

# Chapter 9: Customizing a Virtual Machine with cloud-init

Customizing a virtual machine often seems simple enough – clone it from a template; start; click a couple of **Next** buttons (or text tabs); create some users, passwords, and groups; configure network settings... That might work for a virtual machine or two. But what happens if we have to deploy two or three hundred virtual machines and configure them? All of a sudden, we're faced with a mammoth task – and it's a task that will be prone to errors if we do everything manually. We're wasting precious time while doing that instead of configuring them in a much more streamlined, automated fashion. That's where cloud-init comes in handy, as it can customize our virtual machines, install software on them, and it can do it on first and subsequent virtual machine boots. So, let's discuss cloud-init and how it can bring value to your large-scale configuration nightmares.

In this chapter, we will cover the following topics:

- What is the need for virtual machine customization?
- Understanding cloud-init
- cloud-init architecture
- How to install and configure cloud-init at boot time
- cloud-init images
- cloud-init data sources
- Passing metadata and user data to cloud-init
- Examples on how to use the cloud-config script with cloud-init

## What is the need for virtual machine customization?

Once you really start using virtual machines and learn how to master them, you will notice that one thing seems to be happening a lot: virtual machine deployment. Since everything is so easy to configure and deploy, you will start to create new instances of virtual machines for almost anything, sometimes even to just check whether a particular application works on a particular version of the operating system. This makes your life as a developer and system administrator a lot easier, but creates its own set of problems. One of the most difficult ones is template management. Even if you have a small set of differ-

ent servers and a relatively modest number of different configurations, things will start to add up, and if you decide to manage templates the normal way through the KVM, the sheer number of combinations will soon be too big.

Another problem that you will soon face is compatibility. When you step out of your Linux distribution of choice, and you have to deploy another Linux distribution that has its own rules and deployment strategies, things will start to get complicated. Usually, the biggest problem is system customization. When it comes to network settings and hostnames, every computer on the network should have its own unique identity. Having a template that uses DHCP network configuration can solve one of these problems, but it is not nearly enough to make things simpler. For example, we could use Kickstart for CentOS / RHEL and compatible Linux distributions. Kickstart is a way to configure systems while they are being deployed, and if you are using these specific distributions, this is probably the best way to quickly deploy physical or virtual machines. On the other hand, Kickstart will make your deployments slower than they should be, as it uses a configuration file that enables us to add software and configuration to a clean installation.

Basically, it *fills up* additional configuration prompts with settings we defined earlier. This means that we are basically doing a full installation and creating a complete system from scratch every time we need to deploy a new virtual machine.

The main problem is *other distributions do not use Kickstart*. There are similar systems that enable unattended installations. Debian and Ubuntu use a tool/system called *preseed* and are able to support Kickstart in some parts, SuSe uses AutoYaST, and there are even a couple of tools that offer some sort of cross-platform functionality. One of them, called **Fully Automated Install (FAI)** is able to automate installing and even the online reconfiguration of different Linux distributions. But that still doesn't solve all of the problems that we have. In a dynamic world of virtualization, the main goal is to deploy as quickly as possible and to automate as much as possible, since we tend to use the same agility when it comes to removing virtual machines from production.

Imagine this: you need to create a single application deployment to test your new application with different Linux distributions. All of your future virtual machines will need to have a unique identifier in the form of a hostname, a deployed

SSH identity that will enable remote management through Ansible, and of course, your application. Your application has three dependencies – two in the form of packages that can be deployed through Ansible, but one depends on the Linux distribution being used and has to be tailored for that particular Linux distribution. To make things even more realistic, you expect that you will have to periodically repeat this test, and every time you will need to rebuild your dependencies.

There are a couple of ways you can create this environment. One is to simply manually install all the servers and create templates out of them. This means manually configuring everything and then creating a virtual machine template that will be deployed. If we intend to deploy to more than a couple of Linux distributions this is a lot of work. It becomes even more work once the distributions get upgraded since all the templates we are deploying from must be upgraded, often at different points in time. This means we can either manually update all the virtual machine templates, or perform a post-install upgrade on each of them. This is a lot of work and it is extremely slow. Add to that the fact that a test like this will probably involve running your test application on both new and old versions of vir-

tual machine templates. In addition to all that, we need to solve the problem of customizing our network settings for each and every Linux distribution we are deploying. Of course, this also means that our virtual machine templates become far from generic. After a while, we are going to end up with tens of virtual machine templates for each test cycle.

Another approach to this problem can be using a system like Ansible – we deploy all the systems from virtual machine templates, and then do the customization from Ansible. This is better – Ansible is designed for a scenario just like this, but this means that we must first create virtual machine templates that are able to support Ansible deployment, with implemented SSH keys and everything else Ansible needs to function.

There is one problem neither of these approaches can solve, and that is the mass deployment of machines. This is why a framework called cloud-init was designed.

## Understanding cloud-init

We need to get a bit more technical in order to understand what cloud-init is and to understand what its limitations are. Since we are talking about a way to fully automatically reconfigure a

system using simple configuration files, it means that some things need to be prepared in advance to make this complex process user friendly.

We already mentioned virtual machine templates in ***Chapter 8**, Creating and Modifying VM Disks, Templates, and Snapshots*. Here, we are talking about a specially configured template that has all the elements needed to read, understand, and deploy the configuration that we are going to provide in our files. This means that this particular image has to be prepared in advance, and is the most complicated part of the whole system.

Luckily, cloud-init images can be downloaded already pre-configured, and the only thing that we need to know is which distribution we want to use. All the distributions we have mentioned throughout this book (CentOS 7 or 8, Debian, Ubuntu, and Red Hat Enterprise Linux 7 and 8) have images we can use. Some of them even have different versions of the base operating system available, so we can use those if we need to. Be aware that there may be differences between installed versions of cloud-init, especially on older images.

Why is this image important? Because it is prepared so that it can detect the cloud system it is

running under, it determines whether cloud-init should be used or should be disabled, and after that, it reads and performs the configuration of the system itself.

## Understanding cloud-init architecture

Cloud-init works with the concept of boot stages because it needs fine and granular control over what happens to the system during boot. The prerequisite for cloud-init would, of course, be a cloud-init image. From the documentation available at <https://cloudinit.readthedocs.io>, we can learn that there are five stages to a cloud-init boot:

- The **generator** is the first one, and the simplest one: it will determine whether we are even trying to run cloud-init, and based on that, whether it should enable or disable the processing of data files. Cloud-init will not run if there are kernel command-line directives to disable it, or if a file called `/etc/cloud/cloud-init.disabled` exists. For more information on this and all the other things in this chapter, please read the documentation (start at <https://cloudinit.readthedocs.io/en/latest/topics/boot.html>) since it contains much more detail about



switches and different options that cloud-init supports and that make it tick.

- The **local** phase tries to find the data that we included for the boot itself, and then it tries to create a running network configuration. This is a relatively simple task performed by a **systemd** service called **cloud-init-local.service**, which will run as soon as possible and will block the network until it's done. The concept of blocking services and targets is used a lot in cloud-init initialization; the reason is simple – to ensure system stability. Since cloud-init procedures modify a lot of core settings for a system, we cannot afford to let the usual startup scripts run and create a parallel configuration that could overrun the one created by cloud-init.
- The **network** phase is the next one, and it uses a separate service called **cloud-init.service**. This is the main service that will bring up the previously configured network and try to configure everything we scheduled in the data files. This will typically include grabbing all the files specified in our configuration, extracting them, and executing other preparation tasks. Disks will also be formatted and partitioned in this stage if such a configuration change is specified. Mount points will also get

created, including those that are dynamic and specific to a particular cloud platform.

- The **config** stage follows, and it will configure the rest of the system, applying different parts of our configuration. It uses cloud-init modules to further configure our template. Now that the network is configured, it can be used to add repositories (the **yum\_repos** or **apt** modules), add an SSH key (the **ssh-import-id** module), and perform similar tasks in preparation for the next phase, in which we can actually use the configuration done in this phase.
- The **final** stage is the part of the system boot that runs things that would probably belong in userland – installing the packages, the configuration management plugin deployment, and executing possible user scripts.

After all this has been done, the system will be completely configured and up and running.

The main advantage of this approach, although it seems complicated, is to have only one image stored in the cloud, and then to create simple configuration files that will only cover the differences between the *vanilla* default configuration, and the one that we need. Images can also be relatively small since they do not contain too many packages geared toward an end user.

Cloud-init is often used as the first stage in deploying a lot of machines that are going to be managed by orchestration systems such as Puppet or Ansible since it provides a way to create working configurations that include ways of connecting to each instance separately. Every stage uses YAML as its primary data syntax, and almost everything is simply a list of different options and variables that get translated into configuration information. Since we are configuring a system, we can also include almost any other type of file in the configuration – once we can run a shell script while configuring the system, everything is possible.

