THE POWER OF ANSIBLE

Automating Cisco IOS

Contents at a glance

Introduction

Installing Ansible

Creating an Inventory

Ansible and Cisco IOS

Task automation

From start to finish

1. Introduction

1. Introduction

Ansible is an open source IT configuration management, deployment, and orchestration tool. It is unique from other management tools in many respects, aiming to provide large productivity gains to a wide variety of automation challenges as a more productive drop-in replacement for many core capabilities in other automation solutions.

Ansible seeks to keep automation tasks easy to understand, such that new users can be quickly brought into new IT projects, and longstanding automation content is easily understood even after months of being away from a project. Ansible seeks to make things powerful for expert users, but equally accessible for all skill levels, ensuring a quicker time to market for IT projects and faster, less-error prone turnaround on IT configuration change.

Ansible uses YAML files as its main source of information. YAML is a human readable data language that is commonly used for configuration or dataset files. It does come with some quirks though, such as being whitespace sensitive.

Ansible and all modules are Python 2.6 compatible, which means that they'll work with any version of Python2 above version 2.6. Therefore, there are no additional dependencies on the machines that you want to manage.

Not only are there no additional language dependencies for your machines, but also there are no additional dependencies at all. Ansible works by running commands via SSH or WinRM, so there is no need to install any additional software on the managed nodes. This is a huge benefit for two reasons:

1. The systems you are automating do not use additional resources because of daemons running in the background.
2. All the features that SSH provides can be used. You can use advanced features, such as Control Persist, Kerberos, and jump hosts. In addition, there is no need to implement your own authentication mechanism

1.1 Infrastructure as code

Infrastructure as code, also referred to as IaC, is a type of IT setup wherein developers or operations teams automatically manage and provision the technology stack for their systems through software, rather than using a manual process to configure discrete hardware devices and operating systems. Infrastructure as code is sometimes referred to as programmable or software-defined infrastructure.

The concept of infrastructure as code is similar to programming scripts, which are used to automate IT processes. However, scripts are primarily used to automate a series of static steps that must be repeated numerous times across multiple servers. Infrastructure as code uses higher-level or descriptive language to code more versatile and adaptive provisioning and deployment processes.

Ansible is a key example of an automation tool implementing infrastructure as code provisioning.

1.2 Architecture

Ansible is installed on a control node, which is the machine that communicates with the managed nodes. The managed nodes are the end devices you want to automate.
One of the primary differentiators between Ansible and many other automation tools is the architecture. Ansible is an agentless tool that runs in a 'push' model; no software is required to be installed on the managed nodes.

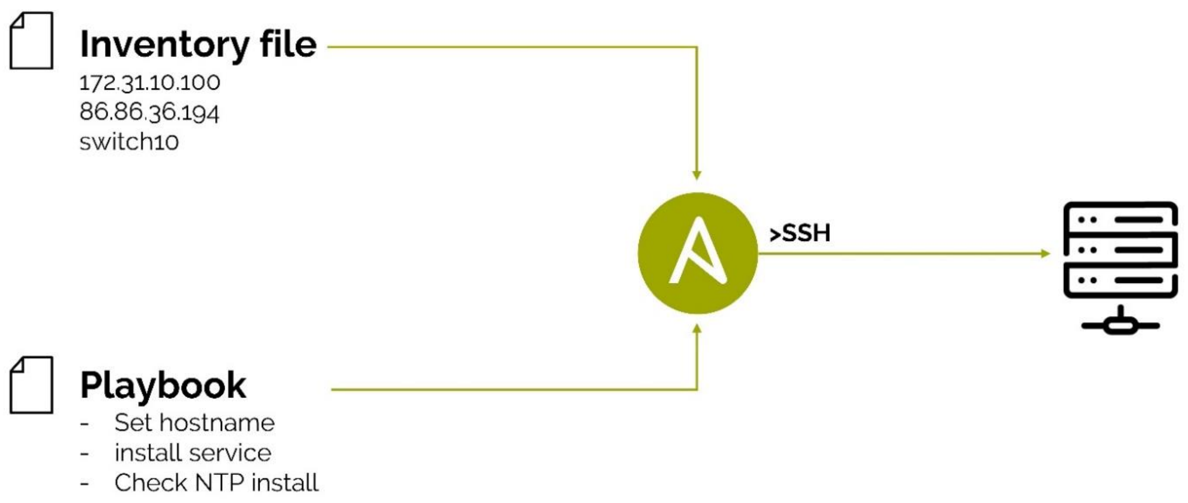
Ansible by default manages remote machines over SSH (Linux and UNIX) or WinRM (Windows), using the remote management frameworks that already exist natively on those platforms. Ansible builds on this by not requiring dedicated users or credentials; it just uses the credentials that the user supplies. Similarly, Ansible does not require administrator access, leveraging sudo, su, and other privilege escalation methods only on request.

1.3 Playbooks

Ansible performs automation and orchestration tasks using Playbooks. Playbooks are a YAML definition of automation tasks that describes how a particular result should be.

Like their namesake, Ansible Playbooks are prescriptive, yet responsive descriptions of how to perform an operation. It clearly states what each individual component of the IT infrastructure needs to do, but still allows components to react to discovered information.

Ansible Playbooks consist of series of 'plays' that define automation across a set of hosts, known as the 'inventory'. Each 'play' consists of multiple 'tasks', that can target one or more machines from the 'Inventory' to perform a specific task.



Each task uses an Ansible module; a python script for executing a specific task. These tasks can be simple, such as copying a configuration file from the control node to the managed node, or installing a software package. They can be complex, such as implementing a complete Virtual Routing domain within an Enterprise network. Ansible includes hundreds of modules, ranging from simple configuration management, to managing network devices, to modules for maintaining infrastructure on every major cloud provider.

The core Ansible modules allow for easy configuration of desired state; they check that the task that is specified needs to be performed before executing it. For example, if an Ansible task is defined to create a VLAN, the VLAN is only created when not yet existent. This desired state configuration, sometimes referred to as ‘idempotency’, ensures that the goal of each task is achieved while configuration can be applied repeatedly without side effects.

Ansible also supports encapsulating Playbook tasks into reusable units called ‘roles.’ Ansible roles can be used to apply common configurations in different scenarios, such as having a common switch configuration role that may be used for all switches in the IT environment. The Ansible Galaxy community site contains thousands of roles that can be used and customized to build Playbooks.

1.4 Modules

As noted above, tasks in Ansible are performed by Ansible ‘modules’; small pieces of code that run on the control node to create configuration data. If you have the need for a new module to handle a specific task of automation your IT infrastructure that is not covered by Ansible’s included set of 450+ modules, Ansible can be extended by writing your own modules. While the modules that ere included with Ansible are implemented in Python and PowerShell, Ansible modules can be written in any language and are only required to take JSON as input and produce JSON as output.

1.5 Network Automation

Ansible strives to automate not just traditional IT server and software, but the entirety of IT infrastructure, including areas not covered by traditional IT automation tools.

On most network devices it is not possible to install custom software, such as the agent daemon used by most automation tools in a PULL model. Ansible’s agentless nature makes it possible to automate network devices, and support is included with Ansible for automating networking from major vendors such as Cisco, Hewlett Packard Enterprise, Juniper and more.

By leveraging this networking support, network automation no longer needs to be done by a separate team but can be done by the same tools and processes used by other automation already implemented. Configuring a new VLAN or access control list becomes just a few additional Ansible tasks in the deployment Playbook, rather than a ticket filed with the separate networking team.

2. Installing Ansible

2. Installing Ansible

Ansible is installed on a control node, which is the machine that communicates with the managed nodes. The managed nodes are the end devices you want to automate.

2.1 Control node requirements

Ansible can be run from any machine with Python version 2 (2.7+) or Python version 3 (3.5+). This includes but is not limited to Debian, macOS, CentOS, RHEL, or any of the Berkeley Software Distributions.

Note

When implementing a Windows machine as control node Windows Subsystem for Linux must be used, as Ansible is not natively supported in python on Windows.

When choosing a control node, it is important to bear in mind that the Ansible platform greatly benefits from being run near the managed nodes. If you are running Ansible in a cloud environment, consider running a machine within the same environment, instead of using a local workstation.

Warning

Some modules and plugins might have additional requirements. These requirements are listed in the module specific docs.

2.2 Managed Node requirements

A managed node must be capable of functioning as an SSH-server.
Furthermore, the nodes also need python. By default, ansible uses python 2 (version 2.6+) but ansible can also use python 3 (version 3.5+). When using python 3 you need to set the ansible_python_interpreter inventory variable.

If the prospect node has selinux installed it is recommended that libselinux-python is also installed as some command can otherwise be interfered with by selinux.

2.3 Selecting an Ansible version

Which Ansible version to install is based on your particular needs and the environment you want to automate. You can choose any of the following ways to install Ansible:

- Install the latest stable release with an OS package manager
- Install with Python Package manager (pip)
- Install from source to access the development (devel) version to develop or test the latest features.

Note

The development version of Ansible should only be used if you are actively devolving content. The source code of the development version is rapidly changing, which can result in instability.

2.2 Installation

In this section we will explain step by step how to install ansible on several different operation systems.

> **Note**
> If your operating system is not listed, please refer to the Ansible documentation

2.2.1 Installing Ansible on RHEL, CentOS, or Fedora

On Fedora:

```
sudo dnf install ansible
```

On RHEL and CentOS:

```
sudo yum install ansible
```

2.2.2 Ansible on Ubuntu

To configure and install Ansible, you have to run these commands:

```
sudo apt update
sudo apt install software-properties-common
sudo apt-add-repository --yes --update ppa:ansible/ansible
sudo apt install ansible
```

> **Note**
> On older versions of Ubuntu you may want to use apt-get instead of apt.

2.2.3 Installing Ansible on Debian

Installing Ansible from the source makes it difficult to uninstall it. Ansible copies files right into the correct directory, so it is difficult to track which files were created. By building an .deb packages, you can easily uninstall Ansible.
Installing Ansible from a .deb package requires the same packages as installing it from the source. The following commands will install and build Ansible from a Debian package:

```
sudo apt-get update
sudo apt-get install build-essential git python-pip python-dev libffi-
dev libssl-dev asciidoc devscripts  debhelper cdbs
sudo pip install setuptools --upgrade
git clone git://github.com/ansible/ansible.git --recursive
cd ansible
make deb
```

Once the last command is done, you have to locate the built Debian package. This can be done with the following command:

```
find . -name "*.deb" ./deb-build/unstable/ansible_2.2.0-
0.git201607051907.d0ccedc.devel~unstable_all.deb
```

Before you can install this package, you'll need to install some packages that Ansible needs by running this command:

```
sudo apt-get install python-jinja2 python-paramiko sshpass python- markupsafe
```

Once the packages are installed, you can install Ansible from the Debian package you just build:

```
sudo dpkg -i ./deb-build/unstable/ansible_2.2.0-
0.git201607051907.d0ccedc.devel~unstable_all.deb
```

You can now verify that Ansible is installed by checking the version, with *ansible --version*. Because Ansible is installed from a package it can be easily uninstalled, with *apt-get remove ansible*.

2.2.4 Installing Ansible on FreeBSD

Ansible works with different versions of Python, FreeBSD has these different packages for each Python version. To install Python you can use this:

```
sudo pkg install py27-ansible
```

Or:

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

The specific version of Ansible can be chosen, i.e. *ansible25*.
Older versions of FreeBSD work with this (depends of your choice of package manager):

```
sudo pgk install ansible
```

2.2.5 Installing Ansible on macOS

The preferred way to install Ansible on macOS is with Python Package manager (pip). To install Ansible with pip can be found in *Installing Ansible with pip.* If you are running macOS version 10.12 or older, it is recommended to upgrade to the latest version of pip to connect to the Python Package Index securely.

2.2.6 Installing Ansible with pip

Ansible can be installed with the Python Package manager. If pip isn't installed on your system, you can run the following commands:

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
python get-pip.py –user
```

Then install Ansible:

```
pip install --user ansible
```

Note

Running pip with sudo will make global changes to the system. Since pip does not coordinate with system package managers, it could make changes to system files, leaving the system in an inconsistent or non-functioning state. This is particularly true for macOS. Installing with --user is recommended unless you understand fully the implications of modifying global files on the system.
Please make sure you have the latest version of pip before installing Ansible. If you have an older version of pip installed, you can upgrade with: pip install -U pip

2.3. Ansible command shell completion

As of Ansible 2.9, shell completion of the Ansible command line utilities is available and provided through an optional dependency called argcomplete. Argcomplete supports bash, and has limited support for zsh and tcsh.
You can install python-argcomplete from EPEL on Red Hat Enterprise based distributions, and or from the standard OS repositories for many other distributions.

2.3.1. Installing argcomplete on RHEL, CentOS, or Fedora

On Fedora:

```
sudo dnf install python-argcomplete
```

On RHEL and CentOS:

```
sudo yum install epel-release
sudo yum install python-argcomplete
```

Installing argcomplete with apt:

```
sudo apt install python-argcomplete
```

Installing argcomplete with pip:

```
python -m pip install argcomplete
```

2.3.2. Configuring argcomplete

There are 2 ways to configure argcomplete to allow shell completion of the Ansible command line utilities: globally or per command.

Globally

Global completion requires bash 4.2.
sudo activate-global-python-argcomplete
This will write a bash completion file to a global location. Use --dest to change the location.

Per command

If you do not have bash 4.2, you must register each script independently.

