Chapter 11: Ansible and Scripting for Orchestration and Automation

Ansible has become the de facto standard in today's open source community because it offers so much while asking so little of you and your infrastructure. Using Ansible with **Kernel-based Virtual Machine** (**KVM**) also makes a lot of sense, especially when you think about larger environments. It doesn't really matter if it's just a simple provisioning of KVM hosts that you want to do (install libvirt and related software), or if you want to uniformly configure KVM networking on hosts – Ansible can be invaluable for both. For example, in this chapter, we will use Ansible to deploy a virtual machine and multi-tier application that's hosted inside KVM virtual machines, which is a very common use case in larger environments. Then, we'll move to more

pedantic subjects of combining Ansible and cloud-init since they differ in terms of timeline when they're applied and a way in which things get done. Cloud-init is an ideal automatic way for initial virtual machine configuration (hostname, network, and SSH keys). Then, we usually move to Ansible so that we can perform additional orchestration post-initial configuration – add software packages, make bigger changes to the system, and so on. Let's see how we can use Ansible and cloud-init with KVM.

In this chapter, we will cover the following topics:

- Understanding Ansible
- Provisioning a virtual machine using the **`kvm_libvirt`** module
- Using Ansible and cloud-init for automation and orchestration
- Orchestrating multi-tier application deployment on KVM VMs
- Learning by example, including various examples on how to use Ansible with KVM

Let's get started!

Understanding Ansible

One of the primary roles of a competent administrator is to try and automate themselves out of everything they possibly can. There is a saying that you must do everything manually at least once. If you must do it again, you will probably be annoyed by it, and the third time you must do it, you will automate the process. When we talk about automation, it can mean a lot of different things.

Let's try to explain this with an example as this is the most convenient way of describing the problem and solution. Let's say that you're working for a company that needs to deploy 50 web servers to host a web application, with standard configuration. Standard configuration includes the software packages that you need to install, the services and network settings that need to be configured, the firewall rules that need to be

configured, and the files that need to be copied from a network share to a local disk inside a virtual machine so that we can serve these files via a web server. How are you going to make that happen?

There are three basic approaches that come to mind:

- Do everything manually. This will cost a lot of time and there will be ample opportunity to do something wrong as we're humans, after all, and we make mistakes (pun intended).
- Try to automate the process by deploying 50 virtual machines and then throwing the whole configuration aspect into a script, which can be a part of the automated installation procedure (for example, kickstart).
- Try to automate the process by deploying a single virtual machine template that will contain all the moving parts already installed. This means we just need to deploy these 50 virtual machines from a virtual machine template and

do a bit of customization to make sure that our virtual machines are ready to be used.

There are different kinds of automation available. Pure scripting is one of them, and it involves creating a script out of everything that needs to run more than once. An administrator that has been doing a job for years usually has a batch of useful scripts. Good administrators also know at least one programming language, even when they hate to admit it, since being an administrator means having to fix things after others break them, and it sometimes involves quite a bit of programming.

So, if you're considering doing automation via a script, we absolutely agree with you that it's doable. But the question remains regarding how much time you'll spend covering every single aspect of that script to get everything right so that the script *always* works properly. Furthermore, if it doesn't, you're going to have to do a lot of manual labor to make it right, without any real way

of amending an additional configuration on top of the previous, unsuccessful one.

This is where procedure-based tools such as Ansible come in handy. Ansible produces **modules** that get pushed to endpoints (in our example, virtual machines) that bring our object to a *desired state*. If you're coming from the Microsoft PowerShell world, yes, Ansible and PowerShell **Desired State Configuration** (**DSC**) are essentially trying to do the same thing. They just go about it in a different way. So, let's discuss these different automatization processes to see where Ansible fits into that world.

Automation approaches

In general, all of this applies to administering systems and their parts, installing applications, and generally taking care of things inside the installed system. This can be considered an *old* approach to administration since it generally deals with services, not servers. At the same time, this kind of automation is decidedly focused on a sin-

gle server or a small number of servers since it doesn't scale well. If we need to work on multiple servers, using regular scripts creates new problems. We need to take a lot of additional variables into account (different SSH keys, hostnames, and IP addresses) since scripts are more difficult to expand to work on multiple servers (which is easy in Ansible).

If one script isn't enough, then we have to move to multiple scripts, which creates a new problem, one of which is script management. Think about it – what happens when we need to change something in a script? How do we make sure that all the instances on all the servers are using the same version, especially if the server IP addresses aren't sequential? So, to conclude, while old and tested, this kind of automation has serious drawbacks.

There's another kind of automation that is gathering traction in the DevOps community – Automation with a capital A. This is a way to automate systems operation across different ma-

chines – even across different operating systems. There are a couple of automation systems that enable this, and they can basically be divided into two groups: systems that use agents and agentless systems.

Systems that use agents

Systems that use agents are more common since they have a few advantages over agentless systems. The first and foremost advantage is their ability to track not only changes that need to be done, but also changes that the user makes to the system. This change tracking means that we can track what is happening across systems and take appropriate actions.

Almost all of them work in the same way. A small application – called an agent – is installed on the system that we need to monitor. After the application has been installed, it connects, or permits connections, from the central server, which handles everything regarding automation. Since you are reading this, you are probably familiar with

systems like this. There are quite a few of them around, and chances are you've already run into one of them. To understand this principle, take a look at the following diagram:

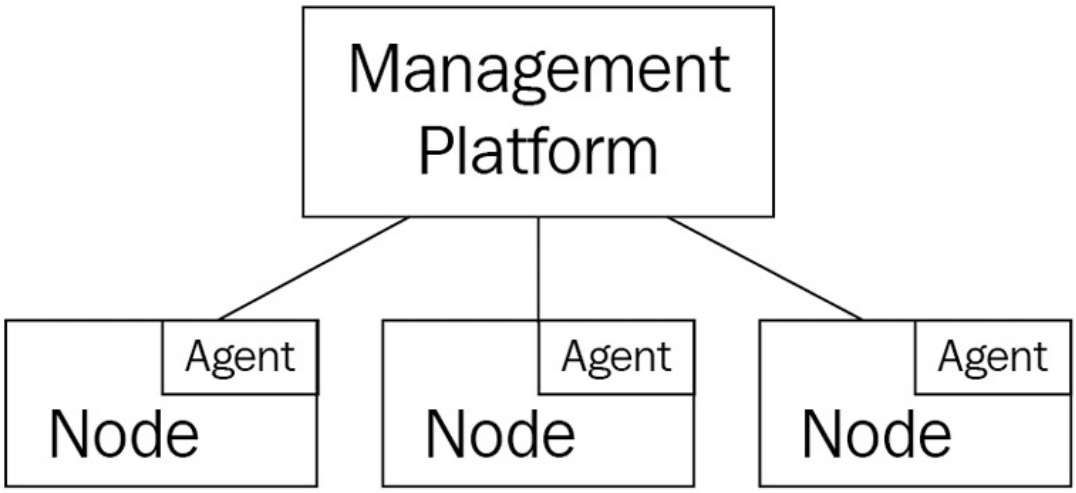


Figure 11.1 – The management platform needs an agent to connect to objects that need orchestration and automation

In these systems, agents have a dual purpose. They are here to run whatever needs to run locally, and to constantly monitor the system for changes. This change-tracking ability can be accomplished in different ways, but the result is similar – the central system will know what has changed and in what way. Change-tracking is an important thing in deployment since it enables

compliance checking in real-time and prevents a lot of problems that arise from unauthorized changes.

Agentless systems

Agentless systems behave differently. Nothing is installed on the system that has to be managed; instead, the central server (or servers) does everything using some kind of command and control channel. On Windows, this may be **PowerShell**, **WinRM**, or something similar, while on Linux, this usually **SSH** or some other remote execution framework. The central server creates a task that then gets executed through the remote channel, usually in the form of a script that is copied and then started on the target system. This is what this principle would look like:

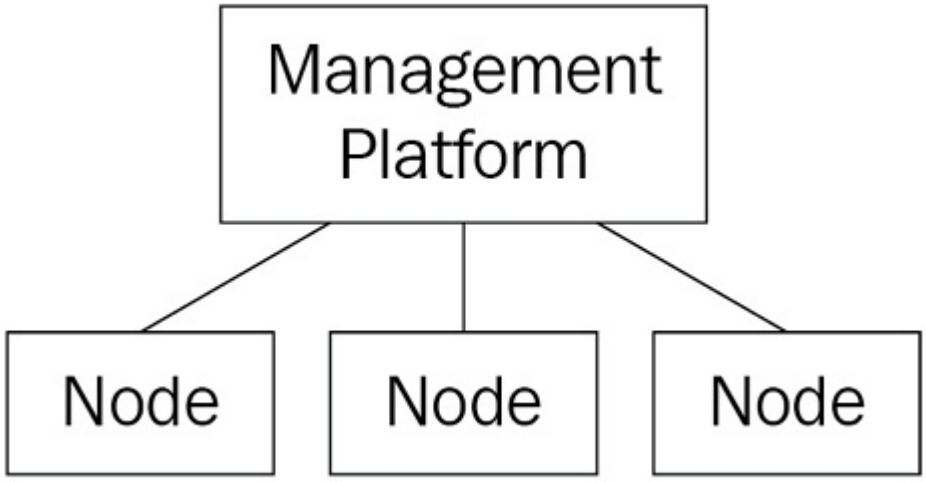


Figure 11.2 – The management platform doesn't need an agent to connect to objects that need orchestration and automation

Regardless of their type, these systems are usually called either automation or configuration management systems, and although these are two de facto standards yet completely different things, in reality, they are used indiscriminately. At the time of writing, two of the most popular are Puppet and Ansible, although there are others (Chef, SaltStack, and so on).

In this chapter, we will cover Ansible since it is easy to learn, agentless, and has a big pool of users on the internet.

Introduction to Ansible

Ansible is an IT automation engine – some call it an automation framework – that enables administrators to automate provisioning, configuration management, and many everyday tasks a system administrator may need to accomplish.

The easiest (and way too simplified) way of thinking about Ansible is that it is a complicated set of scripts that are intended to accomplish administration tasks on a large scale, both in terms of complexity and the sheer number of systems it can control. Ansible runs on a simple server that has all the parts of the Ansible system installed. It requires nothing to be installed on the machines it controls. It is safe to say that Ansible is completely agentless and that in order to accomplish its goal, it uses different ways to connect to remote systems and push small scripts to them.

This also means that Ansible has no way of detecting changes on the systems it controls; it is completely up to the configuration script we cre-

ate to control what happens if something is not as we expect it to be.

There are a couple of things that we need to define before doing everything else – things that we can think of as *building blocks* or modules. Ansible likes to call itself a radically simple IT engine, and it only has a couple of these building blocks that enable it to work.

First, it has **inventories** – lists of hosts that define what hosts a certain task will be performed on. Hosts are defined in a simple text file and can be as simple as a straight list that contains one host per line, or as complicated as a dynamic inventory that is created as Ansible is performing a task. We will cover these in more detail as we show how they are used. The thing to remember is that hosts are defined in text files as there are no databases involved (although there can be) and that hosts can be grouped, a feature that you will use extensively.

Secondly, there's a concept called *play*, which we will define as a set of different tasks run by Ansible on target hosts. We usually use a playbook to start a play, which is another type of object in the Ansible hierarchy.

In terms of playbooks, think of them as a policy or a set of tasks/plays that are required to do something or achieve a certain state on a particular system. Playbooks are also text files and are specifically designed to be readable by humans and are created by humans. Playbooks are used to define a configuration or, to be more precise, declare it. They can contain steps that start different tasks in an ordered manner. These steps are called plays, hence the name playbook. The Ansible documentation is helpful in explaining this as thinking about plays in sports where list of tasks that may be performed are provided and need to be documented, but at the same time may not be called. The important thing to understand here is that our playbooks can have decision-making logic inside them.