*Why is all of this so important?*

Cloud-init stems from a simple idea: create a single template that will define the base content of the operating system you plan to use. Then, we create a separate, specially formatted data file that will hold the customization data, and then combine those two at runtime to create a new instance when you need one. You can even improve things a bit by using a template as a base image and then create different systems as differencing images. Trading speed for convenience in this way can mean deploying in minutes instead of hours.

The way cloud-init was conceived was to be as multiplatform as possible and to encompass as many operating systems as can reasonably be done. Currently, it supports the following:

- Ubuntu
- SLES/openSUSE
- RHEL/CentOS
- Fedora
- Gentoo Linux
- Debian
- Arch Linux
- FreeBSD

We enumerated all the distributions, but cloud-init, as its name suggests is also *cloud-aware*, which means that cloud-init is able to automatically detect and use almost any cloud environment. Running any distribution on any hardware or cloud is always a possibility, even without something like cloud-init, but since the idea is to create a platform-independent configuration that will be deployable on any cloud without any reconfiguration, our system needs to automatically account for any differences between different cloud infrastructures. On top of that, cloud-init can be used for bare-metal deployment, even if it isn't specifically designed for it, or to be more precise, even if it is designed for a lot more than that.

### *Important note*

*Being cloud-aware means that cloud-init gives us tools to do post-deployment checks and configuration changes, another extremely useful option.*

This all sounds a lot more theoretical than it should be. In practice, once you start using cloud-init and learn how to configure it, you will start to create a virtual machine infrastructure that will be almost completely independent of the cloud infrastructure you are using. In this book, we are using KVM as the main virtualization infrastructure, but cloud-init works with any other cloud environment, usually without any modification. Cloud-init was initially designed to enable easy deployment on Amazon AWS but it has long since transcended that limitation.

Also, cloud-init is aware of all the small differences between different distributions, so all the things you put in your configuration file will be translated into whatever a particular distribution uses to accomplish a particular task. In that sense, cloud-init behaves a lot like Ansible – in essence, you define what needs to be done, not how to do it, and cloud-init takes that and makes it happen.

# Installing and configuring cloud-init at boot time

The main thing that we are covering in this chapter is how to get cloud-init to run, and how to get all of its parts in the right place when the machine is being deployed, but this only scratches the surface of how cloud-init actually works.

What you need to understand is that cloud-init runs as a service, configures the system, and follows what we told it to do in a certain way. After the system has booted, we can connect to it and see what was done, how, and analyze the logs.

This could seem contrary to the idea of completely automatic deployment but it is there for a reason – whatever we do, there is always the possibility that we will need to debug the system or do some post-installation tasks that can also be automated.

Using cloud-init is not specifically confined to just debugging. After the system has booted, there is a large amount of data created by the system about how the boot was done, what actual cloud configuration the system is running on, and what was done in regard to customization. Any of your applications and scripts can then rely on this data and use it to run and detect

certain configuration and deployment parameters. Check out this example, taken from a virtual machine in Microsoft Azure, running Ubuntu:

```
Cloud-init v. 19.4-33-gbb4131a2-0ubuntu1~18.04.1 running 'init-local' at Sat, 28 Mar 2020 14:51:09 +0000. Up 16.31 seconds.
Cloud-init v. 19.4-33-gbb4131a2-0ubuntu1~18.04.1 running 'init' at Sat, 28 Mar 2020 15:34:11 +0000. Up 2599.03 seconds.
ci-info: +-----+Net device info+-----+
ci-info: +-----+-----+-----+-----+-----+-----+
ci-info: | Device | Up | Address | Mask | Scope | Hw-Address |
ci-info: +-----+-----+-----+-----+-----+-----+
ci-info: | eth0 | True | 10.0.2.8 | 255.255.255.0 | global | 00:0d:3a:b8:19:73 |
ci-info: | eth0 | True | fe80::20d:3aff:feb8:1973/64 | . | link | 00:0d:3a:b8:19:73 |
ci-info: | lo | True | 127.0.0.1 | 255.0.0.0 | host | . |
ci-info: | lo | True | ::1/128 | . | host | . |
ci-info: +-----+-----+-----+-----+-----+-----+
```

Figure 9.1 – A part of cloud-init output at boot time

Cloud-init actually displays this at boot time (and much more, depending on the cloud-init configuration file), and then puts all of this output into its log files, as well. So, we're really well covered in terms of the additional information that it produces.

The next step in our cloud-init journey is discussing cloud-init images, as these are what we need to make cloud-init work. Let's do that now.

## Cloud-init images

In order to use cloud-init at boot time, we first need a cloud image. At its core, it is basically a semi-installed system that contains specially designed scripts that support cloud-init installation. On all distributions, these scripts are part of a package called cloud-init, but images are usually more prepared than that since they try to negoti-

ate a fine line between size and convenience of installation.

In our examples, we are going to use the ones available at the following URLs:

- <https://cloud.centos.org/>
- <https://cloud-images.ubuntu.com/>

In all the examples we are going to work with, the main intention is to show how the system works on two completely different architectures with minimal to no modifications.

Under normal circumstances, getting the image is all you need to be able to run cloud-init. Everything else is handled by the data files.

For example, these are some of the available images for the CentOS distribution:

























 CentOS-7-x86_64-GenericCloud-1808.qcow2c	2018-09-06 09:18	399M
 CentOS-7-x86_64-GenericCloud-1808.raw.tar.gz	2018-09-06 09:21	384M
 CentOS-7-x86_64-GenericCloud-1809.qcow2	2018-10-05 17:09	873M
 CentOS-7-x86_64-GenericCloud-1809.qcow2.xz	2018-10-05 17:09	252M
 CentOS-7-x86_64-GenericCloud-1809.qcow2c	2018-10-05 17:09	383M
 CentOS-7-x86_64-GenericCloud-1809.raw.tar.gz	2018-10-05 17:09	368M
 CentOS-7-x86_64-GenericCloud-1811.qcow2	2018-12-03 16:21	895M
 CentOS-7-x86_64-GenericCloud-1811.qcow2.xz	2018-12-03 16:21	261M
 CentOS-7-x86_64-GenericCloud-1811.qcow2c	2018-12-03 16:22	396M
 CentOS-7-x86_64-GenericCloud-1811.raw.tar.gz	2018-12-03 16:22	380M
 CentOS-7-x86_64-GenericCloud-1901.qcow2	2019-01-28 21:40	895M
 CentOS-7-x86_64-GenericCloud-1901.qcow2.xz	2019-01-28 21:40	259M
 CentOS-7-x86_64-GenericCloud-1901.qcow2c	2019-01-28 21:40	395M
 CentOS-7-x86_64-GenericCloud-1901.raw.tar.gz	2019-01-28 21:42	379M
 CentOS-7-x86_64-GenericCloud-1905.qcow2	2019-06-04 09:28	898M
 CentOS-7-x86_64-GenericCloud-1905.qcow2.xz	2019-06-04 09:28	262M
 CentOS-7-x86_64-GenericCloud-1905.qcow2c	2019-06-04 09:29	397M
 CentOS-7-x86_64-GenericCloud-1905.raw.tar.gz	2019-06-04 09:29	381M
 CentOS-7-x86_64-GenericCloud-1907.qcow2	2019-08-08 13:30	899M
 CentOS-7-x86_64-GenericCloud-1907.qcow2.xz	2019-08-08 13:30	263M
 CentOS-7-x86_64-GenericCloud-1907.qcow2c	2019-08-08 13:30	398M
 CentOS-7-x86_64-GenericCloud-1907.raw.tar.gz	2019-08-08 14:55	382M

Figure 9.2 – A wealth of available cloud-init images for CentOS

Notice that images cover almost all of the releases of the distribution, so we can simply test our systems not only on the latest version but on all the other versions available. We can freely use all of these images, which is exactly what we are going to do a bit later when we start with our examples.

## Cloud-init data sources

Let's talk a little about data files. Up to now, we have referred to them generically, and we had a big reason to do so. One of the things that make

cloud-init stand out from other utilities is its ability to support different ways of getting the information on what to install and how to install it. We call these configuration files data sources, and they can be separated into two broad categories – **user data** and **metadata**. We will talk a lot more about each of those in this chapter, but as an early introduction, let's say that everything that a user creates as part of the configuration, including YAML files, scripts, configuration files, and possibly other files to be put on a system, such as applications and dependencies that are part of user data. Metadata usually comes directly from the cloud provider or serves the purpose of identifying machines.

It contains instance data, hostnames, network name, and other cloud-specific details that can prove useful when deploying. We can use both these types of data during boot and will be doing so. Everything we put in will be stored in a large JSON store in **/run/cloud-init/instance-data.json** at runtime, or as part of the actual machine configuration. A good example of this is the hostname, part of the metadata that will end up as the actual hostname on the individual machine. This file is populated by cloud-init and can be accessed through the command line or directly.

