Chapter 3: Installing KVM Hypervisor, libvirt, and oVirt

This chapter provides you with an insight into the main topic of our book, which is the **Kernel Virtual Machine** (**KVM**) and its management tools, libvirt and oVirt. We will also learn how to do a complete installation of these tools from scratch using a basic deployment of CentOS 8. You'll find this to be a very important topic as there will be situations where you just don't have all of the necessary utilities installed – especially oVirt, as this is a completely separate part of the overall software stack, and a free management platform for KVM. As oVirt has a lot of moving parts – Python-based daemons and supporting utilities, libraries, and a GUI frontend – we will include a step-by-step guide to make sure that you can install oVirt with ease.

In this chapter, we will cover the following topics:

- Getting acquainted with QEMU and libvirt
- Getting acquainted with oVirt
- Installing QEMU, libvirt, and oVirt
- Starting a virtual machine using QEMU and libvirt

Let's get started!

Getting acquainted with QEMU and libvirt

In ***Chapter 2***, *KVM as a Virtualization Solution*, we started discussing KVM, QEMU, and various additional utilities that we can use to manage our KVM-based virtualization platform. As a machine emulator, QEMU will be used so that we can create and run our virtual machines on any supported platform – be it as an emulator or virtualizer. We're going to focus our time on the second paradigm, which is using QEMU as a virtual-

izer. This means that we will be able to execute our virtual machine code directly on a hardware CPU below it, which means native or near-native performance and less overhead.

Bearing in mind that the overall KVM stack is built as a module, it shouldn't come as a surprise that QEMU also uses a modular approach. This has been a core principle in the Linux world for many years, which further boosts the efficiency of how we use our physical resources.

When we add libvirt as a management platform on top of QEMU, we get access to some cool new utilities such as the **`virsh`** command, which we can use to do virtual machine administration, virtual network administration, and a whole lot more. Some of the utilities that we're going to discuss later on in this book (for example, oVirt) use libvirt as a standardized set of libraries and utilities to make their GUI-magic possible – basically, they use libvirt as an API. There are other commands that we get access to for a variety of purposes. For example, we're going to use a com-

mand called **`virt-host-validate`** to check whether our server is compatible with KVM or not.

Getting acquainted with oVirt

Bear in mind that most of the work that a sizeable percentage of Linux system administrators do is done via command-line utilities, libvirt, and KVM. They offer us a good set of tools to do everything that we need from the command line, as we're going to see in next part of this chapter. But also, we will get a *hint* as to what GUI-based administration can be like, as we're briefly going to discuss Virtual Machine Manager later in this chapter.

However, that still doesn't cover a situation in which you have loads of KVM-based hosts, hundreds of virtual machines, dozens of virtual networks interconnecting them, and a rack full of storage devices that you need to integrate with

your KVM environment. Using the aforementioned utilities is just going to introduce you to a world of pain as you scale your environment out. The primary reason for this is rather simple – we still haven't introduced any kind of *centralized* software package for managing KVM-based environments. When we say centralized, we mean that in a literal sense – we need some kind of software solution that can connect to multiple hypervisors and manage all of their capabilities, including network, storage, memory, and CPU or, what we sometimes refer to as the *four pillars of virtualization*. This kind of software would preferably have some kind of GUI interface from which we can *centrally* manage all of our KVM resources, because – well – we're all human. Quite a few of us prefer pictures to text, and interactivity to text-administration only, especially at scale.

This is where the oVirt project comes in. oVirt is an open source platform for the management of our KVM environment. It's a GUI-based tool that

has a lot of moving parts in the background – the engine runs on a Java-based WildFly server (what used to be known as JBoss), the frontend uses a GWT toolkit, and so on. But all of them are there to make one thing possible – for us to manage a KVM-based environment from a centralized, web-based administration console.

From an administration standpoint, oVirt has two main building blocks – the engine (which we can connect to by using a GUI interface) and its agents (which are used to communicate with hosts). Let's describe their functionalities in brief.

The oVirt engine is the centralized service that can be used to perform anything that we need in a virtualized environment – manage virtual machines, move them, create images, storage administration, virtual network administration, and so on. This service is used to manage oVirt hosts and to do that, it needs to talk to something on those hosts. This is where the oVirt agent (vdsm) comes into play.

Some of the available advanced functionalities of the oVirt engine include the following:

- Live migration of virtual machines
- Image management
- Export and import of virtual machines (OVF format)
- **Virtual-to-virtual conversion (V2V)**
- High availability (restart virtual machines from failed hosts on remaining hosts in the cluster)
- Resource monitoring

Obviously, we need to deploy an oVirt agent and related utilities to our hosts, which are going to be the main part of our environment and a place where we will host everything – virtual machines, templates, virtual networks, and so on. For that purpose, oVirt uses a specific agent-based mechanism, via an agent called **vdsm**. This is an agent that we will deploy to our CentOS 8 hosts so that we can add them to oVirt's inventory, which, in turn, means that we can manage them by using the oVirt engine GUI.

