you want to power off all the machines in your lab for Christmas vacation, you could execute a quick one-liner in Ansible without writing a playbook. An ad-hoc command looks like this:

```
$ ansible [pattern] -m [module] -a "[module options]"
```

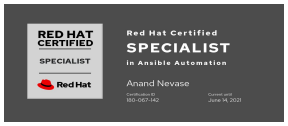
Use cases for ad-hoc tasks

Ad-hoc tasks can be used to reboot servers, copy files, manage packages and users, and much more. You can use any Ansible module in an ad-hoc task. Ad-hoc tasks, like playbooks, use a declarative model, calculating and executing the actions required to reach a specified final state. They achieve a form of idempotence by checking the current state before they begin and doing nothing unless the current state is different from the specified final state.

Ping Test

For testing ansible manages host connectivity, ping module is used

```
ansible <HOST_GROUP> -m ping
```

Managing files

An ad-hoc task can harness the power of Ansible and SCP to transfer many files to multiple machines in parallel. To transfer a file directly to all servers in the [atlanta] group:

```
$ ansible atlanta -m copy -a "src=/etc/hosts dest=/tmp/hosts"
```

The file module allows changing ownership and permissions on files. These same options can be passed directly to the `copy` module as well:

```
$ ansible webservers -m file -a "dest=/srv/foo/a.txt mode=600"

$ ansible webservers -m file -a "dest=/srv/foo/b.txt mode=600 owner=ansible
group=ansible"
```

The `file` module can also create directories, similar to `mkdir -p`:

```
$ ansible webservers -m file -a "dest=/path/to/c mode=755 owner=ansible group=ansible
state=directory"
```

As well as delete directories (recursively) and delete files:

```
$ ansible webservers -m file -a "dest=/path/to/c state=absent"
```

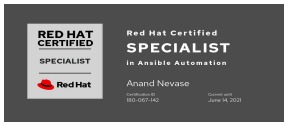
Managing packages

You might also use an ad-hoc task to install, update, or remove packages on managed nodes using a package management module like yum. To ensure a package is installed without updating it:

```
$ ansible webservers -m yum -a "name=acme state=present"
```

To ensure a specific version of a package is installed:

```
$ ansible webservers -m yum -a "name=acme-1.5 state=present"
```

To ensure a package is at the latest version:

```
$ ansible webservers -m yum -a "name=acme state=latest"
```

To ensure a package is not installed:

```
$ ansible webservers -m yum -a "name=acme state=absent"
```

Ansible has modules for managing packages under many platforms. If there is no module for your package manager, you can install packages using the command module or create a module for your package manager.

Managing services

Ensure a service is started on all webservers:

```
$ ansible webservers -m service -a "name=httpd state=started"
```

Alternatively, restart a service on all webservers:

```
$ ansible webservers -m service -a "name=httpd state=restarted"
```

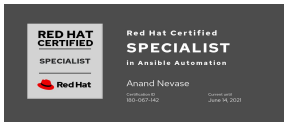
Ensure a service is stopped:

```
$ ansible webservers -m service -a "name=httpd state=stopped"
```

Gathering facts

Facts represent discovered variables about a system. You can use facts to implement conditional execution of tasks but also just to get ad-hoc information about your systems. To see all facts:

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

Guided Exercises : Adhoc Commands

In this exercise, you will customize your Ansible Environment.

5. Create `adhoc` directory in `/root/ansible-training` workspace. Change to this newly created directory.
6. In your `/root/ansible-training/adhoc` directory, create your own **ansible.cfg** file:

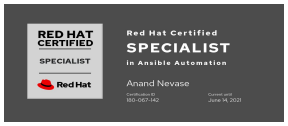
   Create a [defaults] section in that file. In that section, add a line which uses the **inventory** directive to specify the **./inventory** file as the default directory

```
[defaults]
inventory = ./inventory
```

7. Add following hosts is file `/root/ansible-training/adhoc/inventory` for testing:

   192.168.0.238
8. Execute ansible-inventory command without -i option to list hosts :

```
$ ansible all -m ping -u ansible -k
$ ansible all -m setup -u ansible -k
$ ansible all -m yum -a "name=httpd state=latest" -u ansible -k -b
$ ansible all -m service -a "name=httpd state=started" -u ansible -k -b
$ ansible all -m copy -a "content=Hello dest=/var/www/html/index.html" -u
ansible -k -b
$ ansible all -m file -a "path=/var/www/html/index.html state=absent" -u
ansible -k -b
$ ansible -m yum -a "name=httpd state=absent"
$ ansible all -m yum -a "name=httpd state=absent" -u ansible -k -b
```

Working with YAML Syntax

This section provides a basic overview of correct YAML syntax, which is how Ansible playbooks (our configuration management language) are expressed.

We use YAML because it is easier for humans to read and write than other common data formats like XML or JSON. Further, there are libraries available in most programming languages for working with YAML.

You may also wish to read Working With Playbooks at the same time to see how this is used in practice.

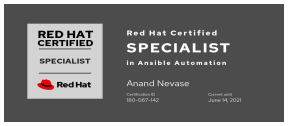
YAML Basics

For Ansible, nearly every YAML file starts with a list. Each item in the list is a list of key/value pairs, commonly called a “hash” or a “dictionary”. So, we need to know how to write lists and dictionaries in YAML.

There’s another small quirk to YAML. All YAML files (regardless of their association with Ansible or not) can optionally begin with `---` and end with `...`. This is part of the YAML format and indicates the start and end of a document.

All members of a list are lines beginning at the same indentation level starting with a `"- "` (a dash and a space):

```
---
# A list of tasty fruits
Fruits:
    - Apple
    - Orange
    - Strawberry
    - Mango
```

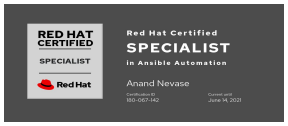
A dictionary is represented in a simple `key: value` form (the colon must be followed by a space):

```
# An employee record
Martin:
    name: Martin D'vloper
    job: Developer
    skill: Elite
```

More complicated data structures are possible, such as lists of dictionaries, dictionaries whose values are lists or a mix of both:

```
# Employee records

-  martin:
    name: Martin D'vloper
    job: Developer
    Skills:
      - python
      - perl
      - pascal
-  tabitha:
    name: Tabitha Bitumen
    job: Developer
    Skills:
      - lisp
      - fortran
      - erlang
```

Dictionaries and lists can also be represented in an abbreviated form if you really want to:



```
---
martin: {name: Martin D'vloper, job: Developer, skill: Elite}
fruits: ['Apple', 'Orange', 'Strawberry', 'Mango']
```

These are called “Flow collections”.

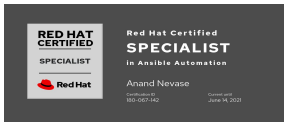
Ansible doesn’t really use these too much, but you can also specify a boolean value (true/false) in several forms:

```
create_key: yes
needs_agent: no
knows_oop: True
likes_emacs: TRUE
uses_cvs: false
```

Values can span multiple lines using `|` or `>`. Spanning multiple lines using a “Literal Block Scalar” `|` will include the newlines and any trailing spaces. Using a “Folded Block Scalar” `>` will fold newlines to spaces; it’s used to make what would otherwise be a very long line easier to read and edit. In either case the indentation will be ignored. Examples are:

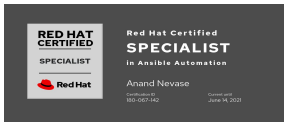
```
include_newlines: |
            exactly as you see
            will appear these three
            lines of poetry
fold_newlines: >
            this is really a
            single line of text
            despite appearances
```

Let’s combine what we learned so far in an arbitrary YAML example. This really has nothing to do with Ansible, but will give you a feel for the format:



```yaml
---
# An employee record
name: Martin D'vloper
job: Developer
skill: Elite
employed: True
Foods:
    - Apple
    - Orange
    - Strawberry
    - Mango
Languages:
    perl: Elite
    python: Elite
    pascal: Lame
education: |
    4 GCSEs
    3 A-Levels
    BSc in the Internet of Things
```

That's all you really need to know about YAML to start writing Ansible playbooks.



Implementing Playbooks

Playbooks are a completely different way to use ansible than in ad-hoc task execution mode, and are particularly powerful.

Simply put, playbooks are the basis for a really simple configuration management and multi-machine deployment system, unlike any that already exist, and one that is very well suited to deploying complex applications.

Playbooks can declare configurations, but they can also orchestrate steps of any manual ordered process, even as different steps must bounce back and forth between sets of machines in particular orders. They can launch tasks synchronously or asynchronously.