When creating any file in the configuration, we can use any file format available, and we are able to compress the files if needed – cloud-init will decompress them before it runs. If we need to pass the actual files into the configuration, there is a limitation though – files need to be encoded as text and put into variables in a YAML file, to be used and written later on the system we are configuring. Just like cloud-init, YAML syntax is declarative – this is an important thing to remember.

Now, let's learn how we pass metadata and user data to cloud-init.

## Passing metadata and user data to cloud-init

In our examples, we are going to create a file that will essentially be an **.iso** image and behave like a CD-ROM connected to the booting machine. Cloud-init knows how to handle a situation like this, and will mount the file, extract all the scripts, and run them in a predetermined order, as we already mentioned when we explained how the boot sequence works (check the *Understanding cloud-init architecture* section earlier in this chapter).

In essence, what we have to do to get the whole thing running is to create an image that will get connected to the cloud template, and that will provide all the data files to the cloud-init scripts inside the template. This is a three-step process:

1. We have to create the files that hold the configuration information.
2. We have to create an image that contains the file data in the right place.
3. We need to associate the image with the template at boot time.

The most complicated part is defining how and what we need to configure when booting. All of this is accomplished on a machine that is running the cloud-utils package for a given distribution.

At this point, we need to make a point about the two different packages that are used in all the distributions to enable cloud-init support:

- **cloud-init** – Contains all that is necessary to enable a computer to reconfigure itself during boot if it encounters a cloud-init configuration
- **cloud-utils** – Is used to create a configuration that is to be applied to a cloud image

The main difference between these packages is the computer we are installing them on. **cloud-**

**init** is to be installed on the computer we are configuring and is part of the deployment image. **cloud-utils** is the package intended to be used on the computer that will create the configuration.

In all the examples and all the configuration steps in this chapter, we are in fact referring to two different computers/servers: one that can be considered primary, and the one that we are using in this chapter – unless we state otherwise – is the computer that we use to create the configuration for cloud-init deployment. This is not the computer that is going to be configured using this configuration, just a computer that we use as a workstation to prepare our files.

In this simplified environment, this is the same computer that runs the entire KVM virtualization and is used both to create and deploy virtual machines. In a normal setup, we would probably create our configuration on a workstation that we work on and deploy to some kind of KVM-based host or cluster. In that case, every step that we present in this chapter basically remains the same; the only difference is the place that we deploy to, and the way that the virtual machine is invoked for the first boot.

We will also note that some virtualization environments, such as OpenStack, oVirt, or RHEV-M, have direct ways to communicate with a cloud-init enabled template. Some of them even permit you to directly reconfigure the machine on first boot from a GUI, but that falls way out of the scope of this book.

The next topic on our list is cloud-init modules. Cloud-init uses modules for a reason – to extend its range of available actions it can take in the virtual machine boot phase. There are dozens of cloud-init modules available – **SSH**, **yum**, **apt**, setting **hostname**, **password**, **locale**, and creating users and groups, to name a few. Let's check how we can use them.

## Using cloud-init modules

When creating a configuration file, in cloud-init, pretty much like in any other software abstraction layer, we are dealing with modules that are going to translate our more-or-less universal configuration demands, such as *this package needs to be installed* into actual shell commands on a particular system. The way this is done is through **modules**. Modules are logical units that break down different functionalities into smaller groups and enable us to use different commands. You can check the list of all available modules at

the following link:

<https://cloudinit.readthedocs.io/en/latest/topics/modules.html>.

It's quite a list, which will just further show you how well developed cloud-init is.

As we can see from the list, some of the modules, such as, for example, **Disk setup** or **Locale**, are completely platform-independent while some, for example, **Puppet**, are designed to be used with a specific software solution and its configuration, and some are specific to a particular distribution or a group of distributions, like **Yum Add Repo** or **Apt Configure**.

This can seem to break the idea of a completely distribution-agnostic way to deploy everything, but you must remember two things – cloud-init is first and foremost cloud-agnostic, not distribution-agnostic, and distributions sometimes have things that are way too different to be solved with any simple solution. So, instead of trying to be everything at once, cloud-init solves enough problems to be useful, and at the same time tries not to create new ones.

### *Important note*

*We are not going to deal with particular modules one by one since it would make this chapter too long and possibly turn it into a book on its own. If*

*you plan on working with cloud-init, consult the module documentation since it will provide all the up-to-date information you need.*

## Examples on how to use a cloud-config script with cloud-init

First, you need to download the cloud images and resize them in order to make sure that the disk size after everything is installed is large enough to accommodate all the files you plan to put in the machine you created. In these examples, we are going to use two images, one for CentOS, and another for Ubuntu Server. We can see that the CentOS image we are using is 8 GB in size, and we will enlarge it to 10 GB. Note that the actual size on the disk is not going to be 10 GB; we are just allowing the image to grow to this size.

We are going to do the same with the Ubuntu image, after we get it from the internet. Ubuntu also publishes cloud versions of their distribution daily, for all supported versions. The main difference is that Ubuntu creates images that are designed to be 2.2 GB when full. We downloaded



an image from <https://cloud.centos.org>; let's now get some information about it:

```
[root@localhost testimages]# ls
bionic-server-cloudimg-amd64.img  CentOS-7-x86_64-GenericCloud-1809.qcow2
[root@localhost testimages]# qemu-img info CentOS-7-x86_64-GenericCloud-1809.qcow2
image: CentOS-7-x86_64-GenericCloud-1809.qcow2
file format: qcow2
virtual size: 8.0G (8589934592 bytes)
disk size: 679M
cluster_size: 65536
Format specific information:
  compat: 0.10
[root@localhost testimages]# qemu-i
qemu-img qemu-io
[root@localhost testimages]# qemu-img info bionic-server-cloudimg-amd64.img
image: bionic-server-cloudimg-amd64.img
file format: qcow2
virtual size: 2.2G (2361393152 bytes)
disk size: 329M
cluster_size: 65536
Format specific information:
  compat: 0.10
[root@localhost testimages]#
```

Figure 9.3 – Cloud-init image sizes

Note that the actual size on the disk is different – **qemu-img** gives us 679 MB and 2.2 GB versus roughly 330 MB and 680 MB of actual disk usage:

```
[root@localhost testimages]# ls -al
total 1031984
drwxr-xr-x. 2 root root      93 Jan 12 16:06 .
dr-xr-x---. 6 root root     242 Jan 12 16:05 ..
-rw-r--r--. 1 root root 344981504 Jan  7 18:29 bionic-server-cloudimg-amd64.img
-rw-r--r--. 1 root root 711770112 Jan 12 01:27 CentOS-7-x86_64-GenericCloud-1809.qcow2
[root@localhost testimages]#
```

Figure 9.4 – Image size via qemu-img differs from the real virtual image size

We can now do a couple of everyday administration tasks on these images – grow them, move them to the correct directory for KVM, use them as a base image, and then customize them via cloud-init:

1. Let's make these images bigger, just so that we can have them ready for future capacity needs (and practice):

```
[root@localhost testimages]# qemu-img resize bionic-server-cloudimg-amd64.img 10G
Image resized.
[root@localhost testimages]# qemu-img resize CentOS-7-x86_64-GenericCloud-1809.qcow2 10G
Image resized.
[root@localhost testimages]# qemu-img info CentOS-7-x86_64-GenericCloud-1809.qcow2
image: CentOS-7-x86_64-GenericCloud-1809.qcow2
file format: qcow2
virtual size: 10G (10737418240 bytes)
disk size: 679M
cluster size: 65536
Format specific information:
  compat: 0.10
[root@localhost testimages]# qemu-img info bionic-server-cloudimg-amd64.img
image: bionic-server-cloudimg-amd64.img
file format: qcow2
virtual size: 10G (10737418240 bytes)
disk size: 329M
cluster size: 65536
Format specific information:
  compat: 0.10
[root@localhost testimages]#
```

Figure 9.5 – Growing the Ubuntu and CentOS maximum image size to 10 GB via `qemu-img`

After growing our images, note that the size on the disk hasn't changed much:

```
[root@localhost testimages]# ls -al
total 1032052
drwxr-xr-x. 2 root root    93 Jan 12 16:06 .
dr-xr-xr-x. 6 root root   242 Jan 12 16:05 ..
-rw-r--r--. 1 root root 344982016 Jan 12 16:13 bionic-server-cloudimg-amd64.img
-rw-r--r--. 1 root root 914948608 Jan 12 16:13 CentOS-7-x86_64-GenericCloud-1809.qcow2
[root@localhost testimages]#
```

Figure 9.6 – The real disk usage has changed only slightly

The next step is to prepare our environment for the cloud-image procedure so that we can enable cloud-init to do its magic.