Vdsm is a Python-based agent that the oVirt engine uses so that it can directly communicate with a KVM host, and vdsm can then talk to the locally installed libvirt engine to do all the necessary operations. It's also used for configuration purposes as hosts need to be configured to be used in the oVirt environment in order to configure virtual networks, storage management and access, and so on. Also, vdsm has **Memory Overcommitment Manager** (**MOM**) integration so that it can efficiently manage memory on our virtualization hosts.

In graphical terms, this is what the architecture of oVirt looks like:

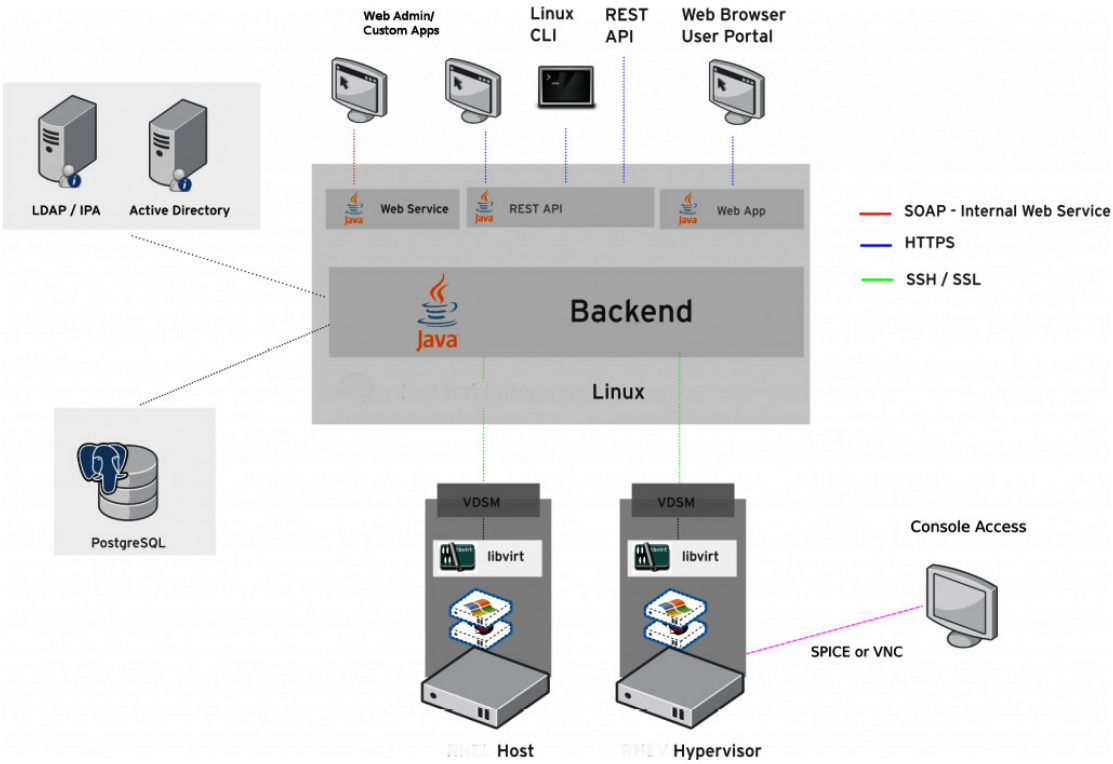

Figure 3.1 – The oVirt architecture (source: http://ovirt.org)

We will take care of how to install oVirt in the next chapter. If you've ever heard or used a product called Red Hat Enterprise Virtualization, it might look very, very familiar.

Installing QEMU, libvirt, and oVirt

Let's start our discussion about installing QEMU, libvirt, and oVirt with some basic information:

- We're going to use CentOS 8 for everything in this book (apart from some bits and pieces that only support CentOS 7 as the last supported version at the time of writing).
- Our default installation profile is always going to be **Server with GUI**, with the premise being that we're going to cover both GUI and text-mode utilities to do almost everything that we're going to do in this book.
- Everything that we need to install on top of our default *Server with GUI* installation is going to be installed manually so that we have a complete, step-by-step guide for everything that we do.
- All the examples that we're going to cover in this book can be installed on a single physical server with 16 physical cores and 64 GB of memory. If you modify some numbers (number of cores assigned to virtual machines, amount of memory assigned to some virtual machines, and so on), you could do this with a 6-core laptop and 16 GB of memory, provided that you're not running all the virtual ma-

chines all the time. If you shut the virtual machines down after you've completed this chapter and start the necessary ones in the next chapter, you'll be fine with that. In our case, we used a HP ProLiant DL380p Gen8, an easy-to-find, second-hand server – and a quite cheap one at that.