```
eval $(register-python-argcomplete ansible)
eval $(register-python-argcomplete ansible-config)
eval $(register-python-argcomplete ansible-console)
eval $(register-python-argcomplete ansible-doc)
eval $(register-python-argcomplete ansible-galaxy)
eval $(register-python-argcomplete ansible-inventory)
eval $(register-python-argcomplete ansible-playbook)
eval $(register-python-argcomplete ansible-pull)
eval $(register-python-argcomplete ansible-vault)
```

You should place the above commands into your shells profile file such as

```
~/.profile or ~/.bash_profile
```

3. Creating an inventory

3. Creating an Inventory

In chapter 2, you installed Ansible on your nodes. In this chapter, we'll look at what an inventory file is, and how to leverage the inventory file when you have a complex inventory of machines you need to interact with.

3.1 What is an Inventory?

In the configuration management, the tool you use needs to know which machine it should run on. This list of machines is called the inventory. Without an inventory, you have a set of playbooks that define the desires of the system. Once the inventory is defined, you can use patterns to select the hosts or groups you want to run Ansible against.

The default location of the inventory file is */etc/ansible/hosts.* Using this file is not recommended. You should maintain a different inventory file for each project you make. You can specify a different inventory file with the command: *ansible all –i <path>.*
The inventory file can be an INI file, a JSON file or a YAML file. The JSON file is only used when the inventory is dynamically created.

Depending on the plugins you have, you can create the inventory file in various ways or formats. The most common formats are INI and YAML. They can be as simple as a list of hostnames to run against. A basic INI file might look like this:

```
mail.example.com

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

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

The heading in brackets are the group names, these are used to define the hosts and deciding what hosts you are controlling and for what purpose.
Here is that same basic inventory file in YAML format:

```
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
```

There are two default groups: *all* and *ungrouped*. The *all* group contains every host.
The *ungrouped* group contains any hosts that doesn't belong to another group besides
from *all*. Every host belongs at least to two groups, namely to *all* and *ungrouped* or *all* and another group. The groups *all* and *ungrouped* are always present.

Each host can be put in more than one group. If there are a lot of hosts with a similar pattern, you can add the hosts as a range, as shown here:

```
host[1:3].example.com
```

This is equivalent to:

```
host1.example.com
host2.example.com
host3.example.com
```

These ranges supports leading zeros ([01:03]) and alphabetic ranges ([a:z]). Only the range [a:z] is supported, you can't specify a range [aa:zz]. If you need to define a range like this you need to use two ranges [a:z][a:z].
There can also be a third parameter specified, this parameter is optional and is called *step*:

```
host[min:max:step].example.com
```

This parameter allows the user to specify the increment between the hosts. It would look like this:

```
host[1:6:2].example.com
```

This is equivalent to:

```
host1.example.com
host3.example.com
host5.example.com
```

3.2 Adding variables to the inventory

You can store variables that are related to hosts in your inventory. These variables can be added directly to the hosts or groups in your inventory file. A variable can be easily assigned to a single host, then use it in your Playbook later. In INI it would look like this:

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

In YAML it would look like this:

```
Delft:
   host1:
        http_port: 80
        maxRequestsPerChild: 808
   host2:
        http_port: 303
        maxRequestsPerChild: 909
```

if you run SSH on a non-standard port, you can specify that certain port in your inventory file. To do this you can add a colon followed with the port number at the end of your hostname:

```
badwolf.example.com:5309
```

Note

If you use non-standard SSH ports in your SSH configuration file, the openssh connection will find and use them, but the paramiko connection will not.

In your inventory you can also define aliases. In INI:

```
jumper ansible_port=5555 ansible_host=192.0.2.50
```

In YAML:

```
...
  hosts:
    jumper:
      ansible_port: 5555
      ansible_host: 192.0.2.50
```

In the example above, Ansible will be running against the host alias “jumper”. This will connect to 192.0.2.50 on port 5555. This only works when the host has a static IP or when it is connected through tunnels.

Note

Values in the INI format using the key=value syntax are interpreted differently depending on where they are declared:

- Declared in line with the host. The INI values are interpreted as Python literal structures.
- Declared in the :vars section, the INI values are interpreted as strings.
- If the value is set in the INI inventory, it must be a certain type (for example, a string or a Boolean).

Try to use YAML format for your inventory to avoid confusion. A YAML inventory will processes variables consistently and correctly.

The variables can also be shared within a group, you can apply that variable to an entire group.
In INI:

```
[Delft]
host1
host2

[Delft:vars]
ntp_server=ntp.Delft.example.com
proxy=proxy.Delft.example.com
```

In YAML:

```
Delft:
  hosts:
    host1:
    host2:
  vars:
    ntp_server: ntp.Delft.example.com
    proxy: proxy.Delft.example.com
```

The group variables are an easy way to apply the variables to multiple hosts. But if a host is a member of different groups and you assign different variables to the same host in different groups, Ansible will choose which variable they use based on the internal rules.

It is possible to make a group out of another group, this can be done using *:children* suffix for INI or *children:* entry for YAML. The variables can be added with *:vars* or *vars:*. The child groups have a couple of properties to note:

- If a host is member of a child group it is automatically member of the parent group.
- The variable of a child group will have a higher precedence than the parent group variables.
- A group can have multiple parents and children, but there can't be circular dependencies.
- A host can be member of multiple groups, but there is only one instance of the host.

3.3 Usage

3.3.1 Organizing host and group variables

You can store every variable in the main inventory file, but sorting them may help to organize your variable values. The host and group files must use one specific syntax, YAML.
Ansible gets the host and group variable files by searching a path that is relative to the inventory file. If the inventory file is located at */etc/ansible/hosts* and contains a host named "soccer" and this host belongs to two groups, "Delft" and "webservers". The host will use the variables in the YAML files at the locations:

/etc/ansible/group_vars/Delft
/etc/ansible/group_vars/webservers
/etc/ansible/host_vars/soccer

For example, if you group hosts in your inventory by datacentre, and each datacentre use its own NTP and database. You can create a file named */etc/ansible/group_vars/Delft* to store the variables for the Delft group.
It is also possible to create directories named after your groups or hosts. Ansible will read these directories in an alphabetic order. All hosts of the groups have variables defined in different files. This can be helpful to keep your variables organized when a single file gets too big. You can also add these directories to your playbook. But if you load inventory files from your playbook directory and inventory directory, the variables in the playbook directory will override the variables in the inventory directory.

3.3.2 Merging variables

By default, the variables are merged to the specific host before the play is run. This will keep Ansible focused on the hosts and tasks. The order of merging variables is (from lowest to highest):

- The *all* group
- Parent group
- Child group
- Host

Ansible merges groups by default alphabetically, and the last group overrides the previous groups. For example, if you have an *a_group* that will merge with a *b_group*. The *b_group vars* that matches will override the ones in the *a_group.*
This behaviour setting can be changed by changing the group variable *ansible_group_priority*. The larger the number you set, the later it will be merged, so you give it a higher priority. This is default 1. The *ansible_group_priority* can only be set in the inventory source.

3.3.3 Multiple inventories

If you want to target multiple inventories at the same time, you can give multiple inventory parameters from the command line or you can configure *ANSIBLE_INVENTORY*. Target two sources from the command line can be done like this:

```
ansible-playbook get_logs.yml -i staging -i production
```

If there are variables that conflict in the inventories, it will be resolved according to the rules described in *3.4 merging variables*.
You can also pass a directory to Ansible. Ansible will read every file in that directory as an inventory and merge them together. The directory would look like this:

```
inventory
|openstack.yml                      # configure inventory plugin
|dynamic-inventory.py               # add additional hosts with dynamic
                                        inventory script
|static-inventory                   # add static hosts and groups
|group_vars
| |all.yml                             # assign variables to all hosts
```

You can target this directory with the following command:

```
ansible-playbook  example.yml -i inventory
```

the inventories are merged based on alphabetic order according to the filenames. This can be controlled by adding prefixes to the files:

```
inventory
|01-openstack.yml                      # configure inventory plugin
|02-dynamic-inventory.py               # add additional hosts with dynamic
                                          inventory script
|03-static-inventory                   # add static hosts and groups
|group_vars
| |all.yml                             # assign variables to all hosts
```

This means that the playbook will be run with number 3. This is because *01-openstack.yml* defines *myvar = 1* for the group all, *02-dynamic-inventory.py* defines *myvar = 2*, and *03-static-inventory* defines *myvar = 3*.

4. Using playbooks

4. Using playbooks

Ansible playbooks are YAML files that use a special keywords to inform Ansible what to do. Ansible works its way through a list of commands and arguments.
Playbooks can be more closely described as a model of a system than rather than to be described as a programming language or script. Each playbook has one or more ‘plays’. A ‘play’ maps a group of hosts to some roles, this is called tasks. A task is a call to an Ansible module, or a custom module.

4.1 Compose a playbook

A playbook is written in YAML, because of this you need to follow all of the same rules and standards a YAML file does. For example: you might find that the whitespace sensitivity of YAML takes some getting used to, especially because an indent is 2 spaces.
In a playbook there can be a quite a lot of plays that can affect your system to do different things.
The first thing you have to do in a playbook is tell Ansible where this playbook should run. This has to be done by specifying a host group. The line where host is defined is a list of one or more groups or hosts. Your playbook should look like this:

```yaml
---
-  hosts: all
```

Ansible now knows where to run, you now can tell what you want to run. This has to be done by adding a tasks section. Inside this section you specify the several tasks Ansible has to run. For example, you can use the task ping to make sure your hosts are connected to each other:

```yaml
 ---
- hosts: all
  tasks:
    -  ping:
```

4.1.1 Tasks list

Each play contains a list of tasks. The tasks are executed one at a time against all machines, that match the defined host group. A playbook runs from top to bottom, when a task failed the hosts, where the task failed, are taken out of the playbook. If it fails, simply correct your playbook and run it again.
The goal of a task is to execute a module, with declared arguments. A playbook should be idempotent, this means that running the playbook multiple times would provide the same output/effect it should have when it runs for the first time. The command and shell modules will return the same output every time, this is because the same command is used for example, chmod.

A task should always have a name to specify it, the name is included in the output from running the playbook. A basic task looks like this:

```yaml
tasks:
 - name: run apache
   service:
     name: httpd
     state: started
```

if you want to execute a command or shell line it will look like this:

```
tasks:
  - name: enable selinux
    command: /sbin/setenforce 1
```

or this:

```
tasks:
  - name: run this command and ignore the result
    shell: /usr/bin/somecommand || /bin/true
```

4.1.2 Handlers

As mentioned before, a playbook should be idempotent and can relay when they made changes. A playbook recognizes this and can respond to these changes. These notifications are triggered at the end of each block of tasks.
For example, multiple resources indicate that nginx should restart because of a change in a config file, but nginx will only restart once to avoid unnecessary restarts. Here is an example of it:
tasks:

```
- name: Install nginx
    package:
     name: nginx
     state: present
    notify:
     Start nginx
```

The thing listed in the notify section is called a handler. Handlers are a list of tasks, it is not very different from the regular tasks. These handlers are only run once, regardless how many tasks notify a particular handler. The handlers are defined in a section, it would look like this:

```
handlers:
 - name: Start nginx
    service:
     name: nginx
     state: started
```

If you want to use variables in handlers, you can't use them in names. This is because Ansible may not have a value available for a handler name. Handlers can also "listen" to generic topics and tasks can notify those topics:

```
handlers:
    - name: restart memcached
      service:
        name: memcached
        state: restarted
      listen: "restart web services"
    - name: restart apache
      service:
        name: apache
        state: restarted
      listen: "restart web services"
 tasks:
    - name: restart everything
      command: echo "this task will restart the web services"
      notify: "restart web services"
```

This use makes it easier to trigger multiple handlers. It also makes it easier to share handlers among playbooks and roles. Roles are described in a later chapter.

5. Defining roles

5. Defining roles

Roles are ways of automatically loading certain vars_files, tasks, and handlers based on a known file structure. Grouping content by roles also allows easy sharing of roles with other users.

5.1 Role directory structure

Example project structure:

```
|site.yml
|webservers.yml
|fooservers.yml
|roles
|  |common
|  |  |tasks
|  |  |handlers
|  |  |files
|  |  |templates
|  |  |vars
|  |  |defaults
|  |  |meta
|  |webservers
|  |  |tasks
|  |  |defaults
|  |  |meta
```

Roles expect files to be in certain directory names. Roles must include at least one of these directories, however it is possible to exclude any directory which is not in use. When a directory is used, each directory must contain a main.yml file, which contains the following:

- tasks - contains the main list of tasks to be executed by the role.
- handlers - contains handlers, which may be used by this role or even outside this role.
- defaults - default variables for the role.
- vars - other variables for the role.
- files - contains files which can be deployed using this role.
- templates - contains templates which can be deployed using this role.
- meta - defines some metadata for this role.

Other YAML files may be included in certain directories. For example, it is common practice to have platform-specific tasks included from the tasks/main.yml file:

```yaml
# roles/example/tasks/main.yml
- name: added in 2.4, previously 'include' was used
  import_tasks: redhat.yml
  when: ansible_facts['os_family']|lower == 'redhat'
- import_tasks: debian.yml
  when: ansible_facts['os_family']|lower == 'debian'

# roles/example/tasks/redhat.yml
- yum:
    name: "httpd"
    state: present

# roles/example/tasks/debian.yml
- apt:
    name: "apache2"
    state: present
```

Roles may also include modules and other plugin types.