The fourth big part of the Ansible puzzle are its **modules**. Think of modules as small programs that are executed on the machines you are trying to control in order to accomplish something. There are literally hundreds of modules included with the Ansible package, and they can be used individually or inside your playbooks.

Modules allow us to accomplish tasks, and some of them are strictly declarative. Others return data, either as the results of the tasks the modules did, or explicit data that the module got from a running system through a process called fact gathering. This process is based on a module called **`gather_facts`**. Gathering correct facts about the system is one of the most important things we can do once we've started to develop our own playbooks.

The following architecture shows all of these parts working together:

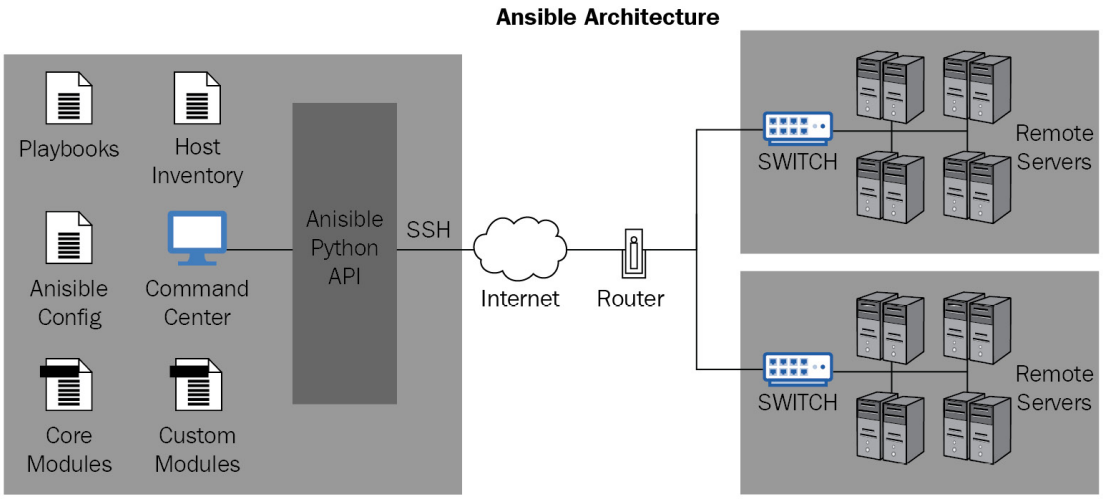


Figure 11.3 – Ansible architecture – Python API and SSH connections

The general consensus among the people working in IT is that management via Ansible is easier to do than via other tools as it doesn't require you to waste days on setup or on playbook development. Make no mistake, however: you will have to learn your way around YAML syntax to use Ansible extensively. That being said, if you're interested in a more GUI-based approach, you can always consider buying Red Hat Ansible Tower.

Ansible Tower is a GUI-based utility that you can use to manage your Ansible-based environments. This started as a project called **AWX**,

which is still very much alive today. But there are some key differences in the way in which AWX gets released versus how Ansible Tower gets released. The main one is the fact that Ansible Tower uses specific release versions while AWX takes a more *what OpenStack used to be* approach – a project that's moving forward rather quickly and has new releases very often.

As Red Hat clearly states on **https://www.ansible.com/products/awx-project/faq**, that:

"Ansible Tower is produced by taking selected releases of AWX, hardening them for long-term supportability, and making them available to customers as Ansible Tower offerings."

Basically, AWX is a community-supported project, while Red Hat directly supports Ansible Tower. Here's a screenshot from **Ansible AWX** so that you can see what the GUI looks like:

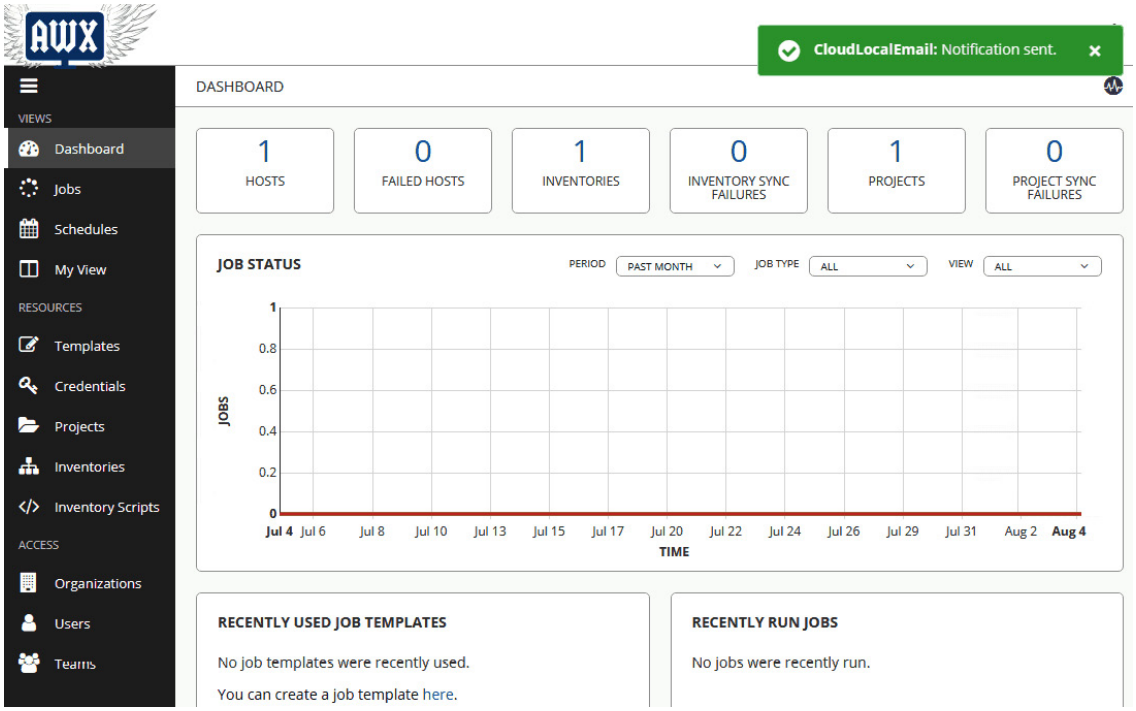


Figure 11.4 – Ansible AWX GUI for Ansible

There are other GUIs available for Ansible, such as **Rundeck**, **Semaphore**, and more. But somehow, AWX seems like the most logical choice for users who don't way to pay additional money for Ansible Tower. Let's spend a bit of time working on AWX before moving on to regular Ansible deployment and usage.

Deploying and using AWX

AWX was announced as an open source project that offers developers access to the Ansible Tower, without need for a license. As with almost

all other Red Hat projects, this one also aims to bridge the gap between a paid product that is production hardened and ready for corporate use, and a community-driven project that has almost all the required functionality, but on a smaller scale and without all the bells and whistles available to corporate customers. But this does not mean that AWX is in any way a *small* project. It builds up the functionality of Ansible and enables a simple GUI that helps you run everything inside your Ansible deployments.

We don't nearly have enough space here to demonstrate how it looks and what it can be used for, so we are just going to go through the basics of installing it and deploying the simplest scenario.

The single-most important address we need to know about when we are talking about AWX is **https://github.com/ansible/awx**. This is the place where the project resides. The most up-to-date information is here, in **`readme.md`**, a file that is shown on the GitHub page. If you are unfamil-

iar with *cloning* from GitHub, do not worry – we are basically just copying from a special source that will enable you to copy only the things that have changed since you last got your version of the files. This means that in order to update to a new version, you only need to clone once more using the same exact command.

On the GitHub page, there is a direct link to the install instructions we are going to follow. Remember, this deployment is from scratch, so we will need to build up our demo machine once again and install everything that is missing.

The first thing we need to do is get the necessary AWX files. Let's clone the GitHub repository to our local disk:

```
[root@awxdemo ~]# git clone -b 13.0.0 https://github.com/ansible/awx.git
Cloning into 'awx'...
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 246524 (delta 0), reused 0 (delta 0), pack-reused 246523
Receiving objects: 100% (246524/246524), 228.71 MiB | 4.23 MiB/s, done.
Resolving deltas: 100% (190421/190421), done.
Note: checking out '69589821ce2fd49c8db8b60bf34ff6b4b0df683a'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

[root@awxdemo ~]#
```

Figure 11.5 – Git cloning the AWX files

Note that we are using 13.0.0 as the version number as this is the current version of AWX at the time of writing.

Then, we need to sort out some dependencies. AWX obviously needs Ansible, Python, and Git, but other than that, we need to be able to support Docker, and we need GNU Make to be able to prepare some files later. We also need an environment to run our VMs. In this tutorial, we opted for Docker, so we will be using Docker Compose.

Also, this is a good place to mention that we need at least 4 GB of RAM and 20 GB of space on our machine in order to run AWX. This differs to the low footprint that we are used to using with Ansible, but this makes sense since AWX is much more than just a bunch of scripts. Let's start by installing the prerequisites.

Docker is the first one we will install. We are using CentOS 8 for this, so Docker is no longer part

of the default set of packages. Therefore, we need to add the repository and then install the Docker engine. We are going to use the **`-ce`** package, which stands for Community Edition. We will also use the **`--nobest`** option to install Docker – without this option, CentOS will report that we are missing some dependencies:

```
[root@awxdemo ~]# dnf config-manager --add-repo=https://download.docker.com/linu
x/centos/docker-ce.repo
Adding repo from: https://download.docker.com/linux/centos/docker-ce.repo
```

Figure 11.6 – Deploying docker-ce package on CentOS 8

After that, we need to run the following command:

```
dnf install docker-ce -y --nobest
```

The overall result should look something like this. Note that the versions of every package on your particular installations will probably be different. This is normal as packages change all the time:

```
Installed:
  containerd.io-1.2.0-3.el7.x86_64          docker-ce-3:18.09.1-3.el7.x86_64
  docker-ce-cli-1:19.03.12-3.el7.x86_64     libcgroup-0.41-19.el8.x86_64

Skipped:
  docker-ce-3:19.03.12-3.el7.x86_64

Complete!
[root@awxdemo ~]# systemctl start docker
[root@awxdemo ~]# systemctl enable docker
Created symlink /etc/systemd/system/multi-user.target.wants/docker.service → /us
r/lib/systemd/system/docker.service.
[root@awxdemo ~]# docker --version
Docker version 19.03.12, build 48a66213fe
```

Figure 11.7 – Starting and enabling the Docker service

Then, we will install Ansible itself using the following command:

```
dnf install ansible
```

If you are running on a completely clean CentOS 8 installation, you might have to install **`epel-release`** before Ansible is available.

Next on our list is Python. Just using the **`dnf`** command is not going to get Python installed as we're going to have to supply the Python version we want. For this, we would do something like this:

```
[root@awxdemo ~]# dnf install python
Last metadata expiration check: 0:14:53 ago on Wed 15 Jul 2020 05:07:09 PM EDT.
No match for argument: python
There are following alternatives for "python": python2, python36, python38
Error: Unable to find a match: python
[root@awxdemo ~]# dnf install python38
```

Figure 11.8 – Installing Python; in this case, version 3.8

After that, we will use pip to install the Docker component for Python. Simply type **`pip3 install docker`** and everything you need will be installed.

We also need to install the **`make`** package:

```
[root@awxdemo ~]# dnf install make
Last metadata expiration check: 0:17:07 ago on Wed 15 Jul 2020 05:07:09 PM EDT.
Dependencies resolved.
================================================================================
 Package       Architecture    Version                   Repository       Size
================================================================================
Installing:
 make          x86_64          1:4.2.1-10.el8            BaseOS          498 k

Transaction Summary
================================================================================
Install  1 Package
```

Figure 11.9 – Deploying GNU Make

Now, it's time for the Docker Compose part. We need to run the **`pip3 install docker-compose`** command to install the Python part and the following command to install docker-compose:

```
curl -L
https://github.com/docker/compose/rel
eases/download/1.25.0/docker-compose-
```

```
  `uname -s`-`uname -m` -o
  /usr/local/bin/docker-compose
```

This command will get the necessary install file from GitHub and use the necessary input parameters (by executing **uname** commands) to start the installation process for docker-compose.