After going through a basic installation of our server – selecting the installation profile, assigning network configuration and root password, and adding additional users (if we need them) – we're faced with a system that we can't do virtualization with because it doesn't have all of the necessary utilities to run KVM virtual machines. So, the first thing that we're going to do is a simple installation of the necessary modules and base applications so that we can check whether our server is compatible with KVM. So, log into your server as an administrative user and issue the following command:

```
yum module install virt
```

```
dnf install qemu-img qemu-kvm libvirt
libvirt-client virt-manager virt-
install virt-viewer -y
```

We also need to tell the kernel that we're going to use IOMMU. This is achieved by editing **/etc/default/grub** file, finding the **GRUB_CMDLINE_LINUX** and adding a statement at the end of this line:

```
intel_iommu=on
```

Don't forget to add a single space before adding the line. Next step is reboot, so, we need to do:

```
systemctl reboot
```

By issuing these commands, we're installing all the necessary libraries and binaries to run our KVM-based virtual machines, as well as to use virt-manager (the GUI libvirt management utility) to manage our KVM virtualization server.

Also, by adding the IOMMU configuration, we're making sure that our host sees the IOMMU and doesn't throw us an error when we use **virt-host-validate** command

After that, let's check whether our host is compatible with all the necessary KVM requirements by issuing the following command:

```
virt-host-validate
```

This command goes through multiple tests to determine whether our server is compatible or not. We should get an output like this:

```
[root@packtVM01 ~]# virt-host-validate
  QEMU: Checking for hardware virtualization                 : PASS
  QEMU: Checking if device /dev/kvm exists                   : PASS
  QEMU: Checking if device /dev/kvm is accessible            : PASS
  QEMU: Checking if device /dev/vhost-net exists             : PASS
  QEMU: Checking if device /dev/net/tun exists               : PASS
  QEMU: Checking for cgroup 'memory' controller support      : PASS
  QEMU: Checking for cgroup 'memory' controller mount-point  : PASS
  QEMU: Checking for cgroup 'cpu' controller support         : PASS
  QEMU: Checking for cgroup 'cpu' controller mount-point     : PASS
  QEMU: Checking for cgroup 'cpuacct' controller support     : PASS
  QEMU: Checking for cgroup 'cpuacct' controller mount-point : PASS
  QEMU: Checking for cgroup 'cpuset' controller support      : PASS
  QEMU: Checking for cgroup 'cpuset' controller mount-point  : PASS
  QEMU: Checking for cgroup 'devices' controller support     : PASS
  QEMU: Checking for cgroup 'devices' controller mount-point : PASS
  QEMU: Checking for cgroup 'blkio' controller support       : PASS
  QEMU: Checking for cgroup 'blkio' controller mount-point   : PASS
  QEMU: Checking for device assignment IOMMU support         : PASS
```

Figure 3.2 – virt-host-validate output

This shows that our server is ready for KVM. So, the next step, now that all the necessary QEMU/libvirt utilities are installed, is to do some pre-flight checks to see whether everything that we installed was deployed correctly and works like it should. We will run the **`virsh net-list`**

and **`virsh list`** commands to do this, as shown in the following screenshot:

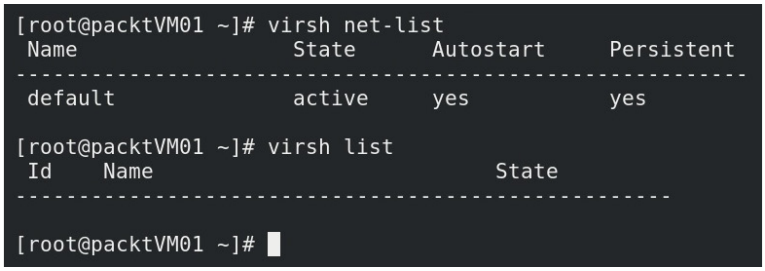

Figure 3.3 – Testing KVM virtual networks and listing the available virtual machines

By using these two commands, we checked whether our virtualization host has a correctly configured default virtual network switch/bridge (more about this in the next chapter), as well as whether we have any virtual machines running. We have the default bridge and no virtual machines, so everything is as it should be.

Installing the first virtual machine in KVM

We can now start using our KVM virtualization server for its primary purpose – to run virtual machines. Let's start by deploying a virtual machine on our host. For this purpose, we copied a

CentOS 8.0 ISO file to our local folder called **/var/lib/libvirt/images**, which we're going to use to create our first virtual machine. We can do that from the command line by using the following command:

```
virt-install --virt-type=kvm --name
MasteringKVM01 --vcpus 2 --ram 4096 -
-os-variant=rhel8.0 --
cdrom=/var/lib/libvirt/images/
CentOS-8-x86_64-1905-dvd1.iso  --
network=default --graphics vnc --disk
size=16
```

There are some parameters here that might be a bit confusing. Let's start with the **--os-variant** parameter, which describes which guest operating system you want to install by using the **virt-install** command. If you want to get a list of supported guest operating systems, run the following command:

```
osinfo-query os
```

The **--network** parameter is related to our default virtual bridge (we mentioned this earlier).

We definitely want our virtual machine to be network-connected, so we picked this parameter to make sure that it's network-connected out of the box.