2. The images that we are going to use are going to be stored in `/var/lib/libvirt/images`:

```
[root@localhost testimages]# mv * /var/lib/libvirt/images/
[root@localhost testimages]# cd /var/lib/libvirt/images/
[root@localhost images]# ls -alh
total 1008M
drwx--x--x. 2 root root    93 Jan 12 16:15 .
drwxr-xr-x. 10 root root  117 Jan 12 01:18 ..
-rw-r--r--. 1 root root 330M Jan 12 16:13 bionic-server-cloudimg-amd64.img
-rw-r--r--. 1 root root 873M Jan 12 16:13 CentOS-7-x86_64-GenericCloud-1809.qcow2
[root@localhost images]#
```

Figure 9.7 – Moving images to the KVM default system directory

We are going to create our first cloud-enabled deployment in the simplest way possible, by only repartitioning the disk and creating a single user with a single SSH key. The key belongs

to the root of the host machine, so we can directly log in to the deployed machine after cloud-init is done.

Also, we are going to use our images as base images by running the following command:

```
[root@localhost images]# qemu-img create -f qcow2 -o backing_file=/var/lib/libvirt/images/CentOS-7-x86_64-GenericCloud-1809.qcow2 deploy-1/centos1.qcow2
Formatting 'deploy-1/centos1.qcow2', fmt=qcow2 size=8589934592 backing_file='/var/lib/libvirt/images/CentOS-7-x86_64-GenericCloud-1809.qcow2' encryption=off cluster_size=65536 lazy_refcounts=off
[root@localhost images]# cd deploy-1/
[root@localhost deploy-1]# ls -alh
total 196K
drwxr-xr-x. 2 root root 27 Jun 21 20:48 .
drwx--x--x. 3 root root 69 Jun 21 20:47 ..
-rw-r--r--. 1 root root 193K Jun 21 20:48 centos1.qcow2
```

Figure 9.8 – Creating an image disk for deployment

The images are now ready. The next step is to start the cloud-init configuration.

3. First, create a local metadata file and put the new virtual machine name in it.
4. The file will be named **meta-data** and we are going to use **local-hostname** to set the name:

```
local-hostname: deploy-1
meta-data (END)
```

Figure 9.9 – Simple meta-data file with only one option

This file is all it takes to name the machine the way we want and is written in a normal YAML notation. We do not need anything else, so this file essentially becomes a one-liner. Then we need an SSH key pair and we need to get it into the configuration. We need to create a file called **user-data** that will look like this:

```
#cloud-config
```

```

users:
  - name: cloud
    ssh-authorized-keys:
      - ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQBAQCZh6
6Gf1lNuMeenGywifUSW1T16uKW0IXnucNwo
IynhymSm1fkTCqyxLk
ImWbyd/tDFkbgTlei3qa245Xwt//5ny2fGi
tcSa7jWvkKvTLiPvxLP
0CvcvGR4aiV/2TuxA1em3JweqpNppyuapH7
u9q0SdxaG2gh3uViYl
/+8uuzJLJJbxb/a8EK+szpdZq7bpLOvigOT
gMan+LGNlsZc6lqE
VD1j40tG3YNtk5lxfKBLxwLpFq7JPfAv8DT
McdYqqqc5PhRnnKLak
SUQ60W0nv4fpa0MKuha1nr072Zyur7FRf9X
FvD+Uc7ABNpeyUTZVI
j2dr5hjjFTPfZWUC96FEh
root@localhost.localdomain
    sudo: [ 'ALL=(ALL)
NOPASSWD:ALL' ]
    groups: users
    shell: /bin/bash
runcmd:
  - echo "AllowUsers cloud" >>
/etc/ssh/sshd_config
  - restart ssh

```

Note that the file must follow the way YAML defines everything including the variables. Pay

attention to the spaces and newlines, as the biggest problems with deployment come from misplaced newlines in the configuration.

There is a lot to parse here. We are creating a user that uses the username **cloud**. This user will not be able to log in using a password since we are not creating one, but we will enable login using SSH keys associated with the local root account, which we will create by using the **ssh-keygen** command. This is just an example SSH key, and SSH key that you're going to use might be different. So, as root, go through the following procedure:

```
[root@localhost ~]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:iNGTi2P1qCnKh19dVZy5hDhdukBbax81VZN2Soc3Etg root@localhost
The key's randomart image is:
+---[RSA 2048]----+
|      .o.+=+++=|
|      . 0000==Eo==|
|      . = o.=..00+o|
|      = * + 0...|
|      = + S . .|
|      . = .|
|      . . + .|
|      oo o|
|      oo.|
+-----[SHA256]-----+
[root@localhost ~]# ls -al .ssh
total 20
drwx-----, 2 root root 80 Jun 22 16:52 .
dr-xr-x---, 20 root root 4096 Jun 21 20:41 ..
-rw-----, 1 root root 1374 Apr 17 01:22 authorized_keys
-rw-----, 1 root root 1679 Jun 22 16:52 id_rsa
-rw-r--r--, 1 root root 396 Jun 22 16:52 id_rsa.pub
-rw-r--r--, 1 root root 366 Apr 27 11:23 known_hosts
```

Figure 9.10 – SSH keygen procedure done, SSH keys are present and accounted for

Keys are stored in the local **.ssh** directory, so we just need to copy them. When we are doing cloud deployments, we usually use this method of authentication, but cloud-init enables us to

define any method of user authentication. It all depends on what we are trying to do and whether there are security policies in place that enforce one authentication method over another.

In the cloud environments, we will rarely define users that are able to log in with a password, but for example, if we are deploying bare-metal machines for workstations, we will probably create users that use normal passwords. When we create a configuration file like this, it is standard practice to use hashes of passwords instead of literal cleartext passwords. The directive you are looking for is probably **passwd:** followed by a string containing the hash of a password.

Next, we configured **sudo**. Our user needs to have root permissions since there are no other users defined for this machine. This means they need to be a member of the **sudo** group and have to have the right permissions defined in the **sudoers** file. Since this is a common setting, we only need to declare the variables, and cloud-init is going to put the settings in the right files. We will also define a user shell. In this file, we can also define all the other users' settings available on Linux, a feature that is intended to help deploy user computers. If you need any of those features, check the

documentation available here:

<https://cloudinit.readthedocs.io/en/latest/topics/modules.html>  
[and-groups](#). All the extended user information fields are supported.

The last thing we are doing is using the **runcmd** directive to define what will happen after the installation finishes, in the last stage. In order to permit the user to log in, we need to put them on the list of allowed users in the **sshd** and we need to restart the service.

Now we are ready for our first deployment.

5. We have three files in our directory: a hard disk that uses a base file with the cloud template, a **meta-data** file that contains just minimal information that is essential for our deployment, and **user-data**, which contains our definitions for our user. We didn't even try to install or copy anything; this install is as minimal as it gets, but in a normal environment this is a regular starting point, as a lot of deployments are intended only to bring our machine online, and then do the rest of the installation by using other tools. Let's move to the next step.

We need a way to connect the files we just created, the configuration, with the virtual machine. Usually, this is done in a couple of ways. The simplest way is usually to generate a **.iso** file that contains the files. Then we just mount

the file as a virtual CD-ROM when we create the machine. On boot, cloud-init will look for the files automatically.

Another way is to host the files somewhere on the network and grab them when we need them. It is also possible to combine these two strategies. We will discuss this a little bit later, but let's finish our deployment first. The local **.iso** image is the way we are going to go on this deployment. There is a tool called **genisoimage** (provided by the package with the same name) that is extremely useful for this (the following command is a one-line command):

```
genisoimage -output deploy-1-  
cidata.iso -volid cidata -joliet -  
rock user-data meta-data
```

What we are doing here is creating an emulated CD-ROM image that will follow the ISO9660/Joliet standard with Rock Ridge extensions. If you have no idea what we just said, ignore all this and think about it this way – we are creating a file that will hold our metadata and user data and present itself as a CD-ROM:

```
[root@localhost deploy-1]# genisoimage -output deploy-1-cidata.iso -volid cidata  
-joliet -rock user-data meta-data  
I: -input-charset not specified, using utf-8 (detected in locale settings)  
Total translation table size: 0  
Total rockridge attributes bytes: 331  
Total directory bytes: 0  
Path table size(bytes): 10  
Max brk space used 0  
183 extents written (0 MB)  
[root@localhost deploy-1]#
```



## Figure 9.11 – Creating an ISO image

In the end, we are going to get something like this:

```
[root@localhost deploy-1]# ls -al
total 38908
drwxr-xr-x. 2 root root      88 Jan 12 17:42 .
drwx--x--x. 5 root root     141 Jan 12 21:36 ..
-rw-r--r--. 1 qemu qemu 39518208 Jan 12 23:37 centos1.qcow2
-rw-r--r--. 1 qemu qemu  374784 Jan 12 18:51 deploy-1-cidata.iso
-rw-r--r--. 1 root root      26 Jan 12 16:27 meta-data
-rw-r--r--. 1 root root     629 Jan 12 17:42 user-data
```

Figure 9.12 – ISO is created and we are ready to start a cloud-init deployment

Please note that images are taken post deployment, so the size of disk can vary wildly based on your configuration. This was all that was needed in the form of preparations. All that's left is to spin up our virtual machine.

