ANSIBLE

QUALITY THOUGHT

INFOSYSTEMS (INDIA) PVT. LTD.

Table of Contents:

Getting Started with Ansible

In this chapter we will cover:

- IT automation
- What is Ansible?
- The secure shell
- Installing Ansible

IT automation

IT automation is in its larger sense—the processes and software that help with the management of the IT infrastructure (servers, networking, and storage). In the current shift, we are assisting to a huge implementation of such processes and software.

The history of IT automation

At the beginning of IT history, there were very few servers and a lot of people were needed to make them work properly, usually more than one person for each machine. Over the years, servers became more reliable and easier to manage so it was possible to have multiple servers managed by a single system administrator. In that period, the administrators manually installed the software, upgraded the software manually, and changed the configuration files manually. This was obviously a very labor-intensive and error-prone process, so many administrators started to implement scripts and other means to make their life easier. Those scripts were (usually) pretty complex and they did not scale very well.

In the early years of this century, data centers started to grow a lot due to companies' needs. Virtualization helped in keeping prices low and the fact that many of these services were web services, meant that many servers were very similar to each other. At this point, new tools were needed to substitute the scripts that were used before, the configuration management tools.

CFEngine was one of the first tools to demonstrate configuration management capabilities way back in the 1990s; more recently, there has been Puppet, Chef, and Salt, besides Ansible.

Advantages of IT automation

People often wonder if IT automation really brings enough advantages considering that implementing it has some direct and indirect costs. The main advantages of IT automation are:

- Ability to provision machines quickly
- Ability to recreate a machine from scratch in minutes
- Ability to track any change performed on the infrastructure

Disadvantages of IT automation

As with any other technology, IT automation does come with some disadvantages. From my point of view these are the biggest disadvantages:

- Automating all of the small tasks that were once used to train new system administrators
- If an error is performed, it will be propagated everywhere

Limiting the possible damages of an error propagation

The second one is trickier. There are a lot of ways to limit this kind of damage, but none of those will prevent it completely. The following mitigation options are available:

- **Always have backups**: Backups will not prevent you from nuking your machine; they will only make the restore process possible.
- **Always test your infrastructure code (playbooks/roles) in a non-production environment**: Companies have developed different pipelines to deploy code and those

usually include environments such as dev, test, staging, and production. Use the same pipeline to test your infrastructure code. If a buggy application reaches the production environment it could be a problem. If a buggy playbook reaches the production environment, it could be catastrophic.

- **Always peer-review your infrastructure code**: Some companies have already introduced peer-reviews for the application code, but very few have introduced it for the infrastructure code. As I was saying in the previous point, I think infrastructure code is way more critical than application code, so you should always peer-review your infrastructure code, whether you do it for your application code or not.
- **Enable SELinux**: SELinux is a security kernel module that is available on all Linux distributions (it is installed by default on Fedora, Red Hat Enterprise Linux, CentOS, Scientific Linux, and Unbreakable Linux). It allows you to limit users and process powers in a very granular way. I suggest using SELinux instead of other similar modules (such as AppArmor) because it is able to handle more situations and permissions. SELinux will prevent a huge amount of damage because, if correctly configured, it will prevent many dangerous commands from being executed.
- **Run the playbooks from a limited account**: Even though user and privilege escalation schemes have been in UNIX code for more than 40 years, it seems as if not many companies use them. Using a limited user for all your playbooks, and escalating privileges only for commands that need higher privileges will help prevent you nuking a machine while trying to clean an application temporary folder.
- **Use horizontal privilege escalation**: The sudo is a well-known command but is often used in its more dangerous form. The sudo command supports the '-u' parameter that will allow you to specify a user that you want to impersonate. If you have to change a file that is owned by another user, please do not escalate to root to do so, just escalate to that user. In Ansible, you can use the become_user parameter to achieve this.
- **When possible, don't run a playbook on all your machines at the same time**: Staged deployments can help you detect a problem before it's too late. There are many

problems that are not detectable in a dev, test, staging, and qa environment. The majority of them are related to load that is hard to emulate properly in those non-production environments. A new configuration you have just added to your Apache HTTPd or MySQL servers could be perfectly OK from a syntax point of view, but disastrous for your specific application under your production load. A staged deployment will allow you to test your new configuration on your actual load without risking downtime if something was wrong.

- **Avoid guessing commands and modifiers**: A lot of system administrators will try to remember the right parameter and try to guess if they don't remember it exactly. I've done it too, a lot of times, but this is very risky. Checking the man page or the online documentation will usually take you less than two minutes and often, by reading the manual, you'll find interesting notes you did not know. Guessing modifiers is dangerous because you could be fooled by a non-standard modifier (that is, -v is not the verbose mode for grep and -h is not the help command for the MySQL CLI).
- **Avoid error-prone commands**: Not all commands have been created equally. Some commands are (way) more dangerous than others. If you can assume a cat command safe, you have to assume that a dd command is dangerous, since dd perform copies and conversion of files and volumes. I've seen people using dd in scripts to transform DOS files to UNIX (instead of dos2unix) and many other, very dangerous, examples. Please, avoid such commands, because they could result in a huge disaster if something goes wrong.
- **Avoid unnecessary modifiers**: If you need to delete a simple file, use rm ${file} not rm -rf ${file}. The latter is often performed by users that have learned that; "to be sure, always use rm -rf", because at some time in their past, they have had to delete a folder. This will prevent you from deleting an entire folder if the ${file} variable is set wrongly.
- **Always check what could happen if a variable is not set**: If you want to delete the contents of a folder and you use the rm -rf ${folder}/* command, you are looking for

trouble. If the ${folder} variable is not set for some reason, the shell will read a rm -rf /* command, which is deadly (considering the fact that the rm -rf / command will not work on the majority of current OSes because it requires a --no-preserve-rootoption, while rm -rf /* will work as expected). I'm using this specific command as an example because I have seen such situations: the variable was pulled from a database which, due to some maintenance work, was down and an empty string was assigned to that variable. What happened next is probably easy to guess. In case you cannot prevent using variables in dangerous places, at least check them to see if they are not empty before using them. This will not save you from every problem but may catch some of the most common ones.

- **Double check your redirections**: Redirections (along with pipes) are the most powerful elements of Linux shells. They could also be very dangerous: a cat /dev/rand > /dev/sda command can destroy a disk even if a cat command is usually overlooked because it's not usually dangerous. Always double-check all commands that include a redirection.
- **Use specific modules wherever possible**: In this list I've used shell commands because many people will try to use Ansible as if it's just a way to distribute them: it's not. Ansible provides a lot of modules and we'll see them in this book. They will help you create more readable, portable, and safe playbooks.

Types of IT automation

There are a lot of ways to classify IT automation systems, but by far the most important is related to how the configurations are propagated. Based on this, we can distinguish between agent-based systems and agent-less systems.

Agent-based systems

Agent-based systems have two different components: a **server** and a client called **agent**.

There is only one server and it contains all of the configuration for your whole environment, while the agents are as many as the machines in the environment.Periodically, client will contact the server to see if a new configuration for its machine is present. If a new configuration is present, the client will download it and apply it.

Agent-less systems

In agent-less systems, no specific agent is present. Agent-less systems do not always respect the server/client paradigm, since it's possible to have multiple servers and even the same number of servers and clients . Communications are initialized by the server that will contact the client(s) using standard protocols (usually via SSH and PowerShell).

What is Ansible?

Ansible is an agent-less IT automation tool developed in 2012 by **Michael DeHaan**, a former Red Hat associate. The Ansible design goals are for it to be: minimal, consistent, secure, highly reliable, and easy to learn. The Ansible company has recently been bought out by Red Hat and now operates as part of Red Hat, Inc.

Ansible primarily runs in push mode using SSH, but you can also run Ansible using ansible-pull, where you can install Ansible on each agent, download the playbooks locally, and run them on individual machines. If there is a large number of machines (large is a relative term; in our view, greater than 500 and requiring parallel updates), and you plan to deploy updates to the machines in parallel, this might be the right way to go about it.

Secure Shell (SSH)

Secure Shell (also known as **SSH**) is a network service that allows you to login and access a shell remotely in a fully encrypted connection. The SSH daemon is today, the standard for UNIX system administration, after having replaced the unencrypted telnet. The most frequently used implementation of the SSH protocol is OpenSSH.

In the last few months, Microsoft has shown an implementation (at the time of writing) of OpenSSH for Windows.

Since Ansible performs SSH connections and commands in the same way any other SSH client would do, no specific configuration has been applied to the OpenSSH server.

To speed up default SSH connections, you can always enable ControlPersist and the pipeline mode, which makes Ansible faster and secure.

Installing Ansible

Installing Ansible is rather quick and simple. You can use the source code directly, by cloning it from the GitHub project (https://github.com/ansible/ansible), install it using your system's package manager, or use Python's package management tool (**pip**). You can use Ansible on any Windows, Mac, or UNIX-like system. Ansible doesn't require any databases and doesn't need any daemons running. This makes it easier to maintain Ansible versions and upgrade without any breaks.

We'd like to call the machine where we will install Ansible our Ansible workstation. Some people also refer to it as the command center.

Installing Ansible using the system's package manager

It is possible to install Ansible using the system's package manager and in my opinion this is the preferred option if your system's package manager ships at least Ansible 2.0. We will look into installing Ansible via **Yum**, **Apt**, **Homebrew**, and **pip**.

Installing via Yum

If you are running a Fedora system you can install Ansible directly, since from Fedora 22, Ansible 2.0+ is available in the official repositories. You can install it as follows:

```
sudo dnf install ansible
```

For RHEL and RHEL-based (CentOS, Scientific Linux, Unbreakable Linux) systems, versions 6 and 7 have Ansible 2.0+ available in the EPEL repository, so you should ensure that you have the EPEL repository enabled before installing Ansible as follows:

```
sudo yum install ansible
```

Note: On Cent 6 or RHEL 6, you have to run the command rpm -Uvh. Refer to http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm for instructions on how to install EPEL.

Installing via Apt

Ansible is available for Ubuntu and Debian. To install Ansible on those operating systems, use the following command:

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

Installing via Homebrew

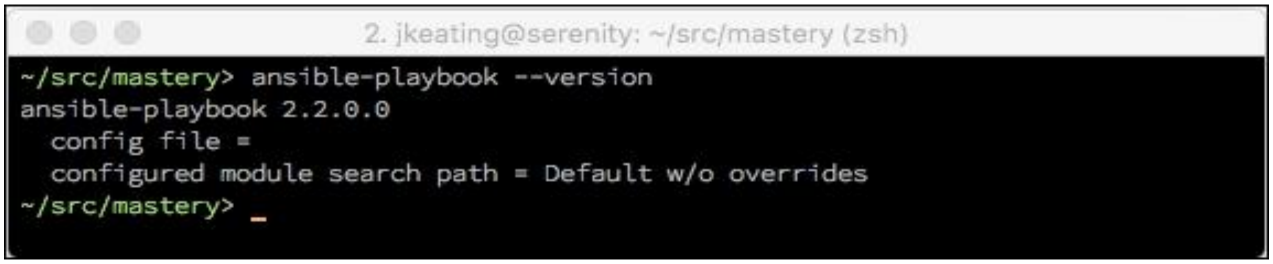
You can install Ansible on Mac OS X using Homebrew, as follows:

```
$ brew update
$ brew install ansible
```

Ansible version and configuration

It is assumed that you have Ansible installed on your system. There are many documents out there that cover installing Ansible in a way that is appropriate for the operating system and version that you might be using. This book will assume the use of the Ansible 2.2.x.x version.

To discover the version in use on a system with Ansible already installed, make use of the version argument, that is, either ansible or ansible-playbook:



```
~/src/mastery> ansible-playbook --version
ansible-playbook 2.2.0.0
  config file =
  configured module search path = Default w/o overrides
~/src/mastery> _
```

Note that ansible is the executable for doing adhoc one-task executions and ansible-playbook is the executable that will process playbooks for orchestrating many tasks.

The configuration for Ansible can exist in a few different locations, where the first file found will be used. The search order changed slightly in version 1.5, with the new order being:

- ANSIBLE_CFG: This is an environment variable
- ~/ansible.cfg: This is in the current directory
- ansible.cfg: This is in the user's home directory
- ./etc/ansible/ansible.cfg

Inventory parsing and data sources

In Ansible, nothing happens without an inventory. Even ad hoc actions performed on localhost require an inventory, even if that inventory consists just of the localhost. The inventory is the most basic building block of Ansible architecture. When executing ansible or ansible-playbook, an inventory must be referenced. Inventories are either files or directories that exist on the same system that runs ansible or ansible-playbook. The location of the inventory can be referenced at runtime with the --inventory-file (-i) argument, or by defining the path in an Ansible config file.

Inventories can be static or dynamic, or even a combination of both, and Ansible is not limited to a single inventory. The standard practice is to split inventories across logical boundaries, such as staging and production, allowing an engineer to run a set of plays against their staging environment for validation, and then follow with the same exact plays run against the production inventory set.

Static inventory

The static inventory is the most basic of all the inventory options. Typically, a static inventory will consist of a single file in the ini format. Here is an example of a static inventory file describing a single host, mastery.example.name:

```
mastery.example.name
```

That is all there is to it. Simply list the names of the systems in your inventory. Of course, this does not take full advantage of all that an inventory has to offer. If every name were listed like this, all plays would have to reference specific hostname, or the special all group. This can be quite tedious when developing a playbook that operates across different sets of your infrastructure. At the very least, hosts should be arranged into groups. A design pattern that works well is to arrange your systems into groups based on expected functionality. At first, this may seem difficult if you have an environment where single systems can play many different roles, but that is perfectly fine. Systems in an inventory can exist in more than one group, and groups can even consist of other groups! Additionally, when listing groups and hosts, it's possible to list hosts without a group. These would have to be listed first, before any other group is defined. Let's build on our previous example and expand our inventory with a few more hosts and some groupings:

```
[web]
mastery.example.name

[dns]
backend.example.name

[database]
backend.example.name
```

```
[frontend:children]
web

[backend:children]
dns
database
```

What we have created here is a set of three groups with one system in each, and then two more groups, which logically group all three together. Yes, that's right; you can have groups of groups. The syntax used here is [groupname:children], which indicates to Ansible's inventory parser that this group by the name of groupname is nothing more than a grouping of other groups. The children in this case are the names of the other groups. This inventory now allows writing plays against specific hosts, low-level role-specific groups, or high-level logical groupings, or any combination.