After starting the **`virt-install`** command, we should be presented with a VNC console window to follow along with the installation procedure. We can then select the language used, keyboard, time and date, and installation destination (click on the selected disk and press **Done** in the top-left corner). We can also activate the network by going to **Network & Host Name**, clicking on the **OFF** button, selecting **Done** (which will then switch to the **ON** position), and connecting our virtual machine to the underlying network bridge (*default*). After that, we can press **Begin Installation** and let the installation process finish. While waiting for that to happen, we can click on **Root Password** and assign a root password for our administrative user.

If all of this seems a bit like *manual labor* to you, we feel your pain. Imagine having to deploy

dozens of virtual machines and clicking on all these settings. We're not in the 19th century anymore, so there must be an easier way to do this.

Automating virtual machine installation

By far, the simplest and the easiest way to do these things in a more *automatic* fashion would be to create and use something called a **kickstart** file. A kickstart file is basically a text configuration file that we can use to configure all the deployment settings of our server, regardless of whether we're talking about a physical or a virtual server. The only caveat is that kickstart files need to be pre-prepared and widely available – either on the network (web) or on a local disk. There are other options that are supported, but these are the most commonly used ones.

For our purpose, we're going to use a kickstart file that's available on the network (via the web server), but we're going to edit it a little bit so

that it's usable, and leave it on our network where **`virt-install`** can use it.

When we installed our physical server, as part of the installation process (called **`anaconda`**), a file was saved in our **`/root`** directory called **`anaconda-ks.cfg`**. This is a kickstart file that contains the complete deployment configuration of our physical server, which we can then use as a basis to create a new kickstart file for our virtual machines.

The simplest way to do that in CentOS 7 was to deploy a utility called **`system-config-kickstart`**, which is not available anymore in CentOS 8. There's a replacement online utility at **https://access.redhat.com/labs/kickstartconfig/** called Kickstart Generator, but you need to have a Red Hat Customer Portal account for that one. So, if you don't have that, you're stuck with text-editing an existing kickstart file. It's not very difficult, but it might take a bit of effort. The most important setting that we need to configure correctly is related to the *location* that we're going

to install our virtual machine from – on a network or from a local directory (as we did in our first **virt-install** example, by using a CentOS ISO from local disk). If we're going to use an ISO file locally stored on the server, then it's an easy configuration. First, we're going to deploy the Apache web server so that we can host our kickstart file online (which will come in handy later). So, we need the following commands:

```
dnf install httpd
systemctl start httpd
systemctl enable httpd
cp /root/anaconda-ks.cfg
/var/www/html/ks.cfg
chmod 644 /var/www/html/ks.cfg
```

Before we start the deployment process, use the vi editor (or any other editor you prefer) to edit the first configuration line in our kickstart file (**/var/www/html/ks.cfg**), which says something like **ignoredisk --only-use=sda**, to **ignoredisk --only-use=vda**. This is because virtual KVM machines don't use **sd*** naming for devices, but

vd naming. This makes it easier for any administrator to figure out if they are administering a physical or a virtual server after connecting to it.

By editing the kickstart file and using these commands, we installed and started **httpd** (Apache web server). Then, we permanently started it so that it gets started after every next server reboot. Then, we copied our default kickstart file (**anaconda-ks.cfg**) to Apache's **DocumentRoot** directory (the directory that Apache serves its files from) and changed permissions so that Apache can actually read that file when a client requests it. In our example, the *client* that's going to use it is going to be the **virt-install** command. The server that we're using to illustrate this feature has an IP address of **10.10.48.1**, which is what we're going to use for our kickstart URL. Bear in mind that the default KVM bridge uses IP address **192.168.122.1**, which you can easily check with the **ip** command:

```
ip addr show virbr0
```

Also, there might be some firewall settings that will need to be changed on the physical server (accepting HTTP connections) so that the installer can successfully get the kickstart file. So, let's try that. In this and the following examples, pay close attention to the **`--vcpus`** parameter (the number of virtual CPU cores for our virtual machine) as you might want to change that to your environment. In other words, if you don't have 4 cores, make sure that you lower the core count. We are just using this as an example:

```
virt-install --virt-type=kvm --
name=MasteringKVM02 --ram=4096 --
vcpus=4 --os-variant=rhel8.0 --
location=/var/lib/libvirt/images/
CentOS-8-x86_64-1905-dvd1.iso --
network=default --graphics vnc --disk
size=16 -x
"ks=http://10.10.48.1/ks.cfg"
```

Important note

Please take note of the parameter that we changed. Here, we must use the **`--location`** *pa-*

rameter, not the **`--cdrom`** *parameter, as we're injecting a kickstart configuration into the boot process (it's mandatory to do it this way).*

After the deployment process is done, we should have two fully functional virtual machines called **`MasteringKVM01`** and **`MasteringKVM02`** on our server, ready to be used for our future demonstrations. The second virtual machine (**`MasteringKVM02`**) will have the same root password as the first one because we didn't change anything in the kickstart file except for the virtual disk option. So, after deployment, we can log into our **`MasteringKVM02`** machine by using the root username and password from the **`MasteringKVM01`** machine.