Now, let's start with our deployments.

## The first deployment

We are going to deploy our virtual machine by using a command line:

```
virt-install --connect qemu:///system
--virt-type kvm --name deploy-1 --ram
2048 --vcpus=1 --os-type linux --os-
variant generic --disk
path=/var/lib/libvirt/images/deploy-
1/centos1.qcow2,format=qcow2 --disk
/var/lib/libvirt/images/deploy-
```

```
1/deploy-1-cidata.iso,device=cdrom --
import --network network=default --
noautoconsole
```

Although it may look complicated, if you came to this part of the book after reading its previous chapters, there should be nothing you haven't seen yet. We are using KVM, creating a name for our domain (virtual machine), we are going to give it 1 CPU and 2 GB of RAM. We are also telling KVM we are installing a generic Linux system. We already created our hard disk, so we are mounting it as our primary drive, and we are also mounting our **.iso** file to serve as a CD-ROM. Lastly, we will connect our virtual machine to the default network:

```
[root@localhost deploy-1]# virt-install --connect qemu:///system --virt-type kvm
--name deploy-1 --ram 2048 --vcpus=1 --os-type linux --os-variant generic --dis
k path=/var/lib/libvirt/images/deploy-1/centos1.qcow2,format=qcow2 --disk /var/l
ib/libvirt/images/deploy-1/deploy-1-cidata.iso,device=cdrom --import --network n
etwork=default --noautoconsole

Starting install...
Domain creation completed.
[root@localhost deploy-1]# virsh domifaddr deploy-1

```

Name	MAC address	Protocol	Address
vnet0	52:54:00:55:93:9e	ipv4	192.168.122.2/24

```

[root@localhost deploy-1]# ssh cloud@192.168.122.120
^C
[root@localhost deploy-1]# ssh cloud@192.168.122.2
The authenticity of host '192.168.122.2 (192.168.122.2)' can't be established.
ECDSA key fingerprint is SHA256:WRuCActXNTbAlvwfuyNPEgqo6FjJoLas6bLKymPJrEQ.
ECDSA key fingerprint is MD5:44:b5:04:e8:87:ad:24:19:01:a3:e9:8d:a7:0e:42:34.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.122.2' (ECDSA) to the list of known hosts.
[cloud@deploy-1 ~]$

```

Figure 9.13 – Deploying and testing a cloud-init customized virtual machine

The deployment will probably take a minute or two. As soon as the machine boots, it will get the IP address and we can SSH to it using our prede-

defined key. The only thing that was not automated is accepting the fingerprint of the newly booted machine automatically.

Now, the time has come to see what happened when we booted the machine. Cloud-init generated a log at `/var/log` named **cloud-init.log**. The file will be fairly large, and the first thing you will notice is that the log is set to provide debug information, so almost everything will be logged:

```
2020-01-12 19:32:56,966 - util.py[DEBUG]: Cloud-init v. 19.3-41-gc4735dd3-0ubuntu1-18.04.1 running 'init-local' at Sun, 12 Jan 2020 19:32:56 +0000. Up 12.55 seconds.
2020-01-12 19:32:56,966 - main.py[DEBUG]: No kernel command line url found.
2020-01-12 19:32:56,966 - main.py[DEBUG]: Closing stdin.
2020-01-12 19:32:56,969 - util.py[DEBUG]: Writing to /var/log/cloud-init.log - ab: [644] 0 bytes
2020-01-12 19:32:56,969 - util.py[DEBUG]: Changing the ownership of /var/log/cloud-init.log to 102:4
2020-01-12 19:32:56,969 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/instance/boot-finished
2020-01-12 19:32:56,969 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/data/no-net
2020-01-12 19:32:56,969 - handlers.py[DEBUG]: start: init-local/check-cache: attempting to read from cache [check]
2020-01-12 19:32:56,969 - util.py[DEBUG]: Reading from /var/lib/cloud/instance/obj.pkl (quiet=False)
2020-01-12 19:32:56,970 - stages.py[DEBUG]: no cache found
2020-01-12 19:32:56,970 - handlers.py[DEBUG]: finish: init-local/check-cache: SUCCESS: no cache found
2020-01-12 19:32:56,970 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/instance
2020-01-12 19:32:56,976 - stages.py[DEBUG]: Using distro class <class 'cloudinit.distros.ubuntu.Distro'>
2020-01-12 19:32:56,977 - init.py[DEBUG]: Looking for data source in: ['NoCloud', 'None'], via packages ['', 'cloudinit.sources'] then matches dependencies ['FILESYSTEM']
2020-01-12 19:32:57,012 - init.py[DEBUG]: Searching for local data source in: ['DataSourceNoCloud']
2020-01-12 19:32:57,012 - handlers.py[DEBUG]: start: init-local/search-NoCloud: searching for local data from DataSourceNoCloud
2020-01-12 19:32:57,012 - init.py[DEBUG]: Seeing if we can get any data from <class 'cloudinit.sources.DataSourceNoCloud.DataSourceNoCloud'>
2020-01-12 19:32:57,012 - init.py[DEBUG]: Update datasource metadata and network config due to events: New instance first boot
2020-01-12 19:32:57,012 - util.py[DEBUG]: Running command ['systemd-detect-virt', '--quiet', '--container'] with allowed return codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,016 - util.py[DEBUG]: Running command ['running-in-container'] with allowed return codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,017 - util.py[DEBUG]: Running command ['lxc-is-container'] with allowed return codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /proc/1/environ (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 176 bytes from /proc/1/environ
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /proc/self/status (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 1290 bytes from /proc/self/status
2020-01-12 19:32:57,019 - util.py[DEBUG]: querying dmi data /sys/class/dmi/id/product_serial
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /sys/class/dmi/id/product_serial (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 1 bytes from /sys/class/dmi/id/product_serial
2020-01-12 19:32:57,019 - util.py[DEBUG]: dmi data /sys/class/dmi/id/product_serial returned
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/user-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/meta-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/vendor-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/network-config (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud-net/user-data (quiet=False)
2020-01-12 19:32:57,020 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud-net/meta-data (quiet=False)
2020-01-12 19:32:57,020 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud-net/vendor-data (quiet=False)
cloud-init.log
```

Figure 9.14 – The cloud-init.log file, used to check what cloud-init did to the operating system

Another thing is how much actually happens below the surface completely automatically. Since this is CentOS, cloud-init has to deal with the SELinux security contexts in real time, so a lot of the information is simply that. There are also a lot of probes and tests going on. Cloud-init has to establish what the running environment is and

what type of cloud it is running under. If something happens during the boot process and it in any way involves cloud-init, this is the first place to look.

Let's now deploy our second virtual machine by using a second (Ubuntu) image. This is where cloud-init really shines – it works with various Linux (and \*BSD) distributions, whatever they might be. We can put that to the test now.

## The second deployment

The next obvious step is to create another virtual machine, but to prove a point, we are going to use Ubuntu Server (Bionic) as our image:

```
[root@localhost deploy-1]# cd ..
[root@localhost images]# mkdir deploy-2
[root@localhost images]# cd deploy-2
[root@localhost deploy-2]# cp ../deploy-1/user-data .
[root@localhost deploy-2]# cp ../deploy-1/meta-data .
[root@localhost deploy-2]# vi meta-data
[root@localhost deploy-2]# genisoimage -output deploy-2-cidata.iso -volid cidata -joliet -rock user-data meta-data
I: -input-charset not specified, using utf-8 (detected in locale settings)
Total translation table size: 0
Total rockridge attributes bytes: 331
Total directory bytes: 0
Path table size(bytes): 10
Max brk space used 0
183 extents written (0 MB)
```

Figure 9.15 – Preparing our environment for another cloud-init-based virtual machine deployment