5.2 Using roles

The classic (original) way to use roles is via the *r*oles: option for a given play:

```yaml
---
- hosts: webservers
  roles:
    - common
    - webservers
```

This designates the following behaviors, for each role 'x':

- If roles/x/tasks/main.yml exists, tasks listed therein will be added to the play.
- If roles/x/handlers/main.yml exists, handlers listed therein will be added to the play.
- If roles/x/vars/main.yml exists, variables listed therein will be added to the play.
- If roles/x/defaults/main.yml exists, variables listed therein will be added to the play.
- If roles/x/meta/main.yml exists, any role dependencies listed therein will be added to the list of roles (1.3 and later).
- Any copy, script, template or include tasks (in the role) can reference files in roles/x/{files,templates,tasks}/ (dir depends on task) without having to path them relatively or absolutely.

When used in this manner, the order of execution for your playbook is as follows:

- Any pre_tasks defined in the play.
- Any handlers triggered so far will be run.
- Each role listed in roles will execute in turn. Any role dependencies defined in the roles meta/main.yml will be run first, subject to tag filtering and conditionals.
- Any tasks defined in the play.
- Any handlers triggered so far will be run.
- Any post_tasks defined in the play.
- Any handlers triggered so far will be run.

Note

When using tags with tasks, unsure that you also tag your pre_tasks, post_tasks, and role dependencies. Pass this along as well, especially if the pre/post tasks and role dependencies are used for monitoring outage window control or load balancing.

As of v2.4, you can now use roles in line with any other tasks using import_role or include_role:

```
---
- hosts: webservers
  tasks:
    - debug:
        msg: "before we run our role"
    - import_role:
        name: example
    - include_role:
        name: example
    - debug:
        msg: "after we ran our role"
```

When roles are defined in the classic manner, they are treated as static imports and processed during playbook parsing.
The name used for the role can be a simple name, or it can be a fully qualified path:

```
---
- hosts: webservers
  roles:
    - role: '/path/to/my/roles/common'
```

Roles can also accept other keywords:

```
---
- hosts: webservers
  roles:
    - common
    - role: foo_app_instance
      vars:
        dir: '/opt/a'
        app_port: 5000
    - role: foo_app_instance
      vars:
        dir: '/opt/b'
        app_port: 5001
```

Or, using the newer syntax:

```
---
- hosts: webservers
  tasks:
    - include_role:
        name: foo_app_instance
      vars:
        dir: '/opt/a'
        app_port: 5000
```

You can conditionally import a role and execute its tasks:

```
---
- hosts: webservers
  tasks:
    - include_role:
        name: some_role
      when: "ansible_facts['os_family'] == 'RedHat'"
```

Finally, you may wish to assign tags to the tasks inside the roles you specify. You can do that as follows:

```
---
- hosts: webservers
  roles:
    - role: foo
      tags:
        - bar
        - baz
    # using YAML shorthand, this is equivalent to the above:
    - { role: foo, tags: ["bar", "baz"] }
```

Or, again, using the newer syntax:

```
---
- hosts: webservers
  tasks:
    - import_role:
        name: foo
      tags:
        - bar
        - baz
```

Note

This tags all of the tasks in that role with the tags specified, appending to any tags that are specified inside the role.

If you want to tag just the import of the role itself:

```
---
- hosts: webservers
  tasks:
    - include_role:
        name: bar
      tags:
        - foo
```

Note

The tags in this example will not be added to tasks inside an include_role, you can use a surrounding block directive to do both.

Warning

It is not possible to import a role while specifying a subset of tags to execute. If you want to construct a role with many tags and you want to call subsets of the role at different times, you should consider just splitting that role into multiple roles.

5.3 Role default variables

Role default variables allow you to set default variables for included or dependent roles (see below). To create defaults, simply add a code[defaults/main.yml] file in your role directory. These variables will have the lowest priority of any variables available, and can be easily overridden by any other variable, including inventory variables.

5.4 Role dependencies

Role dependencies allow you to automatically pull in other roles when using a role. Role dependencies are stored in the meta/main.yml file within the role directory, as noted above. This file should contain a list of roles and parameters to insert before the specified role, such as the following example roles/myapp/meta/main.yml:

```
---
dependencies:
  - role: common
    vars:
      some_parameter: 3
  - role: apache
    vars:
      apache_port: 80
  - role: postgres
    vars:
      dbname: blarg
      other_parameter: 12
```

Note

Role dependencies must use the classic role definition style. Role dependencies are always executed before the role that includes them, and may be recursive. Dependencies also follow the duplication rules specified above. If another role also lists it as a dependency, it will not be run again based on the same rules given above.

Always remember that when using allow_duplicates: true, it needs to be in the dependent role's meta/main.yml, not the parent.
For example, a role named car depends on a role named wheel as follows:

```
---
dependencies:
  - role: wheel
    vars:
      n: 1
  - role: wheel
    vars:
      n: 2
  - role: wheel
    vars:
      n: 3
  - role: wheel
    vars:
      n: 4
```

And the wheel role depends on two roles: tire and brake. The meta/main.yml for wheel would then contain the following:

```
---
dependencies:
  - role: tire
  - role: brake
```

And the meta/main.yml for tire and brake would have to contain the following:

```
 ---
allow_duplicates: true
```

The resulting execution order would be as follows:

```
tire(n=1)
brake(n=1)
wheel(n=1)
tire(n=2)
brake(n=2)
wheel(n=2)
...
car
```

Note that we did not have to use allow_duplicates: true for wheel, because each instance defined by car uses different parameter values.

5.5 Role duplication and execution

Ansible will only allow one role to execute once, even if defined multiple times, if the parameters defined on the role are not different for each definition. For example:

```
---
- hosts: webservers
  roles:
    - foo
    - foo
```

Given the above, the role foo will only be run once.

To make roles run more than once, there are two options:

- Pass different parameters in each role definition.
- Add allow_duplicates: true to the meta/main.yml file for the role.

Example 1 - passing different parameters:

```
---
- hosts: webservers
  roles:
    - role: foo
      vars:
        message: "first"
    - { role: foo, vars: { message: "second" } }
```

In this example, because each role definition has different parameters, foo will run twice.

Example 2 - using allow_duplicates: true:

```
# playbook.yml
---
- hosts: webservers
  roles:
    - foo
    - foo

# roles/foo/meta/main.yml
---
allow_duplicates: true
```

In this example, foo will run twice because we have explicitly enabled it to do so.

5.6 Role search path

Ansible will search for roles in the following way:

- A roles/ directory, relative to the playbook file.
- By default, in /etc/ansible/roles

In Ansible 1.4 and later you can configure an additional roles_path to search for roles. Use this to check all your common roles out to one location and share them easily between multiple playbook projects. See Configuring Ansible for details about how to set this up in ansible.cfg.

6. Creating modules

6. Creating modules

Ansible has modules for all of the common system administration tasks. A module is a reusable, standalone script. Ansible runs this script locally or remotely. These modules can communicate with your local machine, an API or a remote system to do specific tasks, for example changing a database password. All the core modules of Ansible are developed in python. These modules are divided into two groups of modules: *ansible-modules-core* and *ansible-modules-extra*. If you want to write your own module for local use, you can write it in any programming language.

6.1 ansible-modules-core

This group of modules contains all of the core modules that Ansible has, such as apt, template and copy. The modules in this group are solid and reviewed by a team from Ansible before changes are released.

6.2 ansible-modules-extras

The modules that doesn't belong in the core module group are stored in the extra group. These modules are maintained by the community. An example of an extra module is debconf or bundler. If you make changes that effects an extra module, the maintainer has to review your changes and is responsible for the module.

6.3 Setup environment

Before you can write your own module, you have to setup an environment to test your modules. This environment has nothing to do with the pervious steps you have done. First, we are gone make a new folder to work and test in:

```
Mkdir ansible-module
Cd ansible-module
git clone git://github.com/ansible/ansible.git --recursive
source ansible/hacking/env-setup
chmod +x ansible/hacking/test-module
```

Here we cloned Ansible from GitHub, and source a file that manipulate the environment. This makes sure that when you run the Ansible commands it points to the downloaded version instead the globally installed version. The last step we did make the file test-module executable. You may also need to install some packages such as *pyyaml* and *jinja2*. This can be done via pip:

```
pip install pyyaml jinja2
```

6.4 Writing modules in Bash

You can write a simple module using bash. Most of the time you will use python for writing a module. However, bash is a good start to write your first simple module. Let's start with making a simple module that uses a file and convert it into uppercases. To do this we first need to make a file and then make it executable:

```
touch test-file
chmod +x test-file
```

After this we can create a module, first we are going to create a module that returns
data that gets used in making and running a module. Ansible expects a JSON string as return, for this example we write a module that returns a JSON-encoded string. To do this you have to add the following to *test-file:*

```
#! /bin/bash
cat <<EOF
{“content”:”Hello World”}
EOF
```

After you have done this you can run it with the following command:

```
Ansible/hacking/test-module -m test-file
```

You now have written and test your first module. This module doesn’t do anything yet. To make a module that converts a file into uppercases, you have to make the module accept a filename as parameter and then make all the characters in the file uppercase. To do this your script needs to look like this:

```
#!/bin/bash
source $1
content=$(cat $file | tr '[:lower:]' '[:upper:]')
echo $content > $file
cat << EOF
{"content":"$content"}
EOF
```

When you run this module, it will convert a file to uppercases. But you have to make sure that the file exists first.

```
Ansible/hacking/test-module -m test-file -a file=example_file.txt
```

The module will change the characters in example_file.txt to uppercase characters. So the module does what you would expect it to do, but for Ansible there are still missing some metadata that is required. This is because Ansible module should be idempotent. Which means that a module can be executed multiple time the output will always be the same. Ansible modules report back when changes are made using the changed attribute. You can do this by checking in your module, to see if the content is the same both before and after the execution:

```
#!/bin/bash
source $1
original=$(cat $file)
content=$(echo $original | tr '[:lower:]' '[:upper:]')

if [[ "$original" == "$content" ]]; then
 CHANGED="false"
else
 CHANGED="true"
 echo $content > $file
fi

cat <<EOF
{"changed":$CHANGED, "content":"$content"}
EOF
```

This file contains all of the information Ansible needs to run is as a standalone module. You can test this module, the first time you run it you will notice that changed is true. If you run the same file again the output of changed will be false. The module you created is now idempotent.

6.5 Writing modules in python

Writing a module in python is very similar to writing a module in bash. The script is called with a file path, this contains your arguments as the first parameter. Python can be used to read and parse this file and build it out of your module. Ansible is shipped with a module called *ansible.module_utils.basic*, this module provides a boilerplate that you'd otherwise have to yourself. In this chapter we will make a module that sets the system time. We are going to create this module the hard way, so without the boilerplate. This is because you can't use the boilerplates in other programming languages.
Let's get started, you first need to make a file named *time-test.py*. The content of this file is a python script and would look like the following:

```
  #!/usr/bin/python
import datetime
import json

date = str(datetime.datetime.now())
print(json.dumps({
    "time" : date
}))
```

This module can be tested in the same environment you created earlier. To run this script, you can use the following command:

```
ansible/hacking/test-module -m ./time-test.py
```

The output should look like this:

```
{"time": "2020-09-17 11:34:03.432951"}
```

We can modify this module to allow us to set the time. We will do this by passing a key value pair in the form of *time=<string>* into the module. Ansible saves arguments internally in an argument file. This file is just a string, so any form of argument is good. In this case we will do parsing to treat our input as key=value.
We will use the following as an example to set the time:

time time="July 17 12:55"

if you don’t set the parameters the module will just leave the time as it is and return it. Your script would look like the following:

```
#!/usr/bin/python
import datetime
import sys
import json
import os
import shlex

args_file = sys.argv[1]
args_data = file(args_file).read()

arguments = shlex.split(args_data)
for arg in arguments:

    if "=" in arg:
        (key, value) = arg.split("=")

        if key == "time":
            rc = os.system("date -s \"%s\"" % value)
            if rc != 0:
                print(json.dumps({
                    "failed" : True,
                    "msg"    : "failed setting the time"
                }))
                sys.exit(1)
            date = str(datetime.datetime.now())
            print(json.dumps({
                "time" : date,
                "changed" : True
            }))
            sys.exit(0)

date = str(datetime.datetime.now())
print(json.dumps({
    "time" : date
}))
```

This module will look for the inserted key and if this key is time then it will change the time. If no parameters were sent, this module will just return the time.
To test this module, you have to use the following command:

```
ansible/hacking/test-module -m ./time-test.py -a "time=\”July 17 12:55\””
```

It would return something like this:

```
{“changed”: true, “time”: “2020-07-17 12:55:00.000405”}
```

As mentioned before, there are some shortcuts that you can use when you write a module in python. Modules are transferred as one file, so an argument file is no longer needed. This means that the code is getting shorter and the execution time is faster.

6.6 Writing modules in other programming language

To write a module you don’t have to use python or bash, you can use any programming language you want. But if you want to use the core modules, you need to write your module in python, otherwise you have to implement the argument handling yourself. We would recommend to write your modules in python, because of this core modules that handles the arguments.