While you might run the main `/usr/bin/ansible` program for ad-hoc tasks, playbooks are more likely to be kept in source control and used to push out your configuration or assure the configurations of your remote systems are in spec.

Guided Exercise: Sample Playbook - Hello Ansible

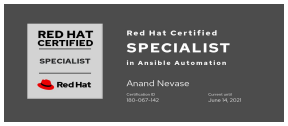
1. **Create following file is workspace** `/root/ansible-training/simple-playbook`

```
├── ansible.cfg
├── inventory
└── playbook.yml
```

2. Write following ansible code in playbook.yml

```
- hosts: all
  gather_facts: yes
  tasks:
    - debug:
        msg: "Hello from ansible"
```

3. Run command: `ansible-playbook playbook.yml -u ansible -k`



Guided Exercise: Sample Playbook - Multi-play

1. **Create following file is workspace** `/root/ansible-training/multi-play`

   ```
   ├── ansible.cfg
   ├── inventory
   └── playbook.yml
   ```

2. Write following ansible code in playbook.yml

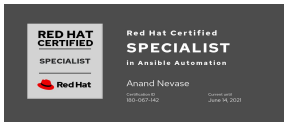
```
- hosts: localhost
  gather_facts: yes
  tasks:
    - debug:
        msg: "Hello from {{ansible_hostname}}"

- hosts: all
  gather_facts: yes
  tasks:
    - debug:
        msg: "Hello from {{ansible_hostname}}"
```

3. Run command: `ansible-playbook playbook.yml -u ansible -k`

Practice Exercise: Write a simple playbook with display following things:

1. Hostname
2. Machine Architecture
3. Machine Date
4. OS distribution
5. MAC Address
6. Machine Cores

   Hint: Use ansible facts



Guided Exercise: Sample Playbook - HTTP server installation and configuration.

Playbook should perform following tasks:

- Instal httpd

- Start httpd service

- Configure server index.html page to display “HELLO”

Steps:

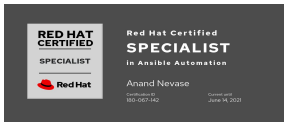
1. Create following file is workspace `/root/ansible-training/httpd-playbook`

```
├── ansible.cfg
├── inventory
└── playbook.yml
```

2. Write following ansible code in playbook.yml

```
- hosts: all
  gather_facts: no
  become: yes
  tasks:
    - name: install httpd package
      yum:
        name: httpd
        state: latest

    - name: Start httpd service
      service:
        name: httpd
        state: started

    - name: Create index.html
      copy:
        content: "HELLO"
        dest: /var/www/html/index.html
```

3. Run command: `ansible-playbook playbook.yml -u ansible -k`

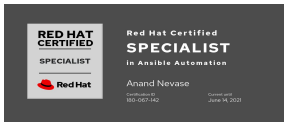


Practice Exercise: Write a simple playbook with perform following tasks:

1. Delete file index.html

2. Stop service httpd

3. Uninstall httpd



Managing Variables

While automation exists to make it easier to make things repeatable, all systems are not exactly alike; some may require configuration that is slightly different from others. In some instances, the observed behavior or state of one system might influence how you configure other systems. For example, you might need to find out the IP address of a system and use it as a configuration value on another system.

Ansible uses *variables* to help deal with differences between systems.

Creating valid variable names

Before you start using variables, it's important to know what are valid variable names.

Variable names should be letters, numbers, and underscores. Variables should always start with a letter.

`foo_port` is a great variable. `foo5` is fine too.

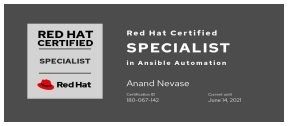
`foo-port`, `foo port`, `foo.port` and `12` are not valid variable names.

YAML also supports dictionaries which map keys to values. For instance:

```
foo:
  field1: one
  field2: two
```

You can then reference a specific field in the dictionary using either bracket notation or dot notation:

```
foo['field1']
foo.field1
```

These will both reference the same value (“one”). However, if you choose to use dot notation be aware that some keys can cause problems because they collide with attributes and methods of python dictionaries. You should use bracket notation instead of dot notation if you use keys which start and end with two underscores (Those are reserved for special meanings in python) or are any of the known public attributes:

`add`, `append`, `capitalize`, `center`, `clear`, `copy`, `count`, `decode`, `difference`, `encode`, `endswith`, `expandtabs`, `extend`, `find`, `format`, `fromhex`, `fromkeys`, `get`, `has_key`, `hex`, `imag`, `index`, `insert`, `intersection`, `intersection_update`, `isalnum`, `isalpha`, `isdecimal`, `isdigit`, `isdisjoint`, `is_integer`, `islower`, `isnumeric`, `isspace`, `issubset`, `issuperset`, `rpartition`, `rsplit`, `rstrip`, `setdefault`, `sort`, `split`, `splitlines`, `startswith`, `strip`, `swapcase`, `title`, `translate`, `union`, `update`, `upper`, `values`, `viewitems`, `viewkeys`, `viewvalues`, `zfill`.

Defining variables in a Inventory

Host Variables

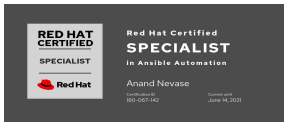
It is easy to assign variables to hosts that will be used later in playbooks:

```
[atlanta]
host1 http_port=80 maxRequestsPerChild=808
host2 http_port=303 maxRequestsPerChild=909
```

Group Variables

Variables can also be applied to an entire group at once:

```
[atlanta]
Host1
Host2
[atlanta:vars]
ntp_server=ntp.atlanta.example.c
```

Defining variables in a playbook

You can define variables directly in a playbook:

```
- hosts: webservers
  vars:
    http_port: 80
```

This can be nice as it's right there when you are reading the playbook.

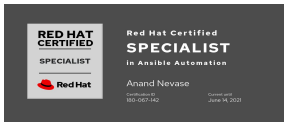
Registering variables

Another major use of variables is running a command and registering the result of that command as a variable. Results will vary from module to module. Use of `-v` when executing playbooks will show possible values for the results.

The value of a task being executed in ansible can be saved in a variable and used later. See some examples of this in the Conditionals chapter.

While it's mentioned elsewhere in that document too, here's a quick syntax example:

```
- hosts: web_servers
  tasks:
     - shell: /usr/bin/foo
       register: foo_result
       ignore_errors: True

     - shell: /usr/bin/bar
       when: foo_result.rc == 5
```

Registered variables are valid on the host the remainder of the playbook run, which is the same as the lifetime of “facts” in Ansible. Effectively registered variables are just like facts.

When using `register` with a loop, the data structure placed in the variable during the loop will contain a `results` attribute, that is a list of all responses from the module. For a more in-depth example of how this works, see the Loops section on using register with a loop.

Accessing complex variable data

We already described facts a little higher up in the documentation.

Some provided facts, like networking information, are made available as nested data structures. To access them a simple `{{ foo }}` is not sufficient, but it is still easy to do. Here’s how we get an IP address:

```
{{ ansible_facts["eth0"]["ipv4"]["address"] }}
```

OR alternatively:

```
{{ ansible_facts.eth0.ipv4.address }}
```

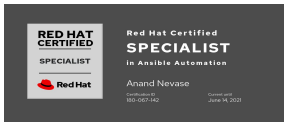
Similarly, this is how we access the first element of an array:

```
{{ foo[0] }}
```

Passing variables on the command line

In addition to `vars_prompt` and `vars_files`, it is possible to set variables at the command line using the `--extra-vars` (or `-e`) argument. Variables can be defined using a single quoted string (containing one or more variables) using one of the formats below key=value format:

```
ansible-playbook release.yml --extra-vars "version=1.23.45 other_variable=foo"
```

Defining variables in files

It's a great idea to keep your playbooks under source control, but you may wish to make the playbook source public while keeping certain important variables private. Similarly, sometimes you may just want to keep certain information in different files, away from the main playbook.

You can do this by using an external variables file, or files, just like this:

```
---

- hosts: all
  remote_user: root
  vars:
    favcolor: blue
  vars_files:
    - /vars/external_vars.yml
  tasks:
  - name: this is just a placeholder
    command: /bin/echo foo
```

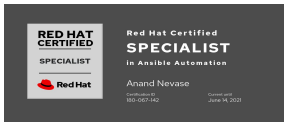
This removes the risk of sharing sensitive data with others when sharing your playbook source with them.

The contents of each variables file is a simple YAML dictionary, like this:

```
---
# in the above example, this would be vars/external_vars.yml
somevar: somevalue
password: magic
```

Variable precedence: Where should I put a variable?