We know this is a lot of dependencies, but AWX is a pretty complex system under the hood. On the surface, however, things are not so complicated. Before we do the final install part, we need to verify that our firewall has stopped and that it is disabled. We are creating a demo environment, and **firewalld** will block communication between containers. We can fix that later, once we have the system running.

Once we have everything running, installing AWX is simple. Just go to the **awx/installer** directory and run the following:

```
  ansible-playbook -i inventory -e
  docker_registry_password=password
  install.yml
```

The installation should take a couple of minutes. The result should be a long listing that ends with the following:

```
PLAY RECAP
**********************************
**********************************
localhost  :
ok=16   changed=8    unreachable=0
 failed=0    skipped=86   rescued=0
  ignored=0
```

This means that the local AWX environment has been deployed successfully.

Now, the fun part starts. AWX is comprised of four small Docker images. For it to work, all of them need to be configured and running. You can check them out by using **`docker ps`** and **`docker logs -t awx_task`**.

The first command lists all the images that got deployed, as well as their status:

```
[root@awxdemo installer]# docker ps
CONTAINER ID        IMAGE                COMMAND                  CREATED
      STATUS              PORTS                    NAMES
ba1dabec8a2a        ansible/awx:13.0.0   "tini -- /usr/bin/la…"   4 minutes ago
      Up 4 minutes        8052/tcp                 awx_task
a5ba16f3529f        ansible/awx:13.0.0   "tini -- /bin/sh -c …"   4 minutes ago
      Up 4 minutes        0.0.0.0:80->8052/tcp     awx_web
8d5ae3c600f8        redis                "docker-entrypoint.s…"   4 minutes ago
      Up 4 minutes        6379/tcp                 awx_redis
09ed8671d88a        postgres:10          "docker-entrypoint.s…"   4 minutes ago
      Up 4 minutes        5432/tcp                 awx_postgres
```

Figure 11.10 – Checking the pulled and started docker images

The second command shows us all the logs that the **awx_task** machine is creating. These are the main logs for the whole system. After a while, the initial configuration will complete:

```
[root@awxdemo installer]# docker logs -f awx_task
Using /etc/ansible/ansible.cfg as config file
127.0.0.1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "elapsed": 0,
    "match_groupdict": {},
    "match_groups": [],
    "path": null,
    "port": 5432,
    "search_regex": null,
    "state": "started"
}
Using /etc/ansible/ansible.cfg as config file
127.0.0.1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
```

Figure 11.11 – Checking the awx_task logs

There will be a lot of logging going on, and you will have to interrupt this command by using *Ctrl + C*.

After this whole process, we can point our web browser to **`http://localhost`**. We should be greeted by a screen that looks like this:

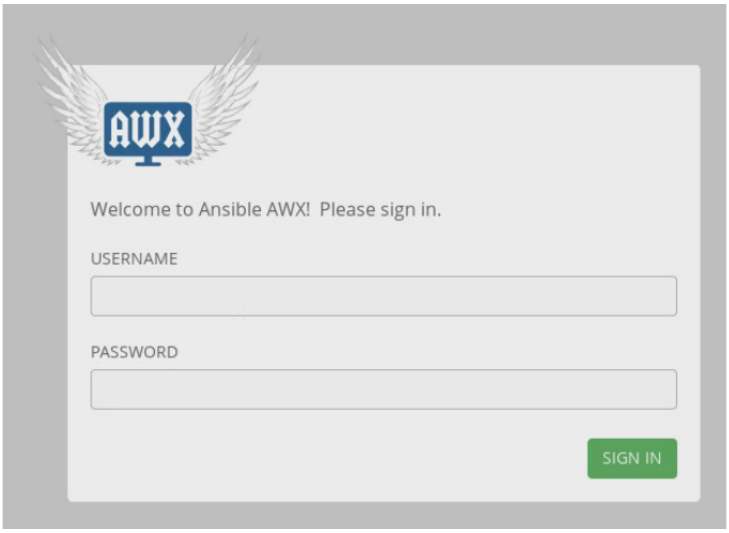


Figure 11.12 – AWX default login screen

The default username is **`admin`**, while the password is **`password`**. After logging in successfully, we should be faced with the following UI:

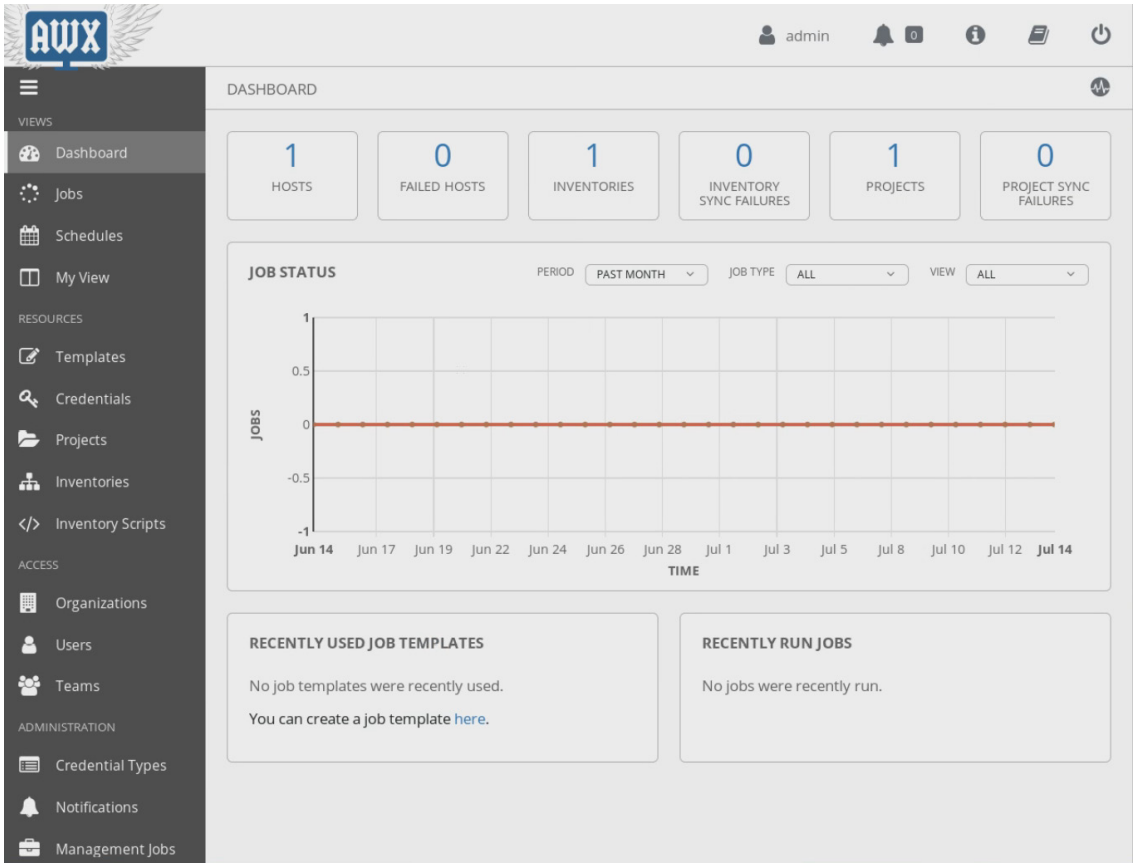


Figure 11.13 – Initial AWX dashboard after logging in

There is a lot to learn here, so we are just going to go through the basics. Basically, what AWX represents is a smart GUI for Ansible. We can see this quickly if we open **Templates** (on the left-hand side of the window) and take a look at the **Demo** template:

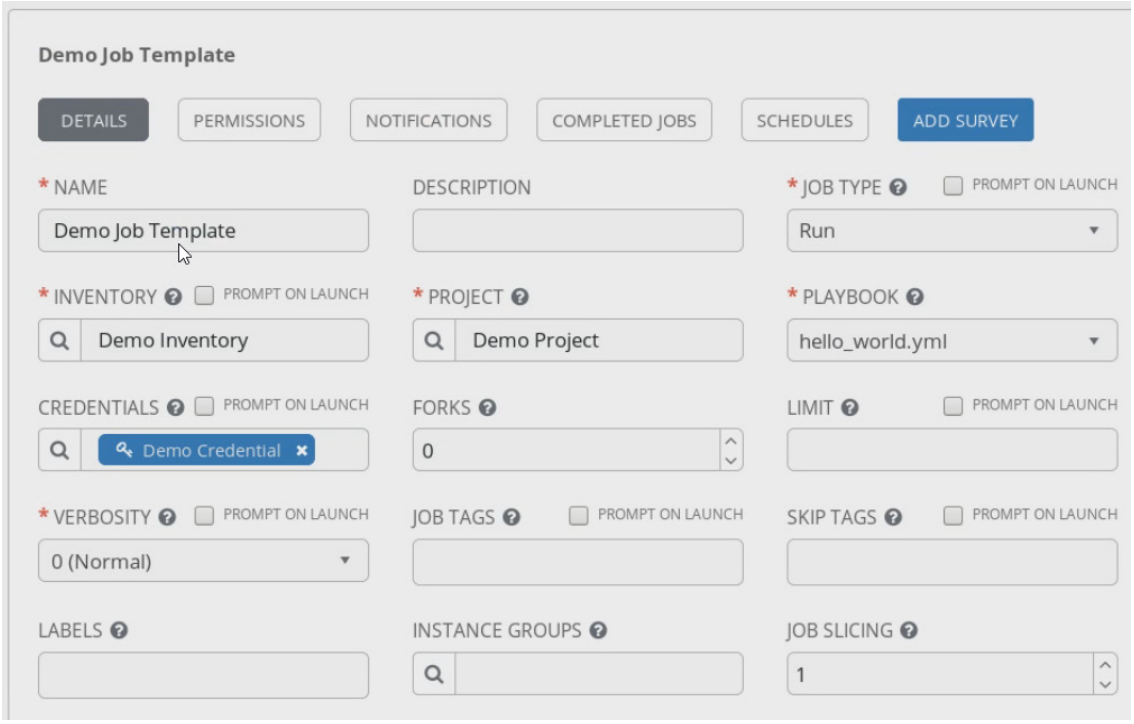


Figure 11.14 – Using a demo template in AWX

What we can see here will become much more familiar to us in the next part of this chapter, when we deploy Ansible. All these attributes are different parts of an Ansible playbook, including the playbook itself, the inventory, the credentials used, and a couple of other things that make using Ansible easier. If we scroll down a bit, there should be three buttons there. Press the **LAUNCH** button. This will play the template and turn it into a **`job`**:

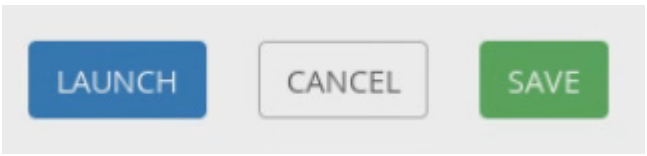


Figure 11.15 – By clicking on the Launch button, we can start our template job

The idea is that we can create templates and run them at will. Once you've run them, the results of the runs will end up under **Jobs** (find it as the second item on the left-hand side of the window):

