Chapter 8: Creating and Modifying VM Disks, Templates, and Snapshots

This chapter represents the end of second part of the book, in which we focused on various **`libvirt`** features—installing **Kernel-based Virtual Machine** (**KVM**) as a solution, **`libvirt`** networking and storage, virtual devices and display protocols, installing **virtual machines** (**VMs**) and configuring them... and all of that as a preparation for things that are coming in the next part of the book, which is about automation, customization, and orchestration. In order for us to be able to learn about those concepts, we must now switch our focus to VMs and their advanced operations—modifying, templating, using snapshots, and so on. Some of these topics will often be referenced later in the book, and some of these topics will be even more valuable for various business reasons in a production environment. Let's dive in and cover them.

In this chapter, we will cover the following topics:

- Modifying VM images using **`libguestfs`** tools
- VM templating
- **`virt-builder`** and **`virt-builder`** repos
- Snapshots
- Use cases and best practices while using snapshots

Modifying VM images using libguestfs tools

As our focus in this book shifts more toward scaling things out, we have to end this part of the book by introducing a stack of commands that will come in handy as we start to build bigger environments. For bigger environments, we really need various automation, customization, and orchestration tools that we will start discussing in our next chapter. But first, we have to focus on various customization utilities that we already have at our disposal. These command-line utilities will be really helpful for many different types of operations, varying from **`guestfish`** (for accessing and modifying VM files) to **`virt-p2v`** (**physical-to-virtual** (**P2V**) conversion) and **`virt-sysprep`** (to *sysprep* a VM before templating and cloning). So, let's approach the subject of these utilities in an engineering fashion—step by step.

`libguestfs` is a command-line library of utilities for working with VM disks. This library consists of roughly 30 different commands, some of which are included in the following list:

- **`guestfish`**
- **`virt-builder`**
- **`virt-builder-repository`**
- **`virt-copy-in`**
- **`virt-copy-out`**
- **`virt-customize`**
- **`virt-df`**

- **`virt-edit`**
- **`virt-filesystems`**
- **`virt-rescue`**
- **`virt-sparsify`**
- **`virt-sysprep`**
- **`virt-v2v`**
- **`virt-p2v`**

We'll start with five of the most important commands—**`virt-v2v`**, **`virt-p2v`**, **`virt-copy-in`**, **`virt-customize`**, and **`guestfish`**. We will cover **`virt-sysprep`** when we cover VM templating, and we have a separate part of this chapter dedicated to **`virt-builder`**, so we'll skip these commands for the time being.

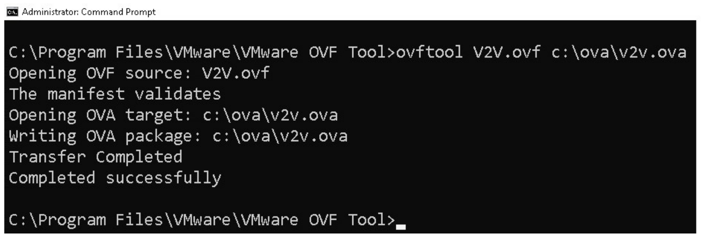
virt-v2v

Let's say that you have a Hyper-V-, Xen-, or VMware-based VM and you want to convert them to KVM, oVirt, Red Hat Enterprise Virtualization, or OpenStack. We'll just use a VMware-based VM as an example here and convert it to a KVM VM that is going to be managed by **`libvirt`** utilities. Because of some changes that were introduced in 6.0+ revisions of VMware platforms (both on the **ESX integrated (ESXi)** hypervisor side and on the vCenter server side and plugin side), it is going to be rather time-consuming to export a VM and convert it to a KVM machine—either by using a vCenter server or a ESXi host as a source. So, the simplest way to convert a VMware VM to a KVM VM would be the following:

1. Shut down the VM in the vCenter or ESXi host.

2. Export the VM as an **Open Virtualization Format** (**OVF**) template (which downloads VM files to your **Downloads** directory).
3. Install the VMware **OVFtool** utility from https://code.vmware.com/web/tool/4.3.0/ovf.
4. Move the exported VM files to the **OVFtool** installation folder.
5. Convert the VM in OVF format to **Open Virtualization Appliance** (**OVA**) format.

The reason why we need **OVFtool** for this is rather disappointing—it seems that VMware removed the option to export the OVA file directly. Luckily, **OVFtool** exists for Windows-, Linux-, and OS X-based platforms, so you'll have no trouble using it. Here's the last step of the process:



```
C:\Program Files\VMware\VMware OVF Tool>ovftool V2V.ovf c:\ova\v2v.ova
Opening OVF source: V2V.ovf
The manifest validates
Opening OVA target: c:\ova\v2v.ova
Writing OVA package: c:\ova\v2v.ova
Transfer Completed
Completed successfully

C:\Program Files\VMware\VMware OVF Tool>_
```

Figure 8.1 – Using OVFtool to convert OVF to OVA template format

After doing this, we can easily upload the **v2v.ova** file to our KVM host and type the following command into the **ova** file directory:

```
virt-v2v -i ova v2v.ova -of qcow2 -o libvirt -n default
```

The **-of** and **-o** options specify the output format (**qcow2** libvirt image), and **-n** makes sure that the VM gets connected to the default virtual network.

If you need to convert a Hyper-V VM to KVM, you can do this:

```
virt-v2v -i disk
/location/of/virtualmachinedisk.vhdx
-o local -of qcow2 -os
/var/lib/libvirt/images
```

Make sure that you specify the VM disk location correctly. The **`-o local`** and **`-os /var/lib/libvirt/images`** options make sure that the converted disk image gets saved locally, in the specified directory (the KVM default image directory).

There are other types of VM conversion processes, such as converting a physical machine to a virtual one. Let's cover that now.

virt-p2v

Now that we've covered **`virt-v2v`**, let's switch to **`virt-p2v`**. Basically, **`virt-v2v`** and **`virt-p2v`** perform a job that seems similar, but the aim of **`virt-p2v`** is to convert a *physical* machine to *VM*. Technically speaking, this is quite a bit different, as with **`virt-v2v`** we can either access a management server and hypervisor directly and convert the VM on the fly (or via an OVA template). With a physical machine, there's no management machine that can provide some kind of support or **application programming interface** (**API**) to do the conversion process. We have to *attack* the physical machine directly. In the real world of IT, this is usually done via some kind of agent or additional application.

Just as an example, if you want to convert a physical Windows machine to a VMware-based VM, you'll have to do it by installing a VMware vCenter Converter Standalone on a system that needs to be converted. Then, you'll have to select a correct mode of operation and *stream* the complete conversion process to vCenter/ESXi. It does work rather well, but—for example—RedHat's approach is a bit different. It uses a boot media to convert a physical server. So, before using this conversion process, you have to log in to the Customer Portal (located at https://access.redhat.com/downloads/content/479/ver=/rhel---8/8.0/x86_64/product-software for **Red Hat Enterprise Linux** (**RHEL**) 8.0, and you can switch versions from the menu). Then, you will have to download a correct image and use the `virt-p2v` and `virt-p2v-make-disk` utilities to create an image. But—lo and behold—the `virt-p2v-make-disk` utility uses `virt-builder`, which we will cover just a bit later in a separate part of this chapter. So, let's table this discussion for just a short while, as we will come back to it soon with full force.

As a side note, on the list of supported destinations of this command, we can use Red Hat Enterprise Virtualization, OpenStack, or KVM/`libvirt`. In terms of supported architectures, `virt-p2v` is only supported for x86_64-based platforms, and only if it is used on RHEL/CentOS 7 and 8. Keep that in mind when planning to do your P2V conversions.