By utilizing generic group names, such as dns and database, Ansible plays can reference these generic groups rather than the explicit hosts within. An engineer can create one inventory file that fills in these groups with hosts from a preproduction staging environment and another inventory file with the production versions of these groupings. The playbook content does not need to change when executing on either staging or production environment because it refers to the generic group names that exist in both inventories. Simply refer to the right inventory to execute it in the desired environment.

Inventory variable data

Inventories provide more than just system names and groupings. Data about the systems can be passed along as well. This can include:

- Host-specific data to use in templates
- Group-specific data to use in task arguments or conditionals
- Behavioral parameters to tune how Ansible interacts with a system

Let's improve upon our existing example inventory and add to it some variable data. We will add some host-specific data as well as group-specific data:

```
[web]
mastery.example.name ansible_host=192.168.10.25

[dns]
backend.example.name

[database]
backend.example.name

[frontend:children]
web

[backend:children]
dns
database

[web:vars]
http_port=88
proxy_timeout=5

[backend:vars]
ansible_port=314

[all:vars]
ansible_ssh_user=otto
```

In this example, we defined ansible_host for mastery.example.name to be the IP address of 192.168.10.25. The ansible_host variable is a **behavioral inventory variable**, which is intended to alter the way Ansible behaves when operating with this host. In this case, the variable instructs Ansible to connect to the system using the provided IP address rather than performing a DNS lookup on the name mastery.example.name. There are a number of other behavioral inventory variables, which are listed at the end of this section along with their intended use.

As of version 2.0, the longer form of some behavioral inventory parameters has been deprecated. The ssh part of ansible_ssh_host, ansible_ssh_user, and ansible_ssh_port is no longer required. A future release may ignore the longer form of these variables.

Our new inventory data also provides group-level variables for the web and backend groups. The web group defines http_port, which may be used in an **nginx** configuration file, and proxy_timeout, which might be used to determine **HAProxy** behavior. The backend group makes use of another behavioral inventory parameter to instruct Ansible to connect to the hosts in this group using port 314 for SSH, rather than the default of 22.

Finally, a construct is introduced that provides variable data across all the hosts in the inventory by utilizing a built-in allgroup. Variables defined within this group will apply to every host in the inventory. In this particular example, we instruct Ansible to log in as the otto user when connecting to the systems. This is also a behavioral change, as the Ansible default behavior is to log in as a user with the same name as the user executing ansible or ansible-playbook on the control host.

Here is a table of behavior inventory variables and the behavior they intend to modify:

Inventory parameters	Behaviour
ansible_host	This is the DNS name or IP address used to connect to the host, if different from the inventory name, or the name of the Docker container to connect to.
ansible_port	This is the SSH port number, if not 22.
ansible_user	This is the default SSH username or user inside a Docker container to use.
ansible_ssh_pass	This is the SSH password to use (this is insecure; we strongly recommend using --ask-pass or the SSH keys).
ansible_ssh_private_key_file	This is the private key file used by SSH. This is useful if you use multiple keys and you don't want to use SSH agent.
ansible_ssh_common_args	This defines SSH arguments to append to the default arguments for ssh, sftp, and scp.
ansible_sftp_extra_args	This setting is always appended to the default sftp command-line arguments.
ansible_scp_extra_args	This setting is always appended to the default scp command-line arguments.
ansible_ssh_extra_args	This setting is always appended to the default ssh command-line arguments.
ansible_ssh_pipelining	This setting uses a Boolean to define whether or not SSH pipelining should be used for this host.
ansible_ssh_executable	This setting overrides the path to the SSH executable for this host.
ansible_become	This defines whether privilege escalation (sudo or otherwise) should be used with this host.
ansible_become_method	The method to use for privilege escalation. One of sudo, su, pbrun, pfexec, doas, dzdo, or ksu.

ansible_become_user	This is the user to become through privilege escalation.
ansible_become_pass	This is the password to use for privilege escalation.
ansible_sudo_pass	This is the sudo password to use (this is insecure; we strongly recommend using --ask-sudo-pass).
ansible_connection	This is the connection type of the host. Candidates are local, smart, ssh, paramiko, or docker. The default is paramiko before Ansible 1.2, and smart afterwards, which detects whether the usage of ssh will be feasible based on whether the SSH feature ControlPersist is supported.
ansible_docker_extra_args	This is a string of any extra arguments that can be passed to Docker. This is mainly used to define a remote Docker daemon to use.
ansible_shell_type	This is the shell type of the target system. By default, commands are formatted using the sh-style syntax. Setting this to csh or fish will cause commands to be executed on target systems to follow the syntax of csh or fish instead.
ansible_shell_executable	This sets the shell tool that will be used on the target system. This should only be used if the default of /bin/sh is not possible to use.
ansible_python_interpreter	This is the target host Python path. This is useful for systems with more than one Python, systems that are not located at /usr/bin/python (such as *BSD), or for systems where /usr/bin/python is not a 2.X series Python. We do not use the /usr/bin/env mechanism as it requires the remote user's path to be set right and also assumes that the Python executable is named Python, where the executable might be

	named something like python26.
ansible_*_interpreter	This works for anything such as Ruby or Perl and works just like ansible_python_interpreter. This replaces the shebang of modules which run on that host.

Automating Simple Tasks

Ansible can be used for both, creating and managing a whole infrastructure, as well as be integrated into an infrastructure that is already working.

But first we will talk about **YAML Ain't Markup Language** (**YAML**), a human-readable data serialization language that is widely used in Ansible.

YAML

YAML, like many other data serialization languages (such as JSON), has very few, basic concepts:

- Declarations
- Lists
- Associative arrays

A declaration is very similar to a variable in any other language, that is:

```
name: 'This is the name'
```

To create a list, we will have to use '-':

```
- 'item1'
- 'item2'
- 'item3'
```

YAML uses indentation to logically divide parents from children. So if we want to create associative arrays (also known as objects), we would just need to add an indentation:

```
item:
  name: TheName
  location: TheLocation
```

Obviously, we can mix those together, that is:

```
people:
  - name: Albert
    number: +1000000000
    country: USA
  - name: David
    number: +44000000000
    country: UK
```

Hello Ansible

Let's start by checking if a remote machine is reachable; in other words, let's start by pinging a machine. The simplest way to do this, is to run the following:

```
$ ansible all -i HOST, -m ping
```

Here, HOST is an IP address, the **Fully Qualified Domain Name** (**FQDN**), or an alias of a machine where you have SSH access

You should receive something like this as a result:

```
test01.fale.io | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Now, let's see what we did and why. Let's start from the Ansible help. To query it, we can use the following command:

To make it easier to be read, we have removed all the output related to options that we have not used:

```
Usage: ansible <host-pattern> [options]
Options:
```

```
-i INVENTORY, --inventory-file=INVENTORY
                  specify inventory host path
                  (default=/etc/ansible/hosts) or comma
                  separated host list.
-m MODULE_NAME, --module-name=MODULE_NAME
                  module name to execute (default=command)
```

So, what we did was:

1. We invoked Ansible.
2. We instructed Ansible to run on all hosts.
3. We specified our inventory (also known as the list of the hosts).
4. We specified the module we wanted to run (ping).

Now that we can ping the server, let's echo hello ansible!

```
ansible all -m shell -a '/bin/echo hello ansible!'
```

You should receive something like this as a result:

```
test01.fale.io | SUCCESS | rc=0 >>
hello ansible!
```

In this example, we used an additional option. Let's check the help to see what it does:

```
Usage: ansible <host-pattern> [options]
Options:
  -a MODULE_ARGS, --args=MODULE_ARGS
                  module arguments
```

As you may have guessed from the context and the name, the args options allow you to pass additional arguments to the module. Some modules (like ping) do not support any arguments, while others (such as shell) will require arguments.

Working with playbooks

Playbooks are one of the core features of Ansible and tell Ansible what to execute. They are like a to-do list for Ansible that contains a list of tasks; each task internally links to a piece of code called a **module**. Playbooks are simple, human-readable YAML files, whereas modules are a piece of code that can be written in any language with the condition that its output be in the JSON format. You can have multiple tasks listed in a playbook and these tasks would be executed serially by Ansible. You can think of playbooks as an equivalent of manifests in Puppet, states in Salt, or cookbooks in Chef; they allow you to enter a list of tasks or commands you want to execute on your remote system.

Studying the anatomy of a playbook

Playbooks can have a list of remote hosts, user variables, tasks, handlers, and so on. You can also override most of the configuration settings through a playbook. Let's start looking at the anatomy of a playbook.

The purpose of the playbook we are going to consider now, is to ensure that the httpd package is installed and the service is **enabled** and **started**. This is the content of the setup_apache.yaml file:

```
---
- hosts: all
  remote_user: fale
  tasks:
  - name: Ensure the HTTPd package is installed
    yum:
      name: httpd
      state: present
      become: True
  - name: Ensure the HTTPd service is enabled and running
    service:
      name: httpd
      state: started
      enabled: True
    become: True
```

The setup_apache.yaml file is an example of a playbook. The file is comprised of three main parts, as follows:

- hosts: This lists the host or host group against which we want to run the task. The hosts field is mandatory and every playbook should have it. It tells Ansible on which hosts to run the listed tasks. When provided with a host group, Ansible will take the host group from the playbook and try look for it in an inventory file . If there is no match, Ansible will skip all the tasks for that host group. The --list-hosts option along with the playbook (ansible-playbook <playbook> --list-hosts) will tell you exactly which hosts the playbook will run against.
- remote_user: This is one of the configuration parameters of Ansible (consider, for example, tom -remote_user) that tells Ansible to use a particular user (in this case, tom) while logging into the system.
- tasks: Finally, we come to tasks. All playbooks should contain tasks. Tasks are a list of actions you want to perform. A tasks field contains the name of the task (that is, the help text for the user about the task), a module that should be executed, and arguments that are required for the module. Let's look at the single task that is listed in the playbook, as shown in the preceding snippet of code:

In the preceding case, there are two tasks. The name parameter represents what the task is doing and is present mainly to improve readability, as we'll see during the playbook run. The name parameter is optional. The modules, yum and service, have their own set of parameters. Almost all modules have the name parameter (there are exceptions such as the debug module), which indicates what component the actions are performed on. Let's look at the other parameters:

- In the yum module's case, the state parameter has the latest value and it indicates that the httpdlatest package should be installed. The command to execute more or less translates to yum install httpd.
- In the service module's scenario, the state parameter with the started value indicates that the httpd service should be started, and it roughly translates to /etc/init.d/httpd start. In this module we also have the "enabled" parameter that defines whether the service should start at boot or not.

- The become: True parameter represents the fact that the tasks should be executed with sudo access. If the sudo user's file does not allow the user to run the particular command, then the playbook will fail when it is run.

Note:

You might have questions about why there is no package module that figures out the architecture internally and runs the yum, apt, or any other package options depending on the architecture of the system. Ansible populates the package manager value into a variable named ansible_pkg_manager.

In general, we need to remember that the number of packages that have a common name across different operating systems is a small subset of the number of packages that are actually present. For example, the httpd package is called httpd in Red Hat systems and apache2 in Debian-based systems. We also need to remember that every package manager has its own set of options that make it powerful; as a result, it makes more sense to use explicit package manager names so that the full set of options are available to the end user writing the playbook.

Running a playbook

Now, it's time (yes, finally!) to run the playbook. To instruct Ansible to execute a playbook instead of a module, we will have to use a different command (ansible-playbooks) that has a syntax very similar to the "ansible" command we already saw:

```
$ ansible-playbook -i HOST setup_apache.yaml
```

As you can see, aside from the host-pattern (that is specified in the playbook) that has disappeared, and the module option that has been replaced by the playbook name, nothing changed. So to execute this command on my machine, the exact command is:

```
$ ansible-playbook   setup_apache.yaml
```

The result is the following:

```
PLAY [all] *******************************************************
TASK [setup] *****************************************************
ok: [test01.fale.io]


TASK [Ensure the HTTPd package is installed] *********************
changed: [test01.fale.io]


TASK [Ensure the HTTPd service is enabled and running] ***********
changed: [test01.fale.io]


PLAY RECAP *******************************************************
test01.fale.io    : ok=3    changed=2    unreachable=0    failed=0
```

Wow! The example worked. Let's now check whether the httpd package is installed and up-and-running on the machine. To check if HTTPd is installed, the easiest way is to ask rpm:

```
$ rpm -qa | grep httpd
```

If everything worked properly, you should have an output like the following:

```
httpd-tools-2.4.6-40.el7.centos.x86_64
httpd-2.4.6-40.el7.centos.x86_64
```

To see the status of the service, we can ask systemd:

```
systemctl status httpd
```

The expected result is something like the following:

```
httpd.service - The Apache HTTP Server
 Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; vendor preset: disabled)
   Active: active (running) since Sat 2016-05-07 13:22:14 EDT; 7min ago
     Docs: man:httpd(8)
           man:apachectl(8)
 Main PID: 2214 (httpd)
   Status: "Total requests: 0; Current requests/sec: 0; Current traffic:   0 B/sec"
   CGroup: /system.slice/httpd.service
```

```
        -2214 /usr/sbin/httpd -DFOREGROUND
        -2215 /usr/sbin/httpd -DFOREGROUND
        -2216 /usr/sbin/httpd -DFOREGROUND
        -2217 /usr/sbin/httpd -DFOREGROUND
        -2218 /usr/sbin/httpd -DFOREGROUND
        -2219 /usr/sbin/httpd -DFOREGROUND
```

The end state, according to the playbook, has been achieved. Let's briefly look at exactly what happens during the playbook run:

```
PLAY [all] ***********************************************************
```

This line advises us that a playbook is going to start here and that it will be executed on "all" hosts

```
TASK [setup] *********************************************************
ok: [test01.fale.io]
```

The TASK lines show the name of the task (setup in this case), and their effect on each host. Sometimes people get confused by the setup task. In fact, if you look at the playbook, there is no setup task. This is because Ansible, before executing the tasks that we have asked it to, will try to connect to the machine and gather information about it that could be useful later. As you can see, the task resulted with a green ok state, so it succeeded and nothing was changed on the server:

```
TASK [Ensure the HTTPd package is installed] **********************
changed: [test01.fale.io]
TASK [Ensure the HTTPd service is enabled and running] ***********
changed: [test01.fale.io]
```

These two task's states are yellow and spell "changed". This means that those tasks were executed and have succeeded but have actually changed something on the machine:

```
PLAY RECAP *********************************************************
test01.fale.io      : ok=3    changed=2    unreachable=0    failed=0
```

Those last few lines are a recapitulation of how the playbook went. Let's rerun the task now and see the output after both the tasks have actually run:

```
PLAY [all] *******************************************************
TASK [setup] *****************************************************
ok: [test01.fale.io]
TASK [Ensure the HTTPd package is installed] *********************
ok: [test01.fale.io]
TASK [Ensure the HTTPd service is enabled and running] ***********
ok: [test01.fale.io]
PLAY RECAP *******************************************************
test01.fale.io    : ok=3    changed=0    unreachable=0    failed=0
```

As you would have expected, the two tasks in question give an output of ok, which would mean that the desired state was already met prior to running the task. It's important to remember that many tasks such as the **Gathering facts** task obtain information regarding a particular component of the system and do not necessarily change anything on the system; hence, these tasks didn't display the changed output earlier.

The PLAY RECAP section in the first and second run are shown as follows. You will see the following output during the first run:

```
PLAY RECAP *******************************************************
test01.fale.io    : ok=3    changed=2    unreachable=0    failed=0
```

You will see the following output during the second run:

```
PLAY RECAP *******************************************************
test01.fale.io    : ok=3    changed=0    unreachable=0    failed=0
```

As you can see, the difference is that the first task's output shows changed=2, which means that the system state changed twice due to two tasks. It's very useful to look at this output, since, if a system has achieved its desired state and then you run the playbook on it, the expected output should be changed=0.

If you're thinking of the word **Idempotency** at this stage, you're absolutely right and deserve a pat on the back! Idempotency is one of the key tenets of configuration management. Wikipedia defines Idempotency as an operation that, if applied twice to any value, gives the same result as if it were applied once. The earliest examples of this that you would have encountered in your childhood would be multiplicative operations on the number 1, where 1*1=1 every single time.

Let's proceed. You could have also written the preceding tasks as follows but when the tasks are run, from an end user's perspective, they are quite readable (we will call this file setup_apache_no_com.yaml):

```
---
- hosts: all
  remote_user: fale
  tasks:
  - yum:
      name: httpd
      state: present
    become: True
  - service:
      name: httpd
      state: started
      enabled: True
    become: True
```

Let's run the playbook again to spot any difference in the output:

```
$ ansible-playbook -i test01.fale.io, setup_apache_no_com.yaml
```

The output would be:

```
PLAY [all] *******************************************************
TASK [setup] *****************************************************
ok: [test01.fale.io]


TASK [yum] *******************************************************
ok: [test01.fale.io]


TASK [service] ***************************************************
ok:[test01.fale.io]


PLAY RECAP *******************************************************
test01.fale.io    : ok=3    changed=0    unreachable=0    failed=0
```

As you can see, the difference is in the readability. Wherever possible, it's recommended to keep the tasks as simple as possible (the **KISS** principle of **Keep It Simple Stupid**) to allow for maintainability of your scripts in the long run.

Ansible verbosity

One of the first options anyone picks up is the debug option. To understand what is happening when you run the playbook, you can run it with the **verbose** (-v) option. Every extra v will provide the end user with more debug output.

Let's see an example of using the playbook debug for a single task using the following debug options:

- The -v option provides the default output, as shown in the preceding examples.
- The -vv option adds a little more information, as shown in the following example:

```
Using /etc/ansible/ansible.cfg as config file

   PLAYBOOK: setup_apache.yaml *********************************
   1 plays in setup_apache.yaml

   PLAY [all] ***************************************************

   TASK [setup] *************************************************
   ok: [test01.fale.io]

   TASK [Ensure the HTTPd package is installed] ***************
   task path: /home/fale/setup_apache.yaml:5
   ok: [test01.fale.io] => {"changed": false, "msg": "", "rc": 0, "results": ["httpd-2.4.6-40.el7.centos.x86_64 providing httpd is already installed"]}

   TASK [Ensure the HTTPd service is enabled and running] ****
   task path: /home/fale/setup_apache.yaml:10
   ok: [test01.fale.io] => {"changed": false, "enabled": true, "name": "httpd", "state": "started"}

   PLAY RECAP ***************************************************
   test01.fale.io   : ok=3  changed=0   unreachable=0  failed=0
```

The -vvv option adds a lot more information, as shown in the following code. This shows the sshcommand Ansible uses to create a temporary file on the remote host and run the script remotely

Variables in playbooks

Sometimes it is important to set and get variables in a playbook.

Very often, you'll need to automate multiple similar operations. In those cases, you'll want to create a single playbook that can be called with different variables to ensure code reusability.

Another case where variables are very important is when you have more than one datacenter and some values will be datacenter-specific. A common example are the DNS servers. Let's analyze the following simple code that will introduce us to the Ansible way to set and get variables:

```
---
- hosts: all
  remote_user: fale
  tasks:
  - name: Set variable 'name'
    set_fact:
      name: Test machine
  - name: Print variable 'name'
    debug:
      msg: '{{ name }}'
```

```
$ ansible-playbook -i test01.fale.io variables.yaml
```

You should see the following result:

```
PLAY [all] ***********************************************************
TASK [setup] *********************************************************
ok: [test01.fale.io]


TASK [Set variable 'name'] ********************************************
ok: [test01.fale.io]


TASK [Print variable 'name'] ******************************************
```

```
ok:[test01.fale.io] => {
"msg": "Test machine"
}


PLAY RECAP ***************************************************************
test01.fale.io      : ok=3    changed=0    unreachable=0    failed=0
```

If we analyze the code we have just executed, it should be pretty clear what's going on. We set a variable (that in Ansible are called facts) and then we print it with the debug function.

Ansible allows you to set your variables in many different ways, that is, either by passing a variable file, declaring it in a playbook, passing it to the ansible-playbook command using -e / --extra-vars, or by declaring it in an inventory file (we will be discussing more in-depth about this in the next chapter).

It's now time to start using some metadata that Ansible obtained during the setup phase. Let's start by looking at the data that is gathered by Ansible. To do this, we will execute:

```
$ ansible all -i HOST -m setup
```

In our specific case, this means executing the following:

```
$ ansible all -i test01.fale.io, -m setup
```

We can obviously do the same with a playbook, but this way is faster. Also, for the "setup" case, you will need to see the output only during the development to be sure to use the right variable name for your goal.

The output will be something like this:

```
test01.fale.io | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "178.62.36.208",
            "10.16.0.7"
        ],
"ansible_all_ipv6_addresses": [
   "fe80::601:e2ff:fef1:1301"
],
"ansible_architecture": "x86_64",
"ansible_bios_date": "04/25/2016",
"ansible_bios_version": "20160425",
"ansible_cmdline": {
   "ro": true,
   "root": "LABEL=DOROOT"
},
"ansible_date_time": {
   "date": "2016-05-14",
   "day": "14",
   "epoch": "1463244633",
   "hour": "12",
   "iso8601": "2016-05-14T16:50:33Z",
   "iso8601_basic": "20160514T125033231663",
   "iso8601_basic_short": "20160514T125033",
   "iso8601_micro": "2016-05-14T16:50:33.231770Z",
   "minute": "50",
   "month": "05",
   "second": "33",
   "time": "12:50:33",
   "tz": "EDT",
   "tz_offset": "-0400",
   "weekday": "Saturday",
   "weekday_number": "6",
   "weeknumber": "19",
   "year": "2016"
},
"ansible_default_ipv4": {
   "address": "178.62.36.208",
   "alias": "eth0",
   "broadcast": "178.62.63.255",
   "gateway": "178.62.0.1",
   "interface": "eth0",
   "macaddress": "04:01:e2:f1:13:01",
   "mtu": 1500,
   "netmask": "255.255.192.0",
   "network": "178.62.0.0",
   "type": "ether"
},
"ansible_default_ipv6": {},
"ansible_devices": {
   "vda": {
      "holders": [],
      "host": "",
      "model": null,
      "partitions": {
         "vda1": {
            "sectors": "41943040",
            "sectorsize": 512,
            "size": "20.00 GB",
            "start": "2048"
         }
      },
      "removable": "0",
      "rotational": "1",
      "scheduler_mode": "",
      "sectors": "41947136",
      "sectorsize": "512",
      "size": "20.00 GB",
      "support_discard": "0",
      "vendor": "0x1af4"
   }
},
"ansible_distribution": "CentOS",
"ansible_distribution_major_version": "7",
"ansible_distribution_release": "Core",
"ansible_distribution_version": "7.2.1511",
"ansible_dns": {
   "nameservers": [
      "8.8.8.8",
      "8.8.4.4"
   ]
},
"ansible_domain": "",
"ansible_env": {
   "HOME": "/home/fale",
   "LANG": "en_US.utf8",
   "LC_ALL": "en_US.utf8",
   "LC_MESSAGES": "en_US.utf8",
   "LESSOPEN": "||/usr/bin/lesspipe.sh %s",
   "LOGNAME": "fale",
   "MAIL": "/var/mail/fale",
   "PATH": "/usr/local/bin:/usr/bin",
   "PWD": "/home/fale",
   "SHELL": "/bin/bash",
   "SHLVL": "2",
   "SSH_CLIENT": "86.187.141.39 37764 22",
   "SSH_CONNECTION": "86.187.141.39 37764 178.62.36.208 22",
   "SSH_TTY": "/dev/pts/0",
  "TERM": "rxvt-unicode-256color",
  "USER": "fale",
  "XDG_RUNTIME_DIR": "/run/user/1000",
  "XDG_SESSION_ID": "180",
  "_": "/usr/bin/python"
},
"ansible_eth0": {
  "active": true,
  "device": "eth0",
  "ipv4": {
      "address": "178.62.36.208",
      "broadcast": "178.62.63.255",
      "netmask": "255.255.192.0",
      "network": "178.62.0.0"
  },
  "ipv4_secondaries": [
      {
          "address": "10.16.0.7",
          "broadcast": "10.16.255.255",
          "netmask": "255.255.0.0",
          "network": "10.16.0.0"
      }
  ],
  "ipv6": [
      {
          "address": "fe80::601:e2ff:fef1:1301",
          "prefix": "64",
          "scope": "link"
      }
  ],
  "macaddress": "04:01:e2:f1:13:01",
  "module": "virtio_net",
  "mtu": 1500,
  "pciid": "virtio0",
  "promisc": false,
  "type": "ether"
},
"ansible_eth1": {
  "active": false,
  "device": "eth1",
  "macaddress": "04:01:e2:f1:13:02",
  "module": "virtio_net",
  "mtu": 1500,
  "pciid": "virtio1",
  "promisc": false,
  "type": "ether"
```

```
},
"ansible_fips": false,
"ansible_form_factor": "Other",
"ansible_fqdn": "test",
"ansible_hostname": "test",
"ansible_interfaces": [
  "lo",
  "eth1",
  "eth0"
],
"ansible_kernel": "3.10.0-327.10.1.el7.x86_64",
"ansible_lo": {
  "active": true,
  "device": "lo",
  "ipv4": {
    "address": "127.0.0.1",
    "broadcast": "host",
    "netmask": "255.0.0.0",
    "network": "127.0.0.0"
  },
  "ipv6": [
    {
      "address": "::1",
      "prefix": "128",
      "scope": "host"
    }
  ],
  "mtu": 65536,
  "promisc": false,
  "type": "loopback"
},
"ansible_machine": "x86_64",
"ansible_machine_id": "fd8cf26e06e411e4a9d004010897bd01",
"ansible_memfree_mb": 6,
"ansible_memory_mb": {
  "nocache": {
    "free": 381,
    "used": 108
  },
  "real": {
    "free": 6,
    "total": 489,
    "used": 483
  },
  "swap": {
    "cached": 0,
            "free": 0,
            "total": 0,
            "used": 0
        }
    },
    "ansible_memtotal_mb": 489,
    "ansible_mounts": [
        {
            "device": "/dev/vda1",
            "fstype": "ext4",
            "mount": "/",
            "options": "rw,relatime,data=ordered",
            "size_available": 18368385024,
            "size_total": 21004894208,
            "uuid": "c5845b43-fe98-499a-bf31-4eccae14261b"
        }
    ],
    "ansible_nodename": "test",
    "ansible_os_family": "RedHat",
    "ansible_pkg_mgr": "yum",
    "ansible_processor": [
        "GenuineIntel",
        "Intel(R) Xeon(R) CPU E5-2630L v2 @ 2.40GHz"
    ],
    "ansible_processor_cores": 1,
    "ansible_processor_count": 1,
    "ansible_processor_threads_per_core": 1,
    "ansible_processor_vcpus": 1,
    "ansible_product_name": "Droplet",
    "ansible_product_serial": "NA",
    "ansible_product_uuid": "NA",
    "ansible_product_version": "20160415",
    "ansible_python_version": "2.7.5",
    "ansible_selinux": {
        "status": "disabled"
    },
    "ansible_service_mgr": "systemd",
    "ansible_ssh_host_key_dsa_public":
"AAAAB3NzaC1kc3MAAACBAPEf4dzeET6ukHemTASsamoRLxo2R8iHg5J1bYQUyug
gtRKlbRrHMtpQ8qN5CQNtp8J+2Hq6/JKiDF+cdxgOehf9b7F4araVvJxqx967RvLNBrM
WXv7/4hi+efgXG9eejGoGQNAD66up/fkLMd0L8fwSwmTJoZXwOxFwcbnxCZsFAAAA
FQDgK7fka+1AKjYZNFIfCB2b0ZitGQAAAIADeofiC5q+SLgEvkBCUCTyJ+EVb6WH
eHbVdrpE2GdnUr03R6MmmYhYZMijruS/rcpzBLmi8juDkqAWy6Xqxd+DwixykntXPe
UFS3F7LK5vNwFalaRltPwr4Azh+EeSUQ2Zz2AdKx6zSqtLOD8ZMPkRDvz4WGHGme
R+i7UFsFDZdgAAAIEAy26Tx0jAlY3mEaTW9lQ9DoGXgPBxsSX/XqeLh5wBaBO6AJa
```