What do we need to do? We need to copy both **meta-data** and **user-data** to the new folder. We need to edit the metadata file since it has the hostname inside it, and we want our new machine to have a different hostname. As for **user-data**, it is going to be completely the same as on

our first virtual machine. Then we need to create a new disk and resize it:

```
[root@localhost deploy-2]# qemu-img create -f qcow2 -o backing_file=
/var/lib/libvirt/images/bionic-server-cloudimg-amd64.img bionic.qcow2
Formatting 'bionic.qcow2', fmt=qcow2 size=10737418240 backing file='
/var/lib/libvirt/images/bionic-server-cloudimg-amd64.img' encryption
=off cluster_size=65536 lazy_refcounts=off
[root@localhost deploy-2]# qemu-img resize bionic.qcow2 10G
Image resized.
```

Figure 9.16 – Growing our virtual machine image for deployment purposes

We are creating a virtual machine from our downloaded image, and just allowing for more space as the image is run. The last step is to start the machine:

```
[root@localhost deploy-2]# virt-install --connect qemu:///system --virt-type kvm --name de
ploy-2 --ram 2048 --vcpus 1 --os-type linux --os-variant generic --disk path=/var/lib/libv
irt/images/deploy-2/bionic.qcow2,format=qcow2 --disk path=/var/lib/libvirt/images/deploy-2
/deploy-2-cidata.iso,device=cdrom --import --network network=default --noautoconsole
Starting install...
Domain creation completed.
```

Figure 9.17 – Deploying our second virtual machine with cloud-init

The command line is almost exactly the same, only the names change:

```
virt-install --connect qemu:///system
--virt-type kvm --name deploy-2 --ram
2048 --vcpus=1 --os-type linux --os-
variant generic --disk
path=/var/lib/libvirt/images/deploy-
2/bionic.qcow2,format=qcow2 --disk
/var/lib/libvirt/images/deploy-
2/deploy-2-cidata.iso,device=cdrom --
```

```
import --network network=default --
noautoconsole
```

Now let's check the IP addresses:

```
[root@localhost deploy-2]# virsh domifaddr deploy-1
Name          MAC address          Protocol    Address
-----
vnet0         52:54:00:55:93:9e     ipv4        192.168.122.2/24

[root@localhost deploy-2]# virsh domifaddr deploy-2
Name          MAC address          Protocol    Address
-----
vnet1         52:54:00:e9:6a:9f     ipv4        192.168.122.126/24
```

Figure 9.18 – Check the virtual machine IP addresses

We can see both of the machines are up and running. Now for the big test – can we connect? Let's use the **SSH** command to try:

```
[root@localhost deploy-2]# ssh cloud@192.168.122.126
The authenticity of host '192.168.122.126 (192.168.122.126)' can't be established.
ECDSA key fingerprint is SHA256:4wD2aNIgUHKYIS8ByndSkH36hEe8xIhfb4Z/lLEyqTE.
ECDSA key fingerprint is MD5:a6:61:06:ac:f3:34:0d:0a:91:df:9c:8f:d7:54:d7:e3.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.122.126' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 18.04.3 LTS (GNU/Linux 4.15.0-74-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Sun Jan 12 19:33:37 UTC 2020

System load:  0.68           Processes:           85
Usage of /:   10.1% of 9.52GB Users logged in:    0
Memory usage: 5%            IP address for ens3: 192.168.122.126
Swap usage:   0%

0 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.
```

Figure 9.19 – Using SSH to verify whether we can connect to our virtual machine

As we can see, the connection to our virtual machine works without any problems.

One more thing is to check the deployment log.  
Note that there is no mention of configuring SELinux since we are running on Ubuntu:

```
2020-01-12 19:32:56,966 - util.py[DEBUG]: Cloud-init v. 19.3-41-gc4735dd3-0ubuntu1-18.04.1 running 'init-local' at Sun, 12 Jan 2020 19:32:56 +0000. Up 12.65 seconds.
2020-01-12 19:32:56,966 - main.py[DEBUG]: No kernel command line url found.
2020-01-12 19:32:56,966 - main.py[DEBUG]: Closing stdin.
2020-01-12 19:32:56,969 - util.py[DEBUG]: Writing to /var/log/cloud-init.log - ab: [644] 0 bytes
2020-01-12 19:32:56,969 - util.py[DEBUG]: Changing the ownership of /var/log/cloud-init.log to 102:4
2020-01-12 19:32:56,969 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/instance/boot-finished
2020-01-12 19:32:56,969 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/data/no-net
2020-01-12 19:32:56,969 - handlers.py[DEBUG]: start: init-local/check-cache: attempting to read from cache [check]
2020-01-12 19:32:56,969 - util.py[DEBUG]: Reading from /var/lib/cloud/instance/obj.pkl (quiet=False)
2020-01-12 19:32:56,970 - stages.py[DEBUG]: no cache found
2020-01-12 19:32:56,970 - handlers.py[DEBUG]: finish: init-local/check-cache: SUCCESS: no cache found
2020-01-12 19:32:56,970 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/instance
2020-01-12 19:32:56,976 - stages.py[DEBUG]: Using distro class <class 'cloudinit.distros.ubuntu.Distro'>
2020-01-12 19:32:56,977 - _init_.py[DEBUG]: Looking for data source in: ['NoCloud', 'None'], via packages ['', 'cloudinit.sources'] that matches dependencies ['FILESYSTEM']
2020-01-12 19:32:57,012 - _init_.py[DEBUG]: Searching for local data source in: ['DataSourceNoCloud']
2020-01-12 19:32:57,012 - handlers.py[DEBUG]: start: init-local/search-NoCloud: searching for local data from DataSourceNoCloud
2020-01-12 19:32:57,012 - _init_.py[DEBUG]: Seeing if we can get any data from <class 'cloudinit.sources.DataSourceNoCloud.DataSourceNoCloud'>
2020-01-12 19:32:57,012 - _init_.py[DEBUG]: Update datasource metadata and network config due to events: New instance first boot
2020-01-12 19:32:57,012 - util.py[DEBUG]: Running command ['systemd-detect-virt', '--quiet', '--container'] with allowed return codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,016 - util.py[DEBUG]: Running command ['running-in-container'] with allowed return codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,017 - util.py[DEBUG]: Running command ['lxc-is-container'] with allowed return codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /proc/1/environ (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 176 bytes from /proc/1/environ
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /proc/self/status (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 1290 bytes from /proc/self/status
2020-01-12 19:32:57,019 - util.py[DEBUG]: querying dmi data /sys/class/dmi/id/product_serial
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /sys/class/dmi/id/product_serial (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 1 bytes from /sys/class/dmi/id/product_serial
2020-01-12 19:32:57,019 - util.py[DEBUG]: dmi data /sys/class/dmi/id/product_serial returned
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/user-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/meta-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/vendor-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/network-config (quiet=False)
cloud-init.log
```

Figure 9.20 – The Ubuntu cloud-init log file has no mention of SELinux

Just for fun, let's do another deployment with a twist – let's use a module to deploy a software package.

## The third deployment

Let's deploy another image. In this instance, we are creating another CentOS 7 but this time we are *installing* (not *starting*) **httpd** in order to show how this type of configuration works. Once again, the steps are simple enough: create a directory, copy the metadata and user data files, modify the files, create the **.iso** file, create the disk, and run the machine.



This time we are adding another section (**packages**) to the configuration, so that we can *tell* cloud-init that we need a package to be installed (**httpd**):

```
cloud-config
users:
  - name: cloud
    ssh-authorized-keys:
      - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCh66Gf1NuMeenGywi fUSw1T16uKw0IXnucNwoIynhynSm1fkTCgyxLkImWbyd/tD
      gitcSa7jWvKKvTLiPvxLP0CvcvGR4a1V/2TuxA1em3JweqpNpyuapH7u9q0SdxaG2gh3uViYl/+8uu2JLJJbxb/a8EK+szpdZq7bpL0vlg0TgMa
      kK5LxfKBLxwLpFq7JPFav8DTHcdYqqqc5PhRnnKLak5UQ60W0nv4fpa0MKuhalnr072Zyur7FRf9XFvD+Uc7ABNpeyUTZVIj2dr5hjJFTPfZWUC9
    domain:
      sudo: ['ALL=(ALL) NOPASSWD:ALL']
      groups: sudo
      shell: /bin/bash
packages:
  - httpd
runcmd:
  - echo "AllowUsers cloud" >> /etc/ssh/sshd_config
  - restart ssh
```

Figure 9.21 – Cloud-init configuration file for the third virtual machine deployment

Since all the steps are more or less the same, we get the same result – success:

```
[root@localhost deploy-3]# qemu-img create -f qcow2 -o backing-file=/var/lib/libvirt/images/CentOS-7-x86_64-GenericCloud-1809.qcow2 centos2.qcow2
Formatting 'centos2.qcow2' fmt=qcow2 size=8589934592 backing_file='/var/lib/libvirt/images/CentOS-7-x86_64-GenericCloud-1809.qcow2' encryption=off cluster_size=65535 lazy_refcounts=off
[root@localhost ~]# qemu-img resize centos2.qcow2 10G
Image resized.
[root@localhost ~]# virt-install --connect qemu:///system --virt-type kvm --name deploy-3 --ram 2048 --vcpus=1 --os-type linux --os-variant generic --disk path=/var/lib/libvirt/images/deploy-3/centos2.qcow2,format=qcow2 --disk /var/lib/libvirt/images/deploy-3-cidata.iso,device=cdrom --import --network network=default --noautoconsole

Starting install...
Domain creation completed.
```

Figure 9.22 – Repeating the deployment process for the third virtual machine

We should wait for a while so that the VM gets deployed. After that, let's log in and check whether the image deployed correctly. We asked for **httpd** to be installed during the deployment. Was it?

```
• httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; disabled; vendor preset: disabled)
   Active: inactive (dead)
   Docs: man:httpd(8)
         man:apachectl(8)
```



## Figure 9.23 – Checking whether httpd is installed but not started

We can see that everything was done as expected. We haven't asked for the service to start, so it is installed with the default settings and is disabled and stopped by default.