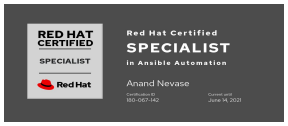
If multiple variables of the same name are defined in different places, they get overwritten in a certain order.



Here is the order of precedence from least to greatest (the last listed variables winning prioritization):

- command line values (eg “-u user”)
- role defaults
- inventory file or script group vars
- inventory group_vars/all
- playbook group_vars/all
- inventory/playbook group_vars/*
- inventory file or script host vars
- inventory/ playbook host_vars/*
- host facts / cached set_facts
- play vars
- play vars_prompt
- play vars_files
- role vars (defined in role/vars/main.yml)
- block vars (only for tasks in block)
- task vars (only for the task)
- include_vars
- set_facts / registered vars
- role (and include_role) params
- include params
- extra vars (always win precedence)

Basically, anything that goes into “role defaults” (the defaults folder inside the role) is the most malleable and easily overridden. Anything in the vars directory of the role overrides previous versions of that variable in namespace. The idea here to follow is that the more explicit you get in scope, the more precedence it takes with command line `-e` extra vars always winning. Host and/or inventory variables can win over role defaults, but not explicit includes like the vars directory or an `include_vars` task.



Guided Exercise: Sample Playbook - which display following vars:

- inventory_var
- play_var
- file_var
- register_var
- task_var
- cmd_line_var

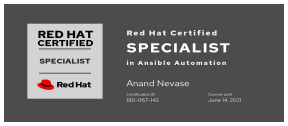
Steps 1. Create following file is workspace `/root/ansible-training/variable-playbook`

```
.
├── ansible.cfg
├── file_vars.yml
├── include_var.yml
├── inventory
└── playbook.yml
```

Steps 2. Write following ansible code in playbook.yml

```
- hosts: all
  gather_facts: no
  vars:
    play_var: "PLAY Variable"
  vars_files:
    - file_vars.yml
  tasks:
    - set_fact:
        task_var: "TASK Variable"
    - include_vars: include_var.yml
    - debug:
        msg:
          - "{{play_var}}"
          - "{{task_var}}"
          - "{{include_var}}"
          - "{{host_var}}"
          - "{{group_var}}"
          - "{{cmd_line_var}}"
          - "{{file_var}}"
```

Steps 3. Run command: `ansible-playbook playbook.yml -u ansible -k -e "cmd_line_var='Command Line Variable'"`



Managing Inclusion:

When working with complex or long playbooks, administrators can use separate files to divide tasks and list of variables into smaller pieces for easier management. There are multiple ways to include tasks files and variables in aa playbook.

- Including task/play (using include module)
- Including variable(using include_vars) - covered in the previous section.

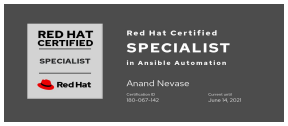
Including play/task (include module)

- Includes a file with a list of plays or tasks to be executed in the current playbook.
- Files with a list of plays can only be included at the top level. Lists of tasks can only be included where tasks normally run (in play).
- Before Ansible 2.0, all includes were ‘static’ and were executed when the play was compiled.
- Static includes are not subject to most directives. For example, loops or conditionals are applied instead to each inherited task.
- Since Ansible 2.0, task includes are dynamic and behave more like real tasks. This means they can be looped, skipped and use variables from any source. Ansible tries to auto detect this, but you can use the `static` directive (which was added in Ansible 2.1) to bypass autodetection.
- This module is also supported for Windows targets.

Guided Exercise: Sample Playbook to include task:

1. Create following file is workspace `/root/ansible-training/inclusion-playbook`

```
├── ansible.cfg
├── inventory
├── install.yml
└── playbook.yml
```

2. Write following ansible code in install.yml

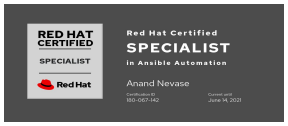
```
- name: install httpd package
  yum:
    name: httpd
    state: latest
- name: Start httpd service
  service:
    name: httpd
    state: started
- name: Create index.html
  copy:
    content: "HELLO"
    dest: /var/www/html/index.html
```

3. Write following ansible code in playbook.yml

```
- hosts: all
  gather_facts: no
  become: yes
  tasks:
    - include: install.yml
```

4. Execute following ansible-playbook command:

```
$ansible-playbook playbook.yml -u ansible -k
```

Conditions:

When statement

Use the when condition to control whether a task or role runs or is skipped. This is normally used to change play behavior based on facts from the destination system. Consider this playbook:

```
- hosts: all
  tasks:
    - include: Ubuntu.yml
      when: ansible_os_family == "Ubuntu"
    - include: RHEL.yml
      when: ansible_os_family == "RedHat"
```

Where `Ubuntu.yml` and `RHEL.yml` include some distribution-specific logic.

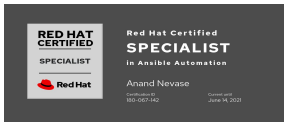
Guided Exercise: Sample Playbook for when condition:

1. Create following file is workspace `/root/ansible-training/when-playbook`

```
├── ansible.cfg
├── inventory
└── playbook.yml
```

2. Write following ansible code in playbook.yml

```
- hosts: all
  tasks:
  - stat:
      path: /home/mdtutorials2/sample.txt
    register: result
  - debug:
      msg: "Ansible When File Not Present Example."
```

```
        when: result.stat.exists == false
```

3. Execute following ansible-playbook command:

```
$ansible-playbook playbook.yml -u ansible -k
```

Loops

Often you'll want to do many things in one task, such as create a lot of users, install a lot of packages, or repeat a polling step until a certain result is reached.

Standard Loops

To save some typing, repeated tasks can be written in short-hand like so:

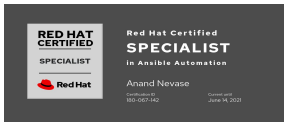
```
- name: add several users
  user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  loop:
     - testuser1
     - testuser2
```

If you have defined a YAML list in a variables file, or the 'vars' section, you can also do:

```
loop: "{{ somelist }}"
```

The above would be the equivalent of:

```
- name: add user testuser1
  user:
    name: "testuser1"
    state: present
    groups: "wheel"
- name: add user testuser2
  user:
    name: "testuser2"
    state: present
    groups: "wheel"
```

Note that the types of items you iterate over do not have to be simple lists of strings. If you have a list of hashes, you can reference subkeys using things like:

```
- name: add several users
  user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

Do-Until Loops

New in version 1.4.

Sometimes you would want to retry a task until a certain condition is met. Here's an example:

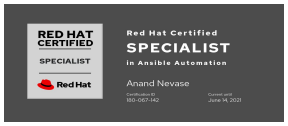
```
- shell: /usr/bin/foo
  register: result
  until: result.stdout.find("all systems go") != -1
  retries: 5
  delay: 10
```

The above example run the shell module recursively till the module's result has "all systems go" in its stdout or the task has been retried for 5 times with a delay of 10 seconds. The default value for "retries" is 3 and "delay" is 5.

The task returns the results returned by the last task run. The results of individual retries can be viewed by -vv option. The registered variable will also have a new key "attempts" which will have the number of the retries for the task.

with_item Loops

With the release of Ansible 2.5, the recommended way to perform loops is the use the new `loop` keyword instead of `with_X` style loops.



In many cases, `loop` syntax is better expressed using filters instead of more complex use of `query` or `lookup`.

The following examples will show how to convert many common `with_` style loops to `loop` and filters.

`with_items` is replaced by `loop` and the `flatten` filter.

```
- name: with_items
  debug:
    msg: "{{ item }}"
  with_items: "{{ items }}"

- name: with_items -> loop
  debug:
    msg: "{{ item }}"
  loop: "{{ items|flatten(levels=1) }}"
```

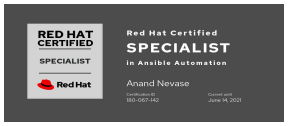
Guided Exercise: Sample Playbook for understanding Loops:

1. Create following file is workspace `/root/ansible-training/when-playbook`

```
├── ansible.cfg
├── inventory
├── loop-playbook.yml
├── with-items-playbook.yml
└── do-until-playbook.yml
```

4. Write following ansible code in loop-playbook.yml (adding testusers)

```
- hosts: all
  tasks:
       - name: add several users
         user:
           name: "{{ item.name }}"
           state: present
           groups: "{{ item.groups }}"
```

```
    loop:
      - { name: 'testuser1', groups: 'wheel' }
      - { name: 'testuser2', groups: 'root' }
```

5. Write following ansible code in with-items-playbook.yml (deleting testusers)

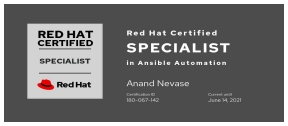
```
- hosts: all
  tasks:
      - name: delete several users
        user:
          name: "{{ item.name }}"
          state: absent
          groups: "{{ item.groups }}"
        with_items:
          - { name: 'testuser1', groups: 'wheel' }
          - { name: 'testuser2', groups: 'root' }
```

6. Write following ansible code in do-until-playbook.yml

```
- hosts: all
  tasks:
      - shell: /usr/bin/foo
        register: result
        until: result.stdout.find("all systems go") != -1
        retries: 5
        delay: 10
```

7. Execute following ansible-playbook command:

```
$ansible-playbook loop-playbook.yml -u ansible -k
$ansible-playbook with-items-playbook.yml -u ansible -k
$ansible-playbook do-until-playbook.yml -u ansible -k
```

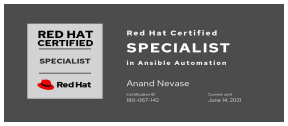



Handlers

A Handler is exactly the same as a Task, but it will run when called by another Task. A Handler will take action when called by an event it listens for.

This is useful for secondary actions that might be required after running a Task, such as starting a new service after installation or reloading a service after a configuration change.