If we wanted to take this a step further, we could create a shell script with a loop that's going to automatically give unique names to virtual machines by using indexing. We can easily implement this by using a **`for`** loop and its counter:

```
#!/bin/bash
```

```
for counter in {1..5}
do
echo "deploying VM $counter"
virt-install --virt-type=kvm --
name=LoopVM$counter --ram=4096 --
vcpus=4 --os-variant=rhel8.0 --
location=/var/lib/libvirt/images/Cent
OS-8-x86_64-1905-dvd1.iso --
network=default --graphics vnc --disk
size=16 -x
"ks=http://10.10.48.1/ks.cfg"
done
```

When we execute this script (don't forget to **`chmod`** it to **`755`**!), we should get 10 virtual machines named **`LoopVM1-LoopVM5`**, all with the same settings, which includes the same root password.

If we're using a GUI server installation, we can use GUI utilities to administer our KVM server. One of these utilities is called **Virtual Machine Manager**, and it's a graphical utility that enables you to do pretty much everything you need for

your basic administration needs: manipulate virtual networks and virtual machines, open a virtual machine console, and so on. This utility is accessible from GNOME desktop – you can use the Windows search key on your desktop and type in **`virtual`**, click on **Virtual Machine Manager**, and start using it. This is what Virtual Machine Manager looks like:

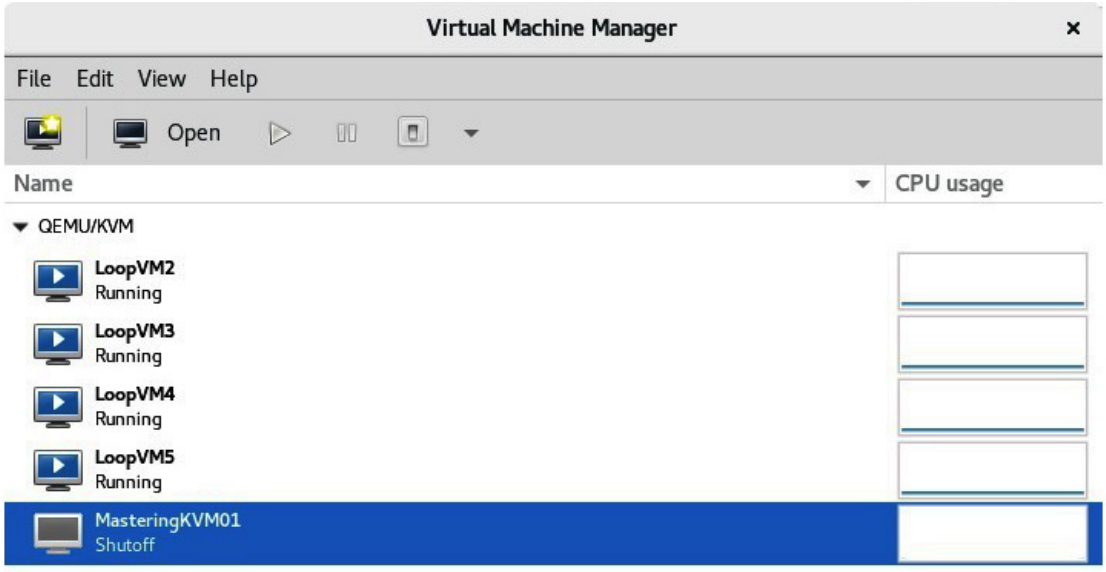

Figure 3.4 – Virtual Machine Manager

Now that we've covered the basic command-line utilities (**`virsh`** and **`virt-install`**) and have a very simple-to-use GUI application (Virtual Machine Manager), let's move away from that perspective a bit and think about what we said about oVirt and managing a lot of hosts, virtual

machines, networks, and storage devices. So, now, let's discuss how to install oVirt, which we will then use to manage our KVM-based environments in a much more centralized fashion.

Installing oVirt

There are different methods of installing oVirt. We can either deploy it as a self-hosted engine (via the Cockpit web interface or CLI) or as a standalone application via package-based installation. Let's use the second way for this example – a standalone installation in a virtual machine. We're going to split the installation into two parts:

1. Installing the oVirt engine for centralized management
2. Deploying oVirt agents on our CentOS 8-based hosts

First, let's deal with oVirt engine deployment. Deployment is simple enough, and people usually use one virtual machine for this purpose.