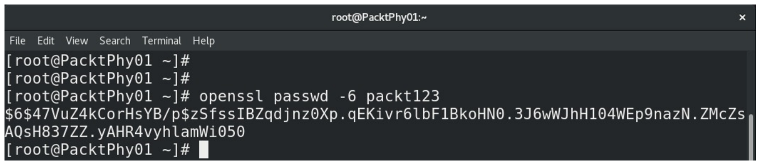
guestfish

The last utility that we want to discuss in this intro part of the chapter is called **`guestfish`**. This is a very, very important utility that enables you to do all sorts of advanced things with actual VM filesystems. We can also use it to do different types of conversion—for example, convert an **International Organization for Standardization** (**ISO**) image to **`tar.gz`**; convert a virtual disk image from an **`ext4`** filesystem to a **Logical Volume Management** (**LVM**)-backed **`ext4`** filesystem; and much more. We will show you a couple of examples of how to use it to open a VM image file and root around a bit.

The first example is a really common one—you have prepared a **`qcow2`** image with a complete VM; the guest operating system is installed; everything is configured; you're ready to copy that VM file somewhere to be reused; and... you remember that you didn't configure a root password according to some specification. Let's say that this was something that you had to do for a client, and that client has specific root password requirements for the initial root password. This makes it easier for the client—they don't need to have a password sent by you in an email; they have only one password to remember; and, after receiving the image, it will be used to create the VM. After the VM has been created and run, the root password will be changed to something—according to security practices—used by a client.

So, basically, the first example is an example of what it means to be *human*—forgetting to do something, and then wanting to repair that, but (in this case) without actually running the VM as that can change quite a few settings, especially if your **qcow2** image was created with VM templating in mind, in which case you *definitely* don't want to start that VM to repair something. More about that in the next part of this chapter.

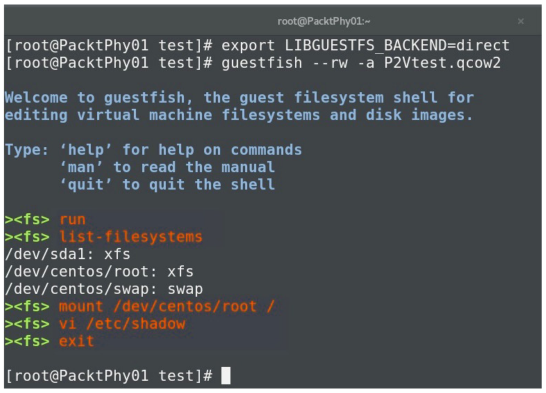
This is an ideal use case for **guestfish**. Let's say that our **qcow2** image is called **template.qcow2**. Let's change the root password to something else—for example, **packt123**. First, we need a hash for that password. The easiest way to do that would be to use **openssl** with the **-6** option (which equals SHA512 encryption), as illustrated in the following screenshot:



```
[root@PacktPhy01 ~]#
[root@PacktPhy01 ~]#
[root@PacktPhy01 ~]# openssl passwd -6 packt123
$6$47VuZ4kCorHsYB/p$zSfssIBZqdjnz0Xp.qEKivr6lbF1BkoHN0.3J6wWJhH104WEp9nazN.ZMcZs
AQsH837ZZ.yAHR4vyhlamWi050
[root@PacktPhy01 ~]#
```

Figure 8.2 – Using openssl to create an SHA512-based password hash

Now that we have our hash, we can mount and edit our image, as follows:

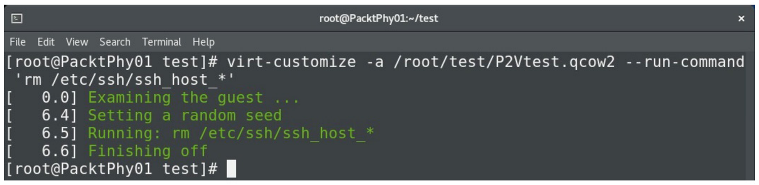


```
[root@PacktPhy01 test]# export LIBGUESTFS_BACKEND=direct
[root@PacktPhy01 test]# guestfish --rw -a P2Vtest.qcow2

Welcome to guestfish, the guest filesystem shell for
editing virtual machine filesystems and disk images.

Type: 'help' for help on commands
      'man' to read the manual
      'quit' to quit the shell

><fs> run
><fs> list-filesystems
/dev/sda1: xfs
/dev/centos/root: xfs
/dev/centos/swap: swap
><fs> mount /dev/centos/root /
><fs> vi /etc/shadow
><fs> exit

[root@PacktPhy01 test]#
```

Figure 8.3 – Using guestfish to edit the root password inside our qcow2 VM image

Shell commands that we typed in were used to get direct access to the image (without **libvirt** involvement) and to mount our image in read-write mode. Then, we started our session (**guest-fish run** command), checked which filesystems are present in the image (**list-filesystems**), and mounted the filesystem on the root folder. In the second-to-last step, we changed the root's password hash to the hash created by **openssl**. The **exit** command closes our **guestfish** session and saves changes.

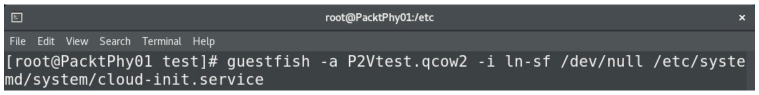
You could use a similar principle to—for example—remove forgotten **sshd** keys from the **/etc/ssh** directory, remove user **ssh** directories, and so on. The process can be seen in the following screenshot:



```
[root@PacktPhy01 test]# virt-customize -a /root/test/P2Vtest.qcow2 --run-command
 'rm /etc/ssh/ssh_host_*'
[   0.0] Examining the guest ...
[   6.4] Setting a random seed
[   6.5] Running: rm /etc/ssh/ssh_host_*
[   6.6] Finishing off
[root@PacktPhy01 test]#
```

Figure 8.4 – Using virt-customize to execute command inside a qcow2 image

The second example is also rather useful, as it involves a topic covered in the next chapter (**cloud-init**), which is often used to configure cloud VMs by manipulating the early initialization of the VM instance. Also, taking a broader view of the subject, you can use this **guestfish** example to manipulate the service configuration *inside* VM images. So, let's say that our VM image was configured so that the **cloud-init** service is started automatically. We want that service to be disabled for whatever reason—for example, to debug an error in the **cloud-init** configuration. If we didn't have the capability to manipulate **`qcow`** image content, we'd have to start that VM, use **`systemctl`** to *disable* the service, and—perhaps—do the whole procedure to reseal that VM if this was a VM template. So, let's use **`guestfish`** for the same purpose, as follows:



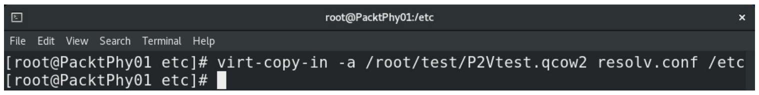
```
[root@PacktPhy01 test]# guestfish -a P2Vtest.qcow2 -i ln-sf /dev/null /etc/syste
md/system/cloud-init.service
```

Figure 8.5 – Using guestfish to disable the cloud-init service on VM startup

Important note

Be careful in this example, as normally we'd use **`ln -sf`** *with a space character between the command and options. Not so in our* **`guestfish`** *example—it needs to be used without a space.*

And lastly, let's say that we need to copy a file to our image. For example, we need to copy our local **`/etc/resolv.conf`** file to the image as we forgot to configure our **Domain Name System** (**DNS**) servers properly. We can use the **`virt-copy-in`** command for that purpose, as illustrated in the following screenshot:



```
[root@PacktPhy01 etc]# virt-copy-in -a /root/test/P2Vtest.qcow2 resolv.conf /etc
[root@PacktPhy01 etc]#
```

Figure 8.6 – Using virt-copy-in to copy a file to our image

Topics that we covered in this part of our chapter are very important for what follows next, which is a discussion about creating VM templates.