```
- hosts: all
  tasks:
    - name: Install httpd
      yum: name=nginx state=latest
      notify:
        - Start Httpd

  handlers:
    - name: Start httpd
      service: name=httpd state=started
```

Tags

If you have a large playbook, it may become useful to be able to run only a specific part of it rather than running *everything* in the playbook. Ansible supports a “tags:” attribute for this reason.

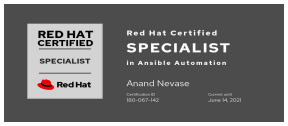
Tags can be applied to *many* structures in Ansible (see “tag inheritance”, below), but its simplest use is with individual tasks. Here is an example that tags two tasks with different tags:

Guided Exercise: Tags Sample playbook:

```
- hosts: localhost
  gather_facts: no
  tasks:
    - debug:
        msg: "Task 1 - 1"
      tags: task1, mix1
    - debug:
        msg: "Task 1 - 2"
      tags: task1, mix2
    - debug:
        msg: "Task 2 - 1"
      tags: task2, mix1
    - debug:
        msg: "Task 2 - 2"
      tags: task2, mix2
```

Command to Execute playbook with tags:

```
$ ansible-playbook playbook.yml -t "task1"
$ ansible-playbook playbook.yml -t "task2"
$ ansible-playbook playbook.yml -t "mix1"
$ ansible-playbook playbook.yml -t "mix2"
$ ansible-playbook playbook.yml -t "task1,task2"
```

Error Handling In Playbooks

Ansible normally has defaults that make sure to check the return codes of commands and modules and it fails fast – forcing an error to be dealt with unless you decide otherwise.

Sometimes a command that returns different than 0 isn't an error. Sometimes a command might not always need to report that it 'changed' the remote system. This section describes how to change the default behavior of Ansible for certain tasks so output and error handling behavior is as desired.

Ignoring Failed Commands

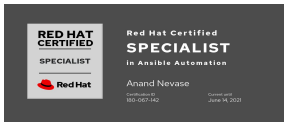
Generally playbooks will stop executing any more steps on a host that has a task fail. Sometimes, though, you want to continue on. To do so, write a task that looks like this:

```
- name: this will not be counted as a failure
  command: /bin/false
  ignore_errors: yes
```

Note that the above system only governs the return value of failure of the particular task, so if you have an undefined variable used or a syntax error, it will still raise an error that users will need to address. Note that this will not prevent failures on connection or execution issues. This feature only works when the task must be able to run and return a value of 'failed'.

Blocks error handling

Blocks also introduce the ability to handle errors in a way similar to exceptions in most programming languages. Blocks only deal with 'failed' status of a task. A bad task definition or an unreachable host are not 'rescuable' errors.

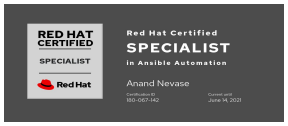


Guided Exercise: Block-rescue Sample playbook:

```
- hosts: localhost
  gather_facts: no
  tasks:
        - name: Attempt and graceful roll back demo
         block:
           - debug:
                msg: 'I execute normally'
            - name: i force a failure
              command: /bin/false
            - debug:
                msg: 'I never execute, due to the above task failing, :-('
         rescue:
           - debug:
                msg: 'I caught an error'
            - name: i force a failure in middle of recovery! >:-)
              command: /bin/false
            - debug:
                msg: 'I also never execute :-('
         always:
           - debug:
                msg: "This always executes"
```

Command to Execute playbook with tags:

```
$ ansible-playbook playbook.yml
```

Roles

Roles provide a framework for fully independent, or interdependent collections of variables, tasks, files, templates, and modules.

In Ansible, the role is the primary mechanism for breaking a playbook into multiple files. This simplifies writing complex playbooks, and it makes them easier to reuse. The breaking of playbook allows you to logically break the playbook into reusable components.

Each role is basically limited to a particular functionality or desired output, with all the necessary steps to provide that result either within that role itself or in other roles listed as dependencies.

Roles are not playbooks. Roles are small functionality which can be independently used but have to be used within playbooks. There is no way to directly execute a role. Roles have no explicit setting for which host the role will apply to.

Top-level playbooks are the bridge holding the hosts from your inventory file to roles that should be applied to those hosts.

Creating a New Role

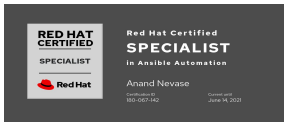
The directory structure for roles is essential to create a new role.

Role Structure

Roles have a structured layout on the file system. The default structure can be changed but for now let us stick to defaults.

Each role is a directory tree in itself. The role name is the directory name within the /roles directory.

```
$ ansible-galaxy -h
```

Usage

```
ansible-galaxy
[delete|import|info|init|install|list|login|remove|search|setup] [--help]
[options] ...
```

Options

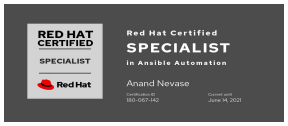
- -h, --help − Show this help message and exit.
- -v, --verbose − Verbose mode (-vvv for more, -vvvv to enable connection debugging)
- --version − Show program's version number and exit.

Creating a Role Directory

The above command has created the role directories.

```
$ ansible-galaxy init roles/httpd
```

```
$ tree httpd/
httpd/
├── defaults
│   └── main.yml
├── files ├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md ├── tasks
│   └── main.yml
├── templates ├── tests │   ├── inventory
│   └── test.yml
└── vars
    └── main.yml
```

```
8 directories, 8 files
```

Not all the directories will be used in the example and we will show the use of some of them in the example.

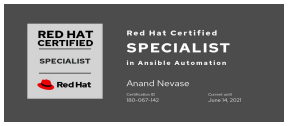
Utilizing Roles in Playbook

This is the code of the playbook we have written for demo purpose. This code is of the playbook vivek_orchestrate.yml. We have defined the hosts: tomcat-node and called the two roles – install-tomcat and start-tomcat.

The problem statement is that we have a war which we need to deploy on a machine via Ansible.

```
---
- hosts: all
roles:
   - {role: httpd }
```

Contents of our directory structure from where we are running the playbook.



Guided Exercise: Sample Playbook for understanding roles:

1. Create following file is workspace `/root/ansible-training/roles-playbook`

```
├── ansible.cfg
├── inventory
└── playbook.yml
```

2. Create `install-httpd` role:

```
$ ansible-galaxy init roles/install-httpd
```

3. Directory structure after role creation:

```
.
├── ansible.cfg
├── inventory
├── playbook.yml
└── roles
    └── install-httpd
        ├── defaults
        │   └── main.yml
        ├── files
        ├── handlers
        │   └── main.yml
        ├── meta
        │   └── main.yml
        ├── README.md
        ├── tasks
        │   └── main.yml
        ├── templates
        ├── tests
        │   ├── inventory
        │   └── test.yml
        └── vars
            └── main.yml
```

4. Add following code in ./roles/install-httpd/tasks/main.yml

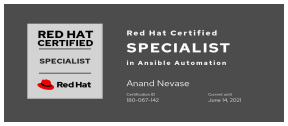
```
- name: install httpd package
  yum:
    name: httpd
    state: latest

- name: Start httpd service
  service:
    name: httpd
    state: started

- name: Create index.html
  copy:
    content: "HELLO"
    dest: /var/www/html/index.html
```

5. Add following code in playbook.yml

```
- hosts: all
  gather_facts: no
  become: yes
  roles:
    - role: install-httpd
```

6. Run following command to execute playbook:

```
$ ansible-playbook playbook.yml -u ansible -k
```

Implementing ansible Vault:

The “Vault” is a feature of Ansible that allows you to keep sensitive data such as passwords or keys protected at rest, rather than as plaintext in playbooks or roles. These vaults can then be distributed or placed in source control.

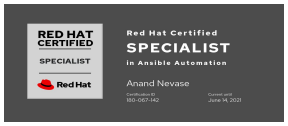
There are 2 types of vaulted content and each has their own uses and limitations:

Vaulted files:

- The full file is encrypted in the vault, this can contain Ansible variables or any other type of content.
- It will always be decrypted when loaded or referenced, Ansible cannot know if it needs the content unless it decrypts it.
- It can be used for inventory, anything that loads variables (i.e vars_files, group_vars, host_vars, include_vars, etc) and some actions that deal with files (i.e M(copy), M(assemble), M(script), etc).

Single encrypted variable:

- Only specific variables are encrypted inside a normal ‘variable file’.
- Does not work for other content, only variables.
- Decrypted on demand, so you can have vaulted variables with different vault secrets and only provide those needed.
- You can mix vaulted and non vaulted variables in the same file, even inline in a play or role.



To enable this feature, a command line tool, ansible-vault is used to edit files, and a command line flag `--ask-vault-pass`, `--vault-password-file` or `--vault-id` is used. You can also modify your `ansible.cfg` file to specify the location of a password file or configure Ansible to always prompt for the password. These options require no command line flag usage.

Running a Playbook With Vault

To run a playbook that contains vault-encrypted data files, you must provide the vault password.

To specify the vault-password interactively:

```
ansible-playbook site.yml --ask-vault-pass
```

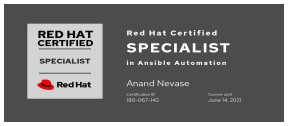
This prompt will then be used to decrypt (in memory only) any vault encrypted files that are accessed.

Alternatively, passwords can be specified with a file or a script (the script version will require Ansible 1.7 or later). When using this flag, ensure permissions on the file are such that no one else can access your key and do not add your key to source control:

```
ansible-playbook site.yml --vault-password-file ~/.vault_pass.txt

ansible-playbook site.yml --vault-password-file ~/.vault_pass.py
```

The password should be a string stored as a single line in the file.

If you are using a script instead of a flat file, ensure that it is marked as executable, and that the password is printed to standard output. If your script needs to prompt for data, prompts can be sent to standard error.



Guided Exercise: Sample Playbook for understanding Vault:

1. Create following file is workspace `/root/ansible-training/roles-playbook`

```
├── ansible.cfg
├── inventory
└── playbook.yml
```

2. Create cred.yml file, add following variable

```
username: testuser
password: password123
```

3. Add following code in playbook.yml

```
- hosts: localhost
  gather_facts: no
  vars_files:
    - cred.yml
  tasks:
    - debug:
        msg:
          - "Username: {{username}}"
          - "Password: {{password}}"
```

4. Encrypt cred.yml

```
$ ansible-vault encrypt cred.yml
New Vault password:
Confirm New Vault password:
Encryption successful
```

5. Execute playbook

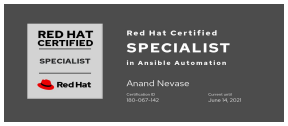
```
$  ansible-playbook playbook.yml
ERROR! Attempting to decrypt but no vault secrets found
```

6. Execute playbook

```
$ ansible-playbook playbook.yml  --ask-vault-pass
```

7. Store vault password in /tmp/vault file and execute playbook using below command:

```
$ ansible-playbook playbook.yml  --vault-password-file /tmp/vault
```

8. Edit Vault variable

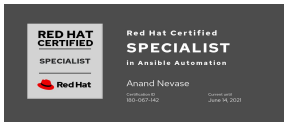
```
$ansible-vault edit cred.yml
Vault password:
```

9. View Vault variable

```
$ansible-vault view cred.yml
Vault password:
```

10. For create new Vault variables