```
Irs0dQJdNeHcMhFy0seVkOMN1SpeoBTJSoTOx15HAGsKsAcmnA5mcJeUZqptVR6Jx
ROztHw3zQePQ3/V3KQzAN31tIm3PbKztlEZbXRUM7RV5WsdRHTb8rutENhY=",
        "ansible_ssh_host_key_ecdsa_public":
"AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBPDXQ9rj
gDmUKsEWH4U2vg4iqtK+75urlj9nwW+rNNTFHTE5oG82sOlO6o0tUY8LXgB/tJnIcJ1
hINdrWrZNpn4=",
        "ansible_ssh_host_key_rsa_public":
"AAAAB3NzaC1yc2EAAAADAQABAAABAQCwQx5EElH7FeD/agB/gCJfBUEVhk44tl
dzdEzwc2IEbI59relTGNOU7soCCMcSH7nwlEbOOvmLa2R/YaXdHv/cb1aXBC/wj/m4Z
HylBeF5qzECUkeaB3+CT+hp8qHHApclFr2lm2CwZ+YXjEyjJ3en4K3gLlIQyQjgE2F57
kmD1FVVDSJFvNTn+NQvb3DPppND+HKEeHwrJ0GgznoP62yobEgriAIBSGf//0WHC
O/9shEvauoRpPM+U9pU7lv637s7qyubIqyrs5fz3u34qBj8oCATOefRN1wsfJDeMG0D5ry
I6BI6t/eAi8BPr7VHJSQBk+buM9Jr1yoMQTEasq2J",
        "ansible_swapfree_mb": 0,
        "ansible_swaptotal_mb": 0,
        "ansible_system": "Linux",
        "ansible_system_vendor": "DigitalOcean",
        "ansible_uptime_seconds": 603067,
        "ansible_user_dir": "/home/fale",
        "ansible_user_gecos": "",
        "ansible_user_gid": 1000,
        "ansible_user_id": "fale",
        "ansible_user_shell": "/bin/bash",
        "ansible_user_uid": 1000,
        "ansible_userspace_architecture": "x86_64",
        "ansible_userspace_bits": "64",
        "ansible_virtualization_role": "host",
        "ansible_virtualization_type": "kvm",
        "module_setup": true
    },
    "changed": false
}
```

As you can see, from this huge list of options, you can gain a huge quantity of information, and you can use them as any other variable. Let's print the OS name and the version. To do so, we can create a new playbook called setup_variables.yamlwith the following content:

```
---
- hosts: all
  remote_user: fale
  tasks:
  - name: Print OS and version
    debug:
      msg: '{{ ansible_distribution }} {{ ansible_distribution_version }}'
```

Run it with the following:

```
$ ansible-playbook -itest01.fale.io setup_variables.yaml
```

This will give us the following output:

```
PLAY [all] ********************************************************
TASK [setup] ******************************************************
ok: [test01.fale.io]


TASK [Print OS and version] ***************************************
ok: [test01.fale.io] => {
  "msg": "CentOS 7.2.1511"

}


PLAYRECAP *********************************************************
test01.fale.io     : ok=2   changed=0    unreachable=0    failed=0
```

As you can see, it printed the OS name and version, as expected. In addition to the methods seen previously, it's also possible to pass a variable using a command-line argument. In fact, if we look in the Ansible help, we will notice the following:

```
-e EXTRA_VARS, --extra-vars=EXTRA_VARS
set additional variables as key=value or YAML/JSON
```

The same lines are present in the ansible-playbook command as well. Let's make a small playbook called cli_variables.yaml with the following content:

```
---
- hosts: all
  remote_user: fale
  tasks:
  - name: Print variable 'name'
    debug:
      msg: '{{ name }}'
```

```
ansible-playbook  cli_variables.yaml -e 'name=test01'
```

We will receive the following:

```
PLAY [all] *******************************************************
TASK [setup] *****************************************************
ok: [test01.fale.io]
TASK [Print variable 'name'] *************************************
ok: [test01.fale.io] => {
   "msg": "test01"
}
PLAY RECAP *******************************************************
test01.fale.io     : ok=2   changed=0    unreachable=0    failed=0
```

Creating the Ansible user

When you create a machine (or rent one from any hosting company) it arrives only with the root user. Let's start creating a playbook that ensures that an Ansible user is created, it's accessible with an SSH key, and is able to perform actions on behalf of other users (sudo) with no password asked. I often call this playbook, firstrun.yaml since I execute it as soon as a new machine is created, but after that, I don't use it since it uses the root user that I disable for security reasons. Our script will look something like the following:

```
---
- hosts: all
  user: root
  tasks:
  - name: Ensure ansible user exists
    user:
      name: ansible
      state: present
      comment: Ansible
  - name: Ensure ansible user accepts the SSH key
    authorized_key:
      user: ansible
      key: https://github.com/fale.keys
   state: present
  - name: Ensure the ansible user is sudoer with no password required
    lineinfile:
      dest: /etc/sudoers
      state: present
      regexp: '^ansible ALL\='
      line: 'ansible ALL=(ALL) NOPASSWD:ALL'
      validate: 'visudo -cf %s'
```

Before running it, let's look at it a little bit. We have used three different modules (user, authorized_key, and lineinfile) that we have never seen. The user module, as the name suggests, allows us to make sure a user is present (or absent).

The authorized_key module allows us to ensure that a certain SSH key can be used to login as a specific user on that machine. This module will not substitute all the SSH keys that are already enabled for that user, but will simply add (or remove) the specified key. If you want to alter this behavior, you can use the **exclusive** option, that allows you to delete all the SSH keys that are not specified in this step.

The lineinfile module allows us to alter the content of a file. It works in a very similar way to **sed** (a stream editor), where you specify the regular expression that will be used to match the line, and then specify the new line that will be used to substitute the matched line. If no line is matched, the line is added at the end of the file. Now let's run it with:

```
$ ansible-playbook firstrun.yaml
```

This will give us the following result:

```
PLAY [all] ***********************************************************
TASK [setup] **********************************************************
ok: [test01.fale.io]


TASK [Ensure ansible user exists] ***********************************
changed: [test01.fale.io]


TASK [Ensure ansible user accepts the SSH key] *******************
changed: [test01.fale.io]
```

```
TASK [Ensure the anisble user is sudoer with no password required] *
changed: [test01.fale.io]


PLAY RECAP *********************************************************
test01.fale.io     : ok=4   changed=3    unreachable=0    failed=0
```

Configuring a basic server

After we have created the user for Ansible with the necessary privileges, we can go on to make some other small changes to the OS. To make it more clear, we will see how each action is performed and then we'll look at the whole playbook.

Enabling EPEL

EPEL is the most important repository for Enterprise Linux and it contains a lot of additional packages. It's also a safe repository since no package in EPEL will conflict with packages in the base repository. To enable EPEL in RHEL/CentOS 7, it is enough to just install the epel-release package. To do so in Ansible, we will use:

```
- name: Ensure EPEL is enabled
  yum:
    name: epel-release
    state: present
  become: True
```

As you can see, we have used the yum module, as we did in one of the first examples of the chapter, specifying the name of the package and that we want it to be present.

Installing Python bindings for SELinux

Since Ansible is written in Python and mainly uses the Python bindings to operate on the operating system, we will need to install the Python bindings for SELinux:

```
- name: Ensure libselinux-python is present
  yum:
    name: libselinux-python
    state: present
  become: True
- name: Ensure libsemanage-python is present
  yum:
    name: libsemanage-python
    state: present
  become: True
```

Upgrading all installed packages

To upgrade all installed packages, we will need to use the yum module again, but with a different parameter, in fact we would use:

```
- name: Ensure we have last version of every package
  yum:
    name: "*"
    state: latest
  become: True
```

As you can see, we have specified "*" as the package name (this stands for a wildcard to match all installed packages) and the state is latest. This will upgrade all installed packages to the latest version available.

If you remember, when we talked about the "present" state, we said that it was going to install the last available version. So what's the difference between "present" and "latest"? Present will install the latest version if the package is not installed, while if the package is already installed (no matter the version) it will go forward without making any change. Latest will install the latest version if the package is not installed, while if the package is already installed will check whether a newer version is available and if it is, Ansible will update the package.

Ensuring that NTP is installed, configured, and running

To make sure NTP is present, we use the yum module:

```
- name: Ensure NTP is installed
  yum:
    name: ntp
    state: present
  become: True
```

Now that we know that NTP is installed, we should ensure that the server is using the timezone that we want. To do so, we will create a symbolic link in /etc/localtime that will point to the wanted zoneinfo file:

```
- name: Ensure the timezone is set to UTC
  file:
    src: /usr/share/zoneinfo/GMT
    dest: /etc/localtime
    state: link
  become: True
```

As you can see, we have used the file module to tell Ansible, specifying that it needs to be a link (state: link).

To complete the NTP configuration, we need to start the ntpd service and ensure that it will run at every, consequent boot:

```
- name: Ensure the NTP service is running and enabled
  service:
    name: ntpd
    state: started
    enabled: True
  become: True
```

Ensuring that FirewallD is present and enabled

As you can imagine, the first step is to ensure that FirewallD is installed:

```
- name: Ensure FirewallD is installed
  yum:
    name: firewalld
    state: present
  become: True
```

Since we want to be sure that, when we enable FirewallD we will not lose our SSH connection, we ensure that SSH traffic can always pass through it:

```
- name: Ensure SSH can pass the firewall
  firewalld:
    service: ssh
    state: enabled
    permanent: True
    immediate: True
  become: True
```

To do so, we have used the firewalld module. This module will take parameters that are very similar to the ones the firewall-cmd console would use. You will have to specify the service that is to be authorized to pass the firewall, whether you want this rule to apply immediately, and whether you want the rule to be permanent so that after a reboot the rule will still be present.

Now that we have installed FirewallD and we are sure that our SSH connection will survive, we can enable it as we do any other service:

```
- name: Ensure FirewallD is running
  service:
    name: firewalld
    state: started
    enabled: True
  become: True
```

Adding a customized MOTD

To add the MOTD, we will need a template that will be the same for all servers and a task to use the template.

I find it very useful to add a MOTD to every server. It's even more useful if you use Ansible, because you can use it to warn your users that changes to the system could be overwritten by Ansible. My usual template is called 'motd', and has this content:

```
This system is managed by Ansible
```

```
  Any change done on this system could be overwritten by Ansible

OS: {{ ansible_distribution }} {{ ansible_distribution_version }}
Hostname: {{ inventory_hostname }}
eth0 address: {{ ansible_eth0.ipv4.address }}
All connections are monitored and recorded
    Disconnect IMMEDIATELY if you are not an authorized user
```

This is a jinja2 template and it allows us to use every variable set in the playbooks. This also allows us to use complex syntax for conditionals and cycles that we will see later in this chapter. To populate a file from a template in Ansible, we will need to use:

```
- name: Ensure the MOTD file is present and updated
  template:
    src: motd
    dest: /etc/motd
    owner: root
    group: root
    mode: 0644
  become: True
```

The template module allows us to specify a local file (src) that will be interpreted by jinja2 and the output of this operation will be saved on the remote machine in a specific path (dest), be owned by a specific user (owner) and group (group), and have a specific access mode (mode).

Changing the hostname

To keep things simple, one way I find useful is to set the hostname of a machine to something meaningful. To do so, we can use a very simple Ansible module called hostname:

```
- name: Ensure the hostname is the same of the inventory
  hostname:
    name: "{{ inventory_hostname }}"
  become: True
```

Reviewing and running the playbook

Putting everything together, we now have the following playbook (called common_tasks.yaml for simplicity):

```
---
- hosts: all
  remote_user: ansible
  tasks:
  - name: Ensure EPEL is enabled
    yum:
      name: epel-release
      state: present
    become: True
  - name: Ensure libselinux-python is present
    yum:
      name: libselinux-python
      state: present
    become: True
  - name: Ensure libsemanage-python is present
    yum:
      name: libsemanage-python
      state: present
    become: True
  - name: Ensure we have last version of every package
    yum:
      name: "*"
      state: latest
    become: True
  - name: Ensure NTP is installed
    yum:
      name: ntp
      state: present
    become: True
  - name: Ensure the timezone is set to UTC
    file:
      src: /usr/share/zoneinfo/GMT
      dest: /etc/localtime
      state: link
    become: True
  - name: Ensure the NTP service is running and enabled
    service:
      name: ntpd
      state: started
      enabled: True
    become: True
  - name: Ensure FirewallD is installed
    yum:
      name: firewalld
      state: present
    become: True
```

```
- name: Ensure FirewallD is running
  service:
    name: firewalld
    state: started
    enabled: True
  become: True
- name: Ensure SSH can pass the firewall
  firewalld:
    service: ssh
    state: enabled
    permanent: True
    immediate: True
  become: True
- name: Ensure the MOTD file is present and updated
  template:
    src: motd
    dest: /etc/motd
    owner: root
    group: root
    mode: 0644
  become: True
- name: Ensure the hostname is the same of the inventory
  hostname:
    name: "{{ inventory_hostname }}"
  become: True
```

Since this playbook is pretty complex, we can run the following:

```
$ ansible-playbook common_tasks.yaml --list-tasks
```

This asks Ansible to print all the tasks in a shorter form so that we can quickly see what tasks a playbook performs. The output should be something like the following:

```
playbook: common_tasks.yaml
  play #1 (all): all TAGS: []
    tasks:
      Ensure EPEL is enabled TAGS: []
      Ensure libselinux-python is present TAGS: []
      Ensure libsemanage-python is present TAGS: []
      Ensure we have last version of every package TAGS: []
      Ensure NTP is installed TAGS: []
      Ensure the timezone is set to UTC TAGS: []
      Ensure the NTP service is running and enabled TAGS: []
      Ensure FirewallD is installed TAGS: []
      Ensure FirewallD is running TAGS: []
      Ensure SSH can pass the firewall TAGS: []
      Ensure the MOTD file is present and updated TAGS: []
      Ensure the hostname is the same of the inventory TAGS: []
```

We can now run the playbook with the following:

```
$ ansible-playbook -itest01.fale.io common_tasks.yaml
```

We will receive the following output:

```
PLAY [all] ***********************************************************

TASK [setup] *********************************************************
ok: [test01.fale.io]

TASK [Ensure EPEL is enabled] ****************************************
changed: [test01.fale.io]

TASK [Ensure libselinux-python is present] ***********************
ok: [test01.fale.io]

TASK [Esure libsemanage-python is present] ***********************
ok: [test01.fale.io]

TASK [Ensure we have last version of every package] **************
changed: [test01.fale.io]

TASK [Ensure NTP is installed] ***********************************
ok: [test01.fale.io]

TASK [Ensure the timezone is set to UTC] *************************
changed: [test01.fale.io]

TASK [Ensure the NTP service is running and enabled] *************
changed: [test01.fale.io]

TASK [Ensure FirewallD is installed] *****************************
ok: [test01.fale.io]

TASK [Ensure FirewallD is running] *******************************
```

```
changed: [test01.fale.io]

TASK [Ensure SSH can pass the firewall] ***************************
ok: [test01.fale.io]

TASK [Ensure the MOTD file is present and updated] ****************
changed: [test01.fale.io]

TASK [Ensure the hostname is the same of the inventory] **********
changed: [test01.fale.io]

PLAY RECAP ********************************************************
test01.fale.io     : ok=9   changed=7    unreachable=0    failed=0
```

Installing and configuring a web server

Now that we have made some generic changes to the operating system, let's move on to actually creating a web server. We are splitting those two phases so we can share the first phase between every machine and apply the second only to web servers.

For this second phase, we will create a new playbook called webserver.yaml with the following content:

```
---
- hosts: all
  remote_user: ansible
  tasks:
  - name: Ensure the HTTPd package is installed
    yum:
      name: httpd
      state: present
    become: True
  - name: Ensure the HTTPd service is enabled and running
    service:
      name: httpd
      state: started
      enabled: True
    become: True
  - name: Ensure HTTP can pass the firewall
    firewalld:
      service: http
      state: enabled
      permanent: True
      immediate: True
    become: True
  - name: Ensure HTTPS can pass the firewall
    firewalld:
      service: https
      state: enabled
      permanent: True
      immediate: True
    become: True
```

As you can see, the first two tasks are the same as the ones in the example at the beginning of this chapter, and the last two tasks are used to instruct FirewallD to let HTTP and HTTPS traffic pass.

Let's run this script with the following:

```
ansible-playbook -i test01.fale.io webserver.yaml
```

This results in the following:

```
PLAY [all] ***********************************************************
TASK [setup] *********************************************************
ok: [test01.fale.io]


TASK [Ensure the HTTPd package is installed] *********************
changed: [test01.fale.io]


TASK [Ensure the HTTPd service is enabled and running] ***********
changed: [test01.fale.io]


TASK [Ensure HTTP can pass the firewall] *************************
changed: [test01.fale.io]
```

```
TASK [Ensure HTTPS can pass the firewall] ************************
changed: [test01.fale.io]


PLAY RECAP ******************************************************
test01.fale.io    : ok=5    changed=4    unreachable=0    failed=0
```

Publishing a website

Since our website will be a simple, single page website, we can easily create it and publish it using a single Ansible task. To make this page a little bit more interesting, we will create it from a template that will be populated by Ansible with a little data about the machine. The script to publish it will be called deploy_website.yaml and will have the following content:

```
---
- hosts: all
  remote_user: ansible
  tasks:
  - name: Ensure the website is present and updated
    template:
      src: index.html.j2
      dest: /var/www/html/index.html
      owner: root
      group: root
      mode: 0644
    become: True
```

Let's start with a simple template that we will call index.html.j2:

```
<html>
<body>
<h1>Hello World!</h1>
</body>
</html>
```

Now we can test our website deployment by running the following:

```
$ ansible-playbook -i test01.fale.io deploy_website.yaml
```

We should receive the following output:

```
PLAY [all] ******************************************************
TASK [setup] *****************************************************
ok: [test01.fale.io]


TASK [Ensure the website is present and updated] *****************
changed: [test01.fale.io]


PLAY RECAP *******************************************************
test01.fale.io    : ok=2    changed=1    unreachable=0    failed=0
```

If you now go to your test machine IP/FQDN in your browser, you'll find the "Hello World!" page.

Jinja2 templates

Jinja2 is a widely-used and fully-featured template engine for Python. Let's look at some syntax that will help us with Ansible. This paragraph does not want to be a replacement for the official documentation, but its goal is to teach you some components that you'll find very useful when using them with Ansible.

As we have seen, we can print variable content simply with the '{{ VARIABLE_NAME }}' syntax. If we want to print just an element of an array we can use '{{ ARRAY_NAME['KEY'] }}', and if we want to print a property of an object, we can use '{{ OBJECT_NAME.PROPERTY_NAME }}'.

So we can improve our previous static page in the following way:

```
<html>
<body>
<h1>Hello World!</h1>
<p>This page was created on {{ ansible_date_time.date }}.</p>
</body>
</html>
```

Filters

From time to time, we may want to change the style of a string a little bit, without writing specific code for it, for example, we may want to capitalize some text. To do so, we can use one of Jinja2's filters, such as: '{{ VARIABLE_NAME | capitalize }}'. There are many filters available for Jinja2 and you can find the full list at: http://jinja.pocoo.org/docs/dev/templates/#builtin-filters.

Conditionals

One thing you may often find useful in a template engine is the possibility of printing different strings depending on the content (or existence) of a string. So we can improve our static web page in the following way:

```
<html>
<body>
<h1>Hello World!</h1>
<p>This page was created on {{ ansible_date_time.date }}.</p>
{% if ansible_eth0.active == True %}
<p>eth0 address {{ ansible_eth0.ipv4.address }}.</p>
{% endif %}
</body>
</html>
```

As you can see, we have added the capability to print the main IPv4 address for the eth0 connection, if the connection is active. With conditionals we can also use the tests.

Note: For a full list, please refer to: http://jinja.pocoo.org/docs/dev/templates/#builtin-tests.

So to obtain the same result we could also have written the following:

```
<html>
<body>
<h1>Hello World!</h1>
<p>This page was created on {{ ansible_date_time.date }}.</p>
{% if ansible_eth0.active is equalto True %}
```

```
<p>eth0 address {{ ansible_eth0.ipv4.address }}.</p>
{% endif %}
</body>
</html>
```

There are a lot of different tests that will really help you to create easy-to-read, effective templates.

Cycles

The jinja2 template system also offers the capability to create cycles. Let's add a feature to our page that will print the main IPv4 network address for each device instead of only eth0. We will then have the following code:

```
<html>
<body>
<h1>Hello World!</h1>
<p>This page was created on {{ ansible_date_time.date }}.</p>
<p>This machine can be reached on the following IP addresses</p>
<ul>
{% for address in ansible_all_ipv4_addresses %}
<li>{{ address }}</li>
{% endfor %}
</ul>
</body>
</html>
```

As you can see, the syntax for cycles is familiar if you already know Python.

These few pages on Jinja2 templating were not a substitute for the official documentation. In fact Jinja2 templates are much more powerful than what we have seen here. The goal here is only to give you the basic Jinja2 templates that are most often used in Ansible.

Working with inventory files

An inventory file is the source of truth for Ansible (there is also an advanced concept called **dynamic inventory**, which we will cover later). It follows the **Initialization** (**INI**) format and tells Ansible whether the remote host or hosts provided by the user are genuine.

Ansible can run its tasks against multiple hosts in parallel. To do this, you can directly pass the list of hosts to Ansible using an inventory file. For such parallel execution, Ansible allows you to group your hosts in the inventory file; the file passes the group name to Ansible. Ansible will search for that group in the inventory file and run its tasks against all the hosts listed in that group.

You can pass the inventory file to Ansible using the -i or --inventory-file option followed by the path to the file. If you do not explicitly specify any inventory file to Ansible, it will take the default path from the host_file parameter of ansible.cfg, which defaults to /etc/ansible/hosts.

The basic inventory file

Before diving into the concept, let's first look at a basic inventory file called **hosts** that we can use instead of the list we used in the previous examples:

```
test01.fale.io
```

We can now perform the same operations that we did in the previous chapter, tweaking the Ansible command parameters. For instance, to install the web server, we used this command:

```
$ ansible-playbook -i test01.fale.iowebserver.yaml
```

Groups in an inventory file

The advantages of inventory files are noticeable when we have more complex situations. Let's say our website is getting more complicated and we now need a more complex environment. In our example, our website will require a MySQL database. Also we decide to have two web servers. In this scenario it makes sense to group different machines based on their role in our infrastructure. Our hosts file would change to:

```
[webserver]
ws01.fale.io
ws02.fale.io

[database]
db01.fale.io
```

Now we can instruct playbooks to run only on hosts in a certain group. We have created three different playbooks for our website example:

- firstrun.yaml is generic and will have to be run on every machine
- common_tasks.yaml is generic and will have to be run on every machine
- webserver.yaml is specific for web servers and therefore should not be run on any other machines

We need to change only the webserver.yaml file which, at the moment, specifies that it has to be run on all machines and should become web server only. To do so, let's open the webserver.yaml file and change content from:

```
- hosts: all
```

To

```
- hosts: webserver
```

With only those three playbooks, we cannot proceed to create our environment with three servers. Since we don't have a playbook to set up the database yet (we will see it in the next chapter), we will provision the two web servers completely and for the database server we will only provision the base system.

We can run the firstrun playbook with the following:

```
$ ansible-playbook -i hosts firstrun.yaml
```

The following would be the result:

```
PLAY [all] *****************************************************
TASK [setup] ***************************************************
ok: [ws02.fale.io]
ok: [db01.fale.io]
ok: [ws01.fale.io]


TASK [Ensure ansible user exists] ******************************
changed: [ws01.fale.io]
changed: [db01.fale.io]
changed: [ws02.fale.io]


TASK [Ensure ansible user accepts the SSH key] *****************
changed: [ws02.fale.io]
changed: [ws01.fale.io]
changed: [db01.fale.io]


TASK [Ensure the ansible user is sudoer with no password required]
changed: [ws01.fale.io]
changed: [db01.fale.io]
changed: [ws02.fale.io]


PLAY RECAP *****************************************************
db01.fale.io    : ok=4   changed=3   unreachable=0   failed=0
ws01.fale.io    : ok=4   changed=3   unreachable=0   failed=0
ws02.fale.io    : ok=4   changed=3   unreachable=0   failed=0
```

As you can see, the output is very similar to what we received with a single host, but with one line per host at each step. In this case, all the machines were in the same state and the same steps have been performed, so we see that they all acted the same, but with more complex scenarios, you can have different machines returning different states on the same step. We can also execute the other two playbooks with similar results.

Regular expressions in the inventory file

When you have a large number of servers, it is common and helpful to give them predictable names, for instance, call all web servers wsXY or webXY, or call the database servers dbXY. If you do so, you can reduce the number of lines in your hosts file increasing its readability. For instance, our hosts file can be simplified as:

```
[webserver]
ws[01:02].fale.io

[database]
db01.fale.io
```

In this example, we have used [01:02] that will match for all occurrences between the first number (01 in our case) and the last (02 in our case). In our case, the gain is not huge, but if you have 40 web servers, you can cut 39 lines from your hosts file.

Working with variables

Ansible allows you to define variables in many ways, from a variable file within a playbook, by passing it from the Ansible command using the -e / --extra-vars option, or by passing it to an inventory file. You can define variables in an inventory file either on a per-host basis, for an entire group, or by creating a variable file in the directory where your inventory file exists.

Host variables

It's possible to declare variables for a specific host, declaring them in the hosts file. For instance, we may want to specify different engines for our web servers. Let's suppose that one needs to reply to a specific domain, while the other to a different domain name. In this case, we would do it with the following hosts file:

```
[webserver]
ws01.fale.io domainname=example1.fale.io
ws02.fale.io domainname=example2.fale.io

[database]
db01.fale.io
```

In this way, all playbooks running on web servers will be able to refer to the domain name variable.

Group variables

There are other cases where you want to set a variable that is valid for the whole group. Let's suppose that we want to declare the variable https_enabled to True and its value has to be equal for all web servers. In this case, we can create a [webserver:vars] section, so we will use the following hosts file:

```
[webserver]
ws01.fale.io
ws02.fale.io

[webserver:vars]
https_enabled=True

[database]
db01.fale.io
```

Variable files

Sometimes, you have a lot of variables to declare for each host and group, and the hosts file gets hard to read. In those cases, you can move the variables to specific files. For host level variables, you'll need to create a file named the same as your host in the host_vars folder, while for group variables you'll have to use the group name for the file name and place them in the group_vars folder.

So, if we want to replicate the previous example of host-based variables using files, we will need to create the host_vars/ws01.fale.io file with the following content:

```
domainname=example1.fale.io
```

Create the host_vars/ws02.fale.io file with the following conten

```
domainname=example2.fale.io
```

While if we want to replicate the group based variables example, we will need to have the group_vars/webserver file with the following content:

```
https_enabled=True
```

Overriding configuration parameters with an inventory file

eters will override all the other parameters that are set either through ansible.cfg, environment variables, or set in the playbooks themselves. Variable passed to the ansible-playbook/ansible command have priority over any other variable, including the ones set in the inventory file.

The following is the list of parameters you can override from an inventory file:

- ansible_user: This parameter is used to override the user that is used for communicating with the remote host. Sometimes, a certain machine needs a different user, in those cases this variable will help you.
- ansible_port: This parameter will override the default SSH port with the user-specified port. Sometimes sysadmin chooses to run SSH on a non-standard port. In this case, you'll need to instruct Ansible about the change.
- ansible_host: This parameter is used to override the host for an alias.
- ansible_connection: This specifies the connection type to the remote host. The values are SSH, Paramiko, or local.
- ansible_private_key_file: This parameter will override the private key used for SSH; this will be useful if you want to use specific keys for a specific host. A common use case is if you have hosts spread across multiple data centers, multiple AWS regions, or different kinds of applications. Private keys can potentially be different in such scenarios.
- ansible__type: By default, Ansible uses the **sh shell**; you can override this using the ansible_shell_type parameter. Changing this to csh, ksh, and so on will make Ansible use the commands of that shell.

Working with dynamic inventory

The idea behind dynamic inventories is that Ansible will not read the hosts file, but instead execute a script that will return the list of hosts to Ansible in JSON format. This allows you, for instance, to query your cloud provider and ask it directly, what machines in your entire infrastructure are running at any given moment.

Many scripts for the most common cloud providers are already present in Ansible at: https://github.com/ansible/ansible/tree/devel/contrib/inventory but you can create a custom script if you have different needs. The Ansible inventory scripts can be written in any language but, for consistency reasons, dynamic inventory scripts should be written in Python. Remember that these scripts need to be executable directly, so please remember to set them with the executable flag (chmod + x inventory.py).

Amazon Web Services

To allow Ansible to gather data from **Amazon Web Services** (**AWS**) about your EC2 instances, you need to download the following two files from Ansible's GitHub repository at https://github.com/ansible/ansible:

- The ec2.py inventory script
- The ec2.ini file, which contains the configuration for your EC2 inventory script

Ansible uses AWS Python library boto to communicate with AWS using APIs. To allow this communication, you need to export the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY variables.

You can use the inventory in two ways:

- Pass it directly to an ansible-playbook command using the -i option and copy the ec2.ini file to your current directory where you are running the Ansible commands
- Copy the ec2.py file to /etc/ansible/hosts, make it executable using chmod +x, and copy the ec2.ini file to /etc/ansible/ec2.ini

The ec2.py file will create multiple groups based on the region, availability zone, tags, and so on. You can check the contents of the inventory file by running ./ec2.py --list.

Let's see an example playbook with EC2 dynamic inventory, which will simply ping all machines in my account.