VM templating

One of the most common use cases for VMs is creating VM *templates*. So, let's say that we need to create a VM that is going to be used as a template. We use the term *template* here literally, in the same manner in which we can use templates for Word, Excel, PowerPoint, and so on, as VM templates exist for the very same reason—to have a *familiar* working environment preconfigured for us so that we don't need to start from scratch. In the case of VM templates, we're talking about *not installing a VM guest operating system from scratch*, which is a huge time-saver. Imagine getting a task to deploy 500 VMs for some kind of testing environment to test how something works when scaled out. You'd lose weeks doing that from scratch, even allowing for the fact that you can do installations in parallel.

VMs need to be looked at as *objects*, and they have certain *properties* or *attributes*. From the *outside* perspective (meaning, from the perspective of `libvirt`), a VM has a name, a virtual disk, a virtual **central processing unit** (**CPU**) and memory configuration, connectivity to a virtual switch, and so on. We covered this subject in *Chapter 7*, *VM: Installation, Configuration, and Life Cycle Management*. That being said, we didn't touch the subject of *inside* a VM. From that perspective (basically, from the guest operating system perspective), a VM also has certain properties—installed guest operating system version, **Internet Protocol** (**IP**) configuration, **virtual local area network** (**VLAN**) configuration… After that, it depends on which operating system the family VM is based. We thus need to consider the following:

- If we're talking about Microsoft Windows-based VMs, we have to consider service and software configuration, registry configuration, and license configuration.
- If we're talking about Linux-based VMs, we have to consider service and software configuration, **Secure Shell** (**SSH**) key configuration, license configuration, and so on.

It can be even more specific than that. For example, preparing a template for Ubuntu-based VMs is different from preparing a template for CentOS 8-based VMs. And to create these templates properly, we need to learn some basic procedures that we can then use repetitively every single time when creating a VM template.

Consider this example: suppose you wish to create four Apache web servers to host your web applications. Normally, with the traditional manual installation method, you would first have to create four VMs with specific hardware configurations, install an operating system on each of them one by one, and then download and install the required Apache packages using `yum` or some other software installation method. This is a time-consuming job, as you will be mostly doing repetitive work. But with a template approach, it can be done in considerably less time. How? Because you will bypass operating system installation and other configuration tasks and directly spawn VMs from a template that consists of a preconfigured operating system image, containing all the required web server packages ready for use.

The following screenshot shows the steps involved in the manual installation method. You can clearly see that *Steps 2-5* are just repetitive tasks performed across all four VMs, and they would have taken up most of the time required to get your Apache web servers ready:

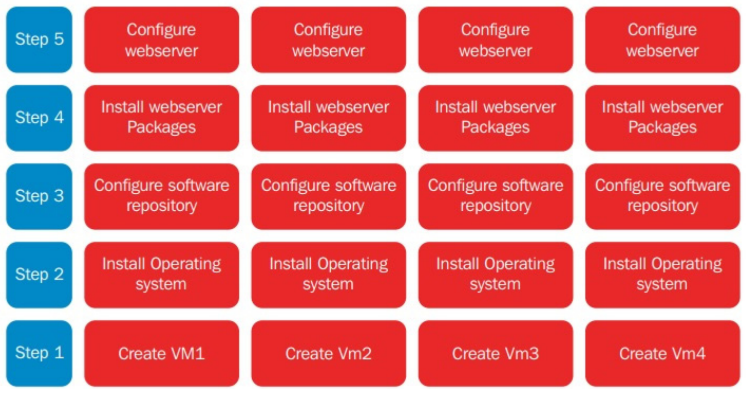


Figure 8.7 – Installing four Apache web servers without VM templates

Now, see how the number of steps is drastically reduced by simply following *Steps 1-5* once, creating a template, and then using it to deploy four identical VMs. This will save you a lot of time. You can see the difference in the following diagram:

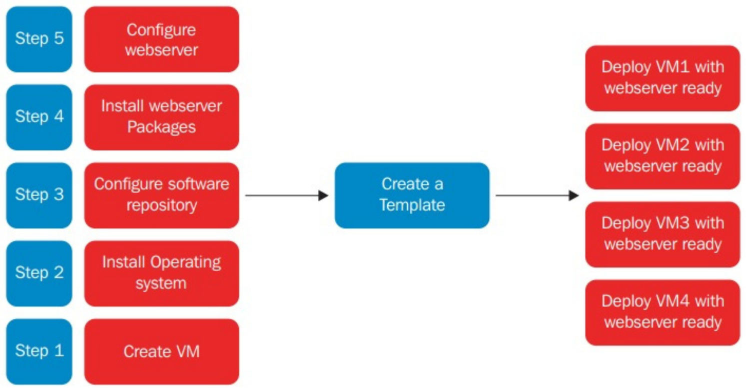

Step 5
Configure webserver
Step 4
Install webserver Packages
Step 3
Configure software repository
Step 2
Install Operating system
Step 1
Create VM
Create a Template
Deploy VM1 with webserver ready
Deploy VM2 with webserver ready
Deploy VM3 with webserver ready
Deploy VM4 with webserver ready

Figure 8.8 – Installing four Apache web servers by using VM templates

This isn't the whole story, though. There are different ways of actually going from *Step 3* to *Step 4* (from **Create a Template** to deployment of VM1-4), which either includes a full cloning process or a linked cloning process, detailed here:

- **Full clone**: A VM deployed using the full cloning mechanism creates a complete copy of the VM, the problem being that it's going to use the same amount of capacity as the original VM.
- **Linked clone**: A VM deployed using the thin cloning mechanism uses the template image as a base image in read-only mode and links an additional **copy-on-write (COW)** image to store newly generated data. This provisioning method is *heavily* used in cloud and **Virtual Desktop Infrastructure** (**VDI**) environments as it saves a lot of disk space. Remember that fast storage capacity is something that's really expensive, so any kind of optimization in this respect will be a big money saver. Linked clones will also have an impact on performance, as we will discuss a bit later.

Now, let's see how the templates work.

Working with templates

In this section, you will learn how to create templates of Windows and Linux VMs using the `virt-clone` option available in `virt-manager`. Although the `virt-clone` utility was not originally intended for creating templates, when used with `virt-sysprep` and other operating system sealing utilities, it serves that purpose. Be aware that there is a difference between a clone and a master image. A clone image is just a VM, and a master image is a VM copy that can be used for deployment of hundreds and thousands of new VMs.

Creating templates

Templates are created by converting a VM into a template. This is actually a three-step procedure that includes the following:

1. Installing and customizing the VM, with all the desired software, which will become the template or base image.
2. Removing all system-specific properties to ensure VM uniqueness—we need to take care of SSH host keys, network configuration, user accounts, **media access control** (**MAC**) address, license information, and so on.
3. Mark the VM as a template by renaming it with a template as a prefix. Some virtualization technologies have special VM file types for this (for example, a VMware `.vmtx` file), which effectively means that you don't have to rename a VM to mark it as a template.

To understand the actual procedure, let's create two templates and deploy a VM from them. Our two templates are going to be the following:

- A CentOS 8 VM with a complete **Linux, Apache, MySQL, and PHP** (**LAMP**) stack
- A Windows Server 2019 VM with SQL Server Express

Let's go ahead and create these templates.