```
$ansible-vault create cred-new.yml
```



Ansible Tower - AWX

AWX is an open source web application that provides a user interface, REST API, and task engine for Ansible. It's the open source version of the Ansible Tower. The AWX allows you to manage Ansible playbooks, inventories, and schedule jobs to run using the web interface.

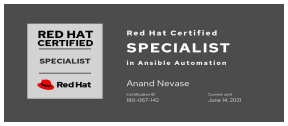
In this tutorial, we will show you basic usage of the Ansible AWX. So, you need a server that already has been installed Ansible AWX. We cover some basic configurations of Ansible AWX that you must know, such as setup credentials, inventories, setup and run job templates etc

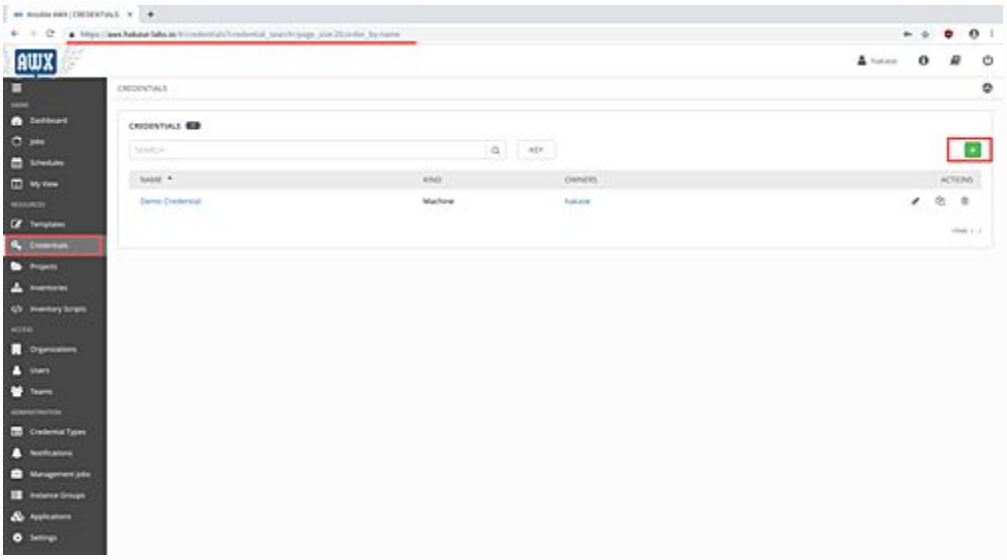
Setup Credentials

First of all, we need to configure the Ansible AWX Credentials. It's used for authentication when launching and running jobs against managed servers, synchronizing with inventory sources and importing projects.

By default, the Ansible AWX supports many credentials, including the VM machine through SSH authentication, Amazon Web Services, Google Compute Engine, OpenStack, Vault password, Source Control etc.

In order to setup credentials, click the '**Credentials**' menu on the left and click the **'+'** button on the right side of it.

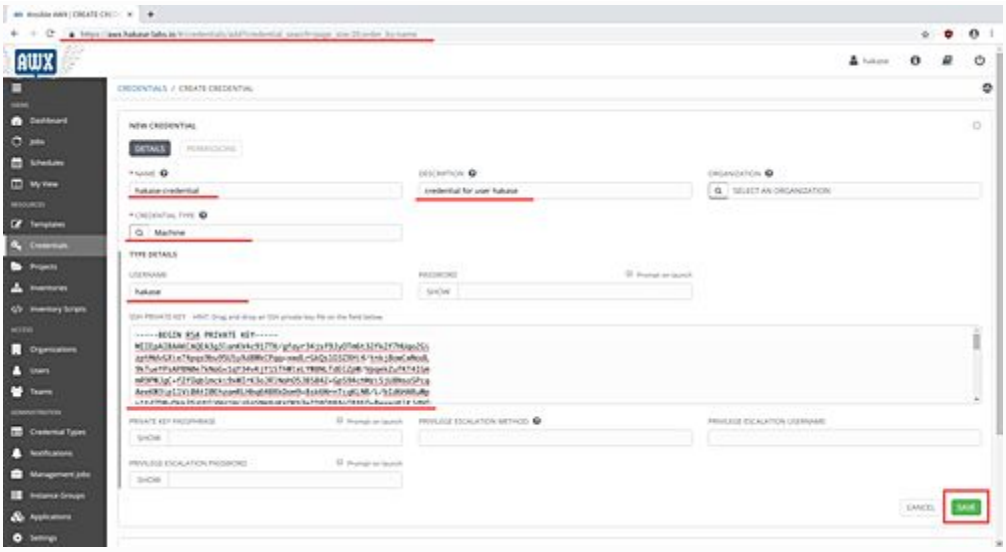




Now type credentials '**NAME**' and '**DESCRIPTION**', then specify the '**CREDENTIAL TYPE**' to '**Machine**'.

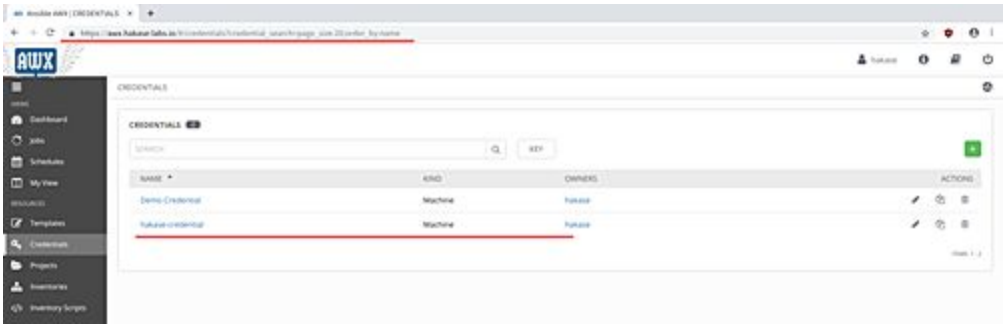
Simply, the 'Machine' credential will allow you to use the SSH authentication for managing servers. Its support for both password and key-based authentications.

For this guide, we will be using the key-based authentication. So, type the username and paste the private keys for that user.



Then click the '**SAVE**' button.

As a result, the new Ansible AWX credentials type 'Machine' has been created.

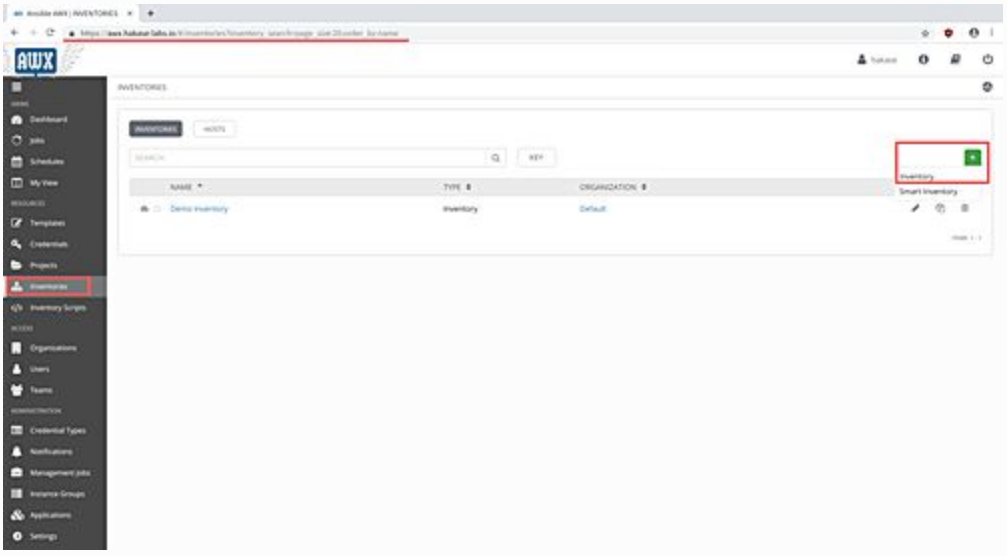


Setup Inventories

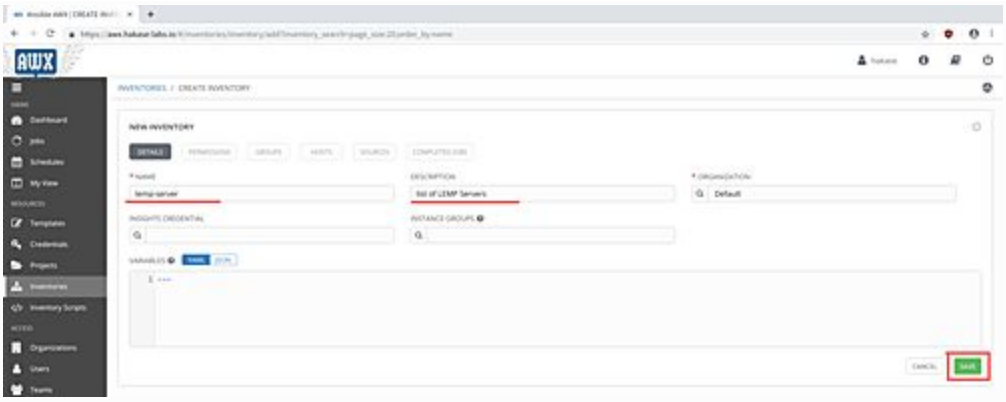