```
ansible -i ec2.py all -m ping
```

As expected, the two droplets I have on my account respond with the following:

```
52.28.138.231 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

In the preceding example, we're using the ec2.py script instead of a static inventory file with the -i option and the ping command.

Similarly, you can use these inventory scripts to perform various types of operations. For example, you can integrate it with your deployment script to figure out all the nodes in a single zone and deploy to them if you're performing your deployment zone-wise (a zone represents a data center) in AWS.

If you simply want to know what the web servers in the cloud are and you've tagged them using a certain convention, you can do that by using the dynamic inventory script by filtering out the tags. Furthermore, if you have special scenarios that are not covered by your present script, you can enhance it to provide the required set of nodes in JSON format and then act on those nodes from the playbooks. If you're using a database to manage your inventory, your inventory script can query the database and dump a JSON. It could even sync with your cloud and update your database on a regular basis.

Working with iterates in Ansible

You may have noticed that up to now we have never used cycles, so every time we had to do multiple, similar operations, we wrote the code multiple times. An example of this is the webserver.yaml code.

In fact, this was the content of the webserver.yaml file:

```
---
- hosts: webserver
  remote_user: ansible
  tasks:
  - name: Ensure the HTTPd package is installed
    yum:
      name: httpd
      state: present
    become: True
  - name: Ensure the HTTPd service is enabled and running
    service:
      name: httpd
      state: started
      enabled: True
    become: True
  - name: Ensure HTTP can pass the firewall
    firewalld:
      service: http
      state: enabled
      permanent: True
      immediate: True
    become: True
  - name: Ensure HTTPS can pass the firewall
    firewalld:
      service: https
      state: enabled
      permanent: True
      immediate: True
    become: True
```

As you can see, the last two blocks do the same operation (ensure that a certain port of the firewall is open).

Standard iteration - with_items

To improve the above code, we can use a simple iteration: with_items.

This allows us to iterate in a list of item, and at every iteration, the designated item of the list will be available to us in the item variable.

We can therefore change that code to the following:

```
---
- hosts: webserver
  remote_user: ansible
  tasks:
  - name: Ensure the HTTPd package is installed
    yum:
      name: httpd
      state: present
    become: True
  - name: Ensure the HTTPd service is enabled and running
    service:
      name: httpd
      state: started
      enabled: True
    become: True
  - name: Ensure HTTP and HTTPS can pass the firewall
    firewalld:
      service: '{{ item }}'
      state: enabled
      permanent: True
      immediate: True
    become: True
    with_items:
    - http
    - https
```

We can execute it as the following:

```
ansible-playbook -i hosts webserver.yaml
```

We receive the following:

```
PLAY [webserver] *****************************************************
TASK [setup] *********************************************************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [Ensure the HTTPd package is installed] *********************
ok: [ws01.fale.io]
ok: [ws02.fale.io]
```

```
TASK [Ensure the HTTPd service is enabled and running] ***********
ok:[ws02.fale.io]
ok:[ws01.fale.io]


TASK [Ensure HTTP can pass the firewall] *************************
ok: [ws02.fale.io]
ok: [ws01.fale.io]


TASK [Ensure HTTP and HTTPS can pass the firewall] ***************
ok: [ws02.fale.io] => (item=http)
ok: [ws01.fale.io] => (item=http)
ok: [ws02.fale.io] => (item=https)
ok: [ws01.fale.io] => (item=https)


PLAY RECAP *******************************************************
ws01.fale.io     : ok=5    changed=0    unreachable=0    failed=0
ws02.fale.io     : ok=5    changed=0    unreachable=0    failed=0
```

As you can see, the output is slightly different from the previous execution, in fact on the lines for operations with loops we can see the item that was processed the "Ensure HTTP and HTTPS can pass the firewall" block

We have now seen that we can iterate on a list of items, but Ansible allows us other kind of iterations as well.

Nested loops - with_nested

There are cases where you have to iterate all elements of a list with all items from other lists (**Cartesian product**). One case that is very common is when you have to create multiple folders in multiple paths. In our example, we will create the folders mail and public_html in the home folders of the users alice and bob.

We can do so with the following code from the with_nested.yaml file:

```
---
- hosts: all
  remote_user: ansible
  vars:
    users:
    - alice
    - bob
    folders:
    - mail
    - public_html
  tasks:
  - name: Ensure the users exist
    user:
      name: '{{ item }}'
    become: True
    with_items:
    - '{{ users }}'
  - name: Ensure the folders exist
    file:
      path: '/home/{{ item.0 }}/{{ item.1 }}'
      state: directory
    become: True
    with_nested:
    - '{{ users }}'
    - '{{ folders }}'
```

Running this with the following:

```
ansible-playbook -i hosts with_nested.yaml
```

We receive the following result:

```
PLAY [all] ********************************************************
TASK [setup] ******************************************************
ok: [ws01.fale.io]
ok: [db01.fale.io]
ok: [ws02.fale.io]


TASK [Ensure the users exist] ************************************
changed: [db01.fale.io] => (item=alice)
changed: [ws01.fale.io] => (item=alice)
changed: [ws02.fale.io] => (item=alice)
```

```
changed: [db01.fale.io] => (item=bob)
changed: [ws01.fale.io] => (item=bob)
changed: [ws02.fale.io] => (item=bob)


TASK [Ensure the folders exist] ***********************************
changed: [ws02.fale.io] => (item=[u'alice', u'mail'])
changed: [ws01.fale.io] => (item=[u'alice', u'mail'])
changed: [db01.fale.io] => (item=[u'alice', u'mail'])
changed: [ws01.fale.io] => (item=[u'alice', u'public_html'])
changed: [ws02.fale.io] => (item=[u'alice', u'public_html'])
changed: [db01.fale.io] => (item=[u'alice', u'public_html'])
changed: [ws02.fale.io] => (item=[u'bob', u'mail'])
changed: [ws01.fale.io] => (item=[u'bob', u'mail'])
changed: [db01.fale.io] => (item=[u'bob', u'mail'])
changed: [ws02.fale.io] => (item=[u'bob', u'public_html'])
changed: [ws01.fale.io] => (item=[u'bob', u'public_html'])
changed: [db01.fale.io] => (item=[u'bob', u'public_html'])


PLAY RECAP ********************************************************
db01.fale.io     : ok=3   changed=2   unreachable=0   failed=0
ws01.fale.io     : ok=3   changed=2   unreachable=0   failed=0
ws02.fale.io     : ok=3   changed=2   unreachable=0   failed=0
```

Ansible supports many more types of loop, but since they are used far less, you can refer directly to the official documentation about loops at http://docs.ansible.com/ansible/playbooks_loops.html.

Delegating a task

Sometimes you want to execute an action on a different system. This could be, for instance, a database node while you are deploying something on an application server node or to the local host. To do so, you can just add the 'delegate_to: HOST' property to your task and it will be run on the proper node. Let's rework the previous example to achieve this:

```
---
  - hosts: database
```

```
      remote_user: ansible
      tasks:
      - name: Count processes running on the remote system
        shell: ps | wc -l
        register: remote_processes_number
      - name: Print remote running processes
        debug:
          msg: '{{ remote_processes_number.stdout }}'
      - name: Count processes running on the local system
        shell: ps | wc -l
        delegate_to: localhost
        register: local_processes_number
      - name: Print local running processes
        debug:
          msg: '{{ local_processes_number.stdout }}'
```

Saving it as delegate_to.yaml, we can run it with the following:

```
ansible-playbook -i hosts delegate_to.yaml
```

We will receive the same output as the previous example:

```
PLAY [database] ***************************************************
TASK [setup] ******************************************************
ok: [db01.fale.io]


TASK [Count processes running on the remote system] ***************
changed: [db01.fale.io]


TASK [Print remote running processes] ****************************
ok: [db01.fale.io] => {
    "msg": "7"
}


TASK [Count processes running on the local system] ***************
changed: [db01.fale.io -> localhost]
```

```
TASK [Print local running processes] ******************************
ok: [db01.fale.io] => {
    "msg": "11"
}


PLAY RECAP ***************************************************************
db01.fale.io      : ok=5    changed=2    unreachable=0    failed=0
```

Working with conditionals

Until now, we have only seen how playbooks work and how tasks are executed. We also saw that Ansible executes all these tasks sequentially. However, this would not help you while writing an advanced playbook that contains tens of tasks and have to execute only a subset of these tasks. For example, let's say you have a playbook that will install Apache HTTPd server on the remote host. Now, the Apache HTTPd server has a different package name for a Debian-based operating system, and it's called apache2; for a Red-Hat-based operating system, it's called httpd.

Having two tasks, one for the httpd package (for Red-Hat-based systems) and the other for the apache2 package (for Debian-based systems) in a playbook, will make Ansible install both packages, and this execution will fail, as apache2 will not be available if you're installing on a Red-Hat-based operating system. To overcome such problems, Ansible provides conditional statements that help run a task only when a specified condition is met. In this case, we do something similar to the following pseudocode:

```
If os = "redhat"
  Install httpd
Else if os = "debian"
  Install apache2
End
```

While installing httpd on a Red-Hat-based operating system, we first check whether the remote system is running a Red-Hat-based operating system, and if it is, we then install the httpd package; otherwise, we skip the task. Without wasting your time, let's dive into an example playbook called conditional_httpd.yaml with the following content:

```
---
  - hosts: webserver
    remote_user: ansible
    tasks:
    - name: Print the ansible_os_family value
      debug:
        msg: '{{ ansible_os_family }}'
    - name: Ensure the httpd package is updated
      yum:
        name: httpd
        state: latest
      become: True
      when: ansible_os_family == 'RedHat'
    - name: Ensure the apache2 package is updated
      apt:
        name: apache2
        state: latest
      become: True
      when: ansible_os_family == 'Debian'
```

Run it with the following:

```
ansible-playbook -i hosts conditional_httpd.yaml
```

This is the result:

```
PLAY [webserver] **************************************************
TASK [setup] *******************************************************
ok: [ws03.fale.io]
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [Print the ansible_os_family value] **************************
ok: [ws01.fale.io] => {
   "msg": "RedHat"
}
ok: [ws02.fale.io] => {
   "msg": "RedHat"
}
ok: [ws03.fale.io] => {
    "msg": "Debian"
}


TASK [Ensure the httpd package is updated] ***********************
skipping: [ws03.fale.io]
changed: [ws01.fale.io]
changed: [ws02.fale.io]


TASK [Ensure the apache2 package is updated] *********************
skipping: [ws02.fale.io]
skipping: [ws01.fale.io]
changed: [ws03.fale.io]


PLAY RECAP *****************************************************
ws01.fale.io      : ok=3   changed=1   unreachable=0   failed=0
ws02.fale.io      : ok=3   changed=1   unreachable=0   failed=0
ws03.fale.io      : ok=3   changed=1   unreachable=0   failed=0
```

As you can see, I've created a new server (ws03) for this example that is Debian-based. As expected, the installation of the httpd package was performed on the two CentOS nodes, while the installation of the apache2 package was performed on the Debian node.

Likewise, you can match for different conditions as well. Ansible supports equal to (==), different than (!=), bigger than (>), smaller than (<), bigger than or equal to (>=), and smaller than or equal to (<=).

The operators we have seen so far will match the entire content of the variable, but what if you just want to check whether a particular character or a string is present in a variable? To perform these kinds of checks, Ansible provides the in and not operators. You can also match multiple conditions using the AND and OR operators. The AND operator will make sure that all conditions are matched before executing this task, whereas the OR operator will make sure that at least one of the conditions there is a match for at least one of the conditions, for example, you can use foo >= 0 and foo <= 5.

Boolean conditionals

Apart from string matching, you can also check whether a variable is True. This type of validation will be useful when you want to check whether a variable was assigned a value or not. You can even execute a task based on the Boolean value of a variable.