Example 1 – Preparing a CentOS 8 template with a complete LAMP stack

Installation of CentOS should be a familiar theme to us by now, so we're just going to focus on the *AMP* part of the LAMP stack and the templating part. So, our procedure is going to look like this:

1. Create a VM and install CentOS 8 on it, using the installation method that you prefer. Keep it minimal as this VM will be used as the base for the template that is being created for this example.
2. SSH into or take control of the VM and install the LAMP stack. Here's a script for you to install everything needed for a LAMP stack on CentOS 8, after the operating system installation has been done. Let's start with the package installation, as follows:

   ```
   yum -y update
   yum -y install httpd httpd-tools
   mod_ssl
   systemctl start httpd
   systemctl enable httpd
   yum -y install mariadb-server
   mariadb
   yum install -y php php-fpm php-
   mysqlnd php-opcache php-gd php-xml
   php-mbstring libguestfs*
   ```

   After we're done with the software installation, let's do a bit of service configuration—start all the necessary services and enable them, and reconfigure the firewall to allow connections, as follows:

```
systemctl start mariadb
systemctl enable mariadb
systemctl start php-fpm
systemctl enable php-fpm
firewall-cmd --permanent --
zone=public --add-service=http
firewall-cmd --permanent --
zone=public --add-service=https
systemctl reload firewalld
```

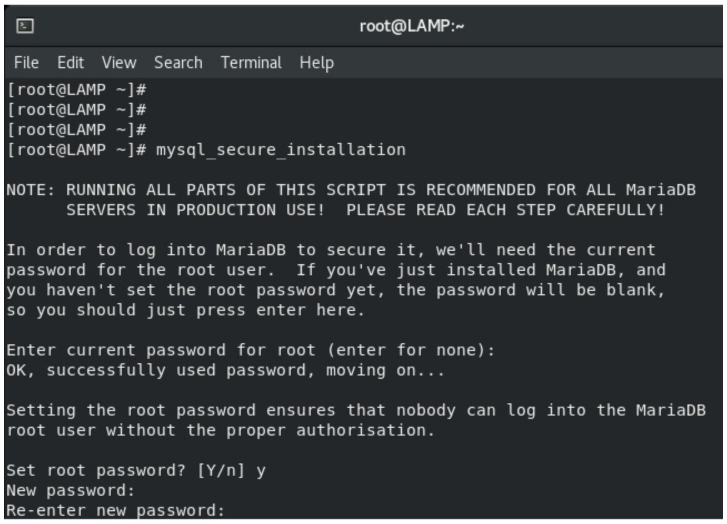
We also need to configure some security settings related to directory ownership—for example, **Security-Enhanced Linux** (**SELinux**) configuration for the Apache web server. Let's do that next, like this:

```
chown apache:apache /var/www/html -
R
semanage fcontext -a -t
httpd_sys_content_t
"/var/www/html(/.*)?"
restorecon -vvFR /var/www/html
setsebool -P httpd_execmem 1
```

3. After this has been done, we need to configure MariaDB, as we have to set some kind of MariaDB root password for the database administrative user and configure basic settings. This is usually done via a **`mysql_secure_installation`** script provided by MariaDB packages. So, that is our next step, as illustrated in the following code snippet:

```
mysql_secure_installation
```

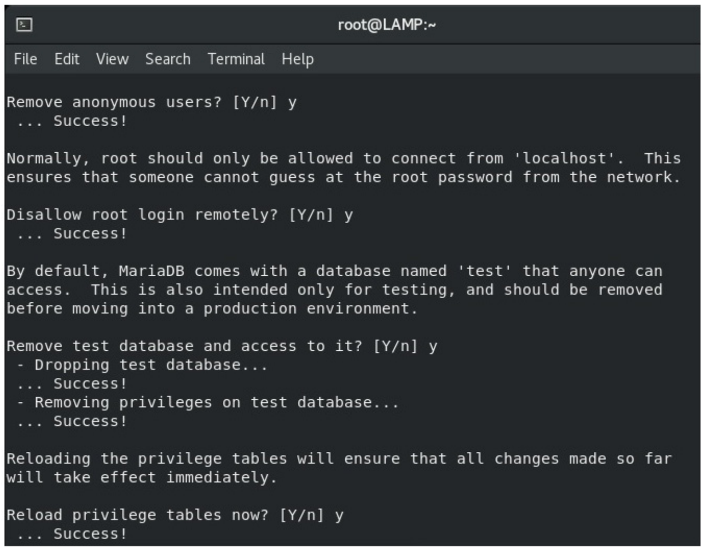
After we start the **`mysql_secure_installation`** script, it is going to ask us a series of questions, as illustrated in the following screenshot:



```
[root@LAMP ~]#
[root@LAMP ~]#
[root@LAMP ~]#
[root@LAMP ~]# mysql_secure_installation

NOTE: RUNNING ALL PARTS OF THIS SCRIPT IS RECOMMENDED FOR ALL MariaDB
      SERVERS IN PRODUCTION USE!  PLEASE READ EACH STEP CAREFULLY!

In order to log into MariaDB to secure it, we'll need the current
password for the root user.  If you've just installed MariaDB, and
you haven't set the root password yet, the password will be blank,
so you should just press enter here.

Enter current password for root (enter for none):
OK, successfully used password, moving on...

Setting the root password ensures that nobody can log into the MariaDB
root user without the proper authorisation.

Set root password? [Y/n] y
New password:
Re-enter new password:
```

Figure 8.9 – First part of MariaDB setup: assigning a root password that is empty after installation

After assigning a root password for the MariaDB database, the next steps are more related to housekeeping—removing anonymous users, disallowing remote login, and so on. Here's what that part of wizard looks like:



```
Remove anonymous users? [Y/n] y
 ... Success!

Normally, root should only be allowed to connect from 'localhost'.  This
ensures that someone cannot guess at the root password from the network.

Disallow root login remotely? [Y/n] y
 ... Success!

By default, MariaDB comes with a database named 'test' that anyone can
access.  This is also intended only for testing, and should be removed
before moving into a production environment.

Remove test database and access to it? [Y/n] y
 - Dropping test database...
 ... Success!
 - Removing privileges on test database...
 ... Success!

Reloading the privilege tables will ensure that all changes made so far
will take effect immediately.

Reload privilege tables now? [Y/n] y
 ... Success!
```

Figure 8.10 – Housekeeping: anonymous users, root login setup, test database data removal

We installed all the necessary services—Apache, MariaDB—and all the necessary additional packages (PHP, **FastCGI Process Manager** (**FPM**)), so this VM is ready for templating. We could also introduce some kind of content to the Apache web server (create a **`sample index.html`** file and place it in **`/var/www/html`**), but we're not going to do that right now. In production environments, we'd just copy web page contents to that directory and be done with it.

4. Now that the required LAMP settings are configured the way we want them, shut down the VM and run the **`virt-sysprep`** command to seal it. If you want to *expire* the root password (translation—force a change of the root password on the next login), type in the following command:

```
passwd --expire root
```

Our test VM is called LAMP and the host is called **`PacktTemplate`**, so here are the necessary steps, presented via a one-line command:

```
virsh shutdown LAMP; sleep 10; virsh
list
```