Inventories are groups of host servers that managed by Ansible AWX. The inventories allow you to create a group with several host server on it. And it makes easier to manage different servers with different environments.

In order to manage and provision of servers, we must create a new inventory group and then add server hosts into that inventory group.

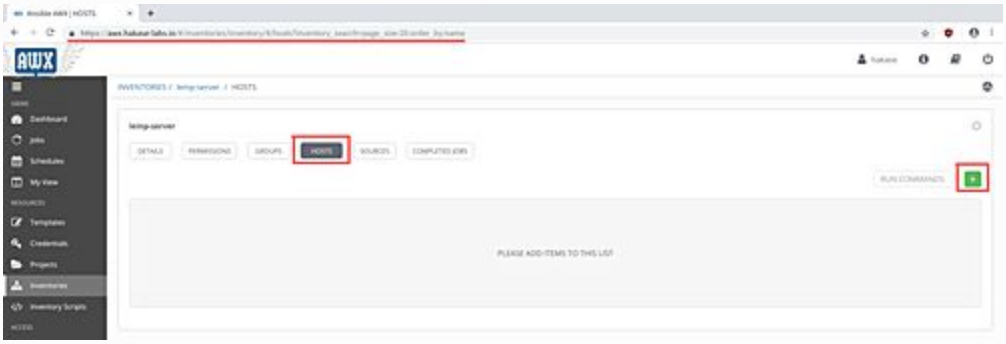
To add new inventory, click the '**Inventories**' menu on the left, then click the **'+'** button and choose the 'Inventory'.



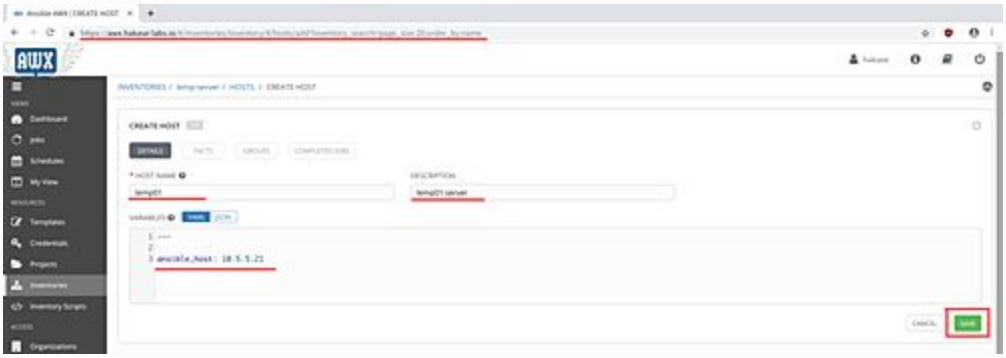
Type the '**NAME**' and '**DESCRIPTION**' of the inventory, then click the '**SAVE**' button.



Now click on the '**HOSTS**' tab, and click the **'+'** button to add new hosts.



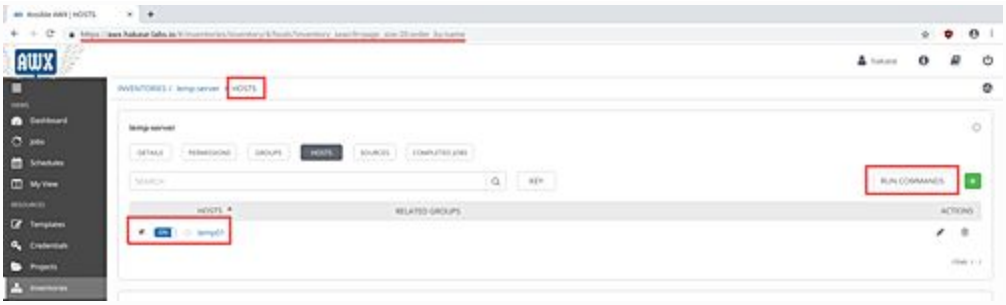
Type the '**HOST NAME**', '**DESCRIPTION**', and the '**VARIABLES**' with additional configuration for the target machine IP address 'ansible_host: 10.5.5.21'.



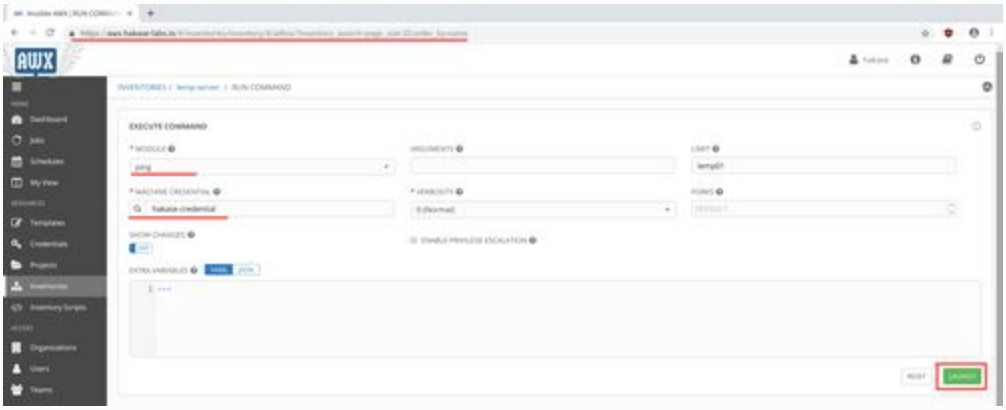
Now click the '**SAVE**' button.

Next, we need to ensure hosts configuration by checking the hosts using the ping command.

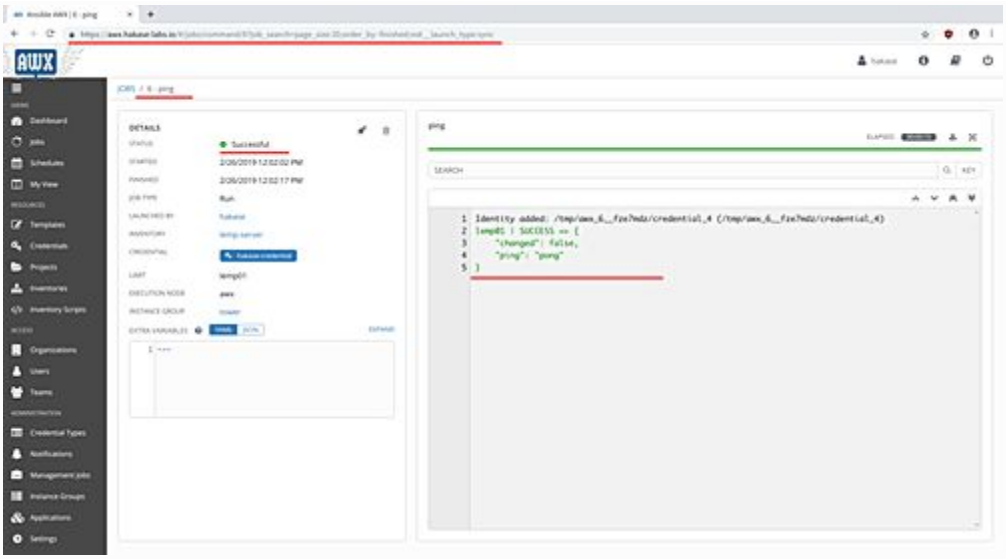
Back to the '**HOSTS**' tab, tick your hosts' name server and click the '**RUN COMMANDS**' button.



Now choose the '**MODULE**' called '**ping**', click the search button inside the '**MACHINE CREDENTIAL**' and '**SELECT**' your own, then click the '**LAUNCH**' button.



And you will be redirected to the new page and below is the result.

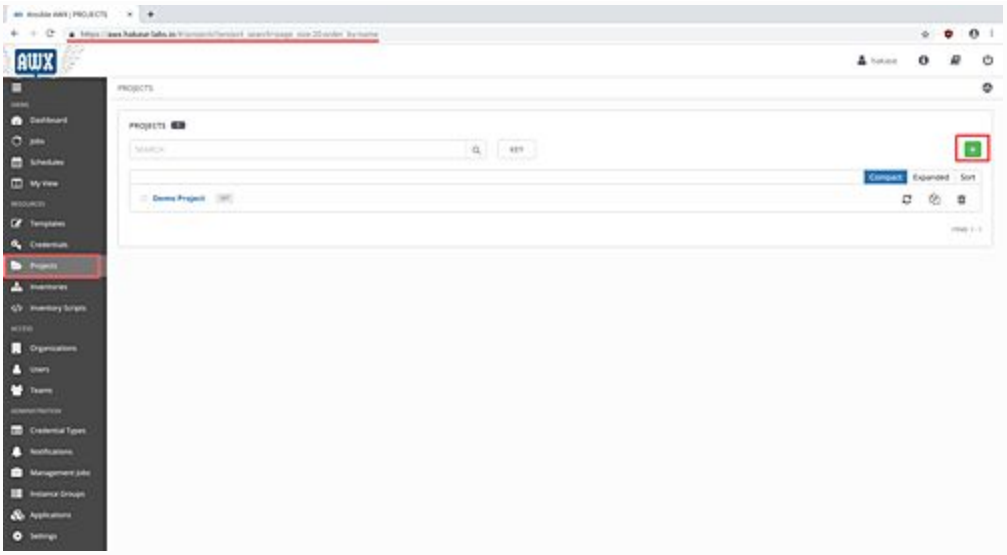


A new inventory has been created, and the target machine server has been added into it.

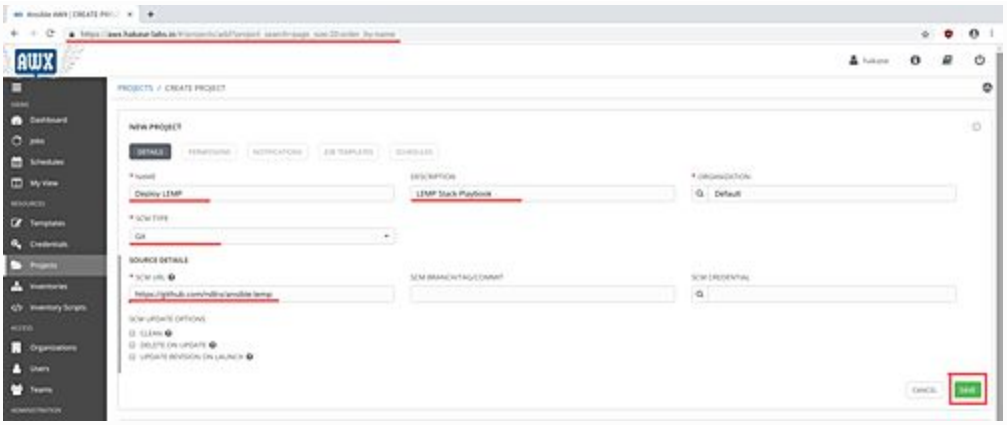
Setup Projects

Projects are represented as Ansible Playbooks on the AWX. Its collections of Ansible Playbooks that we can manage through local project directory or using the SCM system such as Git, Subversion, Mercurial, and RedHat Insights.

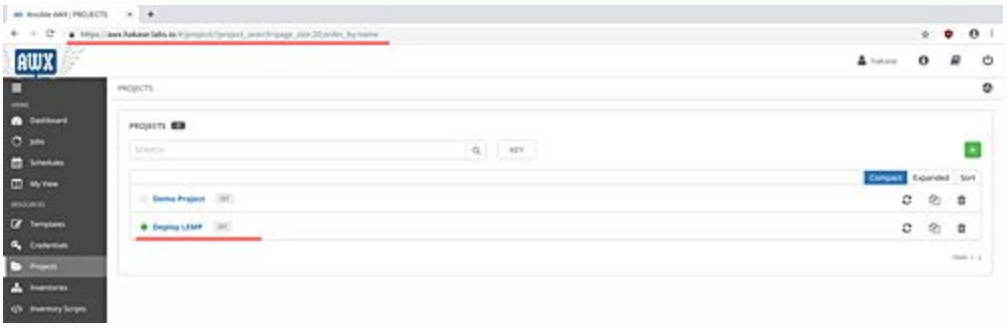
To create new projects, click the '**Projects**' menu on the left and click the **'+'** button.



Type the '**NAME**' of your project and the '**DESCRIPTION**', then choose the '**SCM TYPE**' to '**Git**' and paste your playbook repository.