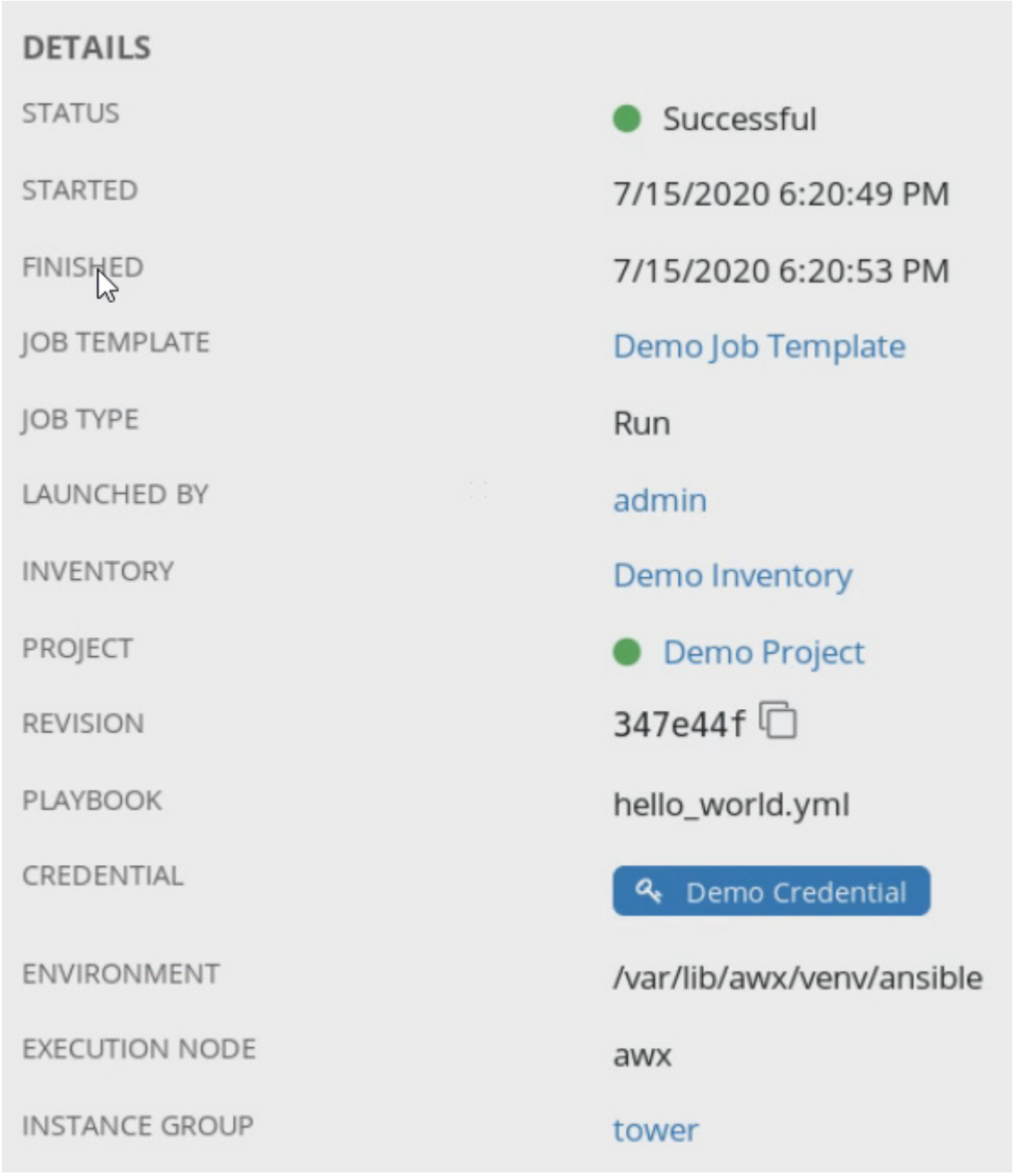


Figure 11.16 – Template job details

The details of the job are basically a summary of what happened, when, and which Ansible elements were used. We can also see the actual result of the playbook we just ran:

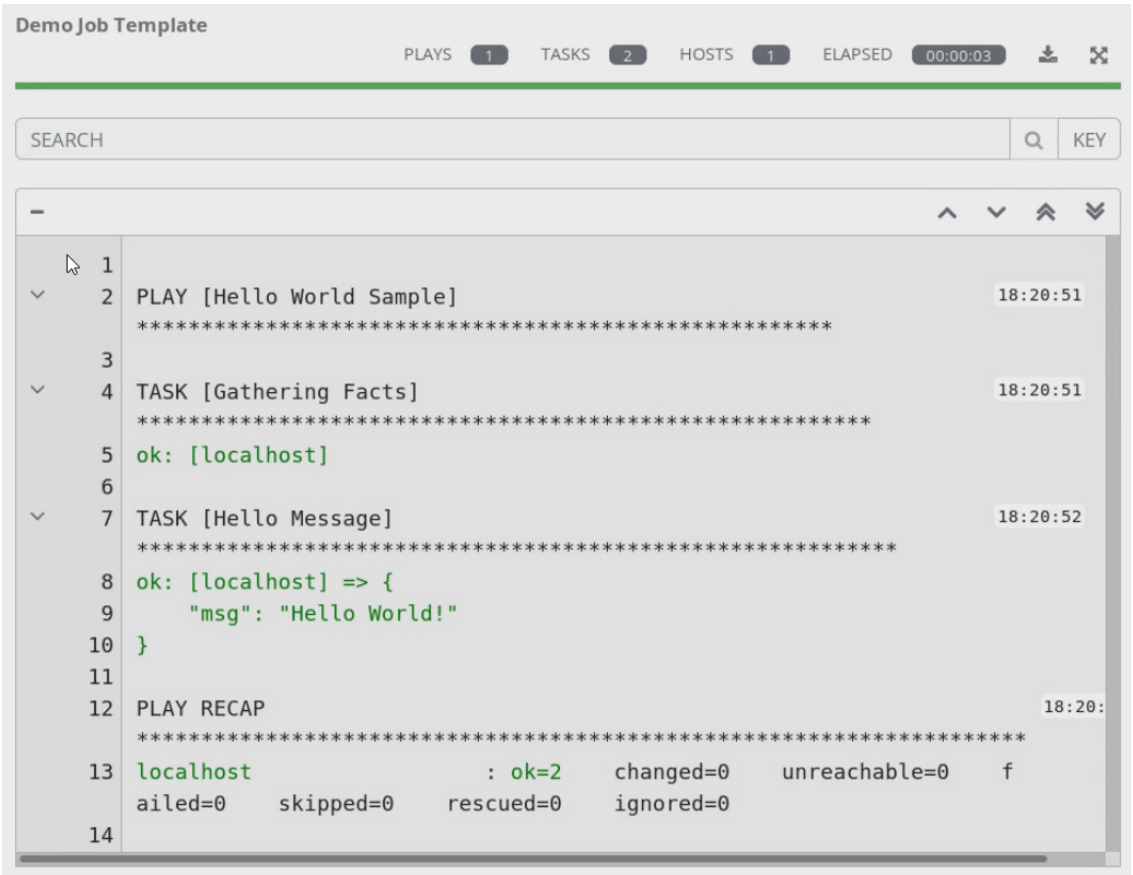


Figure 11.17 – Checking the demo job template's text output

What AWX really does is automate the automation. It enables you to be much more efficient while using Ansible simply because it offers a much more intuitive interface to the different files Ansible uses. It also gives you the ability to track what has been done and when, as well as what the results were. All of this is possible using the Ansible CLI, but AWX saves us a lot of effort while we're keeping control of the whole process.

Of course, because the goal of this chapter is to use Ansible, this means that we need to deploy all of the necessary software packages so that we can use it. Therefore, let's move on to the next phase in our Ansible process and deploy Ansible.

Deploying Ansible

Out of all the similar applications designed for orchestration and systems management, Ansible is probably the simplest one to install. Since it requires no agents on the systems it manages, installation is limited to only one machine – the one that will run all the scripts and playbooks. By default, Ansible uses SSH to connect to machines, so the only prerequisite for its use is that our remote systems have an SSH server up and running.

Other than that, there are no databases (Ansible uses text files), no daemons (Ansible runs on demand), and no management of Ansible itself to speak of. Since nothing is running in the background, Ansible is easily upgraded – the only

thing that can change is the way playbooks are structured, and that can easily be fixed. Ansible is based on the Python programming language, but its structure is simpler than that of a standard Python program. Configuration files and playbooks are either simple text files or YAML formatted text files, with YAML being a file format used to define data structures. Learning YAML is outside the scope of this chapter, so we will just presume that you understand simple data structures. The YAML files we'll be using as examples are simple enough to warrant almost no explanation, but if one is needed, it will be provided.

The installation can be as simple as running the following:

```
  yum install ansible
```

You can run this command as the root user or use the following command:

```
  apt install ansible
```

The choice depends on your distribution (Red Hat/CentOS or Ubuntu/Debian). More information can be found on the Ansible website at **https://docs.ansible.com/**.

RHEL8 users will have to enable the repo containing Ansible RPMs first. At the time of writing, this can be accomplished by running the following:

```
sudo subscription-manager repos --
enable ansible-2.8-for-rhel-8-x86_64-
rpms
```

After running the preceding command, use the following code:

```
dnf install ansible
```

This is all it takes to install Ansible.

One thing that can surprise you is the size of the installation: it really is that small (around 20 MB) and will install Python dependencies as needed.

The machine that Ansible is installed in is also called the *control node*. It must be installed on a

Linux host as Windows is not supported in this role. Ansible control nodes can be run inside virtual machines.

Machines that we control are called managed nodes, and by default, they are Linux boxes controlled through the **`SSH`** protocol. There are modules and plugins that enable extending this to Windows and macOS operating systems, as well as other communication channels. When you start reading the Ansible documentation, you will notice that most of the modules that support more than one architecture have clear instructions regarding how to accomplish the same tasks on different operating systems.

We can configure Ansible's settings using **`/etc/ansible/ansible`**. This file contains parameters that define the defaults, and by itself contains a lot of lines that are commented out but contain default values for all the things Ansible uses to work. Unless we change something, these are the values that Ansible is going to use to run. Let's use Ansible in a practical sense to see how

all of this fits together. In our scenario, we are going to use Ansible to provision a virtual machine by using its built-in module.

Provisioning a virtual machine using the kvm_libvirt module

One thing that you may or may not include is a setting that defines how SSH is used to connect to machines Ansible is going to configure. Before we do that, we need to spend a bit of time talking about security and Ansible. Like almost all things related to Linux (or **`*nix`** in general), Ansible is not an integrated system, instead relying on different services that already exist. To connect to systems it manages and to execute commands, Ansible relies on **`SSH`** (in Linux) or other systems such as **WinRM** or **PowerShell** on Windows. We are going to focus on Linux here, but remember that quite a bit of information about Ansible is completely system-independent.

SSH is a simple but extremely robust protocol that allows us to transfer data (Secure FTP, SFTP, and so on) and execute commands (**SSH**) on remote hosts through a secure channel. Ansible uses SSH directly by connecting and then executing commands and transferring files. This, of course, means that in order for Ansible to work, it is crucial that SSH works.

There are a couple of things that you need to remember when using **SSH** to connect:

- The first is a key fingerprint, as seen from the Ansible control node (server). When establishing a connection for the first time, **SSH** requires the user to verify and accept keys that the remote system presents. This is designed to prevent MITM attacks and is a good tactic in everyday use. But if we are in the position of having to configure freshly installed systems, *all* of them will require for us to accept their keys. This is time-consuming and complicated to do once we start using playbooks, so the first playbook you will start is probably going to disable

key checks and logging into machines. Of course, this should only be used in a controlled environment since this lowers the security of the whole Ansible system.

- The second thing you need to know is that Ansible runs as a normal user. Having said that, maybe we do not want to connect to the remote systems as the current user. Ansible solves that by having a variable that can be set on individual computers or groups that indicates what username the system is going to use to connect to this particular computer. After connecting, Ansible allows us to execute commands on the remote system as a different user entirely. This is something that is commonly used since it enables us to reconfigure the machine completely and change users as if we were at the console.
- The third thing that we need to remember are the keys – **SSH** can log in by using interactive authentication, meaning via password or by using pre-shared keys that are exchanged once and then reused to establish the SSH session.

There is also **ssh-agent**, which can be used to authenticate sessions.

Although we can use fixed passwords inside inventory files (or special key vaults), this is a bad idea. Luckily, Ansible enables us to script a lot of things, including copying keys to remote systems. This means that we are going to have some playbooks that are going to automate deployment of new systems, and these will enable us to take control of them for further configuration.