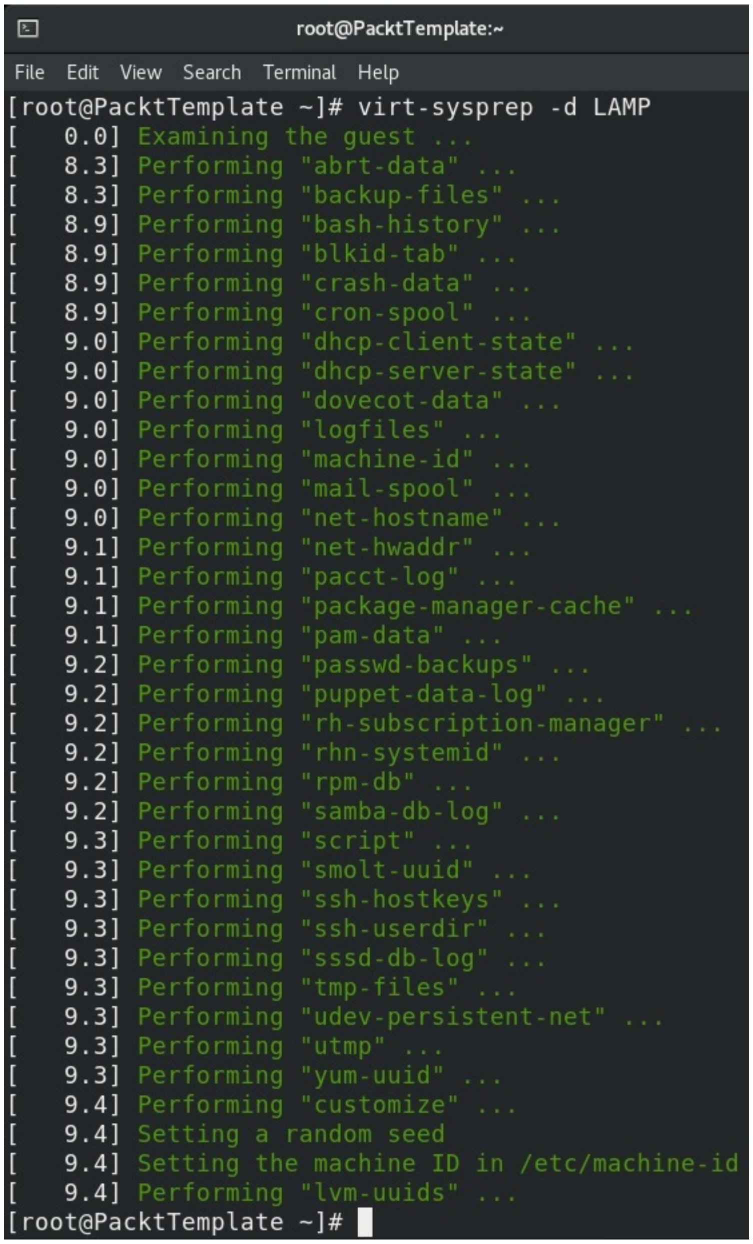
Our LAMP VM is now ready to be reconfigured as template. For that, we will use the **`virt-sysprep`** command.

What is virt-sysprep?

This is a command-line utility provided by the **`libguestfs-tools-c`** package to ease the sealing and generalizing procedure of Linux VM. It prepares a Linux VM to become a template or clone by removing system-specific information automatically so that clones can be made from it. **`virt-sysprep`** can be used to add some additional configuration bits and pieces—such as users, groups, SSH keys, and so on.

There are two ways to invoke `virt-sysprep` against a Linux VM: using the `-d` or `-a` option. The first option points to the intended guest using its name or **universally unique identifier (UUID)**, and the second one points to a particular disk image. This gives us the flexibility to use the `virt-sysprep` command even if the guest is not defined in `libvirt`.

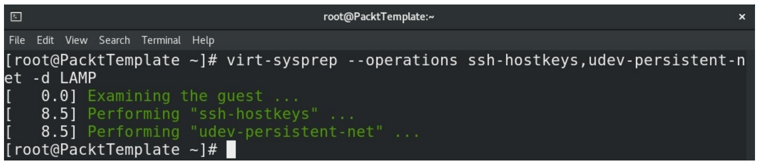
Once the `virt-sysprep` command is executed, it performs a bunch of `sysprep` operations that make the VM image clean by removing system-specific information from it. Add the `--verbose` option to the command if you are interested in knowing how this command works in the background. The process can be seen in the following screenshot:



```
[root@PacktTemplate ~]# virt-sysprep -d LAMP
[   0.0] Examining the guest ...
[   8.3] Performing "abrt-data" ...
[   8.3] Performing "backup-files" ...
[   8.9] Performing "bash-history" ...
[   8.9] Performing "blkid-tab" ...
[   8.9] Performing "crash-data" ...
[   8.9] Performing "cron-spool" ...
[   9.0] Performing "dhcp-client-state" ...
[   9.0] Performing "dhcp-server-state" ...
[   9.0] Performing "dovecot-data" ...
[   9.0] Performing "logfiles" ...
[   9.0] Performing "machine-id" ...
[   9.0] Performing "mail-spool" ...
[   9.0] Performing "net-hostname" ...
[   9.1] Performing "net-hwaddr" ...
[   9.1] Performing "pacct-log" ...
[   9.1] Performing "package-manager-cache" ...
[   9.1] Performing "pam-data" ...
[   9.2] Performing "passwd-backups" ...
[   9.2] Performing "puppet-data-log" ...
[   9.2] Performing "rh-subscription-manager" ...
[   9.2] Performing "rhn-systemid" ...
[   9.2] Performing "rpm-db" ...
[   9.2] Performing "samba-db-log" ...
[   9.3] Performing "script" ...
[   9.3] Performing "smolt-uuid" ...
[   9.3] Performing "ssh-hostkeys" ...
[   9.3] Performing "ssh-userdir" ...
[   9.3] Performing "sssd-db-log" ...
[   9.3] Performing "tmp-files" ...
[   9.3] Performing "udev-persistent-net" ...
[   9.3] Performing "utmp" ...
[   9.3] Performing "yum-uuid" ...
[   9.4] Performing "customize" ...
[   9.4] Setting a random seed
[   9.4] Setting the machine ID in /etc/machine-id
[   9.4] Performing "lvm-uuids" ...
[root@PacktTemplate ~]# 
```

Figure 8.11 – virt-sysprep works its magic on the VM

By default, **`virt-sysprep`** performs more than 30 operations. You can also choose which specific sysprep operations you want to use. To get a list of all the available operations, run the **`virt-sysprep --list-operation`** command. The default operations are marked with an asterisk. You can change the default operations using the **`--operations`** switch, followed by a comma-separated list of operations that you want to use. See the following example:



```
[root@PacktTemplate ~]# virt-sysprep --operations ssh-hostkeys,udev-persistent-n
et -d LAMP
[   0.0] Examining the guest ...
[   8.5] Performing "ssh-hostkeys" ...
[   8.5] Performing "udev-persistent-net" ...
[root@PacktTemplate ~]#
```

Figure 8.12 – Using virt-sysprep to customize operations to be done on a template VM

Notice that this time, it only performed the **`ssh-hostkeys`** and **`udev-persistentnet`** operations instead of the typical operations. It's up to you how much cleaning you would like to undertake in the template.

Now, we can mark this VM as a template by adding the word *template* as a prefix in its name. You can even undefine the VM from **`libvirt`** after taking a backup of its **Extensible Markup Language** (**XML**) file.

Important note

Make sure that from now on, this VM is never started; otherwise, it will lose all sysprep operations and can even cause problems with VMs deployed using the thin method.