### After the installation

The intended use of cloud-init is to configure machines and create an environment that will enable further configuration or straight deployment into production. But to enable that, cloud-init has a lot of options that we haven't even mentioned yet. Since we have an instance running, we can go through the most important and the most useful things you can find in the newly booted virtual machine.

The first thing to check is the `/run/cloud-init` folder:

```
[root@deploy-1 cloud-init]# ls -al
total 4
drwxr-xr-x. 2 root root 140 Jan 12 16:43 .
drwxr-xr-x. 23 root root 740 Jan 12 16:43 ..
-rw-r--r--. 1 root root  0 Jan 12 16:43 enabled
-rw-r--r--. 1 root root  8 Jan 12 16:43 instance-id
-rw-r--r--. 1 root root  0 Jan 12 16:43 network-config-ready
lrwxrwxrwx. 1 root root 36 Jan 12 16:43 result.json -> ../../var/lib/cloud/data/result.json
lrwxrwxrwx. 1 root root 36 Jan 12 16:43 status.json -> ../../var/lib/cloud/data/status.json
[root@deploy-1 cloud-init]#
```

## Figure 9.24 – /run/cloud-init folder contents

Everything that is created at runtime is written here, and available for users. Our demo machine was run under the local KVM hypervisor so

cloud-init is not detecting a cloud, and consequently is unable to provide more data about the cloud, but we can see some interesting details.

The first one is two files named **enabled** and **network-config-ready**. Both of them are empty but very important. The fact that they exist signifies that cloud-init is enabled, and that network has been configured and is working. If the files are not there, something went wrong and we need to go back and debug. More about debugging can be found at

<https://cloudinit.readthedocs.io/en/latest/topics/debugging.html>

The **results.json** file holds this particular instance metadata. **status.json** is more concentrated on what happened when the whole process was running, and it provides info on possible errors, the time it took to configure different parts of the system, and whether everything was done.

Both those files are intended to help with the configuration and orchestration, and, while some things inside these files are important only to cloud-init, the ability to detect and interact with different cloud environments is something that other orchestration tools can use. Files are just a part of it.

Another big part of this scheme is the command-line utility called **cloud-init**. To get information from it, we first need to log in to the machine that we created. We are going to show the differences between machines that were created by the same file, and at the same time demonstrate similarities and differences between distributions.

Before we start talking about this, be aware that cloud-init, as with all Linux software, comes in different versions. CentOS 7 images use an old version, 0.7.9:

```
[cloud@deploy-1 ~]$ cloud-init -v
cloud-init 0.7.9
[cloud@deploy-1 ~]$
```

Figure 9.25 – CentOS cloud-init version – quite old

Ubuntu comes with a much fresher version, 19.3:

```
cloud@deploy-2:~$ cloud-init -v
/usr/bin/cloud-init 19.3-41-gc4735dd3-0ubuntu1~18.04.1
cloud@deploy-2:~$
```

Figure 9.26 – Ubuntu cloud-init version – up to date

Before you freak out, this is not as bad as it seems. Cloud-init decided to switch its versioning system a couple of years ago, so after 0.7.9 came 17.1. There were many changes and most of them are directly connected to the cloud-init command and configuration files. This means

that the deployment will work, but a lot of things after we deploy will not. Probably the most visible difference is when we run **cloud-init --help**. For Ubuntu, this is what it looks like:

```
cloud@deploy-2:/run/cloud-init$ cloud-init -v
/usr/bin/cloud-init 19.3-41-gc4735dd3-0ubuntu1-18.04.1
cloud@deploy-2:/run/cloud-init$ cloud-init --help
usage: /usr/bin/cloud-init [-h] [--version] [--file FILES] [--debug] [--force]
                        {init,modules,single,query,dhclient-hook,features,analyze,devel,collect-logs,clean,status}
                        ...

optional arguments:
  -h, --help            show this help message and exit
  --version, -v         show program's version number and exit
  --file FILES, -f FILES
                        additional yaml configuration files to use
  --debug, -d          show additional pre-action logging (default: False)
  --force              force running even if no datasource is found (use at
                        your own risk)

Subcommands:
  {init,modules,single,query,dhclient-hook,features,analyze,devel,collect-logs,clean,status}
  init                 initializes cloud-init and performs initial modules
  modules              activates modules using a given configuration key
  single               run a single module
  query                Query standardized instance metadata from the command
                        line.
  dhclient-hook        Run the dhclient hook to record network info.
  features             list defined features
  analyze              Devel tool: Analyze cloud-init logs and data
  devel               Run development tools
  collect-logs         Collect and tar all cloud-init debug info
  clean                Remove logs and artifacts so cloud-init can re-run.
  status               Report cloud-init status or wait on completion.
```

Figure 2.27 – Cloud-init features on Ubuntu

Realistically, a lot of things are missing for CentOS, some of them completely:

```
[cloud@deploy-1 ~]$ cloud-init -v
cloud-init 0.7.9
[cloud@deploy-1 ~]$ cloud-init --help
usage: cloud-init [-h] [--version] [--file FILES] [--debug] [--force]
                        {init,modules,query,single,dhclient-hook} ...

positional arguments:
  {init,modules,query,single,dhclient-hook}
  init                 initializes cloud-init and performs initial modules
  modules              activates modules using a given configuration key
  query                query information stored in cloud-init
  single               run a single module
  dhclient-hook        run the dhclient hookto record network info

optional arguments:
  -h, --help            show this help message and exit
  --version, -v         show program's version number and exit
  --file FILES, -f FILES
                        additional yaml configuration files to use
  --debug, -d          show additional pre-action logging (default: False)
  --force              force running even if no datasource is found (use at
                        your own risk)
```

Figure 9.28 – Cloud-init features on CentOS

Since our example has a total of three running instances – one Ubuntu and two CentOS virtual machines – let's try to manually upgrade to the latest stable version of cloud-init available on

CentOS. We can use our regular **yum update** command to achieve that, and the result will be as follows:

```
[cloud@deploy-3 ~]$ cloud-init -v
/usr/bin/cloud-init 18.5
[cloud@deploy-3 ~]$ cloud-init --help
usage: /usr/bin/cloud-init [-h] [--version] [--file FILES] [--debug] [--force]

        {init,modules,single,query,dhclient-hook,features,analyze,devel,collect-logs,clean,status}
        ...

optional arguments:
  -h, --help            show this help message and exit
  --version, -v          show program's version number and exit
  --file FILES, -f FILES additional yaml configuration files to use
  --debug, -d           show additional pre-action logging (default: False)
  --force               force running even if no datasource is found (use at
                        your own risk)

subcommands:
  {init,modules,single,query,dhclient-hook,features,analyze,devel,collect-logs,clean,status}
  init                  initializes cloud-init and performs initial modules
  modules               activates modules using a given configuration key
  single                run a single module
  query                 Query standardized instance metadata from the command
                        line.
  dhclient-hook          Run the dhclient hook to record network info.
  features               list defined features
  analyze                Devel tool: Analyze cloud-init logs and data
  devel                 Run development tools
  collect-logs           Collect and tar all cloud-init debug info
  clean                  Remove logs and artifacts so cloud-init can re-run.
  status                 Report cloud-init status or wait on completion.
```

Figure 9.29 – After a bit of yum update, an up-to-date list of cloud-init features

As we can see, this will make things a lot easier to work with.

We are not going to go into too much detail about the cloud-init CLI tool, since there is simply too much information available for a book like this, and as we can see, new features are being added quickly. You can freely check additional options by browsing at

<https://cloudinit.readthedocs.io/en/latest/topics/cli.html>.

In fact, they are being added so quickly that there is a **devel** option that holds new features while they are in active development. Once they are finished, they become commands of their own.