To sum this up, the Ansible steps for deploying a system will probably start like this:

1. Install the core system and make sure that **SSHD** is running.
2. Define a user that has admin rights on the system.
3. From the control node, run a playlist that will establish the initial connection and copy the local **SSH** key to a remote location.
4. Use the appropriate playbooks to reconfigure the system securely, and without the need to

store passwords locally.

Now, let's dig deeper.

Every reasonable manager will tell you that in order to do anything, you need to define the scope of the problem. In automation, this means defining systems that Ansible is going to work on. This is done through an inventory file, located in **`/etc/Ansible`**, called **`hosts`**.

`Hosts` can be grouped or individually named. In text format, that can look like this:

```
[servers]
srv1.local
srv2.local
srv3.local
[workstations]
wrk1.local
wrk2.local
wrk3.local
```

Computers can be part of multiple groups simultaneously, and groups can be nested.

The format we used here is straight text. Let's re-write this in YAML:

```
All:
  Servers:
     Hosts:
Srv1.local:
Srv2.local:
Srv3.local:
Workstations:
     Hosts:
Wrk1.local:
Wrk2.local:
Wrk3.local:
Production:
   Hosts:
Srv1.local:
Workstations:
```

Important Note

We created another group called Production that contains all the workstations and one server.

Anything that is not part of the default or standard configuration can be included individually in the host definition or in the group definition as variables. Every Ansible command has some way of giving you flexibility in terms of partially or completely overriding all the items in the configuration or inventory.

The inventory supports ranges in host definitions. Our previous example can be written as follows:

```
[servers]
Srv[1:3].local
[workstations]
Wrk[1:3].local
```

This also works for characters, so if we need to define servers named **srva**, **srvb**, **srvc**, and **srvd**, we can do that by stating the following:

```
srv[a:d]
```

IP ranges can also be used. So, for instance, **10.0.0.0/24** would be written down as follows:

```
10.0.0.[1:254]
```

There are two predefined default groups that can also be used: **`all`** and **`ungrouped`**. As their names suggest, if we reference **`all`** in a playbook, it will be run on every server we have in our inventory. **`Ungrouped`** will reference only those systems that are not part of any group.

Ungrouped references are especially useful when setting up new computers – if they are not in any group, we can consider them *new* and set them up to be joined to a specific group.

These groups are defined implicitly and there is no need to reconfigure them or even mention them in the inventory file.

We mentioned that the inventory file can contain variables. Variables are useful when we need to have a property that is defined inside a group of computers, a user, password, or a setting specific to that group. Let's say that we want to define a user that is going to be using on the **`servers`** group:

1. First, we define a group:

```
[servers]
srv[1:3].local
```

2. Then, we define the variables that are going to be used for the whole group:

```
[servers:vars]
ansible_user=Ansibleuser
ansible_connection=ssh
```

This will use the user named **`Ansibleuser`** to connect using **`SSH`** when asked to perform a playbook.

Important Note

Note that the password is not present and that this playbook will fail if either the password is not separately mentioned or the keys are not exchanged beforehand. For more on variables and their use, consult Ansible documentation.

Now that we've created our first practical Ansible task, it's time to talk about how to make Ansible do many things at once while using a more *objective* approach. It's important to be

able to create a single task or a couple of tasks that we can combine through a concept called a *playbook*, which can include multiple tasks/plays.

Working with playbooks

Once we've decided how to connect to the machines we plan to administer, and once we have created the inventory, we can start actually using Ansible to do something useful. This is where playbooks start to make sense.

In our examples, we've configured four CentOS7 systems, gave them consecutive addresses in the range of **`10.0.0.1`** to **`10.0.0.4`**, and used them for everything.

Ansible is installed on the system with the IP address **`10.0.0.1`**, but as we already said, this is completely arbitrary. Ansible has a minimal footprint on the system that is used as a control node and can be installed on any system as long as it has connectivity to the rest of the network we are going to manage. We simply chose the first

computer in our small network. One more thing to note is that the control node can be controlled by itself through Ansible. This is useful, but at the same time not a good thing to do. Depending on your setup, you will want to test not only playbooks, but individual commands before they are deployed to other machines – doing that on your control server is not a wise thing to do.

Now that Ansible is installed, we can try and do something with it. There are two distinct ways that Ansible can be run. One is by running a playbook, a file that contains tasks that are to be performed. The other way is by using a single task, sometimes called **ad hoc** execution. There are reasons to use Ansible either way – playbooks are our main tool, and you will probably use them most of the time. But ad hoc execution also has its advantages, especially if we are interested in doing something that we need done once, but across multiple servers. A typical example is using a simple command to check the version of an installed application or application

state. If we need it to check something, we are not going to write a playbook.

To see if everything works, we are going to start by simply using ping to check if the machines are online.

Ansible likes to call itself *radically simple automation*, and the first thing we are going to do proves that.

We are going to use a module named ping that tries to connect to a host, verifies that it can run on local Python environment, and returns a message if everything is ok. Do not confuse this module with the **`ping`** command in Linux; we are not pinging through a network; we are only *pinging* from the control node to the server we are trying to control. We will use a simple **`ansible`** command to ping all the defined hosts by issuing the following command:

```
  ansible all -m ping
```

The following is the result of running the preceding command:

```
[root@vm0-101 ~]# ansible all -m ping
10.0.0.1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
10.0.0.4 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
10.0.0.2 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
10.0.0.3 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
```

Figure 11.18 – Our first Ansible module – ping, checks for Python and reports its state

What we did here is run a single command called **`ansible all -m ping`**.

`ansible` is the simplest command available and runs a single task. The **`all`** parameter means run it on all the hosts in the inventory, and **`-m`** is used to call a module that will be run.

This particular module has no parameters or options, so we just need to run it in order to get a result. The result itself is interesting; it is in YAML format and contains a few things other than just the result of the command.

If we take a closer look at this, we will see that Ansible returned one result for each host in the inventory. The first thing we can see is the final result of the command – **`SUCCESS`** means that the task itself ran without a problem. After that, we can see data in form of an array – **`ansible_facts`** contains information that the module returns, and it is used extensively when writing playbooks. Data that is returned this way can vary. In the next section, we will show a much bigger dataset, but in this particular case, the only thing that is shown is the location of the Python interpreter. After that, we have the **`changed`** variable, which is an interesting one.

When Ansible runs, it tries to detect whether it ran correctly and whether it has changed the system state. In this particular task, the com-

mand that ran is just informative and does not change anything on the system, so the system state was unchanged.

In other words, this means that whatever was run did not install or change anything on the system. States will make more sense later when we need to check if something was installed or not, such as a service.

The last variable we can see is the return of the **`ping`** command. It simply states **pong** since this is the correct answer that the module gives if everything was set up correctly.

Let's do something similar, but this time with an argument, such as an ad hoc command that we want to be executed on remote hosts. So, type in the following command:

```
ansible all -m shell -a "hostname"
```

The following is the output:

```
[root@vm0-101 ~]# ansible all -m shell -a "hostname"
10.0.0.3 | CHANGED | rc=0 >>
vm0-104.vua.cloud

10.0.0.1 | CHANGED | rc=0 >>
vm0-101.vua.cloud

10.0.0.4 | CHANGED | rc=0 >>
vm0-103.vua.cloud

10.0.0.2 | CHANGED | rc=0 >>
vm0-102.vua.cloud
```

Figure 11.19 – Using Ansible to explicitly execute a specific command on Ansible targets

Here, we called another module called **`shell`**. It simply runs whatever is given as a parameter as a shell command. What is returned is the local hostname. This is functionally the same as what would happen if we connected to each host in our inventory using **`SSH`**, executed the command, and then logged out.

For a simple demonstration of what Ansible can do, this is OK, but let's do something more complex. We are going to use a module called **`yum`** that is specific to CentOS/Red Hat to check if there is a web server installed on our hosts. The web server we are going to check for is going to

be **`lighttpd`** since we want something lightweight.

When we talked about states, we touched on a concept that is both a little confusing at first and extremely useful once we start using it. When calling a command like this, we are declaring a desired state, so the system itself will change if the state is not the one we are demanding. This means that, in this example, we are not actually testing if **`lighttpd`** is installed – we are telling Ansible to check it and that if it's not installed to install it. Even this is not completely true – the module takes two arguments: the name of the service and the state it should be in. If the state on the system we are checking is the same as the state we sent when invoking the module, we are going to get **`changed: false`** since nothing changed. But if the state of the system is not the same, Ansible will make the current state of the system the same as the state we requested.

To prove this, we are going to see if the service is *not* installed or *absent* in Ansible terms.

Remember that if the service was installed, this will uninstall it. Type in the following command:

```
ansible all -m yum -a "name=lighttpd state=absent"
```

This is what you should get as the result of running the preceding command:

```
[root@vm0-101 ~]# ansible all -m yum -a "name=lighttpd state=absent"
10.0.0.2 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "msg": "",
    "rc": 0,
    "results": [
        "lighttpd is not installed"
    ]
}
10.0.0.3 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "msg": "",
    "rc": 0,
    "results": [
        "lighttpd is not installed"
    ]
}
10.0.0.4 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "msg": "",
    "rc": 0,
    "results": [
        "lighttpd is not installed"
    ]
}
```

Figure 11.20 – Using Ansible to check the state of a service

Then, we can say that we want it present on the system. Ansible is going to install the services as needed:

```
[root@vm0-101 ~]# ansible all -m yum -a "name=lighttpd state=present"
10.0.0.4 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": true,
    "changes": {
        "installed": [
            "lighttpd"
        ]
    },
    "msg": "",
    "rc": 0,
    "results": [
        "Loaded plugins: fastestmirror, langpacks\nLoading mirror speeds from cached hostfile\n * base: centos.lonyai
.com\n * epel: mirror.niif.hu\n * extras: centos.mirror.ba\n * updates: centos.lonyai.com\nResolving Dependencies\n--
> Running transaction check\n---> Package lighttpd.x86_64 0:1.4.54-1.el7 will be installed\n--> Processing Dependency
: libfam.so.0()(64bit) for package: lighttpd-1.4.54-1.el7.x86_64\n--> Running transaction check\n---> Package gamin.x
86_64 0:0.1.10-16.el7 will be installed\n--> Finished Dependency Resolution\n\nDependencies Resolved\n\n============
================================================================\n Package           Arch            Version
           Repository     Size\n================================================================================\nIn
stalling:\n lighttpd         x86_64          1.4.54-1.el7            epel          438 k\nInstalling for dependenci
es:\n gamin             x86_64          0.1.10-16.el7           base          128 k\n\nTransaction Summary\n========
================================================================================\nInstall  1 Package (+1 Dependent package)\n
\nTotal download size: 567 k\nInstalled size: 1.6 M\nDownloading packages:\n----------------------------------------
----------------------------------------\nTotal                                                1.5 MB/s | 567 kB  00:00
    \nRunning transaction check\nRunning transaction test\nTransaction test succeeded\nRunning transaction\n  Install
ing : gamin-0.1.10-16.el7.x86_64                                          1/2 \n  Installing : lighttpd-1.4.54-1.el7.x86_64
                              2/2 \n  Verifying  : lighttpd-1.4.54-1.el7.x86_64                                 1/2
 \n  Verifying  : gamin-0.1.10-16.el7.x86_64                                  2/2 \n\nInstalled:\n  lighttpd.x86_64
0:1.4.54-1.el7                                        \n\nDependency Installed:\n  gamin.x86_64 0:0.1.10-16.e
l7                                       \n\nComplete!\n"
    ]
}
```

Activate Windows

Figure 11.21 – Using the yum install command on all Ansible targets

Here, we can see that Ansible simply checked and installed the service since it wasn't there. It also provided us with other useful information, such as what changes were done on the system and the output of the command it performed. Information was provided as an array of variables; this usually means that we will have to do some string manipulation in order to make it look nicer.