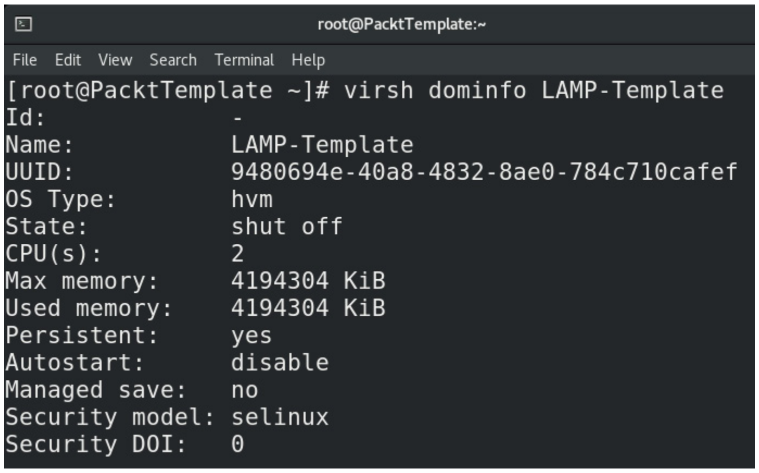
In order to rename a VM, use **`virsh domrename`** as root, like this:

```
# virsh domrename LAMP LAMP-Template
```

`LAMP-Template`, our template, is now ready to be used for future cloning processes. You can check its settings by using the following command:

```
# virsh dominfo LAMP-Template
```

The end result should be something like this:



```
[root@PacktTemplate ~]# virsh dominfo LAMP-Template
Id:             -
Name:           LAMP-Template
UUID:           9480694e-40a8-4832-8ae0-784c710cafef
OS Type:        hvm
State:          shut off
CPU(s):         2
Max memory:     4194304 KiB
Used memory:    4194304 KiB
Persistent:     yes
Autostart:      disable
Managed save:   no
Security model: selinux
Security DOI:   0
```

Figure 8.13 – virsh dominfo on our template VM

The next example is going to be about preparing a Windows Server 2019 template with a pre-installed Microsoft **Structured Query Language** (**SQL**) database—a common use case that many of us will need to use in our environments. Let's see how we can do that.

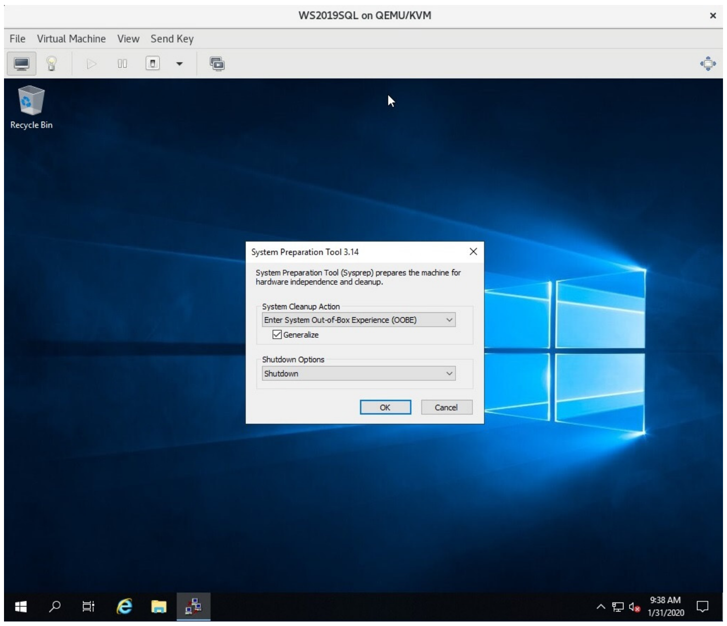
Example 2 – Preparing a Windows Server 2019 template with a Microsoft SQL database

`virt-sysprep` does not work for Windows guests, and there is little chance support will be added any time soon. So, in order to generalize a Windows machine, we would have to access the Windows system and directly run `sysprep`.

The **System Preparation** (`sysprep`) tool is a native Windows utility for removing system-specific data from Windows images. To know more about this utility, refer to this article: https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/sysprep--generalize--a-windows-installation.

Our template preparation process is going to work like this:

1. Create a VM and install the Windows Server 2019 operating system on it. Our VM is going to be called **`WS2019SQL`**.
2. Install the Microsoft SQL Express software and, once it's configured the way you want, restart the VM and launch the **`sysprep`** application. The **`.exe`** file of **`sysprep`** is present in the **`C:\Windows\System32\sysprep`** directory. Navigate there by entering **`sysprep`** in the run box and double-click on **`sysprep.exe`**.
3. Under **System Cleanup Action**, select **Enter System Out-of-Box Experience (OOBE)** and click on the **Generalize** checkbox if you want to do **system identification number** (**SID**) regeneration, as illustrated in the following screenshot:



System Preparation Tool 3.14

System Preparation Tool (Sysprep) prepares the machine for hardware independence and cleanup.

System Cleanup Action: Enter System Out-of-Box Experience (OOBE)

[✓] Generalize

Shutdown Options: Shutdown

OK Cancel

Figure 8.14 – Be careful with sysprep options; OOBE, generalize, and shutdown options are highly recommended

4. Under **Shutdown Options**, select **Shutdown** and click on the **OK** button. The **`sysprep`** process will start after that, and when it's done, it will be shut down.

5. Rename the VM using the same procedure we used on the LAMP template, as follows:

   ```
   # virsh domrename WS2019SQL
   WS2019SQL-Template
   ```

Again, we can use the **dominfo** option to check basic information about our newly created template, as follows:

```
# virsh dominfo WS2019SQL-Template
```

Important note

Be careful when updating templates in the future—you need to run them, update them, and reseal them. With Linux distributions, you won't have many issues doing that. But serializing Microsoft Windows **sysprep** *(start template VM, update,* **sysprep***, and repeating that in the future) will get you to a situation in which* **sysprep** *will throw you an error. So, there's another school of thought that you can use here. You can do the whole procedure as we did it in this part of our chapter, but don't* **sysprep** *it. That way, you can easily update the VM, then clone it, and then* **sysprep** *it. It will save you a lot of time.*

Next, we will see how to deploy VMs from a template.

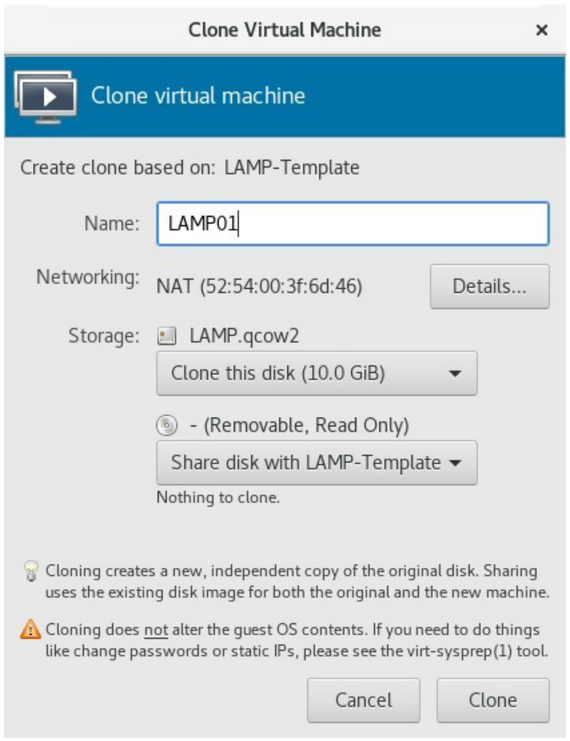
Deploying VMs from a template

In the previous section, we created two template images; the first template image is still defined in **libvirt** as **VM** and named **LAMP-Template**, and the second is called **WS2019SQL-Template**. We will now use these two VM templates to deploy new VMs from them.