There are two commands that you need to know about, both of which give an enormous amount of information about the boot process and the state of the booted system. The first one is **cloud-init analyze**. It has two extremely useful sub-commands: **blame** and **show**.

The aptly named **blame** is actually a tool that returns how much time was spent on things that happened during different procedures cloud-init did during boot. For example, we can see that configuring **grub** and working with the filesystem was the slowest operation on Ubuntu:

```
cloud@deploy-2:/run/cloud-init$ cloud-init analyze blame
-- Boot Record 01 --
00.58700s (modules-config/config-grub-dpkg)
00.46700s (init-network/config-growpart)
00.37500s (init-network/config-resizefs)
00.20000s (init-network/config-ssh)
00.19400s (init-network/config-users-groups)
00.16300s (init-local/search-NoCloud)
00.08900s (modules-final/config-keys-to-console)
00.08000s (modules-config/config-apt-configure)
00.05500s (modules-final/config-ssh-authkey-fingerprints)
00.01600s (init-network/check-cache)
00.01500s (modules-final/config-scripts-user)
00.00900s (modules-config/config-runcmd)
00.00400s (modules-final/config-final-message)
00.00400s (init-network/consume-user-data)
00.00200s (modules-config/config-timezone)
00.00100s (modules-final/config-snappy)
00.00100s (modules-final/config-scripts-vendor)
00.00100s (modules-final/config-salt-minion)
00.00100s (modules-final/config-puppet)
00.00100s (modules-final/config-phone-home)
00.00100s (modules-final/config-package-update-upgrade-install)
00.00100s (modules-final/config-lxd)
00.00100s (modules-config/config-ubuntu-advantage)
00.00100s (modules-config/config-snap_config)
00.00100s (modules-config/config-snap)
00.00100s (modules-config/config-set-passwords)
00.00100s (modules-config/config-ntp)
00.00100s (modules-config/config-locale)
00.00100s (modules-config/config-byobu)
00.00100s (modules-config/config-apt-pipelining)
00.00100s (init-network/consume-vendor-data)
00.00100s (init-network/config-write-files)
00.00100s (init-network/config-update_hostname)
00.00100s (init-network/config-seed_random)
00.00100s (init-network/config-mounts)
00.00100s (init-network/config-ca-certs)
```

## Figure 9.30 – Checking time consumption for cloud-init procedures

The third virtual machine that we deployed uses CentOS image and we added **httpd** to it. By extension, it was by far the slowest thing that happened during the cloud-init process:

```
[cloud@deploy-3 ~]$ sudo cloud-init analyze blame
-- Boot Record 01 --
 34.06400s (modules-config/config-package-update-upgrade-install)
 00.33700s (init-network/config-growpart)
 00.16000s (modules-final/config-keys-to-console)
 00.13300s (init-network/config-users-groups)
 00.11500s (modules-config/config-set-passwords)
 00.11300s (init-local/search-NoCloud)
 00.10500s (init-network/config-set_hostname)
 00.05200s (init-network/check-cache)
 00.04500s (init-network/config-resizefs)
 00.03800s (modules-final/config-ssh-authkey-fingerprints)
 00.01900s (modules-config/config-mounts)
 00.01800s (modules-final/config-scripts-user)
 00.01400s (modules-final/config-power-state-change)
 00.01300s (init-network/consume-user-data)
 00.00600s (modules-config/config-salt-minion)
 00.00500s (init-network/config-update_hostname)
 00.00500s (init-network/config-ssh)
 00.00400s (modules-final/config-final-message)
 00.00400s (modules-config/config-locale)
 00.00300s (modules-config/config-timezone)
 00.00200s (modules-final/config-rightscales_userdata)
 00.00200s (modules-final/config-phone-home)
 00.00200s (modules-config/config-yum-add-repo)
 00.00200s (modules-config/config-runcmd)
 00.00200s (modules-config/config-rh_subscription)
 00.00200s (modules-config/config-puppet)
 00.00200s (modules-config/config-chef)
 00.00100s (modules-final/config-scripts-per-once)
 00.00100s (modules-final/config-scripts-per-instance)
 00.00100s (modules-config/config-mcollective)
 00.00100s (init-network/config-write-files)
 00.00100s (init-network/config-update_etc_hosts)
 00.00100s (init-network/config-rsyslog)
 00.00000s (modules-final/config-scripts-per-boot)
 00.00000s (modules-config/config-disable-ec2-metadata)
 00.00000s (init-network/consume-vendor-data)
 00.00000s (init-network/config-migrator)
 00.00000s (init-network/config-bootcmd)
 00.00000s (init-local/check-cache)
```

Figure 9.31 – Checking time consumption – it took quite a bit of time for cloud-init to deploy the necessary httpd packages

A tool like this makes it easier to optimize deployments. In our particular case, almost none of



this makes sense, since we deployed simple machines with almost no changes to the default configuration, but being able to understand why the deployment is slow is a useful, if not essential, thing.

Another useful thing is being able to see how much time it took to actually boot the virtual machine:

```
cloud@deploy-2:/run/cloud-init$ cloud-init analyze boot
-- Most Recent Boot Record --
  Kernel Started at: 2020-01-12 19:32:45.305890
  Kernel ended boot at: 2020-01-12 19:32:52.213325
  Kernel time to boot (seconds): 6.907434940338135
  Cloud-init activated by systemd at: 2020-01-12 19:32:56.283861
  Time between Kernel end boot and Cloud-init activation (seconds): 4.070536136627197
  Cloud-init start: 2020-01-12 19:32:56.966000
successful
```

Figure 9.32 – Checking the boot time

We are going to end this part with a query – **cloud-init query** enables you to request information from the service, and get it in a useable structured format that you can then parse:

```
cloud@deploy-2:/var/log$ cloud-init query --all
{
  "beta keys": [
    "subplatform"
  ],
  "availability zone": null,
  "base64 encoded keys": [],
  "cloud_name": "unknown",
  "ds": {
    "doc": "EXPERIMENTAL: The structure and format of content scoped under the 'ds' key may change in subsequent releases of cloud-init.",
    "meta_data": {
      "dsmode": "net",
      "instance_id": "nocloud",
      "local_hostname": "deploy-2"
    }
  },
  "instance_id": "nocloud",
  "local_hostname": "deploy-2",
  "platform": "nocloud",
  "public ssh keys": [],
  "region": null,
  "sensitive keys": [],
  "subplatform": "config-disk (/dev/sr0)",
  "userdata": "<redacted for non-root user> file:/var/lib/cloud/instance/user-data.txt",
  "v1": {
    "beta keys": [
      "subplatform"
    ],
    "availability zone": null,
    "cloud_name": "unknown",
    "instance_id": "nocloud",
    "local_hostname": "deploy-2",
    "platform": "nocloud",
    "public ssh keys": [],
    "region": null,
    "subplatform": "config-disk (/dev/sr0)"
  },
  "vendordata": "<redacted for non-root user> file:/var/lib/cloud/instance/vendor-data.txt"
}
```

Figure 9.33 – Querying cloud-init for information



After working with it for even a few hours, cloud-init becomes one of those indispensable tools for a system administrator. Of course, its very essence means it will be much more suited to those of us who have to work in the cloud environment, because the thing it does best is the quick and painless deployment of machines from scripts. But even if you are not working with cloud technologies, the ability to quickly create instances that you can use for testing, and then to remove them without any pain, is something that every administrator needs.

## Summary

In this chapter, we covered cloud-init, its architecture, and the benefits in larger deployment scenarios, where configuration consistency and agility are of utmost importance. Pair that with the paradigm change in which we don't do everything manually – we have a tool that does it for us – and it's an excellent addition to our deployment processes. Make sure that you try to use it as it will make your life a lot easier, while preparing you for using cloud virtual machines, where cloud-init is extensively used.

In the next chapter, we're going to learn how to expand this usage model to Windows virtual ma-

chines by using cloudbase-init.

## Questions

1. Recreate our setup using CentOS 7 and Ubuntu base cloud-init images.
2. Create one Ubuntu and two CentOS instances using the same base image.
3. Add a fourth virtual machine using Ubuntu as a base image.
4. Try using some other distribution as a base image without changing any of the configuration files. Give FreeBSD a try.
5. Instead of using SSH keys, use predefined passwords. Is this more or less secure?
6. Create a script that will create 10 identical instances of a machine using cloud-init and a base image.
7. Can you find any reason why it would be more beneficial to use a distribution-native way of installing machines instead of using cloud-init?

## Further reading

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

- Cloud-init documentation hub:  
<https://cloudinit.readthedocs.io/en/latest/>
- Project home page for cloud-init: <https://cloud-init.io/>
- Source code:  
<https://code.launchpad.net/cloud-init>
- Particularly good examples of config files:  
<https://cloudinit.readthedocs.io/en/latest/topics/examples.htmr>