Now, let's run the command again:

```
ansible all -m yum -a "name=lighttpd
state=absent"
```

This should be the result:

```
[root@vm0-101 ~]# ansible all -m yum -a "name=lighttpd state=present"
10.0.0.3 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "msg": "",
    "rc": 0,
    "results": [
        "lighttpd-1.4.54-1.el7.x86_64 providing lighttpd is already installed"
    ]
}
10.0.0.2 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "msg": "",
    "rc": 0,
    "results": [
        "lighttpd-1.4.54-1.el7.x86_64 providing lighttpd is already installed"
    ]
}
10.0.0.1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "msg": "",
    "rc": 0,
```

Figure 11.22 – Using Ansible to check the service state after service installation

As we can see, there were no changes here since the service is installed.

These were all just starting examples so that we could get to know Ansible a little bit. Now, let's

expand on this and create an Ansible playbook that's going to install KVM on our predefined set of hosts.

Installing KVM

Now, let's create our first playbook and use it to install KVM on all of our hosts. For our playbook, we used an excellent example from the GitHub repository, created by Jared Bloomer, that we changed a bit since we already have our options and inventory configured. The original files are available at **https://github.com/jbloomer/Ansible---Install-KVM-on-CentOS-7.git**.

This playbook will show everything that we need to know about automating simple tasks. We chose this particular example because it shows not only how automation works, but also how to create separate tasks and reuse them in different playbooks. Using a public repository has an added benefit that you will always get the latest

version, but it may differ significantly than the one presented here:

1. First, we created our main playbook – the one that will get called – and named it **`installkvm.yaml`**:

```
---
- name: Install KVM
  hosts: all
  remote_user: root

  roles:
    - checkVirtualization
    - installKVM
```

Figure 11.23 – The main Ansible playbook, which checks for virtualization support and installs KVM

As we can see, this is simple declaration, so let's analyze it line by line. First, we have the playbook name, a string that can contain whatever we want:

The **`hosts`** variable defines what part of the inventory this playbook is going to be performed on – in our case, all the hosts. We can override this (and all the other variables) at runtime, but it helps to limit the playbook to just the hosts we need to control. In our particular

case, this is actually all the hosts in our inventory, but in production, we will probably have more than one group of hosts.

The next variable is the name of the user that is going to perform the task. What we did here is not recommended in production since we are using a superuser account to perform tasks. Ansible is completely capable of working with non-privileged accounts and elevating rights when needed, but as in all demonstrations, we are going to make mistakes so that you don't have to and all in order to make things easier to understand.

Now comes the part that is actually performing our tasks. In Ansible, we declare roles for the system. In our example, there are two of them. Roles are really just tasks to be performed, and that will result in a system that will be in a certain state. In our first role, we are going to check if the system supports virtualization, and then in the second one, we will install KVM services on all the systems that do.

2. When we downloaded the script from the GitHub, it created a few folders. In the one named **`roles`**, there are two subfolders that each contain a file; one is called **`checkVirtualization`** and the other is called **`installKVM`**. You can probably already see where this is heading. First, let's see what **`checkVirtualization`** contains:

```
---
- name: Check for CPU Virtualization
  shell: "lscpu | grep -i virtualization"
  register: result
  failed_when: "result.rc != 0"
```

Figure 11.24 – Checking for CPU virtualization via the lscpu command

This tasks simply calls a shell command and tries to **`grep`** for the lines containing virtualization parameters for the CPU. If it finds none, it fails.

3. Now, let's see the other task:

```
---
- name: Installing KVM Packages
  package:
    name: "{{ item }}"
    state: present
  with_items:
    - qemu-kvm
    - libvirt
    - libvirt-python
    - libguestfs-tools
    - virt-install

- name: Enable and Start libvirtd
  systemd:
    name: libvirtd
    state: started
    enabled: yes

- name: Verify KVM module is loaded
  shell: "lsmod | grep -i kvm"
  register: result
  failed_when: "result.rc != 0"
```

Figure 11.25 – Ansible task for installing the necessary libvirt packages

The first part is a simple loop that will just install five different packages if they are not present. We are using the package module here, which is a different approach than the one we used in our first demonstration regarding how to install packages. The module that we used earlier in this chapter is called **`yum`** and is specific to CentOS as a distribution. The **`package`** module is a generic module that will translate to whatever package manager a specific distribution is using. Once we've installed all the packages we need, we need to make sure that **`libvirtd`** is enabled and started.

We are using a simple loop to go through all the packages that we are installing. This is not necessary, but it is a better way to do things than copying and pasting individual commands since it makes the list of packages that we need much more readable.

Then, as the last part of the task, we verify if the KVM has loaded.

As we can see, the syntax for the playbook is a simple one. It is easily readable, even by somebody who has only minor knowledge of scripting or programming. We could even say that having a firm understanding of how the Linux command line works is more important.

4. In order to run a playbook, we use the **`ansible-playbook`** command, followed by the name of the playbook. In our case, we're going to use the **`ansible-playbook main.yaml`** command. Here are the results:

```
PLAY [Install KVM] *************************************************************

TASK [Gathering Facts] *********************************************************
ok: [10.0.0.3]
ok: [10.0.0.2]
ok: [10.0.0.4]
ok: [10.0.0.1]

TASK [checkVirtualization : Check for CPU Virtualization] **********************
changed: [10.0.0.4]
changed: [10.0.0.2]
changed: [10.0.0.1]
changed: [10.0.0.3]

TASK [installKVM : Installing KVM Packages] ************************************
ok: [10.0.0.2] => (item=qemu-kvm)
ok: [10.0.0.4] => (item=qemu-kvm)
ok: [10.0.0.3] => (item=qemu-kvm)
ok: [10.0.0.1] => (item=qemu-kvm)
changed: [10.0.0.1] => (item=libvirt)
changed: [10.0.0.2] => (item=libvirt)
changed: [10.0.0.4] => (item=libvirt)
changed: [10.0.0.3] => (item=libvirt)
changed: [10.0.0.1] => (item=libvirt-python)
changed: [10.0.0.4] => (item=libvirt-python)
changed: [10.0.0.3] => (item=libvirt-python)
changed: [10.0.0.2] => (item=libvirt-python)
changed: [10.0.0.2] => (item=libguestfs-tools)
changed: [10.0.0.4] => (item=libguestfs-tools)
changed: [10.0.0.1] => (item=libguestfs-tools)
changed: [10.0.0.3] => (item=libguestfs-tools)
changed: [10.0.0.2] => (item=virt-install)
changed: [10.0.0.3] => (item=virt-install)
changed: [10.0.0.4] => (item=virt-install)
changed: [10.0.0.1] => (item=virt-install)
```

Figure 11.26 – Interactive Ansible playbook monitoring

5. Here, we can see that Ansible breaks down everything it did on every host, change by change. The end result is a success:

```
TASK [installKVM : Enable and Start libvirtd] **********************************
changed: [10.0.0.3]
changed: [10.0.0.4]
changed: [10.0.0.2]
changed: [10.0.0.1]

TASK [installKVM : Verify KVM module is loaded] ********************************
changed: [10.0.0.1]
changed: [10.0.0.3]
changed: [10.0.0.4]
changed: [10.0.0.2]

PLAY RECAP *********************************************************************
10.0.0.1                   : ok=5    changed=4    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
10.0.0.2                   : ok=5    changed=4    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
10.0.0.3                   : ok=5    changed=4    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
10.0.0.4                   : ok=5    changed=4    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 11.27 – Ansible playbook report

Now, let's check if our freshly installed KVM *cluster* is working.

6. We are going to start **virsh** and list the active VMs on all the parts of the cluster:

```
[root@vmc-101 ~]# ansible all -m shell -a "virsh list --all"
10.0.0.2 | CHANGED | rc=0 >>
 Id    Name                           State
----------------------------------------------------

10.0.0.1 | CHANGED | rc=0 >>
 Id    Name                           State
----------------------------------------------------

10.0.0.4 | CHANGED | rc=0 >>
 Id    Name                           State
----------------------------------------------------

10.0.0.3 | CHANGED | rc=0 >>
 Id    Name                           State
----------------------------------------------------
```

Figure 11.28 – Using Ansible to check all the virtual machines on Ansible targets

Having finished this simple exercise, we have a running KVM on four machines and the ability to control them from one place. But we still have no VMs running on the hosts. Next, we are going to show you how to create a CentOS installation inside the KVM environment, but we are going to use the most basic method to do so – **virsh**.

We are going to do two things: first, we are going to download a minimal ISO image for CentOS from the internet. Then, we are going to call

virsh. This book will show you different ways to accomplish this task; downloading from the internet is one of the slowest:

1. As always, Ansible has a module dedicated to downloading files. The parameters it expects are the URL where the file is located and the location of the saved file:

```
---
    - name: download Centos core image
      hosts: all
      tasks:
       - name: download from official repository
         get_url:
            url: http://mirror.eu.oneandone.net/linux/distributions/centos/7.6.1810/isos/x86_64/CentOS-7-x86_64-Minimal-1810.iso
            dest: /var/lib/libvirt/boot/
```

Figure 11.29 – Downloading files in Ansible playbooks

2. After running the playbook, we need to check if the files have been downloaded:

```
[root@vm0-101 ~]# ansible all -m shell -a "ls -al /var/lib/libvirt/boot"
10.0.0.3 | CHANGED | rc=0 >>
total 940032
drwx--x--x.  2 root root        46 Sep  6 22:05 .
drwxr-xr-x. 10 root root       117 Sep  6 16:58 ..
-rw-r--r--.  1 root root 962592768 Sep  6 22:05 CentOS-7-x86_64-Minimal-1810.iso

10.0.0.2 | CHANGED | rc=0 >>
total 940032
drwx--x--x.  2 root root        46 Sep  6 22:06 .
drwxr-xr-x. 10 root root       117 Sep  6 16:58 ..
-rw-r--r--.  1 root root 962592768 Sep  6 22:06 CentOS-7-x86_64-Minimal-1810.iso

10.0.0.4 | CHANGED | rc=0 >>
total 940032
drwx--x--x.  2 root root        46 Sep  6 22:06 .
drwxr-xr-x. 10 root root       117 Sep  6 16:58 ..
-rw-r--r--.  1 root root 962592768 Sep  6 22:06 CentOS-7-x86_64-Minimal-1810.iso

10.0.0.1 | CHANGED | rc=0 >>
total 940032
drwx--x--x.  2 root root        46 Sep  6 22:06 .
drwxr-xr-x. 10 root root       117 Sep  6 16:58 ..
-rw-r--r--.  1 root root 962592768 Sep  6 22:06 CentOS-7-x86_64-Minimal-1810.iso
```

Figure 11.30 – Status check – checking if the files have been downloaded to our targets

3. Since we are not automating this and instead creating a single task, we are going to run it in a local shell. The command to run for this would be something like the following:

```
ansible all -m shell -a "virt-
install --name=COS7Core --ram=2048
--vcpus=4 --
cdrom=/var/lib/libvirt/boot/CentOS-
7-x86_64-Minimal-1810.iso --os-
type=linux --os-variant=rhel7 --
disk
path=/var/lib/libvirt/images/cos7vm
.dsk,size=6"
```

4. Without a kickstart file or some other kind of preconfiguration, this VM makes no sense since we will not be able to connect to it or even finish the installation. In the next task, we will remedy that using cloud-init.
   Now, we can check if everything worked:

```
[root@vm0-101 virt-manager]# ansible all -m shell -a "virsh list"
10.0.0.3 | CHANGED | rc=0 >>
 Id    Name                           State
----------------------------------------------------
 2     COS7Core                       running

10.0.0.4 | CHANGED | rc=0 >>
 Id    Name                           State
----------------------------------------------------
 2     COS7Core                       running

10.0.0.1 | CHANGED | rc=0 >>
 Id    Name                           State
----------------------------------------------------
 4     COS7Core                       running