Deploying VMs using full cloning

Perform the following steps to deploy the VM using clone provisioning:

1. Open the VM Manager (**`virt-manager`**), and then select the **`LAMP-Template`** VM. Right-click on it and select the **Clone** option, which will open the **Clone Virtual Machine** window, as illustrated in the following screenshot:



Create clone based on: LAMP-Template

Name: LAMP01

Networking: NAT (52:54:00:3f:6d:46)

Storage: LAMP.qcow2

Clone this disk (10.0 GiB)

- (Removable, Read Only)

Share disk with LAMP-Template

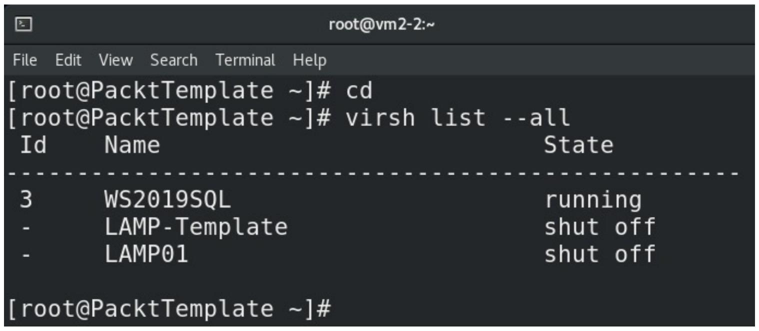
Nothing to clone.

Cloning creates a new, independent copy of the original disk. Sharing uses the existing disk image for both the original and the new machine.

Cloning does not alter the guest OS contents. If you need to do things like change passwords or static IPs, please see the virt-sysprep(1) tool.

Figure 8.15 – Cloning a VM from VM Manager

2. Provide a name for the resulting VM and skip all other options. Click on the **Clone** button to start the deployment. Wait till the cloning operation finishes.
3. Once it's finished, your newly deployed VM is ready to use and you can start using it. You can see the output from the process in the following screenshot:



```
[root@PacktTemplate ~]# cd
[root@PacktTemplate ~]# virsh list --all
 Id    Name                           State
----------------------------------------------------
 3     WS2019SQL                      running
 -     LAMP-Template                  shut off
 -     LAMP01                         shut off

[root@PacktTemplate ~]#
```

Figure 8.16 – Full clone (LAMP01) has been created

As a result of our previous operation, the **LAMP01** VM was deployed from **LAMP-Template**, but as we used the full cloning method, they are independent, and even if you remove **LAMP-Template**, they will operate just fine.

We can also use linked cloning, which will save us a whole lot of disk space by creating a VM that's anchored to a base image. Let's do that next.

Deploying VMs using linked cloning

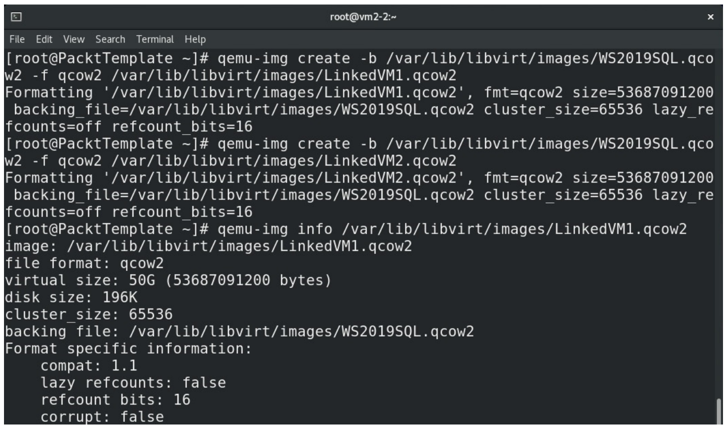
Perform the following steps to get started with VM deployment using the linked cloning method:

1. Create two new **qcow2** images using **/var/lib/libvirt/images/WS2019SQL.qcow2** as the backing file, like this:

```
# qemu-img create -b
/var/lib/libvirt/images/WS2019SQL.q
cow2 -f qcow2
/var/lib/libvirt/images/LinkedVM1.q
cow2
# qemu-img create -b
/var/lib/libvirt/images/WS2019SQL.q
cow2 -f qcow2
```

```
/var/lib/libvirt/images/LinkedVM2.q
cow2
```

2. Verify that the backing file attribute for the newly created **qcow2** images is pointing correctly to the **/var/lib/libvirt/images/WS2019SQL.qcow2** image, using the **qemu-img** command. The end result of these three procedures should look like this:



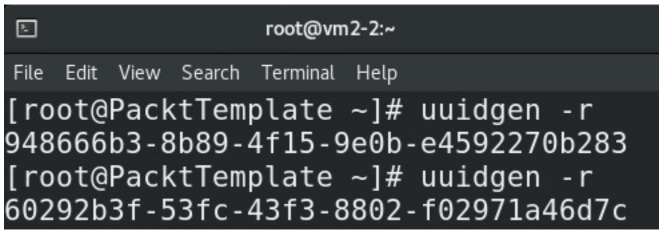
```
[root@PacktTemplate ~]# qemu-img create -b /var/lib/libvirt/images/WS2019SQL.qco
w2 -f qcow2 /var/lib/libvirt/images/LinkedVM1.qcow2
Formatting '/var/lib/libvirt/images/LinkedVM1.qcow2', fmt=qcow2 size=53687091200
 backing_file=/var/lib/libvirt/images/WS2019SQL.qcow2 cluster_size=65536 lazy_re
fcounts=off refcount_bits=16
[root@PacktTemplate ~]# qemu-img create -b /var/lib/libvirt/images/WS2019SQL.qco
w2 -f qcow2 /var/lib/libvirt/images/LinkedVM2.qcow2
Formatting '/var/lib/libvirt/images/LinkedVM2.qcow2', fmt=qcow2 size=53687091200
 backing_file=/var/lib/libvirt/images/WS2019SQL.qcow2 cluster_size=65536 lazy_re
fcounts=off refcount_bits=16
[root@PacktTemplate ~]# qemu-img info /var/lib/libvirt/images/LinkedVM1.qcow2
image: /var/lib/libvirt/images/LinkedVM1.qcow2
file format: qcow2
virtual size: 50G (53687091200 bytes)
disk size: 196K
cluster_size: 65536
backing file: /var/lib/libvirt/images/WS2019SQL.qcow2
Format specific information:
    compat: 1.1
    lazy refcounts: false
    refcount bits: 16
    corrupt: false
```

Figure 8.17 – Creating a linked clone image

3. Let's now dump the template VM configuration to two XML files by using the **virsh** command. We're doing this twice so that we have two VM definitions. We will import them as two new VMs after we change a couple of parameters, as follows:

```
virsh dumpxml WS2019SQL-Template >
/root/SQL1.xml
virsh dumpxml WS2019SQL-Template >
/root/SQL2.xml
```

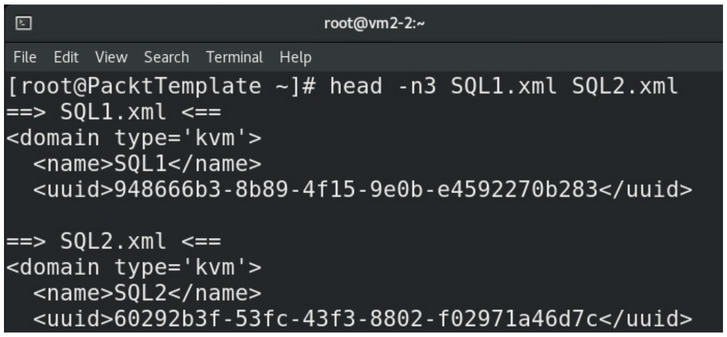
4. By using the **uuidgen -r** command, generate two random UUIDs. We will need them for our VMs. The process can be seen in the following screenshot:



```
[root@PacktTemplate ~]# uuidgen -r
948666b3-8b89-4f15-9e0b-e4592270b283
[root@PacktTemplate ~]# uuidgen -r
60292b3f-53fc-43f3-8802-f02971a46d7c
```