For example, let's put the following code in a file called crontab_backup.yaml:

```
---
  - hosts: all
    remote_user: ansible
    vars:
      backup: True
    tasks:
    - name: Copy the crontab in tmp if the backup variable is true
      copy:
        src: /etc/crontab
        dest: /tmp/crontab
        remote_src: True
      when: backup
```

```
ansible-playbook -i hosts crontab_backup.yaml
```

We will obtain the following:

```
PLAY [all] ******************************************************
TASK [setup] ****************************************************
ok: [db01.fale.io]
ok: [ws02.fale.io]
ok: [ws01.fale.io]


TASK [Copy the crontab in tmp if the backup variable is true] ****
changed: [ws02.fale.io]
changed: [db01.fale.io]
changed: [ws01.fale.io]
```

```
PLAY RECAP ********************************************************
db01.fale.io     : ok=2   changed=1   unreachable=0   failed=0
ws01.fale.io     : ok=2   changed=1   unreachable=0   failed=0
ws02.fale.io     : ok=2   changed=1   unreachable=0   failed=0
```

But if we change the command slightly, to

```
ansible-playbook -i hosts crontab_backup.yaml --extra-vars="backup=False"
```

We will receive this output:

```
PLAY [all] ********************************************************
TASK [setup] ******************************************************
ok: [db01.fale.io]
ok: [ws02.fale.io]
ok: [ws01.fale.io]


TASK [Copy the crontab in tmp if the backup variable is true] ****
skipping: [ws01.fale.io]
skipping: [ws02.fale.io]
skipping: [db01.fale.io]


PLAY RECAP ********************************************************
db01.fale.io     : ok=1   changed=0   unreachable=0   failed=0
ws01.fale.io     : ok=1   changed=0   unreachable=0   failed=0
ws02.fale.io     : ok=1   changed=0   unreachable=0   failed=0
```

Checking if a variable is set

Sometimes you find yourself having to use a variable in a command. Every time you do so, you have to ensure that the variable is **set**. This is because some commands could be catastrophic if called with an **unset** variable (that is: if you execute `rm -rf $VAR/*` and `$VAR` is not set or empty, it will nuke your machine). To do so, Ansible provides a way to check whether a variable is defined or not.

We could improve the previous example in the following way:

```
---
  - hosts: all
    remote_user: ansible
    vars:
      backup: True
    tasks:
    - name: Check if the backup_folder is set
      fail:
        msg: 'The backup_folder needs to be set'
      when: backup_folder is not defined
    - name: Copy the crontab in tmp if the backup variable is true
      copy:
        src: /etc/crontab
        dest: '{{ backup_folder }}/crontab'
        remote_src: True
      when: backup
```

Working with include

The include feature helps you to reduce duplicity while writing tasks. This also allows us to have smaller playbooks by including reusable code in separate tasks using the **Don't Repeat Yourself** (**DRY**) principle.

To trigger the inclusion of another file, you need to put the following under the tasks object:

```
- include: FILENAME.yaml
```

You can also pass some variables to the included file. To do so, we can specify them in the following way:

```
- include: FILENAME.yaml variable1="value1" variable2="value2"
```

In addition of passing variables, you can also use conditionals to include a file only when certain conditions are matched, for instance to include the redhat.yaml file only if the machine is running an OS in the Red Hat family using the following code:

```
- name: Include the file only for Red Hat OSes
include: redhat.yaml
when: ansible_os_family == "RedHat"
```

Working with handlers

In many situations, you will have a task or a group of tasks that change certain resources on the remote machines, which need to trigger an event to become effective. For example, when you change a service configuration, you will need to restart or reload the service itself. In Ansible you can trigger this event using the notify action.

Every handler task will run at the end of the playbook if notified. For example, you changed your HTTPd server configuration multiple times and you want to restart the HTTPd service so that the changes are applied. Now, restarting HTTPd every single time you make a configuration change is not a good practice; it is not a good practice to restart the server even if no changes has been made to its configurations. To deal with such a situation, you can notify Ansible to restart the HTTPd service on every configuration change, but Ansible will make sure that no matter how many times you notify it for the HTTPd restart, it will call that task just once after all other tasks complete. Let's change the webserver.yaml file we created in the previous chapters a little bit, in the following way:

```
---
- hosts: webserver
  remote_user: ansible
  tasks:
  - name: Ensure the HTTPd package is installed
    yum:
      name: httpd
      state: present
    become: True
  - name: Ensure the HTTPd service is enabled and running
    service:
      name: httpd
      state: started
      enabled: True
    become: True
```

```
- name: Ensure HTTP can pass the firewall
  firewalld:
    service: http
    state: enabled
    permanent: True
    immediate: True
  become: True
- name: Ensure HTTPd configuration is updated
  copy:
    src: website.conf
    dest: /etc/httpd/conf.d
  become: True
  notify: Restart HTTPd
handlers:
- name: Restart HTTPd
  service:
    name: httpd
    state: restarted
  become: True
```

Run this script with:

```
ansible-playbook -i hosts webserver.yaml
```

We will have the following output:

```
PLAY [webserver] ***************************************************
TASK [setup] *******************************************************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [Ensure the HTTPd package is installed] ***********************
ok: [ws02.fale.io]
ok: [ws01.fale.io]


TASK [Ensure the HTTPd service is enabled and running] ***********
ok: [ws02.fale.io]
ok: [ws01.fale.io]


TASK [Ensure HTTP can pass the firewall] *************************
ok: [ws02.fale.io]
```

```
ok: [ws01.fale.io]


TASK [Ensure HTTPd configuration is updated] *********************
changed: [ws02.fale.io]
changed: [ws01.fale.io]


RUNNING HANDLER [Restart HTTPd] ********************************
changed: [ws02.fale.io]
changed: [ws01.fale.io]


PLAY RECAP *****************************************************
ws01.fale.io      : ok=6    changed=2    unreachable=0    failed=0
ws02.fale.io      : ok=6    changed=2    unreachable=0    failed=0
```

In this case, the handler has been triggered from the configuration file change. But if we run it a second time, the configuration will not change and therefore we will have the following result:

```
PLAY [webserver] ***********************************************
TASK [setup] ***************************************************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [Ensure the HTTPd package is installed] *********************
ok: [ws02.fale.io]
ok: [ws01.fale.io]


TASK [Ensure the HTTPd service is enabled and running] ***********
ok: [ws02.fale.io]
ok: [ws01.fale.io]


TASK [Ensure HTTP can pass the firewall] *************************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [Ensure HTTPd configuration is updated] *********************
ok: [ws02.fale.io]
```

```
ok: [ws01.fale.io]


PLAY RECAP *******************************************************
ws01.fale.io     : ok=5   changed=0   unreachable=0   failed=0
ws02.fale.io     : ok=5   changed=0   unreachable=0   failed=0
```

When using handlers, those are triggered only a single time, even if they are called multiple times during the playbook execution. By default, handlers are executed at the end of the playbook execution, but you can force them to be run when you want using the meta task with the flush_handlers option like: - meta: flush_handlers

Working with roles

We have seen how we can automate simple tasks, but what we have seen up till now will not solve all your problems. This is because playbooks are very good at executing operations, but are not very good for configuring huge amounts of machines, because they will soon become messy. To solve this, Ansible has roles.

My definition of a role is a set of playbooks, templates, files, or variables to achieve a specific goal. For instance, we could have a database role and a web server role so that those configurations stay cleanly separated.

Before starting to look inside a role, let's talk about a project organization.

Project organization

In the last few years, I've worked on multiple Ansible repositories for multiple organizations and many of them were very chaotic. To ensure that your repository is easy to manage, I'm going to give you a template that I always use.

First of all, I always create three files in the root folder:

- ansible.cfg: A small configuration file to explain to Ansible where to find the files in our folder structure
- hosts: The hosts file we have already seen in the previous chapters
- master.yaml: A playbook that aligns the whole infrastructure

In addition to those three files, I create two folders:

- playbooks: This will contain the playbooks and a folder called **groups** for groups management
- roles: This will contain all the roles we need

To clarify this, let's use the Linux tree command to see the structure of an Ansible repository for a simple web application needing web servers and database servers:

```
ansible.cfg
   hosts
   master.yaml
   playbooks
      firstrun.yaml
      groups
         database.yaml
         webserver.yaml
   roles
       common
       database
       webserver
```

As you can see, I've added a common role as well. This is very useful for putting in all the things that should be performed for every server. Usually, I configure NTP, motd, and other similar services in this role, as well as the machine hostname.

We will now see how to structure a role.

Anatomy of a role

The structure of folders in a role is standard and you cannot change it much.

The most important folder within the role is the tasks folder because this is the only mandatory folder in it. It has to contain a main.yaml file that will be the list of tasks to be executed. Other folders that are often present in the roles are templates and files. The first one will be used to store templates used by the **template task**, while the second will be used to store files that are used by the **copy task**.

Transforming your playbooks in a full Ansible project

Let's see how to transform the three playbooks we used to set up our web infrastructure (common_tasks.yaml, firstrun.yaml, and webserver.yaml) to fit this file organization. We have to remember that we also used two files (index.html.j2 and motd) in those roles, so we have to place these files properly too.

First, we are going to create the folder structure we have seen in the previous paragraph.

The easiest playbook to port is the firstrun.yaml since we only need to copy it into the playbooks folder. This playbook will remain a playbook because it's a set of operations that will have to be run just one time for each server.

We now move to the common_tasks.yaml playbook, which will need a little bit of rework to match the role paradigm.

Transforming a playbook into a role

The first thing we need is to create the roles/common/tasks and roles/common/templates folders. In the first one we will add the following main.yaml file:

```
- name: Ensure SSH can pass the firewall
    firewalld:
      service: ssh
      state: enabled
      permanent: True
      immediate: True
    become: True
  - name: Ensure the MOTD file is present and updated
    template:
      src: motd
      dest: /etc/motd
      owner: root
      group: root
      mode: 0644
    become: True
  - name: Ensure the hostname is the same of the inventory
    hostname:
      name: "{{ inventory_hostname }}"
    become: True
```

As you can see, this is very similar to our common_tasks.yaml playbooks. In fact, there are only two differences:

- The lines; hosts, remote_user, and tasks (lines 2,3, and 4) have been deleted
- The indentation of the rest of the file has been fixed accordingly

In this role, we used the template task to create a motd file on the server with the IP of the machine and other interesting information. For this reason, we need to create roles/common/templates and put the motd template in it.

At this point, our common task will have this structure:

```
common/
    tasks
        main.yaml
    templates
        motd
```

We now need to instruct Ansible on the machines that will need to perform all the tasks specified in the common role. To do so, we should look at the playbooks/groups directory. In this directory, it is handy to have one file for each group of logically similar machines (that is, machines that are performing the same kind of operation). In our case, database and web server.

So, let's create a database.yaml file in playbooks/groups with the following content:

```
---
- hosts: database
  user: ansible
  roles:
  - common
```

Create a webserver.yaml file in the same folder with the following content:

```
---
- hosts: webserver
  user: ansible
  roles:
  - common
```

As you can see, those files specify the group of hosts that we want to operate on, the remote user to use on those hosts, and the roles that we want to execute.

Helper files

When we created the hosts file in the previous chapter, we noticed that it helps to simplify our command lines. So, let's start copying the hosts files we previously used in the root folder of our Ansible repository. Up to now, we have always specified the path of this file on the command line. This is no longer necessary if we create an ansible.cfg file that tells Ansible the location of our hosts file. For this reason, let's create an ansible.cfg file in the root of our Ansible repository with the following content:

```
[defaults]
```

```
hostfile = hosts
host_key_checking = False
roles_path = roles
```

In this file, we have also specified another two variables in addition to the hostfile one that we already talk about, and those are host_key_checking and roles_path.

The host_key_checking flag is useful to not require the verification of the remote system SSH key. This is not suggested for use in production, since the usage of a public key propagation system is suggested for such environments, but is very handy in testing environments since it will help you to reduce the time Ansible hangs waiting for user input.

The roles_path is used to tell Ansible where to find the roles for our playbooks.

I usually add one additional file, which is master.yaml. I find it very useful as you will often need to keep your infrastructure aligned with your Ansible code. To do it in a single command, you'll need a file that will run all of the files in playbooks/groups. So, let's create a master.yaml file in the Ansible repository root folder with the following content:

```
---
- include: playbooks/groups/database.yaml
- include: playbooks/groups/webserver.yaml
```

At this point, we can execute the following:

```
ansible-playbook master.yaml
```

The result will be the following:

```
PLAY [database] ******************************************************
TASK [setup] *********************************************************
ok: [db01.fale.io]
```

```
TASK [common : Ensure EPEL is enabled] ****************************
ok: [db01.fale.io]

TASK [common : Ensure libselinux-python is present] ***************
ok: [db01.fale.io]

TASK [common : Ensure libsemanage-python is present] **************
ok: [db01.fale.io]

TASK [common : Ensure we have last version of every package] *****
changed: [db01.fale.io]

TASK [common : Ensure NTP is installed] ***************************
ok: [db01.fale.io]

TASK [common : Ensure the timezone is set to UTC] ****************
ok: [db01.fale.io]

TASK [common : Ensure the NTP service is running and enabled] ****
ok: [db01.fale.io]

TASK [common : Ensure FirewallD is installed] ********************
ok: [db01.fale.io]

TASK [common : Ensure FirewallD is running] **********************
ok: [db01.fale.io]

TASK [common : Ensure SSH can pass the firewall] *****************
ok: [db01.fale.io]

TASK [common : Ensure the MOTD file is present and updated] ******
ok: [db01.fale.io]

TASK [common : Ensure the hostname is the same of the inventory] *
```

```
ok: [db01.fale.io]


PLAY [webserver] ****************************************************
TASK [setup] ********************************************************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [common : Ensure EPEL is enabled] ****************************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [common : Ensure libselinux-python is present] ***************
ok: [ws02.fale.io]
ok: [ws01.fale.io]


TASK [common : Ensure libsemanage-python is present] *************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [common : Ensure we have last version of every package] *****
changed: [ws01.fale.io]
changed: [ws02.fale.io]


TASK [common : Ensure NTP is installed] **************************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [common : Ensure the timezone is set to UTC] ****************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [common : Ensure the NTP service is running and enabled] ****
ok: [ws02.fale.io]
ok: [ws01.fale.io]


TASK [common : Ensure FirewallD is installed] ********************
ok: [ws02.fale.io]
```

```
ok: [ws01.fale.io]


TASK [common : Ensure FirewallD is running] **********************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [common : Ensure SSH can pass the firewall] *****************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [common : Ensure the MOTD file is present and updated] ******
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [common : Ensure the hostname is the same of the inventory] *
ok: [ws02.fale.io]
ok: [ws01.fale.io]


PLAY RECAP *****************************************************
db01.fale.io    : ok=13  changed=1  unreachable=0  failed=0
ws01.fale.io    : ok=13  changed=1  unreachable=0  failed=0
ws02.fale.io    : ok=13  changed=1  unreachable=0  failed=0
```

As you can see, the actions listed in the common role have been executed on the node in the database group first and then on the nodes in the webserver group.