7. Advanced Ansible CLI

7. Advanced Ansible CLI

For this chapter we will look at two of the advanced features of Ansible: *Ansible Vault* and *Ansible Galaxy*.
While these features aren't a must to run Ansible they can come in handy when either making your playbooks, files, passwords and/or strings more secure or speeding up your automation through 3rd party roles and or playbooks made by the ansible community.

7.1 Ansible Vault

7.1.1 Why you might want to consider Ansible Vault:

When starting off with Ansible Vault, it's important to understand why someone would choose to use this.
Whilst it's not recommended for your run-of-the-mill quick test lab. The moment you want to take your Ansible use to the next level there will be some confidential information you might not want to share or have someone take advantage of in case the project gets leaked.
Therefore, Ansible created Ansible VAULT: This is a tool that enables their users to encrypt their playbooks and password-protect them as its main feature. The other notable feature is encrypting specific variables inside your YAML files.

Warning

While ansible vault offers encryption it's not an end-all be-all solution as you will still be asked for your password and/or the file containing your password. If you want your network to be secure it is still recommended to take the extra necessary measures.

7.1.2 How does one use Ansible Vault?

Tip

When running ansible vault commands from the CLI it is recommended to first edit your $EDITOR environment variable to the one you're most comfortable with inside of your .bashrc file. To check which editor is your default type in echo $EDITOR. Alternatively you can also use EDITOR=nano ansible-vault edit example.yml to edit an encrypted file.

To start/enable this feature it is required to type the following command inside of your command line window:

```
ansible-vault
```

Followed by one of its uses directly after prompting for a password (or the password file location).

- Encrypt example.yml (this encrypts an already existing file)
- Decrypt example.yml (this decrypts an already existing file)
- Create example.yml (creates an encrypted file)
- Edit example.yml (only way to edit a file which has been encrypted by Ansible vault – *see above TIP for inline editor change)*

- Rekey example.yml (to change the original password – you will be prompted first for the original before the new one)
- Encrypt_string (this encrypts a specific variable)
- View example.yml (to view an encrypted file)

Warning
Once you lose your password you will also lose the data, so proceed carefully.

If everything went correctly, and you'd like to use an encrypted playbook instead of using `Ansible-playbook example.yaml`.To run the file, you will now have to provide the ansible with a password or a password location to be able to run the playbook and this would be done by adding `--ask vault pass` at the end of your command forming `ansible-playbook example.yml --ask-vault-pass` for the password prompt version.
And `--vault-password-file pass.txt` at the end of your command forming `ansible-playbook --vault-password-file pass.txt` for the password file version which doesn't require any further actions from the user. However, doing so will leave the password file unencrypted so at the very least it is recommended to change the user rights on it with `chmod 400 pass.txt` so that only the file owner has read only access to it.

Note
The following explanation is for Ansible 2.4+ only.

Lastly there is one more option to store and retrieve your vault passwords assuming you run Ansible version 2.4 or higher; Vault ID's.
Before I continue with the explanation on how to set them up, I'll describe how to use the password to finish of the quick overview. To use an ansible vault id password you must type in the following: `--vault-id passfile1@vault1` at the end of your command forming `ansible-playbook example.yml –vault-id pass1@vault_pass_file.txt` for the vault id version. Do note that using vault id you can have multiple passwords inside of 1 file. The needed password is specified with "pass1" In this case from the "vault_pass_file.txt" file. In case you'd like to enter the password manually you can also use `–vault-id pass1@prompt` to be prompted for the file.
And now for the actual explanation of Vault IDs.
Simply Vault IDs provide a way to identify passwords from a source thanks to a label.

```
[defaults]
inventory = inventory
remote_user = root
vault_identity_list = inline@~/ansible/.inline_pass , files@~/ansible/.files_p
ass
```

`--vault-id label@source` in this case ansible vault would look for a password labeled "label" in a file called "source". Alternatively, you can use `--vault-id label@prompt` to have Ansible prompt you for the password.

However, it is not enough to type in the above code snippets as Vault IDs need to be preconfigured in your ansible.cfg file before use. As an example, we have 2 password files declared here: inline and files at their corresponding locations. Both contain an unencrypted passphrase with the password and will have to be called using inline@prompt or inline@file_source (~/ansible/.inline_pass)

7.1.3 Encrypt_string output breakdown:

Seeing as the encrypt_string output might be confusing at first, we will also explain what all the visible variables mean from the initial code snippet to the output.

```
ansible-vault encrypt_string --vault-password-file pass.txt 'testing' --
name 'secret_password'

Secret_password: !vault |
     $ANSIBLE_VAULT;1.1;AES256
     62313365396662343061393464336163383764373764613633653634306231386433626436623361
     6134333665353966363534333632666535333761666131620a663537646436643839616531643561
     63396265333966386166373632626539326166353965363262633030333630313338646335303630
     3438626666666137650a353638643435666633363396436633863306662323461643237323133331
     6564
```

Breakdown:

- The variable name **secret_password**, followed by !vault |, indicates that the vault is **encrypted**.
- The vault version that supports the vault ID is **1.1.**
- The AES cipher in 256 bits is represented by **AES256**.

--vault-id example

```
ansible-vault encrypt_string --vault-id test@source 'testing' --
name 'secret_password'

Secret_password: !vault |
         $ANSIBLE_VAULT;1.2;AES256;test
         30613233633461343837653833666333643061636561303338373661313838333565653635353162
         3263363434623733343538653462613064333634333464660a663633623939393439316636633863
         61636237636537333938306331383339353265363239643939666639386530626330633337633833
         6664656334373166630a363736393262666465663432613932613036303963343263623137386239
         6330
```

Breakdown:

- The variable name **secret_password**, followed by !vault |, indicates that the vault is **encrypted**.
- The vault version that supports the vault ID is **1.2.**
- The AES cipher in 256 bits is represented by **AES256**.
- The vault ID in use is **test**.

Tip

To speed up the decryption process at startup it is recommended to install the cryptography package using pip install cryptography.

7.2 Ansible Galaxy

7.2.1 Why you might want to consider Ansible Galaxy:

As promised at the beginning of this chapter we will also look at something that might help you speed up the process of automation.we're talking about *Ansible Galaxy* of course*.* Ansible Galaxy is a community driven tool. Or rather a repository of community made roles and collections for all your automation needs.
The reason you may want to consider using the resources provided by Ansible Galaxy is quite simple: *Time.*

Every time we come up with the need to automate we tend to be either short on time or not knowledgeable enough of the subject to be efficient in the process. That's where something like a community made resource can come in handy as a reference or a quick and dirty way to see how far along you are with your test environments without the need to start from ground up.

Warning

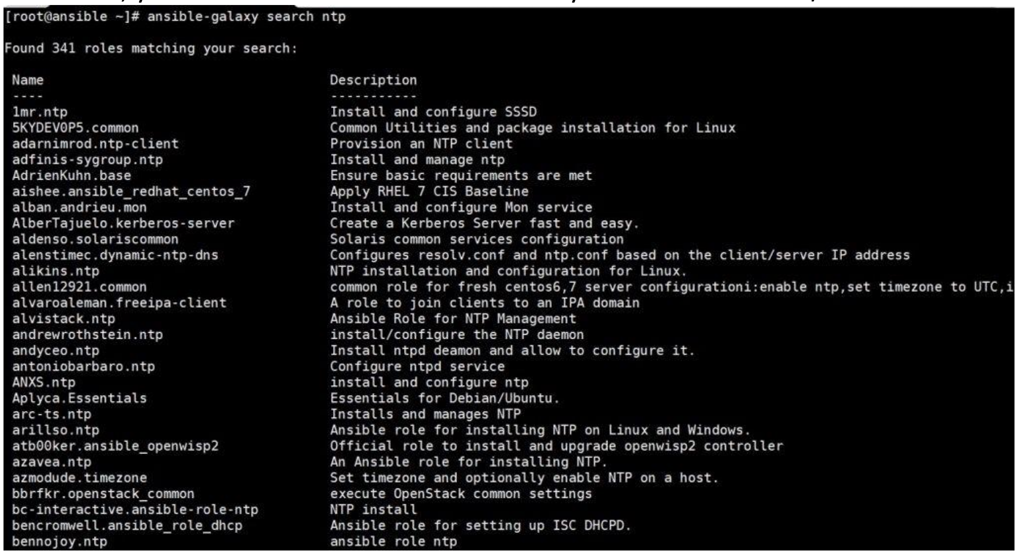
Ansible galaxy provides community driven resources therefore we advise to either read and edit the files you download and or run them first in your test environment before you deploy them in your production environment.

7.2.2 How does someone find content on Ansible Galaxy?

Now there are 2 main ways to search for content available on Ansible Galaxy.

- **CLI**

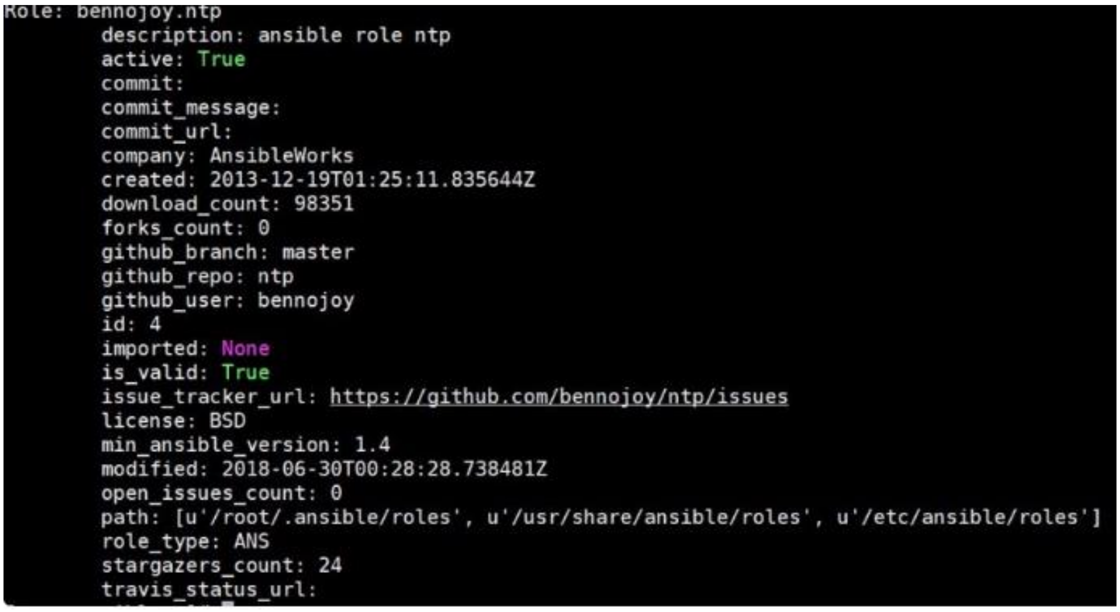
CLI is the quick and dirty way seeing as it doesn't provide detailed information (right of the bat) except for the provided description and the name of your search. The way to do this is as follows:
On the machine you've downloaded and installed Ansible you type in:
`ansible-galaxy search xxxx` (substitute xxxx for your search).
As a general rule of thumb the Top results will show you the most popular and therefore (more often than not) most robust roles.
Of course, it is not impossible to get more information and following the example of a NTP ansible galaxy search we'll show you the steps.
First of all, you want to find a suitable NTP role you'd want to install/download.



```
[root@ansible ~]# ansible-galaxy search ntp

Found 341 roles matching your search:

 Name                                   Description
 ----                                   -----------
 1mr.ntp                                Install and configure SSSD
 5KYDEV0P5.common                       Common Utilities and package installation for Linux
 adarnimrod.ntp-client                  Provision an NTP client
 adfinis-sygroup.ntp                    Install and manage ntp
 AdrienKuhn.base                        Ensure basic requirements are met
 aishee.ansible_redhat_centos_7         Apply RHEL 7 CIS Baseline
 alban.andrieu.mon                      Install and configure Mon service
 AlberTajuelo.kerberos-server           Create a Kerberos Server fast and easy.
 aldenso.solariscommon                  Solaris common services configuration
 alenstimec.dynamic-ntp-dns             Configures resolv.conf and ntp.conf based on the client/server IP address
 alikins.ntp                            NTP installation and configuration for Linux.
 allen12921.common                      common role for fresh centos6,7 server configurationi:enable ntp,set timezone to UTC,i
 alvaroaleman.freeipa-client            A role to join clients to an IPA domain
 alvistack.ntp                          Ansible Role for NTP Management
 andrewrothstein.ntp                    install/configure the NTP daemon
 andyceo.ntp                            Install ntpd deamon and allow to configure it.
 antoniobarbaro.ntp                     Configure ntpd service
 ANXS.ntp                               install and configure ntp
 Aplyca.Essentials                      Essentials for Debian/Ubuntu.
 arc-ts.ntp                             Installs and manages NTP
 arillso.ntp                            Ansible role for installing NTP on Linux and Windows.
 atb00ker.ansible_openwisp2             Official role to install and upgrade openwisp2 controller
 azavea.ntp                             An Ansible role for installing NTP.
 azmodude.timezone                      Set timezone and optionally enable NTP on a host.
 bbrfkr.openstack_common                execute OpenStack common settings
 bc-interactive.ansible-role-ntp        NTP install
 bencromwell.ansible_role_dhcp          Ansible role for setting up ISC DHCPD.
 bennojoy.ntp                           ansible role ntp
```

Image 1: Search for NTP with Ansible Galaxy

Following that you'd type in the following in your CLI `ansible-galaxy info xxxx` (substitute xxxx for the name of your search – bennojoy.ntp for out example) .



```
Role: bennojoy.ntp
        description: ansible role ntp
        active: True
        commit:
        commit_message:
        commit_url:
        company: AnsibleWorks
        created: 2013-12-19T01:25:11.835644Z
        download_count: 98351
        forks_count: 0
        github_branch: master
        github_repo: ntp
        github_user: bennojoy
        id: 4
        imported: None
        is_valid: True
        issue_tracker_url: https://github.com/bennojoy/ntp/issues
        license: BSD
        min_ansible_version: 1.4
        modified: 2018-06-30T00:28:28.738481Z
        open_issues_count: 0
        path: [u'/root/.ansible/roles', u'/usr/share/ansible/roles', u'/etc/ansible/roles']
        role_type: ANS
        stargazers_count: 24
        travis_status_url:
```

Image 2: Get the data from the NTP download

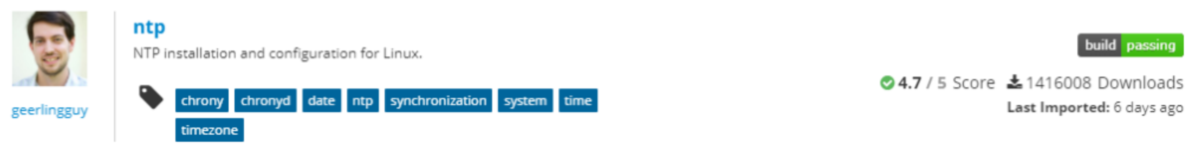
And by repeating these 2 steps you can get to know what your search is all about and on which versions it works.