Keeping in mind that CentOS 8 is not supported for oVirt, in our CentOS 8 virtual machine, we need to punch in a couple of commands:

```
yum install
https://resources.ovirt.org/pub/yum-
repo/ovirt-release44.rpm
yum -y module enable javapackages-
tools pki-deps postgresql:12
yum -y update
yum -y install ovirt-engine
```

Again, this is just the installation part; we haven't done any configuration as of yet. So, that's our logical next step. We need to start a shell application called **`engine-setup`**, which is going to ask us 20 or so questions. They're rather descriptive and explanations are actually provided by the engine setup directly, so these are the settings that we've used for our testing environment (FQDN will be different in your environment):

```
Firewall manager                          : firewalld
Update Firewall                           : True
Set up Cinderlib integration              : False
Configure local Engine database           : True
Set application as default page           : True
Configure Apache SSL                      : True
Engine database host                      : localhost
Engine database port                      : 5432
Engine database secured connection        : False
Engine database host name validation      : False
Engine database name                      : engine
Engine database user name                 : engine
Engine installation                       : True
PKI organization                          : Test
Set up ovirt-provider-ovn                 : True
Grafana integration                       : True
DWH database host                         : localhost
DWH database port                         : 5432
DWH database secured connection           : False
DWH database host name validation         : False
DWH database name                         : ovirt_engine_history
Grafana database user name                : ovirt_engine_history_grafana
Configure WebSocket Proxy                 : True
DWH installation                          : True
Configure local DWH database              : True
Configure VMConsole Proxy                 : True
```

Figure 3.5 – oVirt configuration settings

After typing in **OK**, the engine setup will start. The end result should look something like this:

```
          --== SUMMARY ==--

[ INFO  ] Restarting httpd
          Please use the user 'admin@internal' and password specified in order to login
          Web access is enabled at:
              http://oVirt:80/ovirt-engine
              https://oVirt:443/ovirt-engine
          Internal CA 71:CC:F5:A2:23:0A:9E:63:0C:CC:AF:A3:96:80:75:C3:56:F7:9F:93
          SSH fingerprint: SHA256:4ZcwRZepbLKKHDva5Ww+TiEonlj5sCdRyezPdZPXaMk
          Web access for grafana is enabled at:
              https://oVirt/ovirt-engine-grafana/
          Please run the following command on the engine machine kvmsource, for SSO to work:
          systemctl restart ovirt-engine

          --== END OF SUMMARY ==--

[ INFO  ] Stage: Clean up
          Log file is located at /var/log/ovirt-engine/setup/ovirt-engine-setup-20200921000112-jld2m
2.log
[ INFO  ] Generating answer file '/var/lib/ovirt-engine/setup/answers/20200921000513-setup.conf'
[ INFO  ] Stage: Pre-termination
[ INFO  ] Stage: Termination
[ INFO  ] Execution of setup completed successfully
```

Figure 3.6 – oVirt engine setup summary

Now, we should be able to log into our oVirt engine by using a web browser and pointing it to the URL mentioned in the installation summary. During the installation process, we're asked to provide a password for the **admin@internal** user – this is the oVirt administrative user that we're

going to use to manage our environment. The oVirt web interface is simple enough to use, and for the time being, we just need to log into the Administration Portal (a link is directly available on the oVirt engine web GUI before you try to log in). After logging in, we should be greeted with the oVirt GUI:

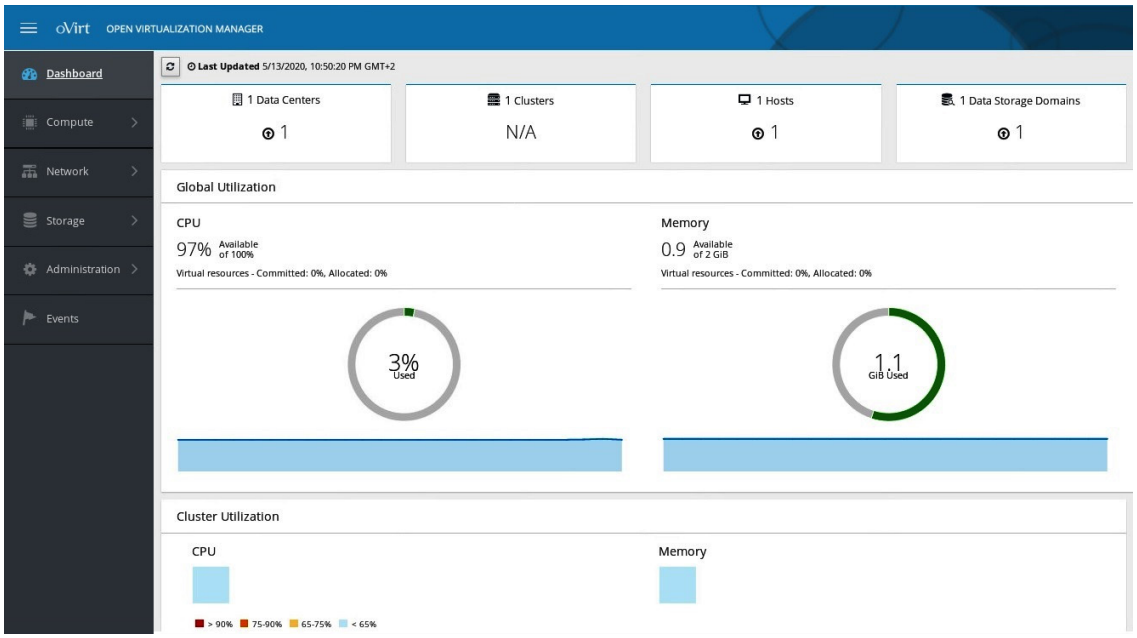

Figure 3.7 – oVirt Engine Administration Portal

We have various tabs on the left-hand side of the screen – **Dashboard**, **Compute**, **Network**, **Storage**, and **Administration** – and each and every one of these has a specific purpose:

- **Dashboard**: The default landing page. It contains the most important information, a visual representation of the state of the health of our environment, and some basic information, including the amount of virtual data centers that we're managing, clusters, hosts, data storage domains, and so on.
- **Compute**: We go to this page to manage hosts, virtual machines, templates, pools, data centers, and clusters.
- **Network**: We go to this page to manage our virtualized networks and profiles.
- **Storage**: We go to this page to manage storage resources, including disks, volumes, domains, and data centers.
- **Administration**: For the administration of users, quotas, and so on.

We will deal with many more oVirt-related operations in ***Chapter 7***, *Virtual Machine – Installation, Configuration, and Life Cycle Management*, which is all about oVirt. But for the time being, let's keep the oVirt engine up and

running so that we can come back to it later and use it for all of our day-to-day operations in our KVM-based virtualized environment.

Starting a virtual machine using QEMU and libvirt

After the deployment process, we can start managing our virtual machines. We will use **`MasteringKVM01`** and **`MasteringKVM02`** as an example. Let's start them by using the **`virsh`** command, along with the **`start`** keyword:

```
[root@packtVM01 ~]# virsh start MasteringKVM01
Domain MasteringKVM01 started

[root@packtVM01 ~]# virsh start MasteringKVM02
Domain MasteringKVM02 started
```

Figure 3.8 – Using the virsh start command

Let's say that we created all five of our virtual machines from the shell script example and that we left them powered on. We can easily check their status by issuing a simple **`virsh list`** command:

```
[root@packtVM01 ~]# virsh list
 Id    Name                           State
----------------------------------------------------
 45    LoopVM2                        running
 46    LoopVM3                        running
 47    LoopVM4                        running
 48    LoopVM5                        running
 49    LoopVM1                        running
 50    MasteringKVM01                 running
 51    MasteringKVM02                 running
```

Figure 3.9 – Using the virsh list command

If we want to gracefully shut down the **MasteringKVM01** virtual machine, we can do so by using the **virsh shutdown** command:

```
[root@packtVM01 ~]# virsh shutdown MasteringKVM01
Domain MasteringKVM01 is being shutdown
```

Figure 3.10 – Using the virsh shutdown command

If we want to forcefully shut down the **MasteringKVM02** virtual machine, we can do so by using the **virsh destroy** command:

```
[root@packtVM01 ~]# virsh destroy MasteringKVM02
Domain MasteringKVM02 destroyed
```

Figure 3.11 – Using the virsh destroy command

If we want to completely remove a virtual machine (for example, **MasteringKVM02**), you'd nor-

mally shut it down first (gracefully or forcefully) and then use the **virsh undefine** command:

```
[root@packtVM01 ~]# virsh destroy MasteringKVM02
Domain MasteringKVM02 destroyed

[root@packtVM01 ~]# virsh undefine MasteringKVM02
Domain MasteringKVM02 has been undefined
```

Figure 3.12 – Using the virsh destroy and undefine commands

Bear in mind that you can actually do **virsh undefine** first, and then **destroy**, and that the end result is going to be the same. However, that may go against the *expected behavior* in which you first shut down an object before you actually remove it.

We just learned how to use the **virsh** command to manage a virtual machine – start it and stop it – forcefully and gracefully. This will come in handy when we start extending our knowledge of using the **virsh** command in the following chapters, in which we're going to learn how to manage KVM networking and storage.

We could do all these things from the GUI as well. As you may recall, earlier in this chapter, we installed a package called **`virt-manager`**. That's actually a GUI application for managing your KVM host. Let's use that to play with our virtual machines some more. This is the basic GUI interface of **`virt-manager`**:

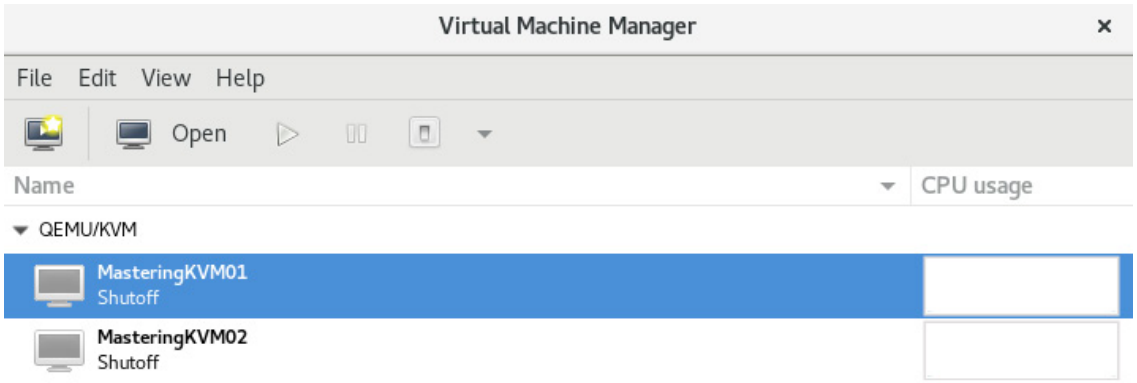

Figure 3.13 – The virt-manager GUI – we can see the list of registered virtual machines and start managing them

If we want to do our regular operations on a virtual machine – start, restart, shut down, turn off – we just need to right-click it and select that option from the menu. For all the operations to become visible, first, we must start a virtual machine; otherwise, only four actions are usable out of the available seven – **Run**, **Clone**, **Delete**,

and **Open**. The **Pause**, **Shut Down** sub-menu, and **Migrate** options will be grayed-out as they can only be used on a virtual machine that's powered on. So, after we – for example – start **`MasteringKVM01`**, the list of available options is going to get quite a bit bigger:

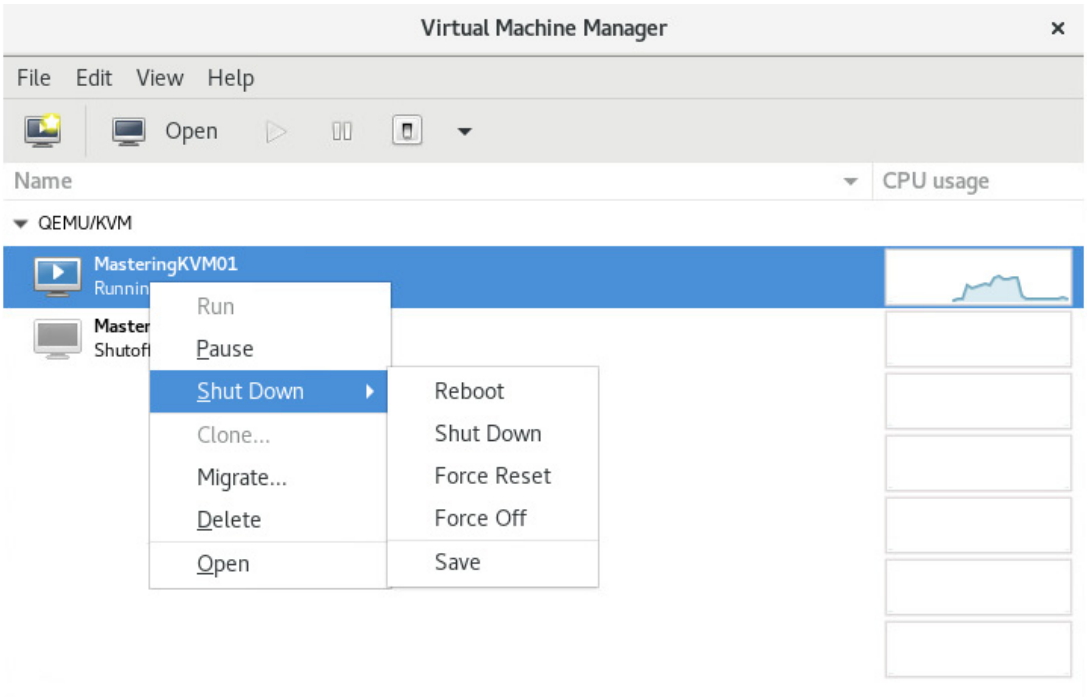

Figure 3.14 – The virt-manager options – after powering the virtual machine on, we can now use many more options

We will use **`virt-manager`** for various operations throughout this book, so make sure that you familiarize yourself with it. It is going to make our

administrative jobs quite a bit easier in many situations.

Summary

In this chapter, we laid some basic groundwork and prerequisites for practically everything that we're going to do in the remaining chapters of this book. We learned how to install KVM and a libvirt stack. We also learned how to deploy oVirt as a GUI tool to manage our KVM hosts.

The next few chapters will take us in a more technical direction as we will cover networking and storage concepts. In order to do that, we will have to take a step back and learn or review our previous knowledge about networking and storage as these are extremely important concepts for virtualization, and especially the cloud.

Questions

1. How can we validate whether our host is compatible with the KVM requirements?
2. What's the name of oVirt's default landing page?
3. Which command can we use to manage virtual machines from the command line?
4. Which command can we use to deploy virtual machines from the command line?

Further reading

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

- Kickstart Generator: **https://access.redhat.com/labs/kickstartconfig/**. Just to remind you, you need to have a RedHat support account to access this link.
- oVirt: **https://www.ovirt.org/**.