10.0.0.2 | CHANGED | rc=0 >>
 Id    Name                           State
----------------------------------------------------
 2     COS7Core                       running
```

Figure 11.31 – Using Ansible to check if all our VMs are running

Here, we can see that all the KVMs are running and that each of them has its own virtual machine online and running.

Now, we are going to wipe our KVM cluster and start again, but this time with a different configuration: we are going to deploy the cloud version of CentOS and reconfigure it using cloud-init.

Using Ansible and cloud-init for automation and orchestration

Cloud-init is one of the more popular ways of machine deployment in private and hybrid cloud environments. This is because it enables machines to be quickly reconfigured in a way that enables just enough functionality to get them connected to an orchestration environment such as Ansible.

More details can be found at **cloud-init.io**, but in a nutshell, cloud-init is a tool that enables the creation of special files that can be combined with VM templates in order to rapidly deploy them. The main difference between cloud-init and unattended installation scripts is that cloud-init is more or less distribution-agnostic and much easier to change with scripting tools. This means less work during deployment, and less time from start of deployment until machines are online and working. On CentOS, this can be accomplished with kickstart files, but this not nearly as flexible as cloud-init.

Cloud-init works using two separate parts: one is the distribution file for the operating system we

are deploying. This is not the usual OS installation file, but a specially configured machine template intended to be used as a cloud-init image.

The other part of the system is the configuration file, which is *compiled*–or to be more precise, *packed* – from a special YAML text file that contains configuration for the machine. This configuration is small and ideal for network transmission.

These two parts are intended to be used as a whole to create multiple instances of identical virtual machines.

The way this works is simple:

1. First, we distribute a machine template that is completely identical for all the machines that we are going to create. This means having one master copy and creating all the instances out of it.
2. Then, we pair the template with a specially crafted file that is created using cloud-init. Our template, regardless of the OS it uses, is capa-

ble of understanding different directives that we can set in the cloud-init file and will be reconfigured. This can be repeated as needed.

Let's simplify this even more: if we need to create 100 servers that will have four different roles using the unattended installation files, we would have to boot 100 images and wait for them to go through all the installation steps one by one. Then, we would need to reconfigure them for the task we need. Using cloud-init, we are booting one image in 100 instances, but the system takes only a couple of seconds to boot since it is already installed. Only critical information is needed to put it online, after which we can take over and completely configure it using Ansible.

We are not going to dwell too much on cloud-init's configuration; everything we need is in this example:

```
#cloud-config
package_upgrade: true
users:
    - name: ansible
      groups: wheel
      lock_passwd: false
      passwd: F1AppspmE+Lz8lMLW2PK5ohcuogevH
      shell: /bin/bash
      sudo: ['ALL=(ALL) NOPASSWD:ALL']
      ssh-authorized-keys:
       - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDf7E8+1x0AW3WxezOmx1tOrLx
```

Figure 11.32 – Using cloud-init for additional configuration

As always, we will explain what's going on step by step. One thing we can see from the start is that it uses straight YAML notation, the same as Ansible. The first directive is here to make sure that our machine is updated as it enables automatically updating the packages on the cloud instance.

Then, we are configuring users. We are going to create one user named **`ansible`** who will belong to group **`wheel`**.

`Lock_passwd` means that we are going to permit using the password to log in. If nothing is configured, then the default is to permit logging in only using **`SSH`** keys and disabling password login completely.

Then, we have the password in hash format. Depending on the distribution, this hash can be created in different ways. Do *not* put a plaintext password here.

Then, we have a shell that this user will be able to use if something needs to be added to the **`/etc/sudoers`** file. In this particular case, we are giving this user complete control over the system.

The last thing is probably the most important. This is the public **`SSH`** key that we have on our system. It's used to authorize the user when they're logging in. There can be multiple keys here, and they are going to end up in the **`SSHD`** configuration to enable users to perform a passwordless login.

There are plenty more variables and directives we can use here, so consult the **`cloud-config`** documentation for more information.

After we have created this file, we need to convert it into an **`.iso`** file that is going to be used for installation. The command to do this is **`cloud-localds`**. We are using our YAML file as one parameter and the **`.iso`** file as another.

After running **`cloud-localds config.iso config.yaml`**, we are ready to begin our deployment.

The next thing we need is the cloud image for CentOS. As we mentioned previously, this is a special image that is designed to be used for this particular purpose.

We are going to get it from **https://cloud.centos.org/centos/7/images**.

There are quite a few files here denoting all the available versions of the CentOS image. If you need a specific version, pay attention to the numbers denoting the month/year of the image release. Also, note that images come in two flavors – compressed and uncompressed.

Images are in **qcow2** format and intended to be used in the cloud as a disk.

In our example, on the Ansible machine, we created a new directory called **/clouddeploy** and saved two file into it: one that contains the OS cloud image and **config.iso**, which we created using **cloud-init**:

```
[root@vm0-101 cloud1]# ls -alh /clouddeploy/
total 899M
drwxr-xr-x.  2 root root   71 Sep  9 18:32 .
dr-xr-xr-x. 18 root root  243 Sep  9 18:07 ..
-rw-r--r--.  1 root root 899M Aug  8 15:30 CentOS-7-x86_64-GenericCloud-1907.qcow2
-rw-r--r--.  1 root root 366K Sep  9 18:32 config.iso
```

Figure 11.33 – Checking the content of a directory

All that remains now is to create a playbook to deploy these. Let's go through the steps:

1. First, we are going to copy the cloud image and our configuration onto our KVM hosts. After that, we are going to create a machine out of these and start it:

```
---
  - name: download Centos core image
    hosts: cloudhosts
    tasks:
     - name: Copy to cloud instance
       copy:
          src: /clouddeploy/CentOS-7-x86_64-GenericCloud-1907.qcow2
          dest: /var/lib/libvirt/images/cloudsrv1/
          owner: qemu
          group: qemu
          mode: u=rw,g=rw,o=r

     - name: Copy cloud-init configuration
       copy:
          src: /clouddeploy/config.iso
          dest: /var/lib/libvirt/images/cloudsrv1/
          owner: qemu
          group: qemu
          mode: u=rw,g=rw,o=r

     - name: Create machine
       command: >
                 virt-install --name=COS7Cloud --ram=1024 --vcpus=1 --os-type=linux --os-variant=rhel7
                 --disk path=/var/lib/libvirt/images/cloudsrv1/CentOS-7-x86_64-GenericCloud-1907.qcow2,device=disk
                 --disk /var/lib/libvirt/images/cloudsrv1/config.iso,device=cdrom --graphics none --import --noautoconsole

     - name: Start VM
       virt:
          name: COS7Cloud
          state: running
```

Figure 11.34 – The playbook that will download the required image, configure cloud-init, and start the VM deployment process

Since this is our first *complicated* playbook, we need to explain a few things. In every play or task, there are some things that are important. A name is used to simplify running the playbook; this is what is going to be displayed when the playbook runs. This name should be explanatory enough to help, but not too long in order to avoid clutter.

After the name, we have the business part of each task – the name of the module being called. In our example, we are using three distinct ones: **`copy`**, **`command`**, and **`virt`**. **`copy`** is used to copy files between hosts, **`command`** executes commands on the remote machine, and

virt contains commands and states needed to control the virtual environment.

You will notice when reading this that **copy** looks strange; **src** denotes a local directory, while **dest** denotes a remote one. This is by design. To simplify things, **copy** works between the local machine (the control node running Ansible) and the remote machine (the one being configured). Directories will get created if they do not exist, and **copy** will apply the appropriate permissions.

After that, we are running a command that will work on local files and create a virtual machine. One important thing here is that we are basically running the image we copied; the template is on the control node. At the same time, this saves disk space and deployment time – there is no need to copy the machine from local to remote disk and then duplicate it on the remote machine once again; as soon as the image is there, we can run it.

Back to the important part – the local installation. We are creating a machine with 1 GB of

RAM and one CPU using the disk image we just copied. We're also attaching our **`config.iso`** file as a virtual CD/DVD. We are then importing this image and using no graphic terminal.

2. The last task is starting the VM on the remote KVM host. We will use the following command to do so:

```
ansible-playbook installvms.yaml
```

If everything ran OK, we should see something like this:

```
[root@vm0-101 ~]# ansible-playbook installvms.yaml

PLAY [download Centos core image] **********************************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************************
ok: [10.0.0.4]
ok: [10.0.0.2]
ok: [10.0.0.3]

TASK [Copy to cloud instance] **************************************************************************************************************************
ok: [10.0.0.2]
ok: [10.0.0.4]
ok: [10.0.0.3]

TASK [Copy cloud-init configuration] *******************************************************************************************************************
changed: [10.0.0.3]
changed: [10.0.0.2]
changed: [10.0.0.4]

TASK [Create machine] **********************************************************************************************************************************
changed: [10.0.0.4]
changed: [10.0.0.3]
changed: [10.0.0.2]

TASK [Start VM] ****************************************************************************************************************************************
ok: [10.0.0.3]
ok: [10.0.0.4]
ok: [10.0.0.2]

PLAY RECAP *********************************************************************************************************************************************
10.0.0.2                   : ok=5    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
10.0.0.3                   : ok=5    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
10.0.0.4                   : ok=5    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 11.35 – Checking our installation process

We can also check this using the command line:

```
ansible cloudhosts -m shell -a "virsh
list –all"
```

The output of this command should look something like this:

```
[root@vm0-101 ~]# ansible cloudhosts -m shell -a "virsh list --all"
10.0.0.4 | CHANGED | rc=0 >>
 Id    Name                           State
----------------------------------------------------
 3     COS7Cloud                      running

10.0.0.3 | CHANGED | rc=0 >>
 Id    Name                           State
----------------------------------------------------
 3     COS7Cloud                      running

10.0.0.2 | CHANGED | rc=0 >>
 Id    Name                           State
----------------------------------------------------
 3     COS7Cloud                      running
```

Figure 11.36 – Checking our VMs

Let's check two more things – networking and the machine state. Type in the following command:

```
ansible cloudhosts -m shell -a "virsh
net-dhcp-leases –-network default"
```

We should get something like this:

```
[root@vm0-101 ~]# ansible cloudhosts -m shell -a "virsh net-dhcp-leases --network default"
10.0.0.2 | CHANGED | rc=0 >>
 Expiry Time          MAC address        Protocol  IP address                Hostname        Client ID or DUID
-------------------------------------------------------------------------------------------------------------------------------
 2019-09-09 19:33:19  52:54:00:9a:e0:20  ipv4      192.168.122.38/24         -               -

10.0.0.4 | CHANGED | rc=0 >>
 Expiry Time          MAC address        Protocol  IP address                Hostname        Client ID or DUID
-------------------------------------------------------------------------------------------------------------------------------
 2019-09-09 19:33:19  52:54:00:a4:b8:21  ipv4      192.168.122.119/24        -               -

10.0.0.3 | CHANGED | rc=0 >>
 Expiry Time          MAC address        Protocol  IP address                Hostname        Client ID or DUID
-------------------------------------------------------------------------------------------------------------------------------
 2019-09-09 19:33:19  52:54:00:31:fc:7c  ipv4      192.168.122.161/24        -               -
```

Figure 11.37 – Checking our VM network connectivity and network configuration

This verifies that our machines are running correctly and that they are connected to their local network on the local KVM instance. Elsewhere in this book, we will deal with KVM networking in more detail, so it should be easy to reconfigure machines to use a common network, either by bridging adapters on the KVMs or by creating a separate virtual network that will span across hosts.

Another thing we wanted to show is the machine status for all the hosts. The point is that we are not using the shell module this time; instead, we are relying on the **`virt`** module to show us how to use it from the command line. There is only one subtle difference here. When we are calling shell (or **`command`**) modules, we are calling parameters that are going to get called. These modules basically just spawn another process on the remote machine and run it with the parameters we provided.