- **Through their website**

Now the other way is to use the Ansible Galaxy website https://galaxy.ansible.com/home
This way you can search by:

- Type (e.g Development, Networking etc.)
- Keywords (through plain search)
- Community authors (for your trusted authors and their commits)
- Partners (not a standardized option but available through https://galaxy.ansible.com/partners)

Whilst the end results would be the same with a list of search results you will also be provided with some additional information like the amount of downloads, community score, whether the code compiles, and the last time it was imported. With even more detailed information once the search result is clicked and links to GitHub so that you can first investigate the resource before you download it.



geerlingguy

ntp

NTP installation and configuration for Linux.

chrony chronyd date ntp synchronization system time timezone

build passing

4.7 / 5 Score 1416008 Downloads

Last Imported: 6 days ago

Image 3: Web download of NTP

How does someone find content?
As with every previous theme around Ansible Galaxy installation and usage is simple and fast. Ansible Galaxy is installed by default with your Ansible installation and to install a role you've found you simply need to type in the following in your CLI: `ansible-galaxy install contributor_name.role`.

Note
All ANSIBLE Galaxy roles will be stored in /etc/ansible/roles directory by default. To change the directory in which the roles are installed you need to modify your installation command to: ansible-galaxy install --roles-path ~/ansible-roles contributor_name.role OR to modify your ansible.cfg file roles_path variable.

Now there are a couple of variables you can use to change the behavior of your installation provided that your chosen role allows for it. The options are:

- Installing a specific version by adding a coma and the version number as stated below
    - `ansible-galaxy install geerlingguy.apache,v1.0.0`
- Installing a specific GIT commit by following install with git+branch name or git+commit hash
    - `ansible-galaxy install git+https://github.com/geerlingguy/ansible-role-apache.git,0b7cd353c0250e87a26e0499e59e7fd265cc2f25`
- Installing multiple roles from one or multiple files
    - To do this you'd make a file named `requirements.yml` with .yml or .yaml extension. And execute the following command: `ansible-galaxy install -r requirements.yml`. Below you can find an example requirements.yml file referenced from the Galaxy documentation.

Tip
To list your installed roles type in: ansible-galaxy list.

```
 # from Galaxy
- src: yatesr.timezone

# from GitHub
- src: https://github.com/bennojoy/nginx

# from GitHub, overriding the name and specifying a specific tag
- src: https://github.com/bennojoy/nginx
  version: master
  name: nginx_role

# from a webserver, where the role is packaged in a tar.gz
- src: https://some.webserver.example.com/files/master.tar.gz
  name: http-role

# from Bitbucket
- src: git+http://bitbucket.org/willthames/git-ansible-galaxy
  version: v1.4

# from Bitbucket, alternative syntax and caveats
- src: http://bitbucket.org/willthames/hg-ansible-galaxy
  scm: hg

# from GitLab or other git-based scm
- src: git@gitlab.company.com:mygroup/ansible-base.git
  scm: git
  version: "0.1" # quoted, so YAML doesn't parse this as a floating-point value
```

Now seeing as this block has multiple variables added to the roles we will now go through them:

- **Src**
  - Source of the role and its required attribute.
- **Scm**
  - If the src field is a URL it needs to be specified with either git or hg. The default is set to *git*.
- **Version**
  - Version of the role. The following options are possible: tag value, commit hash or branch name. The default is set to *master.*
- **Name**
  - To override the name with which the role is installed.

And to install the roles from multiple files you'd have to edit your *requirements.yml* file with an `- include: <path_to_requirements>/additions.yml` whereby *additions.yml* file would have the same structure as *requirements.yml.*
To proceed with the installation, you'd simply provide the root file – in our case requirements.yml – with the same command as stated for singular installations.

```
ansible-galaxy install -r requirements.yml
```

Note

Some roles are dependent on other roles which are automatically installed. To view these dependencies please consult your /meta/main.yml file of your role. They will be listed below the dependencies keyword.

And last but not least if you wish to remove a certain role you can proceed with that using the following command: `ansible-galaxy remove role1 role2` with role1 and role2 being the names of the roles you wish to remove.

7.2.3 How does one contribute content?

The last topic around Ansible-Galaxy that we're going to explore is the creation and contribution of content.
Which will also introduce the last of the most common commands used with Ansible Galaxy `init`.
For example, if you'd decide to create a new role the command, you'd be looking for is:
`ansible-galaxy init my_new_role`

By doing this ansible-galaxy will create a new directory structure for you called my_new_role.
Which would contain the following tree:

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

Note

For a more detailed explanation on the directory structure please re-visit Chapter 5 of our manual.

Once you populate your role with the desired configurations the next step you will need to take is to populate the metadata of your role. Seeing as Galaxy takes a look into your metadata file for all its content information it is important to populate it as fully and concise as possible. You can find the default metadata file created by `init` command below.

```
galaxy_info:
  role_name: foo
  author: your name
  description: your description
  company: your company (optional)

  # If the issue tracker for your role is not on github, uncomment the
  # next line and provide a value
  # issue_tracker_url: http://example.com/issue/tracker

  # Some suggested licenses:
  # - BSD (default)
  # - MIT
  # - GPLv2
  # - GPLv3
  # - Apache
  # - CC-BY
  license: license (GPLv2, CC-BY, etc)

  min_ansible_version: 1.2

  # If this a Container Enabled role, provide the minimum Ansible Container
  # version.
  # min_ansible_container_version:

  # Optionally specify the branch Galaxy will use when accessing the GitHub
  # repo for this role. During role install, if no tags are available,
  # Galaxy will use this branch. During import Galaxy will access files on
  # this branch. If Travis integration is configured, only notifications for
  # this branch will be accepted. Otherwise, in all cases,
  # the repo's default branch (usually master) will be used.
  # github_branch:
  #
  # platforms is a list of platforms, and each platform has a name and a
  # list of versions.
  #
  # platforms:
  # - name: Fedora
  #   versions:
  #   - all
  #   - 25
  # - name: SomePlatform
  #   versions:
  #   - all
  #   - 1.0
  #   - 7
  #   - 99.99

  galaxy_tags: []
```

```
    # List tags for your role here, one per line. A tag is a keyword
    # that describes and categorizes the role. Users find roles by searching
    # for tags. Be sure to remove the '[]' above, if you
    # add tags to this list.
    #
    # NOTE: A tag is limited to a single word comprised of
    # alphanumeric characters.
    #       Maximum 20 tags per role.



dependencies: []

  # List your role dependencies here, one per line. Be sure to remove the '[]'
  # above,if you add dependencies to this list.
```

Whilst most of the information in your metadata file is optional the only required field is `Platforms.` In which you need to provide the platform that is compatible with your role and the versions you've tested on.
All the other fields are optional. After which you can upload your file to GitHub or Travis and from there import it to your Ansible Galaxy account.

The other "thing" you can create and share is your Playbook bundle. The principle is mostly the same with the command changing slightly to:
`ansible-galaxy init –type apb my_new_apb`

Following this command Ansible will create a directory named my_new_apb containing the following:

```
|.travis.yml
|Dockerfile
|Makefile
|README.md
|apb.yml
|defaults
|  |main.yml
|files
|handlers
|  |main.yml
|meta
|  |main.yml
|playbooks
|  |deprovision.yml
|  |provision.yml
|tasks
|  |main.yml
|templates
|tests
|  |ansible.cfg
|  |inventory
|  |test.yml
|vars
|  |main.yml
```

And as with roles once you're satisfied with the content you've written out in your playbook it is mostly important to populate your metadata file with all the relevant information so that the community has an easier time finding it.

Note

For the more advanced users RedHat also provides the "Red Hat Ansible Automation" subscription. Which contains certified and maintained roles, playbooks and newly added collections. Collections are basically fully functional environments with sample playbooks roles and scripts ready to go for your needs. The prices range from $5000 to $14000 depending on your support and node needs and provide access to Ansible Tower.

8. Ansible and Cisco IOS

8. Ansible and Cisco IOS

To give a better explanation of IOS configuration with Ansible, we will now setup a simple environment. With the following setup you will be able to retrieve data from the IOS devices and diagnose/troubleshoot your Ansible environment.

8.1 Configuration

In this section we will describe how to use ansible for the configuration of cisco device.

8.1.1 The environment

The file location of the environment is in the /etc/ansible directory. Within this directory we will add the following files: Hosts, Ansible.cfg and show.yml.

The hosts file will be the inventory file (described in chapter 2) containing the address information about the IOS devices. It will also contain the connection information and password for the IOS devices for Ansible to push/receive data from these devices.

The Ansible.cfg file is a configuration file for Ansible that is needed to disable SSH key checks when connecting to a device with SSH. Because this is a test environment and the IOS devices are not setup with SSH keys this must be disabled or Ansible will refuse to communicate with the devices.

The show.yml is the playbook that we will create to retrieve data from the IOS devices.

8.1.2 Disable SSH key checks

On order for the test environment to work we first will create the Ansible.cfg file in the directory /etc/ansible. Once you have done that you can add the following into the file:

```
[defaults]
host_key_checking = no

[paramiko_connection]
host_key_checking = no
```

8.1.3 Setting up the inventory file

In order to push configurations to the IOS devices we need to specify the addresses of the IOS devices. To do this we will create the file hosts in the directory /etc/ansible Within this file we will create a group called IOS and there specify the ip-addresses of our IOS devices. Our test environment will consist out of 2 routers. After we have specified the addresses of the IOS devices the connection method and password of these devices also must be configured for Ansible in order to use these devices.
The first part of the hosts file we use for the test environment contains the following entries:

```
[IOS]
Router1 ansible_host=10.0.1.14
Router2 ansible_host=10.0.1.15
```

The layout of the file (as mention in chapter 3) is as follows:

```
(hostname) ansible_host=(ip-address)
```

Then we will add the following entries to the hosts file in order to setup the connection type and password that Ansible will use to communicate with the IOS devices:

```
[all:vars]
ansible_become=yes
ansible_become_method=enable
ansible_network_os=ios
ansible_connection=network_cli
ansible_user=cisco
ansible_password=cisco
```

Ansible uses existing privileges to execute tasks with root privileges or with another's user permissions. Gaining these permissions can be achieved using the become statement . In our example these are the lines: *ansible_become=yes* and *ansible_become_method=enable*. The third and fourth line are to determine how to connect to a remote device and which network platform the host needs to correspond to. The last two lines are to set a user with password to remote log in.

With all these entries added the hosts file will contain the following:

```
[IOS]
Router1 ansible_host=10.0.1.14
Router2 ansible_host=10.0.1.15

[all:vars]
ansible_become=yes
ansible_become_method=enable
ansible_network_os=ios
ansible_user=cisco
ansible_password=cisco
ansible_connection=network_cli
```

8.1.4 Setting up a Playbook

In the last file that we will create for the test environment we will configure a playbook that sends a command to the IOS devices and then retrieves the information that the IOS device gives. For this we will create the file show.yml in the /etc/ansible directory with the following entries:

```
- name: Get data
  hosts: IOS
  tasks:

    - name: Gather interface data.
      ios_command:
        commands:
          - show ip int brief
      register: if_data

    - name: Interface output
      debug:
        var: if_data['stdout_lines'][0]
```

In the top of the Playbook we specify the name and what devices are used with the “hosts:” line. This is a reference to the hosts file where we specified the devices under the group [IOS].
Once we have done that, we create a task that gathers the information of the show ip interface brief command. This is done by using ios_command module, within this module we will use “commands:” to list the commands Ansible will run on the IOS devices. In the last part we take the information that the IOS devices gives us and we save it with the name if_data.

```
- name: Gather interface data.
      ios_command:
        commands:
          - show ip int brief
      register: if_data
```

Note

The commands will be executed from top to bottom. So, if we would add the 2 lines:

- show ip int brief
- show run

The show ip int brief command will be executed first. This is important because some commands for IOS devices are used in different modes.