Now click the '**SAVE**' button and as a result, a new project for the playbook has been created.



Additionally:

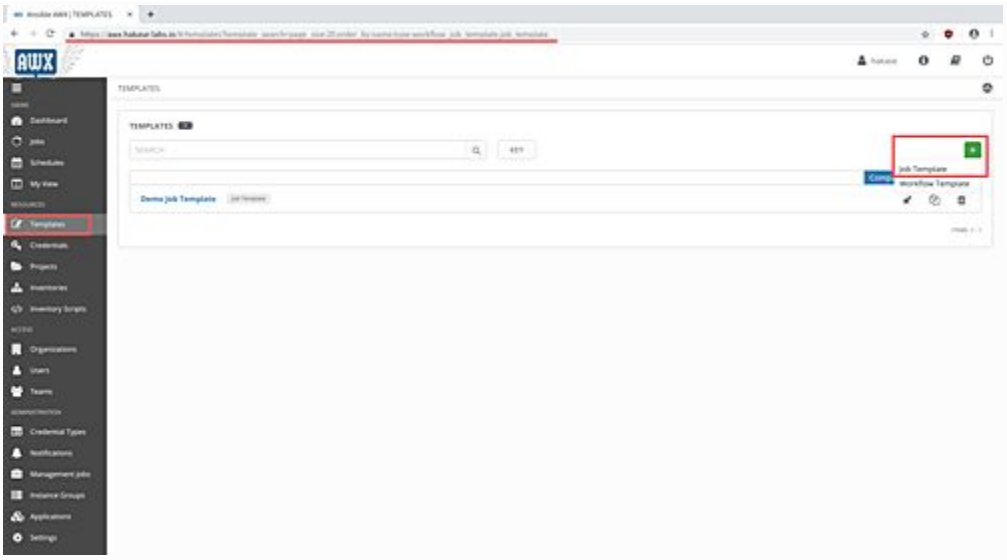
If you want to create a new project with the SCM type '**Manual**', you can create a new playbooks directory located at under the '**/var/lib/awx/projects**' directory.

The '/var/lib/awx/projects' directory is the default project directory for your Ansible playbooks if you're using the AWX docker version with the '**project_data_dir**' enabled.

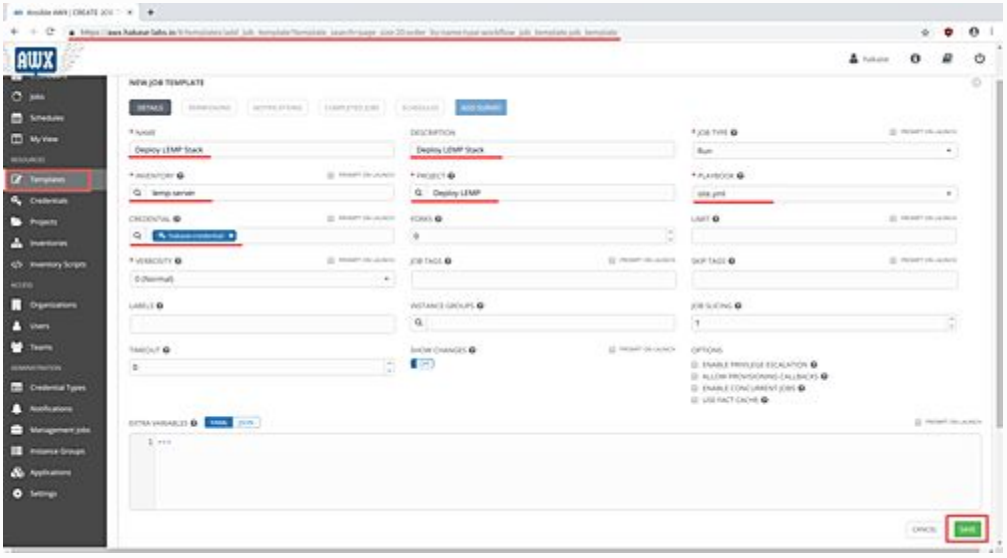
Create New Job Templates

The job template is the definition of running Ansible playbooks itself. So, in order to create a new job template or run the job template, we need to add Ansible playbook from our 'Project', the 'Credentials' for authentication, and the target machines that stored on the 'Inventories'.

For this guide, we've already created a new Project, Credential, and Inventory. So, just click the '**Templates**' menu on the left, then click the **'+'** button and choose the '**Job template**' option.



Now you need to type the '**NAME**' and '**DESCRIPTION**' of the job. Then choose the '**INVENTORY**', '**CREDENTIAL**', and the '**PROJECT**'. And after that, specify the '**PLAYBOOK**' that you want to run and deploy.

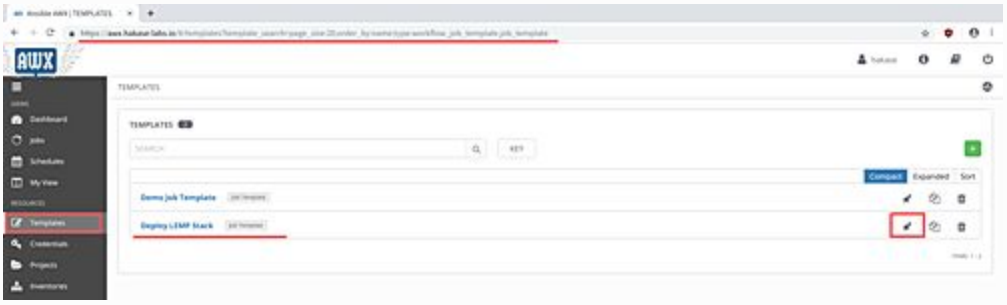


And as a result, the new job template Ansible AWX has been created.

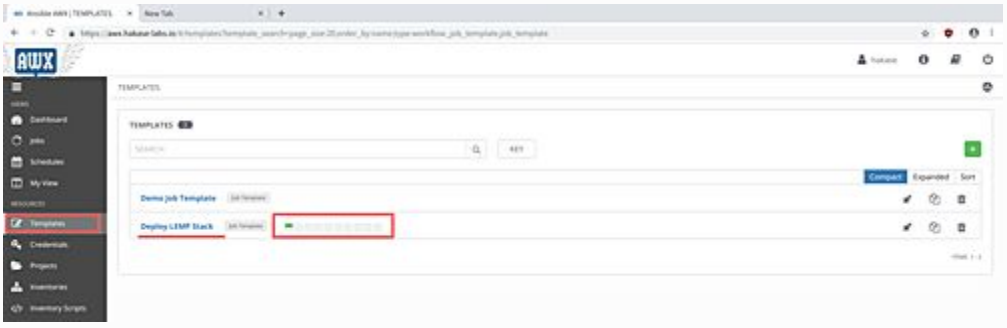
Run the Job Template

After creating the new job template, we will run the job template and deploy the Playbooks 'Projects' to target hosts on the 'Inventory'.

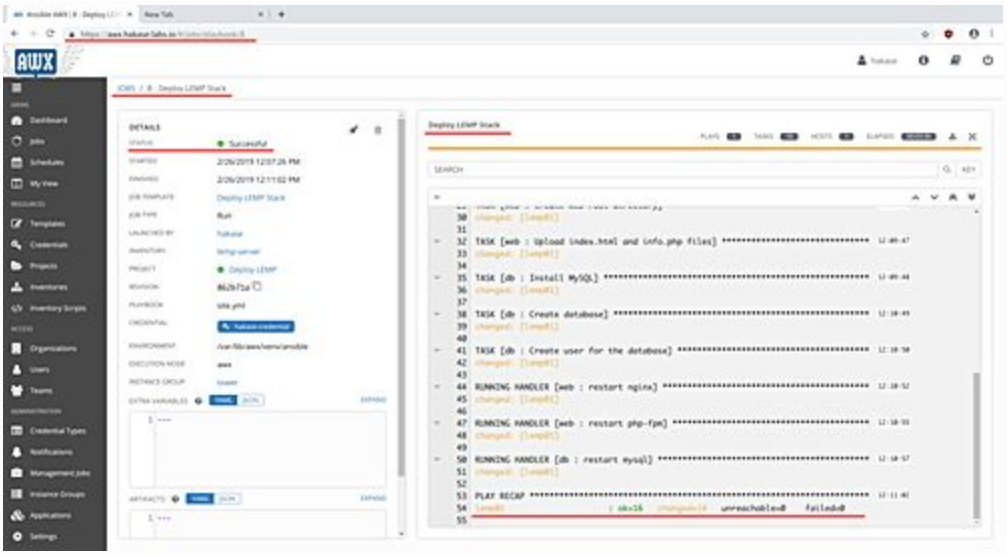
Click the '**Templates**' menu on the left and you will get lists of available job templates.



Once the job is finished, you will get the green sign inside the job template name.



Click on the green sign inside the job template name and you will be shown the actual result of that job.



The job was completed successfully, and the target machine has been installed the LAMP Stack though the Ansible AWX.