In contrast, the **`virt`** module takes the variable declaration as its parameter since we are run-

ning **`virt`** with **`command=info`**. When using Ansible, you will notice that, sometimes, variables are just states. If we wanted to start a particular instance, we would just add **`state=running`**, along with an appropriate name, and Ansible would make sure that the VM is running. Let's type in the following command:

```
ansible cloudhosts -m virt -a
"command=info"
```

The following is the expected output:

```
[root@vm0-101 ~]# ansible cloudhosts -m virt -a "command=info"
10.0.0.4 | SUCCESS => {
    "COS7Cloud": {
        "autostart": 0,
        "cpuTime": "35130000000",
        "maxMem": "1048576",
        "memory": "1048576",
        "nrVirtCpu": 1,
        "state": "running"
    },
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false
}
10.0.0.3 | SUCCESS => {
    "COS7Cloud": {
        "autostart": 0,
        "cpuTime": "34500000000",
        "maxMem": "1048576",
        "memory": "1048576",
        "nrVirtCpu": 1,
        "state": "running"
    },
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false
}
10.0.0.2 | SUCCESS => {
    "COS7Cloud": {
        "autostart": 0,
        "cpuTime": "34260000000",
        "maxMem": "1048576",
        "memory": "1048576",
        "nrVirtCpu": 1,
        "state": "running"
    },
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false
}
```

Figure 11.38 – Using the virt module with Ansible

There is only one thing that we haven't covered yet – how to install multi-tiered applications. Pushing the definition to its smallest extreme, we

are going to install a LAMP server using a simple playbook.

Orchestrating multi-tier application deployment on KVM VM

Now, let's learn how to install multi-tiered applications. Pushing the definition to its smallest extreme, we are going to install a LAMP server using a simple Ansible playbook.

The tasks that need to be done are simple enough – we need to install Apache, MySQL, and PHP. The *L* part of LAMP is already installed, so we are not going to go through that again.

The difficult part is the package names: in our demonstration machine, we are using CentOS7 as the operating system and its package names are a little different. Apache is called **`httpd`** and **`mysql`** is replaced with **`mariaDB`**, another engine

that is compatible with MySQL. PHP is luckily the same as on other distributions. We also need another package named **`python2-PyMySQL`** (the name is case sensitive) in order to get our playbook to work.

The next thing we are going to do is test the installation by starting all the services and creating the simplest **`.php`** script possible. After that, we are going to create a database and a user that is going to use it. As a warning, in this chapter, we are concentrating on Ansible basics, since Ansible is far too complex to be covered in one chapter of a book. Also, we are presuming a lot of things, and our biggest assumption is that we are creating demo systems that are not in any way intended for production. This playbook in particular lacks one important step: creating a root password. Do not go into production with your SQL password not set.

One more thing: our script presumes that there is a file named **`index.php`** in the directory our

playbook runs from, and that file will get copied to the remote system:

```
---
- hosts: 127.0.0.1
  connection: local
  tasks:
    - name: install httpd
      package:
         name: "{{ item }}"
         state: present
      with_items:
           - httpd
           - php
           - mariadb-server
           - python2-PyMySQL

    - name: Copy php test file
      copy:
       src: index.php
       dest: /var/www/html

    - name: Start Apache
      systemd:
        name: httpd
        state: started
        enabled: yes

    - name: Start mariadb
      systemd:
        name: mariadb
        state: started
        enabled: yes

    - name: Create database
      mysql_db: name=ansible state=present login_user=root

    - name: Create user
      mysql_user: name=ansible password=ansible priv=*.*:ALL host=localhost state=present login_user=root
```

Figure 11.39 – Ansible LAMP playbook

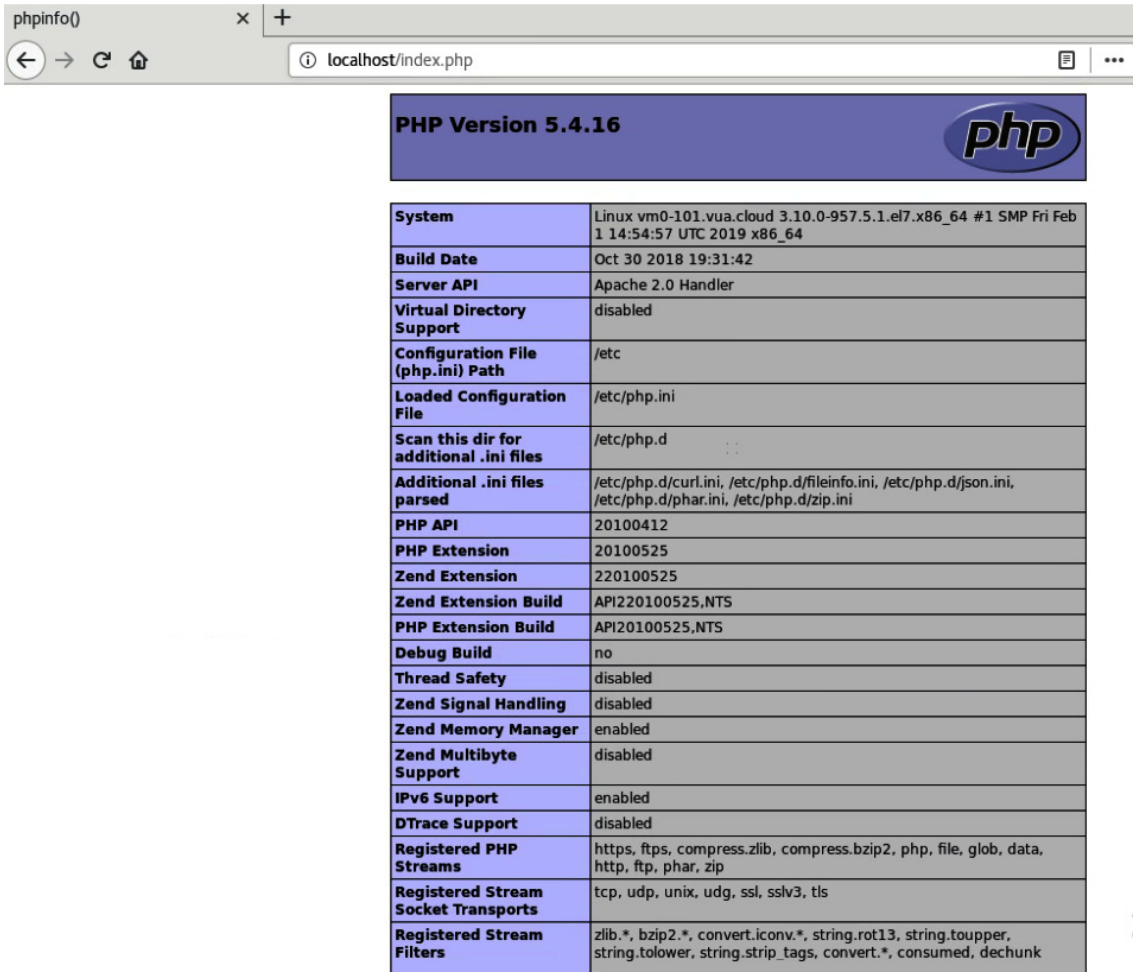
As we can see, there is nothing complicated going on, just a simple sequence of steps. Our **`.php`** file looks like this:

```
<?php
phpinfo();
```

Figure 11.40 – Testing if PHP works

Things can't get any simpler than that. In a normal deployment scenario, we would have something more complicated in the web server direc-

tory, such as a WordPress or Joomla installation, or even a custom application. The only thing that needs to change is the file that is copied (or a set of files) and the location of the database. Our file just prints information about the local `.php` installation:



System	Linux vm0-101.vua.cloud 3.10.0-957.5.1.el7.x86_64 #1 SMP Fri Feb 1 14:54:57 UTC 2019 x86_64
Build Date	Oct 30 2018 19:31:42
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc
Loaded Configuration File	/etc/php.ini
Scan this dir for additional .ini files	/etc/php.d
Additional .ini files parsed	/etc/php.d/curl.ini, /etc/php.d/fileinfo.ini, /etc/php.d/json.ini, /etc/php.d/phar.ini, /etc/php.d/zip.ini
PHP API	20100412
PHP Extension	20100525
Zend Extension	220100525
Zend Extension Build	API220100525,NTS
PHP Extension Build	API20100525,NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	disabled
Zend Memory Manager	enabled
Zend Multibyte Support	disabled
IPv6 Support	enabled
DTrace Support	disabled
Registered PHP Streams	https, ftps, compress.zlib, compress.bzip2, php, file, glob, data, http, ftp, phar, zip
Registered Stream Socket Transports	tcp, udp, unix, udg, ssl, sslv3, tls
Registered Stream Filters	zlib.*, bzip2.*, convert.iconv.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed, dechunk

Figure 11.41 – Checking if PHP works on Apache using a web browser and a previously configured PHP file

Ansible is much more complex than what we showed you here in this chapter, so we strongly suggest you do some further reading and learning. What we did here was just the simplest example of how we can install KVM on multiple hosts and control all of them at once using the command line. What Ansible does best is save us time – imagine having a couple of hundred hypervisors and having to deploy thousands of servers. Using playbooks and a couple of preconfigured images, we can not only configure KVM to run our machines, but reconfigure anything on the machines themselves. The only real prerequisites are a running SSH server and an inventory that will enable us to group machines.

Learning by example – various examples of using Ansible with KVM

Now that we've covered simple and more complex Ansible tasks, let's think about how to use

Ansible to further our configuration skills and overall compliance based on some kind of policy. The following are some things that we are going to leave as exercises for you:

- Task 1:
  We configured and ran one machine per KVM host. Create a playbook that will form a pair of hosts – one running a website and another running a database. You can use any open source CMS for this.
- Task 2:
  Use Ansible and the **`virt-net`** module to reconfigure the network so that the entire cluster can communicate. KVM accepts **`.xml`** configuration for networking, and **`virt-net`** can both read and write XML. Hint: If you get confused, use a separate RHEL8 machine to create a virtual network in the GUI and then use the **`virsh net-dumpxml`** syntax to output a virtual network configuration to standard output, which you can then use as a template.
- Task 3:

Use **`ansible`** and **`virsh`** to auto-start a specific VM that you created/imported on the host.

- Task 4:

  Based on our LAMP deployment playbook, improve on it by doing the following:

  a) Create a playbook that will run on a remote machine.

  b) Create a playbook that will install different roles on different servers.

  c) Create a playbook that will deploy a more complex application, such as WordPress.

If you managed to solve these five tasks, then congratulations – you're *en route* to becoming an administrator who can use Automation, with a capital *A*.

Summary

In this chapter, we discussed Ansible – a simple tool for orchestration and automation. It can be used both in open source and Microsoft-based environments as it supports both natively. Open

source systems can be accessed via SSH keys, while Microsoft operating systems can be accessed by using WinRM and PowerShell. We learned a lot about simple Ansible tasks and more complex ones since deploying a multi-tier application that's hosted on multiple virtual machines isn't an easy task to do – especially if you're approaching the problem manually. Even deploying a KVM hypervisor on multiple hosts can take quite a bit of time, but we managed to solve that with one simple Ansible playbook. Mind you, we only needed some 20 configuration lines to do that, and the upshot of that is that we can easily add hundreds of more hosts as targets for this Ansible playbook.

The next chapter takes us to a world of cloud services – specifically OpenStack – where our Ansible knowledge is going to be very useful for large-scale virtual machine configuration as it's impossible to configure all of our cloud virtual machines by using any kind of manual utilities. Apart from that, we'll extend our knowledge of

Ansible by integrating OpenStack and Ansible so that we can use both of these platforms to do what they do really well – manage cloud environments and configure their consumables.

Questions

1. What is Ansible?
2. What does an Ansible playbook do?
3. Which communication protocol does Ansible use to connect to its targets?
4. What is AWX?
5. What is Ansible Tower?

Further reading

Please refer to the following links for more information regarding what was covered in this chapter:

- What is Ansible?: **https://www.ansible.com/**
- Ansible documentation: **https://docs.ansible.com/**

- Ansible overview: **https://www.ansible.com/overview/it-automation**
- Ansible use cases: **https://www.ansible.com/use-cases**
- Ansible for continuous delivery: **https://www.ansible.com/use-cases/continuous-delivery**
- Integrating Ansible with Jenkins: **https://www.redhat.com/en/blog/integrating-ansible-jenkins-cicd-process**