In order to display the data we requested from the IOS devices we must use the following lines:

```
- name: Interface output
      debug:
        var: if_data['stdout_lines'][0]
```

With this we request the data in the variable if_data.

You can now run the playbook with the following line:

```
ansible-playbook show.yml
```

The result should be as follows:

```
PLAY [Get data]
TASK [Gathering Facts] ***** ******** ****** **** ***** ***** ***** *****
ok: [Router2]
ok: [Router1]
TASK [Gather interface data.] ***** ******** ****** **** ***** ***** *****
ok: [Routerl]
ok: [Router2]
TASK [Interface output) ***** ******** ****** **** ***** ***** ***** *****
ok: (Router1) -> {
    "if_data[ istdout_lines' ][0]": [
        "Interface           IP-Address        OK?         Method          Status                 Protocol",
        "GigabitEthernet1    10.0.1.14         YES         DHCP            up                     up      ",
        "GigabitEthernet2    unassigned        YES         NVRM            down                   down    ",
        "GigabitEthernet3    unassigned        YES         NVRM            administratively down  down    ",
        "GigabitEthernet4    unassigned        YES         NVRM            administratively down  down    ",
        "Loopback1           192.168.0.3       YES         NVRM            up                     up      ",
        "Loopback2           192.168.1.3       YES         NVRM            up                     up      "
    ]
}
ok: (Router2) -> {
    "if_data[ istdout_lines' ][0]": [
        "Interface           IP-Address        OK?         Method          Status                 Protocol",
        "GigabitEthernet1    10.0.1.15         YES         DHCP            up                     up      ",
        "GigabitEthernet2    unassigned        YES         NVRM            administratively down  down    ",
        "GigabitEthernet3    unassigned        YES         NVRM            administratively down  down    ",
        "GigabitEthernet4    unassigned        YES         NVRM            administratively down  down    ",
        "Loopback1           192.168.0.4       YES         NVRM            up                     up      ",
        "Loopback2           192.168.1.4       YES         NVRM            up                     up      "
    ]
}
```

8.2 Cisco IOS modules

In this section we will describe how to use ansible for the configuration of cisco device.

8.2.1 ios_logging

This module provides the logging of the Cisco IOS devices.

```
- name: configure host logging
  cisco.ios.ios_logging:
    dest: host
    name: 172.28.10.1
    state: present
```

The example above configures a host logging. The destination of the logs in set on host, the name is set with an IP-address of the destination. It is required to set the name when *dest* is set on *host.*

8.2.2 ios_facts

This module collects a base set of device facts from a device. The module will always collect a base set of facts from a device and can enable or disable the collection of additional facts.

```
- name: Gather all legacy facts
  cisco.ios.ios_facts:
    gather_subset: all

- name: Gather only the configuration and default facts
  cisco.ios.ios_facts:
    gather_subset:
    - config
```

In the first part of the example above we want to gather all facts. In the second part we gather only the configuration facts. With the line gather_subset you can gather the different facts that you want to collect.

8.2.3 ios_system

This module allows the management to configure host system parameters or remove parameters from the Cisco IOS devices.

```
- name: configure hostname and domain name
  cisco.ios.ios_system:
    hostname: Ansible_Cisco_IOS
    domain_name: test.example.com
```

In this example we configured a hostname and a domain name. The host and domain name can also be deleted with the following:

```
- name: remove complete configuration
  cisco.ios.ios_system:
    state: absent
```

8.2.4 ios_user

This module provides management of the local usernames configured on the network devices. This module also allows playbooks to manage each username individual or the total of the usernames in the current running configuration. It also supports deleting the usernames from the configuration that are not explicitly defined.

You can create a new user as follows:

```
- name: create a new user
  cisco.ios.ios_user:
    name: ansible
    nopassword: true
    sshkey: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"
    state: present
```

In this example a new user is created without a password. This allows the user to login to the system without being authenticated by a password. In this example SSH key are used. This key will be found in a file that is given as a parameter. The last line sets the state of the username, when the state is set on present, the username should be configured in the active running configuration. When the state is set to absent the username should not be in the active configuration.

8.2.5 ios_vlan

This module provides management of the VLANs on Cisco IOS devices.

```
- name: Create vlan
  cisco.ios.ios_vlan:
    vlan_id: 10
    name: testvlan
    state: present

- name: Add interfaces to VLAN
  cisco.ios.ios_vlan:
    vlan_id: 10
    interfaces:
    - GigabitEthernet0/0
    - GigabitEthernet0/1

- name: Check if interfaces is assigned to VLAN
  cisco.ios.ios_vlan:
    vlan_id: 10
    associated_interfaces:
    - GigabitEthernet0/0
    - GigabitEthernet0/1

- name: Delete vlan
  cisco.ios.ios_vlan:
    vlan_id: 10
    state: absent
```

In the example above we create a VLAN and added interface to the VLAN. After the interface are added we checked if the interfaces were added to the VLAN. In the last part of our example we deleted the VLAN. The only crucial part of adding, deleting or assigning interface to a VLAN is the vlan_id, without this your playbook won't work or the wrong VLANs are deleted.

8.2.6 ios_ping

This module tests the reachability using a ping to a remote destination. For Windows targets, you have to use the win_ping module instead.

```
- name: Ping 10.0.0.4 from the source using prod vrf and setting a count
  cisco.ios.ios_ping:
    dest: 10.0.0.4
    source: loopback1
    vrf: prod
    count: 20
```

In the example above we want to ping 10.0.0.4 from the destination loopback1. This is defined in the third and fourth line. In the last line we define the number of packets to send to the destination host.

8.2.7 ios_banner

This module configures the banner for both login and Message Of The Day (MOTD) on the remote devices that are running Cisco IOS. It also allows a playbook to add or remove banner text from a running configuration.

```
- name: configure the login banner
  cisco.ios.ios_banner:
    banner: login
    text: |
      Welcome to the Ansible guide.
      This guide will help you to
      install and understand Ansible
    state: present

- name: remove the MOTD banner
  cisco.ios.ios_banner:
    banner: MOTD
    state: absent
```

In the first part of our example we configure a login banner with the text: "Welcome to the Ansible guide. This guide will help you to install and understand Ansible". In the second part of the example we delete the MOTD from the running configuration.

9. Task automation

9. Task automation

In chapter 8 we started with the configuration of Cisco IOS devices with their hostnames and loopback interfaces. But Ansible has more modules for IOS, whether they are for interfaces or just to set the banner for the login, combining these modules could create powerful automated configuration of Cisco IOS devices. In this chapter we will go through some configuration examples and what exactly happens.

9.1 Environment

In chapter eight we described what the entries in the hosts file are, the entries in Ansible.cfg are just to make Ansible skip the SSH key checking. Thus, when adding a switch to test it will not require me to trust the key. I recommend to only keep the password in plain text for a development environment.

9.1.1 Environment used

The environment used has been configured with the following files in the /etc/ansible directory.

Hosts:

```
[all:vars]
ansible_become=yes
ansible_become_method=enable
ansible_network_os=ios
ansible_user=cisco
ansible_password=cisco
ansible_connection=network_cli

[IOS]
Router1 ansible_host=10.0.1.14
Router2 ansible_host=10.0.1.15

[Switches]
Switch1 ansible_host=10.0.1.254
Switch2 ansible_host=10.0.1.16
```

Ansible.cfg:

```
[defaults]
host_key_checking = no
[paramiko_connection]
host_key_checking = no
```

9.1.2 OSPF example

Taking OSPF as example to configure on a Cisco IOS device. For example, the following code can be used to configure OSPF on IOS devices.

```
    - name: Set OSPF.
      ios_config:
        parents: router ospf 1
        lines:
          - network 10.2.0.0 0.0.0.255 area 0
```

Taking a closer look to the code we see that the module “ios_config” is used to configure OSPF, this module is used to configure Cisco IOS devices in configuration mode. This configuration can range from configuring the banner of a login to setting up SNMP.
As the name parents state, it will run the line(s) of code after getting into the router ospf 1 configuration mode. Thus, any lines that are added to “lines:” will be run in it. This could also be useful when setting up loopback interfaces or configuring physical interfaces.

9.2 Using host variables

In chapter 3 we talked about the host_vars and where they could be located, in this chapter we are working with the following directory structure:

```
Ansible
|host_vars
| | Switch1.yml
| |Switch2.yml
|Example.yml
```

Ansible will automatically look up a host file, whether it is in the working directory or in /etc/ansible/…, based on the names/IP addresses specified in the host file in/etc/ansible/host. With these variables we could easily create an ansible playbook that could serve as template as any data in it will be retrieved from the host vars file.

9.2.1 Configuration contents

Let's look to the following files and their configuration.

Note

Switch1.yml and Switch2.yml are almost the same config, except a different IP address.

Example.yml:

```
---
- name: Create loopback
  hosts: Switches
  tasks:
    - name: Set Loopbacks.
      with_items: "{{ local_loopback }}"
      ios_config:
        lines:
          - description {{ item.desc }}
          - ip address {{ item.ip_address }}
        parents: interface {{ item.name }}

    - name: save running to startup when modified
      ios_config:
        save_when: modified
```

Switch1.yml:

```
---
local_loopback:
  - name: Loopback1
    desc: 'Sample config'
    ip_address: 192.168.2.3 255.255.255.0
  - name: Loopback2
    desc: 'Sample config'
    ip_address: 192.168.3.3 255.255.255.0
```

The name "Switches" is specified, in the host file of Ansible I specified the group "Switches" contains "Switch1" and "Switch2", this will result that the playbook will be run on every device added to the group "Switches". The default behaviour for Ansible is to look for <ip_address>.yaml in any of the known hosts_vars locations. This could be either the current work directory or /etc/ansible/... . But in our case where the hosts are given a hostname in addition to their IP address Ansible will look for Switch1.yml and Switch2.yml. From here on we will call "Example.yml" the playbook, as it is an Ansible playbook.

In Switch1.yml I specified a variable local_loopback with two entries, named Loopback1 and Loopback2 both with their own IP address and description. Keep in mind that; name, description and ip_address are nothing more than names for variables.

In the playbook we want to configure an interface based on details specified in the host_var file "Switch1.yml" and "Switch2.yml". Taking a closer look again we see that the playbook does not contain any predefined details anymore, as told before this playbook will be used as template to configure both switches.

As explained before, Ansible will automatically look for any host file with the same name as the items in the "hosts" file, thus there is no need to specify any location or file that needs to be used. To make things easier I will only mention Switch1 from now on, but keep in mind that with the specified group for the hosts in the playbook, it will run on both Switch1 and Switch2.

9.2.2 With items

We also added the line: "with_items:", what this does is, ansible will look for an entry in the host_vars file of Switch1 with the name local_loopback resulting in Ansible knowing and able to provide data to where we need it later on in the task of configuring the loopback interfaces. "local_loopback" is also here nothing more than a variable name for data in it.

With the line "with_items" now defined we are able to get data from the file through calling {{ item.<variable> }}, in the line "parents" we want to get the name of the loopback addresses thus we need to call the variable {{ item.name }}, this will result in Ansible setting "interface loopback1" as parent for the configuration that will follow. As explained above, the variables in "lines" will result in: 'Sample config' and 192.168.2.3 255.255.255.0.

Remember that we talked before about how Ansible will run this playbook on both switches? The same logic could be applied to what happens next, in "Switch1.yml" we defined two names. This will result in ansible running the configuration for the parents: "interface Loopback1" and "interface Loopback2" with each their defined description and IP address.

Note

You may have noticed that the line "parents" is now underneath the line of "lines", Ansible will read the task first and run the code in the order defined in the "ios_module" thus looking at "parents" first before running the code in "lines".

Note

The quotes around the variables are only needed when the line the variable call is in is not a string, thus "with_items" will require quotes as Ansible needs to look for a variable and not a directory.

9.3 Cisco IOS configuration

This module may be one of the most important modules in Ansible for Cisco IOS, as you are able to change anything with it from a banner to configuring a VPN. But there are also a few more configuration options in the module to help you configure a Cisco IOS device, and I will provide some information of some of them.

9.3.1 Before & after

Either command could be used in any task to run a command, as the name implies, before or after a task. For example, we want to set some access list rules but we want to remove the old configuration first. Thus:

```
- name: load new acl into device
  ios_config:
    lines:
      - 10 permit ip host 192.0.2.1 any log
    parents: ip access-list extended test
    before: no ip access-list extended test
    match: exact
```

This will result in the list test being deleted before the list will be created again and configured. The command after will run after the task has configured the things specified.

9.3.2 Replace

The replace command could be quite useful when you want to be sure you want to replace just one line or a whole block. For example, we want to make sure an access list is exact the same as provided thus adding the line “replace” to the task.

```
- name: load new acl into device
  ios_config:
    lines:
      - 10 permit ip host 192.0.2.1 any log
    parents: ip access-list extended test
    replace: block
```

9.3.3 Diff against

Another useful module is diff_against, this module allows you to compare the intended config to the running config. Keep in mind when running this module in a playbook the argument --diff need to be passed with the command to run the playbook to show the results.

```
- name: check the running-config against master config
  ios_config:
    diff_against: intended
    intended_config: "{{ lookup('file', 'master.cfg') }}"
```

With the configuration as is, Ansible will look for a file named “master.cfg” and will compare the running config against the master config, and with the --diff argument passed with the command it will return the results of what is different between the two files.