Transforming the webserver role

As we transformed the common playbook into the common role, we can do the same for the webserver role.

In roles, we need to have the webserver folder with the tasks subfolder inside it. In this folder, we have to put the main.yaml file containing the tasks copied from the playbooks, that should look like:

```
---
  - name: Ensure the HTTPd package is installed
    yum:
      name: httpd
      state: present
    become: True
  - name: Ensure the HTTPd service is enabled and running
    service:
      name: httpd
      state: started
      enabled: True
    become: True
  - name: Ensure HTTP can pass the firewall
    firewalld:
      service: http
      state: enabled
      permanent: True
      immediate: True
    become: True
  - name: Ensure HTTPd configuration is updated
    copy:
      src: website.conf
      dest: /etc/httpd/conf.d
    become: True
    notify: Restart HTTPd
  - name: Ensure the website is present and updated
    template:
      src: index.html.j2
      dest: /var/www/html/index.html
      owner: root
      group: root
      mode: 0644
    become: True
```

In this role, we have used multiple tasks that will need additional resources to work properly, more specifically we need to:

- Put the website.conf file in roles/webserver/files
- Put the index.html.j2 template in roles/webserver/templates
- Create the Restart HTTPd handler

The first two should be pretty straightforward. The first one, in fact, is an empty file (we have not yet put anything in it since the default configuration was good enough for our use) and the index.html.j2 file should contain the following content:

```
<html>
<body>
<h1>Hello World!</h1>
<p>This page was created on {{ ansible_date_time.date }}.</p>
<p>This machine can be reached on the following IP addresses</p>
<ul>
   {% for address in ansible_all_ipv4_addresses %}
<li>{{ address }}</li>
   {% endfor %}
</ul>
</body>
</html>
```

Handlers in roles

The last thing we need to do to complete this role is to create the handler for the Restart HTTPd notification. To do so, we will need to create a main.yaml file in roles/webserver/handlers with the following content:

```
---
- name: Restart HTTPd
  service:
    name: httpd
    state: restarted
  become: True
```

As you may notice, this is very similar to the handler we used in the playbook if not for the file location and indentation.

The only thing that we still need to do to make our role applicable is to add the entry in the playbooks/groups/webserver.yaml file so that Ansible is informed that the servers in the webserver group should apply the webserver role as well as the common role. Our playbooks/groups/webserver.yaml will need to be like the following:

```
---
- hosts: webserver
  user: ansible
  roles:
  - common
  - webserver
```

We could now execute the master.yaml again to apply the webserver role to the relevant servers, but we can also just execute the playbooks/groups/webserver.yaml, since the change we just did is relevant only to this group of servers. To do so we run:

```
ansible-playbook playbooks/groups/webserver.yaml
```

We should receive an output similar to the following:

```
PLAY [webserver] *************************************************
TASK [setup] *******************************************************
ok: [ws02.fale.io]
ok: [ws01.fale.io]


TASK [common : Ensure EPEL is enabled] ****************************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [common : Ensure libselinux-python is present] **************
ok: [ws01.fale.io]
ok: [ws02.fale.io]


TASK [common : Ensure libsemanage-python is present] *************
ok: [ws01.fale.io]
ok: [ws02.fale.io]
```

As you can see, both the common and the webserver roles has been applied to the webserver nodes.

It's very important to apply all roles concerning a specific node and not just the one you changed because more often than not, when there is a problem on one or more nodes in a group but not on other nodes of the same group, the problem is some roles have been applied unequally in the group. Only by applying all concerned roles to a group, will it grant you the equality of the nodes of that group.

Execution strategies

Before Ansible 2, every task needed to be executed (and completed) on each machine before Ansible issued a new task to all machines. This meant that if you are performing tasks on a hundred machines and one of them is under-performing, all machines will go at the under-performing machine's speed.

With Ansible 2, the execution strategies have been made modular and therefore you can now choose which execution strategy you prefer for your playbooks. You can also write custom execution strategies, but this is beyond the scope of this book. At the moment (in Ansible 2.1) there are only three execution strategies: **linear**, **serial**, and **free**:

- **Linear execution**: This strategy behaves exactly as Ansible did prior to version 2. This is the default strategy.
- **Serial execution**: This strategy will take a subset of hosts (the default is five) and execute all tasks against those hosts before moving to the next subset and starting from the beginning. This kind of execution strategy could help you to work on a limited number of hosts so that you always have some hosts that are available to your users. If you are looking for this kind of deployment, you will need a load balancer in front of your hosts that needs to be informed about which nodes are in maintenance at every given moment.
- **Free execution**: This strategy will serve a new task to each host as soon as that host has completed the previous task. This will allow faster hosts to complete the playbook before slower nodes. If you choose this execution strategy you have to remember that

some tasks could require a previous task to be completed on all nodes (for instance, clustering databases require all database nodes to have the database installed and running) and in this case they will probably fail.

Tasks blocks

In Ansible 2.0 blocks have been made available. Blocks allow you to group tasks in a logical way and they can also help for a better error handling. The majority of properties you can add to a standard task, you can also add it to the blocks. You may need to perform a yum task to install NTPd and enable of the service only if the machine is CentOS. To do so, the following code can be used:

```
tasks:
- block:
   - name: Ensure NTPd is present
   yum:
     name: ntpd
     state: present
   - name: Ensure NTPd is running
   service:
     name: ntpd
     state: started
   enabled: True
 when: ansible_distribution == 'CentOS'
```

As you can notice, the when clause has been applied to the block so all tasks within the block will be performed only if the when clause will be true.

Security management

The last section in this chapter is about security management. If you tell your sysadmin that you want to introduce a new feature or a tool, one of the first questions they would ask you would be; "what security feature(s) are present with your tool?". We'll try to answer these questions from an Ansible perspective in this section. Let's look at them in greater detail.

Using Ansible vault

Ansible vault is an exciting feature of Ansible that was introduced in Ansible version 1.5. This allows you to have encrypted passwords as part of your source code. A recommended practice is to NOT have passwords (as well as any other sensitive information such as private keys, SSL certificates, and so on.) in plain text as part of your repository because anyone who checks out your repository can view your passwords. Ansible vault can help you to secure your confidential information by encrypting and decrypting them on your behalf.

Ansible vault supports an interactive mode in which it will ask you for the password, or a non-interactive mode where you will have to specify the file containing the password and Ansible vault will read it directly.

For these examples, we will use the password ansible, so let's start creating a hidden file called .password with the string ansible in it. To do so, let's execute:

```
echo 'ansible' > .password
```

We can now create an ansible-vault both in the interactive and non-interactive modes. If we want to do it in interactive mode, we will need to execute:

```
ansible-vault create secret.yaml
```

Ansible will ask us for the vault password and then confirm it. Later it will open the default text editor (in my case **vi**) to add the content in clear. I have used the password ansible and the text is This is a password protected file. We can now save and close the editor and check that ansible-vault has encrypted our content, in fact if we run:

```
cat secret.yaml
```

This will output the following:

```
$ANSIBLE_VAULT;1.1;AES256
66346431333933663461383331393763666538373163336536353335646532323135383630646366
3432353561393533623764323961666639326132323331370a636363613032616664333039356565
64643735626162646166313861366532323161646137333634333336393062303461343638333737
6534326135326430390a643739336461616334313833313363343030666662653864353138666233
38386266383866353836373036303339383962363362333364346432613062363830316330653866
6431343764386132663066303761346532643632633432643861
```

In the same way, we can invoke the ansible-vault command with the -vault-password-file=VAULT_PASSWORD_FILEoption to specify our .password file. We can, for instance, edit our secret.yaml file with the command:

C

```
ansible-vault --vault-password-file=.password edit secret.yaml
```

This will open your default text editor where you'll be able to change the file as if it was a plain file. When you save the file, Ansible vault will perform the encryption before saving it, assuring the confidentiality of your content.

Sometimes you need to look at the content of a file but you don't want to open it in a text editor, so you usually use catcommand. Ansible vault has a similar feature called view, so you can run:

```
ansible-vault --vault-password-file=.password view secret.yaml
```

Ansible vault allows you to decrypt a file, replacing its encrypted content with its plain text content. To do so, you can execute:

Copy

```
ansible-vault --vault-password-file=.password decrypt secret.yaml
```

At this point, we can the cat command on the secret.yaml file and the result is the following:

```
This is a password protected file
```

```
ansible-vault --vault-password-file=.password encrypt secret.yaml
```

You can now check that the secret.yaml file is now encrypted again.

The last option of the Ansible vault is very important since it's a rekey function. This function will allow you to change the encryption key in a single command. You could perform the same operation with two commands (decrypt the secret.yamlfile with the **old key** and then encrypt it with the **new key**) but being able to perform it in a single step has major advantages since the file in its clear-text form will not be stored on the disk at any moment of the process. To do so we need a file containing the new password (in our case, the file called .newpassword and containing the string ansible2), and you need to execute the following command:

```
ansible-vault --vault-password-file=.password --new-vault-password-file=.newpassword
rekey secret.yaml
```

We can now use the cat command on to the secret.yaml file and we will see the following output:

```
$ANSIBLE_VAULT;1.1;AES256
63313864643434663939333132333537336362313133616430376463613833353366326662303832
6431316131613033343266373137356166383564326234300a38623663363533393933323464
3435
64353932383930613934343730386635333030373663313631646462613566313362313363393135
3935613661373263330a3166343335336653461356535383662376464656466623536363537386462
```

```
31636637346538636161616632313866366365666361633138666134303433316665376237326162
36386537383838303234303131613364653232646136343234 34
```

Vaults and playbooks

You can also use vaults with ansible-playbook. You'll need to decrypt the file on-the-fly using a command such as the following:

```
$ ansible-playbook site.yml --vault-password-file .password
```

There is yet another option that allows you to decrypt files using a script, which can then look up some other source and decrypt the file. This can also be a useful option to provide more security. However, make sure that the get_password.pyscript has executable permissions:

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

Before closing this chapter, I'd like to speak a little bit about the password file. This file needs to be present on the machine where you execute your playbooks, in a location and with permissions so that is readable by the user who is executing the playbook. You can create the .password file at startup. The '.' character in the .password filename is to make sure that the file is hidden by default when you look for it. This is not directly a security measure, but could help mitigate cases where an attacker does not know exactly what he is looking for.

The .password file content should be a password or key that is secure and accessible only to folks who have permission to run Ansible playbooks.

Finally, make sure that you're not encrypting every file that's available! Ansible vault should be used only for important information that needs to be secure.

Every time you'll save an encrypted file, no matter if changes have been applied or not, the file will be re-encrypted and therefore will change in encrypted content. This will cause your SCM tool to mark the file as modified.

Encrypting user passwords

Ansible vault takes care of passwords that are checked in and helps you handle them while running Ansible playbooks or commands. However, when Ansible plays are run, at times you might need your users to enter passwords. You also want to make sure that these passwords don't appear in the comprehensive Ansible logs (the default location: /var/log/ansible.log) or on stdout.

Ansible uses Passlib, which is a password hashing library for Python, to handle encryption for prompted passwords. You can use any of the following algorithms supported by Passlib:

- des_crypt: DES Crypt
- bsdi_crypt: BSDi Crypt
- bigcrypt: BigCrypt
- crypt16: Crypt16
- md5_crypt: MD5 Crypt
- bcrypt: BCrypt
- sha1_crypt: SHA-1 Crypt
- sun_md5_crypt: Sun MD5 Crypt
- sha256_crypt: SHA-256 Crypt
- sha512_crypt: SHA-512 Crypt
- apr_md5_crypt: Apache's MD5-Crypt variant
- phpass: PHPass' Portable Hash

- pbkdf2_digest: Generic PBKDF2 Hashes
- cta_pbkdf2_sha1: Cryptacular's PBKDF2 hash
- dlitz_pbkdf2_sha1: Dwayne Litzenberger's PBKDF2 hash
- scram: SCRAM Hash
- bsd_nthash: FreeBSD's MCF-compatible nthash encoding

Let's now see how encryption works with a variable prompt:

```
vars_prompt:
- name: ssh_password
  prompt: Enter ssh_password
  private: True
  encryption: md5_crypt
  confirm: True
  salt_size: 7
```

In the preceding snippet, vars_prompt is used to prompt users for some data. The vars_prompt is not a task but is another section at the same level as the tasks: one.

The name module indicates the actual variable name where Ansible will store the user password, as shown in the following command:

```
name: ssh_password
```

We are using the prompt utility to prompt users for the password as follows:

```
prompt: Enter ssh password
```

We are explicitly asking Ansible to hide the password from stdout by using private module; this works like any other password prompt on a Unix system. The private module is accessed as follows:

```
private: True
```

We are using the md5_crypt algorithm over here with a salt size of 7:

```
encrypt: md5_crypt
salt_size: 7
```

Moreover, Ansible will prompt for the password twice and compare both passwords:

```
confirm: True
```

Hiding passwords

Ansible, by default, filters output that contains the login_password key, the password key, and the user:passformat. For example, if you are passing a password in your module using login_password or the password key, then Ansible will replace your password with VALUE_HIDDEN. Let's now see how you can hide a password using the passwordkey:

```
- name: Running a script
  shell: script.sh
    password: my_password
```

In the preceding shell task, we use the password key to pass passwords. This will allow Ansible to hide it from stdout and its log file.

Now, when you run the preceding task in the **verbose** mode, you should not see your mypass password; instead Ansible, with VALUE_HIDDEN, will replace it as follows:

```
REMOTE_MODULE command script.sh password=VALUE_HIDDEN #USE_SHELL
```

Using no_log

Ansible will hide your passwords only if you are using a specific set of keys. However, this might not be the case every time; moreover, you might also want to hide some other confidential data. The no_log feature of Ansible will hide your entire task from logging it to the syslog file. It will still print your task on stdout and log it to other Ansible logfiles.

Another way to prevent Ansible from logging is to set in the ansible.cfg file, in the [defaults] section, log_path with the value /dev/null so that all logs are saved in /dev/null, and therefore lost.

Let's now see how you can hide an entire task with no_log as follows:

```
- name: Running a script
  shell: script.sh
    password: my_password
  no_log: True
```