9.3.4 Save

Another feature I would like to mention is the save function of “ios_config”. With this feature we could tell the device to save its configuration based on what happened, whether we want it to happen always, never, modified or when changed. The first two explain themselves, when defined in a playbook it will always or never save the configuration. But the other two may be a bit confusing.

When the flag is set to modified it will only save the running config when it has changed since the last save, while with the flag set to changed it will save when the playbook made a change to the configuration.

10. From start to finish

10. From start to finish

For this last chapter we will go through an example scenario to demonstrate how a user like you could implement Ansible in your own network. The explanation below will follow a simple step by step guide with additional explanation provided by each step, detailing the usage and reasoning behind the taken step.

Before we continue it is recommended that you read through the previous chapters or at least have a basic understanding of both Ansible and Cisco IOS.

Case introduction:
The Company called "Company A" has taken over another company with 1 site. The site of that company was fully implemented with Cisco devices in 2019. Thanks to the fact that the previous company used Cisco devices it was decided to not replace the devices themselves and simply amend the existing configurations to "Company A" standards.
The network administrators decided to streamline the process using Ansible making it as idempotent as possible. After thorough inspection it was decided to amend the following configurations:

- MOTD banner
- ACL's
- VLANs
- SNMP

Requirements to follow along with the walkthrough:

- (Home/Virtual) Lab environment containing: 2 Routers, 2 L3 Switches and 2 L2 switches.
- Working knowledge of virtualization environments (for the Ansible server) or a physical Linux host.
- General understanding of IOS commands and Ansible playbooks (how to save, edit and run).

10.1 Step 1 – Current Network and Init

Initial setup and configurations:
To start off we would strongly advise you to copy our Topology to your lab environment of choice and configure the devices as stated in the code blocks named after the device. This will provide you with the smoothest start as it won't require any additional steps which are not covered by this walkthrough. In case you've been following along with chapter 9 as well you can keep using your current configuration as the ansible playbooks used would lead you to our base configurations. Otherwise just copy the configurations below into your devices CLI.

Topology:

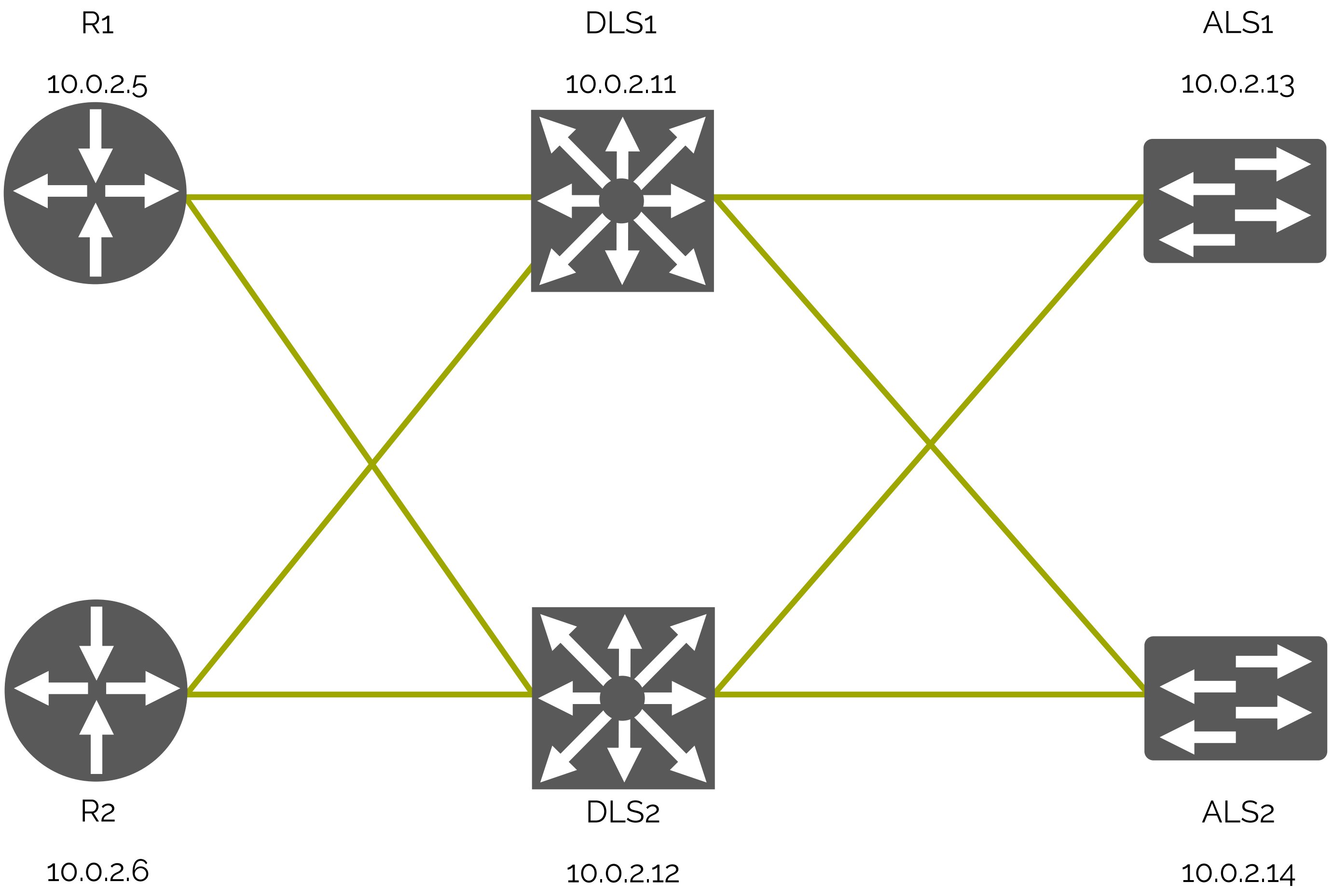


Image 4: Architecture example

The configuration files for the different devices can be downloaded by clicking on the devices above, or by downloading them as a compressed folder with this link.

10.2 Step 2 – Setting up your Linux and Ansible

Linux Host
Now that the Cisco devices are configured it's time to get the Linux Host in working order. For walkthrough of this walkthrough we will use Ubuntu 20.04 as it's the most commonly used Linux system - you can use any distro you want but for the sake of streamlining the walkthrough we will proceed with Ubuntu 20.04. In case you chose a different distro please refer to Chapter 2 for all the requirements and installation steps.

Once you get your Linux machine installed by going through all the steps, open up your CLI and type in the following commands.

```
sudo apt update
sudo apt install software-properties-common
sudo apt-add-repository --yes --update ppa:ansible/ansible
sudo apt install ansible
```

These commands will update your distribution, add the required repository for Ansible and install Ansible. After completing the installation steps we recommend to test out if Ansible is working by using the following command in your CLI:

```
ansible all -m ping -v
```

If everything went correctly you are done with the basic configuration and you're ready to proceed onto the Ansible configuration proceed to implementing Ansible for this specific use case. If you received an error try to read over the guide again and verify you've followed the steps correctly.

10.3 Step 3 – Creation of the Inventory file

Now that everything is set up and the connections have been tested it's time to aggregate the addresses into 1 inventory file for Ansible to use. To do this we need to open up the CLI on our Linux host and navigate to the /etc/ansible directory.

```
cd /etc/ansible
```

Once we're there we will add a file called "Hosts". This file will provide us with a working environment for Ansible.
The Hosts file will be the inventory file (as described in Chapter 2) containing the address information about the IOS devices. It will also contain the connection information and password for the IOS devices for Ansible to push/receive data from these devices.

```
Sudo nano Hosts
```

This will open up the Nano editor and enable us to type in all the devices which would be used later on with our playbooks. For now you can use the code provided below and once you're done we will explain the code line by line or you can revisit Chapter 3 for a detailed explanation on Inventory files.

```
[all:vars]
ansible_become=yes
ansible_become_method=enable
ansible_network_os=IOS
ansible_user=cisco
ansible_password=cisco
ansible_connection=network_cli

[Routers]
Router1 ansible_host=10.0.2.5
Router2 ansible_host=10.0.2.6

[L2Switches]
Switch1 ansible_host=10.0.2.13
Switch2 ansible_host=10.0.2.14

[L3Switches]
DLS1 ansible_host=10.0.2.11
DLS2 ansible_host=10.0.2.12
```

Table 1: explanation inventory file

Hosts file	Explanation
[all:vars]	Marking followed by all set up variables.
ansible_become=yes	Root access for Ansible.
ansible_become_method=enable	Enables the change of privilege modes.
ansible_network_os=IOS	Defines the OS of targets for Ansible.
ansible_user=cisco	Defines the user which Ansible will use to connect.
ansible_password=cisco	Defines the password which Ansible will use to connect.
ansible_connection=network_cli	Defines where Ansible will connect to.
[IOS]	Groups the devices under it to group called "IOS".
Router1 ansible_host=10.0.1.14	Sets up name and tells Ansible which IP the host has.

Finally we can test our groups by using the ping Ansible command again. Instead of defining the all group, the newly created Routers, L2Switches or L3Switches groups can be used to only ping the members of said group.

```
ansible Routers -m ping -v
ansible L2Switches -m ping -v
ansible L3Switches -m ping -v
```

10.4 Step 4 – Usage of playbooks and modules

Having achieved connectivity using Ansible, we will now discuss the modules and playbooks that we will use to amend the configuration as we discussed in the introduction. (For an explanation on modules please revisit chapter 6)

To streamline the commands that need to be used, we will use Ansible Galaxy. To find any relevant modules simply go to *galaxy.ansible.com* and search for Cisco. This returns multiple modules and collections provided by Cisco. We will use the collection cisco.IOS . To install this collection go to your Ansible host machine and, using the CLI, enter the following command:

```
ansible-galaxy collection install cisco.IOS
```

The addition of this collection will change the way we configured Cisco devices (as described in Chapter 9) to a more specialized way of doing instead of the broad commands that were available with IOS_config.

What we want to achieve with Ansible playbooks is idempotency. Idempotency means that whatever function you call it will always provide the same result without fault. To explain it with a simple mathematical example – think of multiplication by zero. It doesn't matter how often you will do it, the result will always remain the same zero.
With automation you don't want to configure every device separately, that wouldn't save you any time at all, but instead you want to be able to send a task to multiple machines at once, where the tasks are repeatable but the result will stay fixed. And that's exactly what we're trying to achieve with the new configurations for the network devices.

To recap what we needed to do from the introduction step. We need to implement a new *Banner* message, set up *ACLs*, configure *VLANs*, install and check the implementation of *SNMP*.

Starting off with the banner**.** The banner is something that's company-wide for the majority of devices. For a banner we can make a playbook that will use all the hosts from our inventory file. And because we installed the cisco.ios collection it might be a good idea to check the modules included, to see if there are any specific modules that will help us with the setup.

This can be done by revisiting Ansible galaxy website and following the links provided until the GitHub explanation of provided modules or by visiting https://github.com/ansible-collections/cisco.IOS/tree/main/docs .

Using the explanation provided in the GitHub docs and combining it with the examples provided from Cisco we can now easily make a playbook that will be idempotent and will satisfy our needs.

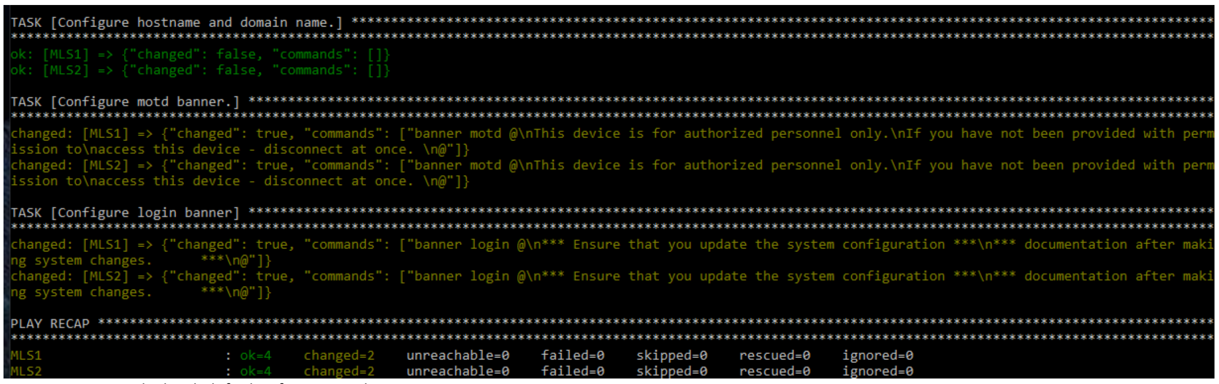
```
---
- name: Configure default info on all devices.
  hosts: all
  tasks:
    - name: Configure hostname and domain name.
      cisco.ios.ios_system:
        hostname: "{{ inventory_hostname }}"

    - name: Configure motd banner.
      cisco.ios.ios_banner:
        banner: motd
        text: |
          This device is for authorized personnel only.
          If you have not been provided with permission to
          access this device - disconnect at once.
        state: present

    - name: Configure login banner
      cisco.ios.ios_banner:
        banner: login
        text: |
          *** Ensure that you update the system configuration ***
          *** documentation after making system changes.      ***
        state: present
```

To reiterate what is happening in this playbook we will go through it line by line. Or you can revisit Chapter 4 and learn more about playbooks in detail.

Table 2: Explanation playbook banner

Configuration	Explanation
- name	The name to identify what our playbook does.
hosts: all	In this case all hosts contained in our inventory file.
- name	Name of a specific task (will be included in the output window).
cisco.ios.ios_banner	Name of the module used – when specified all its commands can be used below in the same task
banner: login	Specifies which banner should be configured on the remote device. In Ansible 2.4 and earlier only login and motd were supported.
text :	The banner text that should be present in the remote device running configuration. This argument accepts a multiline string, with no empty lines. Requires state=present.
state:	Specifies whether or not the configuration is present in the current devices active running configuration.

Running this playbook would configure the hostnames and domain name. These names are defined in the inventory file. The output of the playbook would look like the following in image 6.



```
TASK [Configure hostname and domain name.] *************************************************************************
*******************************************************************************************************************
ok: [MLS1] => {"changed": false, "commands": []}
ok: [MLS2] => {"changed": false, "commands": []}

TASK [Configure motd banner.] **************************************************************************************
*******************************************************************************************************************
changed: [MLS1] => {"changed": true, "commands": ["banner motd @\nThis device is for authorized personnel only.\nIf you have not been provided with perm
ission to\naccess this device - disconnect at once. \n@"]}
changed: [MLS2] => {"changed": true, "commands": ["banner motd @\nThis device is for authorized personnel only.\nIf you have not been provided with perm
ission to\naccess this device - disconnect at once. \n@"]}

TASK [Configure login banner] **************************************************************************************
*******************************************************************************************************************
changed: [MLS1] => {"changed": true, "commands": ["banner login @\n*** Ensure that you update the system configuration ***\n*** documentation after maki
ng system changes.      ***\n@"]}
changed: [MLS2] => {"changed": true, "commands": ["banner login @\n*** Ensure that you update the system configuration ***\n*** documentation after maki
ng system changes.      ***\n@"]}

PLAY RECAP *********************************************************************************************************
*******************************************************************************************************************
MLS1                       : ok=4    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
MLS2                       : ok=4    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Image 6: Output playbook default information devices

Next on our list are the ACLs. Every network needs some form of security and in our case we will place the ACLs on the Distribution Layer Switches which are L3 switches. Unlike the previous playbook we will now specify just one group with the ACLs allowing for an SSH connection to or from the 10.1.1.0, 10.1.2.0 and 10.1.3.0 networks.
We will accomplish this using the cisco.ios.ios_acls module.

```yaml
---
- name: Overwrite all ACL configuration.
  hosts: L3Switches
  tasks:
    - name: Merge provided configuration with device configuration
      cisco.ios.ios_acls:
        config:
        - afi: ipv4
          acls:
          - name: std_acl
            acl_type: standard
            aces:
            - grant: deny
              source:
                address: 192.168.1.200
            - grant: deny
              source:
                address: 192.168.2.0
                wildcard_bits: 0.0.0.255
        state: merged
```

Notable differences from the breakdown of our previous playbook are listed in the table 3 below.

Table 3: Explanation playbook ACL

Configuration	Explanation
hosts: L3Switches	Use all hosts from L3Switches group.
cisco.ios.ios_acls	A more general use module focused on line configuration.
- afi	The Address Family Indicator (AFI) for the Access Control Lists (ACL). This can be IPv4 or IPv6.
grant	This specifies the action. It can be permit or deny.

Going further down the list we arrive at VLANs**.** As our company uses two VLANs it is imperative that the switches know of them and are able to transfer the frames tagged with those VLAN-IDs.

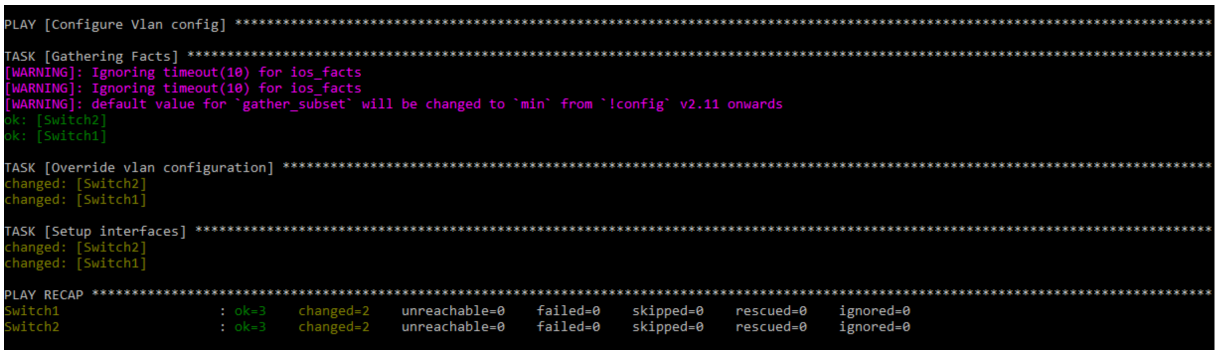
This time we only need to make these changes on the Layer 2 switches. So our goal is to create two new VLANs on each of the layer 2 switches and set the interfaces to access mode for the correct interfaces. As previously mentioned during the banner setup it's always a good idea to check whether or not your collection has modules specialized for your tasks which helps with idempotency and most of the time makes it an easier and clearer to read playbook.
The *cisco.ios* collection also provides us with an *ios_vlans* and *ios_l3_interfaces* module. These modules simplify the actions we need, and are easy to use owing to the well documented module information on the GitHub page of the collection with examples.

Having a clear goal and the help of the documentation we can now write our playbook containing 3 tasks. Creation of the first VLAN, second VLAN and lastly setting up the access mode on the interfaces for our VLANs.

```yaml
- name: Set vlans
  hosts: L3Switches
  tasks:
    - name: Configure Vlans
      cisco.ios.ios_vlans:
        config:
        - name: Users
          vlan_id: 10
          state: active
        - name: Servers
          vlan_id: 20
          state: active
        state: overridden

    - name: Replace L3 configuration
      cisco.ios.ios_l3_interfaces:
        config:
        - name: vlan10
          ipv4:
          - address: 10.0.3.1/24
        - name: vlan20
          ipv4:
          - address: 10.0.4.1/24
        state: replaced
    - name: Configure interfaces
      cisco.ios.ios_l2_interfaces:
        config:
        - name: Ethernet2/1
          mode: trunk
        - name: Ethernet2/2
          mode: trunk
        - name: Ethernet2/3
          mode: trunk
        state: replaced
```

With this we should be able to differentiate between each line of the playbook noticing the group change, different modules used their breakdown as it looks almost identical as to what you would normally write on your switch but slightly more organized.

The output of the VLAN playbook should give you the following output as in image 7.



```
PLAY [Configure Vlan config] ***********************************************************************************************************

TASK [Gathering Facts] *****************************************************************************************************************
[WARNING]: Ignoring timeout(10) for ios_facts
[WARNING]: Ignoring timeout(10) for ios_facts
[WARNING]: default value for `gather_subset` will be changed to `min` from `!config` v2.11 onwards
ok: [Switch2]
ok: [Switch1]

TASK [Override vlan configuration] *****************************************************************************************************
changed: [Switch2]
changed: [Switch1]

TASK [Setup interfaces] ****************************************************************************************************************
changed: [Switch2]
changed: [Switch1]

PLAY RECAP *****************************************************************************************************************************
Switch1                    : ok=3    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
Switch2                    : ok=3    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Image 7: Output playbook VLAN

Lastly we will create a playbook that checks for SNMP and collects the data, removes it and installs the newest version.
Seeing as Ansible works with a push model and each of our goals needs different results we will have to split each of our planned actions into different playbooks.

“Company A” as many other companies values data and wants to be up to date on the state of its devices to diagnose and improve over time. The usual configuration that they have runs on SNMP version 3. It is important to first check for the current diagnostics data and then upgrade the SNMP version that is set up on the new devices as they might have an older version configured.

While there is an SNMP specific module in the cisco.ios collection, the same idea as with ACLs applies. The cisco.ios.ios_config module provides an easy way to visualize and manage SNMP, as opposed to SNMP in the SNMP specific module.

So to check whether or not the devices are configured with SNMP we will run a show run command that grabs the lines which include “snmp-server” if they exist at all and show them directly in the output screen of the CLI on the Ansible host. To do this we should write the following playbook:

```
---
- name: Get SNMP data from all devices.
  hosts: all
  tasks:

    - name: Get SNMP data from running config.
      IOS_command:
        commands:
          - show running-config | include snmp-server
      register: if_data

    - name: Show SNMP data from running config.
      debug:
        var: if_data['stdout_lines'][0]
```

Now while this playbook looks quite similar to the ones we’ve used before there are a few small additions which are explained below.

Table 4: Explanation playbook SNMP

Configuration	Explanation
Register:if_data	Register saves data output as a variable.
Debug:	Prints statements during execution.
Var: `if_data['stdout_lines'][0]`	The Variable “if_data” procured from register are put on screen thanks to the stdout_lines.

Having this done and received data, we now know that there is an SNMP server configured on one/several of our devices which means that we can proceed to the removal. In case we haven’t received any data output we can instead proceed to the installation of SNMPv3.

The removal of SNMP is quite simple and possible the easiest playbook yet in our walkthrough. As usual you want to open your text editor in the CLI of your Ansible host and type / copy the following in:

```
---
- name: Remove any SNMP configuration.
  hosts: all
  tasks:
    - name: Clear SNMP configuration.
      IOS_config:
        lines:
          - no snmp-server
```

As we can see there was nothing new in this playbook as far as complexity goes which is good as the task itself was simple and shouldn't be made complicated. Now it's time to save it and run again by typing the following command:

```
Ansible-playbook play_name.yml
```

In case data was received the first time we ran our Check SNMP playbook, we can run it again to double check if there is still data showing up which shouldn't be there in this case.

And last but not least we have to configure SNMPv3**.** Now that we know for sure that none of our devices are configured with SNMP we can finally set it up to our needs. We will configure it to our needs with password authentication, version 3 and traps centred around OSPF and VLANs. We will also add a new feature to our Cisco playbook where the running config will be saved to the start-up configuration when modifications occur.

```
---
- name: Configure all devices with SNMPv3.
  hosts: all
  tasks:
    - name: Configure SNMP V3.
      IOS_config:
        lines:
          - snmp-server engineid remote 10.0.2.1 446172742E506F776
          - snmp-
server user ADMIN ADMINGROUP v3 auth md5 AUTHPASS priv aes 128 PRIVPASS
          - snmp-server group ADMINGROUP v3 priv
          - snmp-server host 10.0.2.1 traps version 3 priv ADMIN
          - snmp-server enable traps vlancreate
          - snmp-server enable traps vlandelete
          - snmp-server enable traps ospf
          - snmp-server enable traps ospf errors
    - name: save running to start up when modified
      cisco.IOS.IOS_config:
          save_when: modified
```

As previously explained we use IOS_config for simplicity and a neat code. The two noticeable changes this time come in the name of modules and the new task which helps with the longevity of your changes by saving your configuration in start-up config therefore not being reset by a reboot as would be the case with regular running config.

You may have noticed that we used the IOS_config module initially, followed by the cisco.IOS.IOS_config module. These are essentially the same if a newer version of Ansible is used, however if an older version of ansible is used it might be impossible to run only IOS_config, as abbreviations were not fully supported yet.

The second change is the second task which saves the running configuration to start-up configuration when there were any modifications. A simple and small task but definitely really important when you want your changes to be more than just temporary.

10.5 Step 5 – Ansible in retrospective

First of all – we would like to congratulate and thank you for following our Manual and guide so far.

Looking back at what we've achieved we can conclude a few things about automation and Ansible. Just in this chapter alone we learned the value of modules, idempotency and Ansible itself.

Starting with modules we found out how we can search for pre-made functionalities with galaxy and discovered how to find more information/explanation provided by the creators of these modules. This allowed us to expand on the base functionality that Ansible gives us for automation. Discovering this ability expands upon simple automation tasks and adds almost any automation task we can imagine. One thing of note however is that not all modules are made evenly as we discussed in step 4. In certain cases, specialized modules may not be optimal as these modules are often more convoluted in use. In these cases the ios_config module is recommended as this module exposes the standard IOS interface commands for use in ansible.

Next up we have Idempotency – as we learned before it is imperative for automation to be able to provide the one config on all devices which would provide the same results on all of the targets. If we ignore idempotency and proceed to write our automation tasks not caring for uniformity and deploying each task separately, we achieve nothing. Whilst nothing may be worded a bit too strongly, doing that and typing out your configs directly on the device has little to no difference.

And arriving at this point we have to talk about Ansible and the benefits of automation. A recurring theme in this manual is uniformity as well as the ease of code. Having used Ansible from the beginning of this walkthrough together it's safe to say that we haven't encountered any type of code that didn't come out as extremely difficult or convoluted. This is because Ansible tries to make it a selling point to be as easy to use as possible. Having a low barrier of entry helps while making both simple and complex automation tasks which can be differentiated with the use of modules. In case the pre-made modules would not be enough Ansible gives us the possibility of writing our own and or integrating coding languages like python with it, expanding on the functionalities. Now if we didn't use Ansible to automate our tasks we would be most likely forced to install an agent on our target devices making the setup step longer and more prone to mistakes or in other instances forced to re-copy our code for each device separately not adding to the process of automation but instead slightly cutting off the time you'd use while configurating your devices normally.