Figure 8.18 – Generating two new UUIDs for our VMs

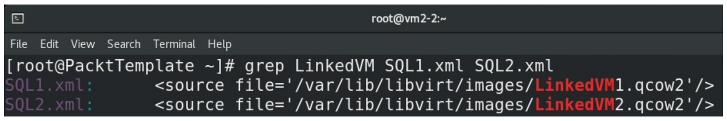
5. Edit the **SQL1.xml** and **SQL2.xml** files by assigning them new VM names and UUIDs. This step is mandatory as VMs have to have unique names and UUIDs. Let's change the name in the first XML file to **SQL1**, and the name in the second XML file to **SQL2**. We can achieve that by changing the **<name></name>** statement. Then, copy and paste the UUIDs that we created with the **uuidgen** command in the **SQL1.xml** and **SQL2.xml** **<uuid></uuid>** statement. So, relevant entries for those two lines in our configuration files should look like this:



```
[root@PacktTemplate ~]# head -n3 SQL1.xml SQL2.xml
==> SQL1.xml <==
<domain type='kvm'>
  <name>SQL1</name>
  <uuid>948666b3-8b89-4f15-9e0b-e4592270b283</uuid>

==> SQL2.xml <==
<domain type='kvm'>
  <name>SQL2</name>
  <uuid>60292b3f-53fc-43f3-8802-f02971a46d7c</uuid>
```

Figure 8.19 – Changing the VM name and UUID in their respective XML configuration files

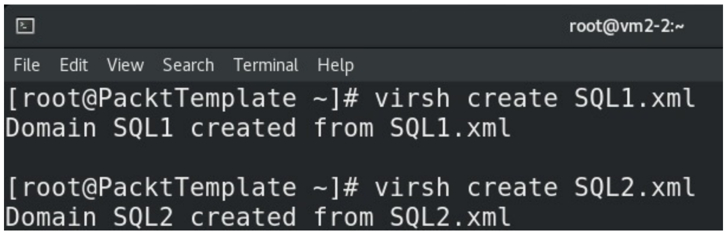
6. We need to change the virtual disk location in our **SQL1** and **SQL2** image files. Find entries for **.qcow2** files later in these configuration files and change them so that they use the absolute path of files that we created in *Step 1*, as follows:



```
[root@PacktTemplate ~]# grep LinkedVM SQL1.xml SQL2.xml
SQL1.xml:      <source file='/var/lib/libvirt/images/LinkedVM1.qcow2'/>
SQL2.xml:      <source file='/var/lib/libvirt/images/LinkedVM2.qcow2'/>
```

Figure 8.20 – Changing the VM image location so that it points to newly created linked clone images

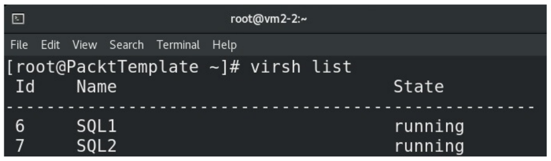
7. Now, import these two XML files as VM definitions by using the **`virsh`** create command, as follows:



```
[root@PacktTemplate ~]# virsh create SQL1.xml
Domain SQL1 created from SQL1.xml

[root@PacktTemplate ~]# virsh create SQL2.xml
Domain SQL2 created from SQL2.xml
```

Figure 8.21 – Creating two new VMs from XML definition files

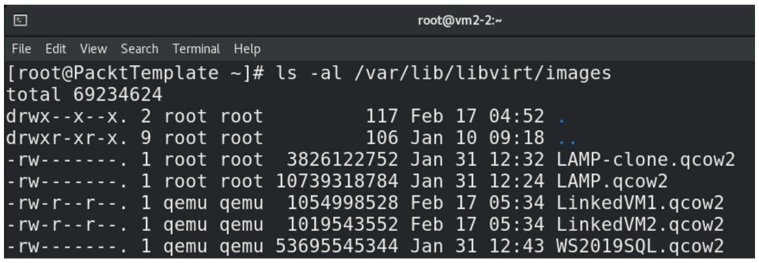
8. Use the **`virsh`** command to verify if they are defined and running, as follows:



```
[root@PacktTemplate ~]# virsh list
 Id    Name                           State
----------------------------------------------------
 6     SQL1                           running
 7     SQL2                           running
```

Figure 8.22 – Two new VMs up and running

9. The VMs are already started, so we can now check the end result of our linked cloning process. Our two virtual disks for these two VMs should be rather small, as they're both using the same base image. Let's check the guest disk image size—notice in the following screenshot that both **`LinkedVM1.qcow`** and **`LinkedVM2.qcow`** files are roughly 50 times smaller than their base image:



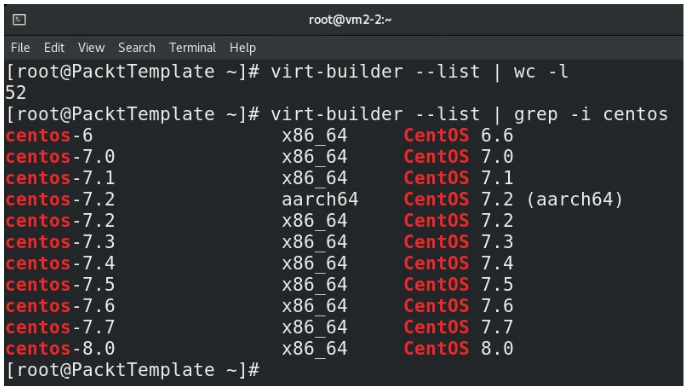
```
[root@PacktTemplate ~]# ls -al /var/lib/libvirt/images
total 69234624
drwx--x--x. 2 root root         117 Feb 17 04:52 .
drwxr-xr-x. 9 root root         106 Jan 10 09:18 ..
-rw-------. 1 root root  3826122752 Jan 31 12:32 LAMP-clone.qcow2
-rw-------. 1 root root 10739318784 Jan 31 12:24 LAMP.qcow2
-rw-r--r--. 1 qemu qemu  1054998528 Feb 17 05:34 LinkedVM1.qcow2
-rw-r--r--. 1 qemu qemu  1019543552 Feb 17 05:34 LinkedVM2.qcow2
-rw-------. 1 qemu qemu 53695545344 Jan 31 12:43 WS2019SQL.qcow2
```

Figure 8.23 – Result of linked clone deployment: base image, small delta images

This should provide plenty of examples and info about using the linked cloning process. Don't take it too far (many linked clones on a single base image) and you should be fine. But now, it's time to move to our next topic, which is about **virt-builder**. The **virt-builder** concept is very important if you want to deploy your VMs quickly – that is, without actually installing them. We can use **virt-builder** repos for that. Let's learn how to do that next.

virt-builder and virt-builder repos

One of the most essential tools in the **libguestfs** package is **virt-builder**. Let's say that you *really* don't want to build a VM from scratch, either because you don't have the time or you just cannot be bothered. We will use CentOS 8 for this example, although the list of supported distributions is now roughly 50 (distributions and their sub-versions), as you can see in the following screenshot:



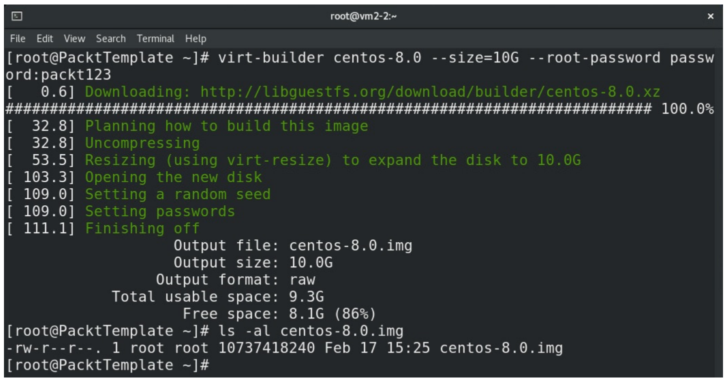
```
[root@PacktTemplate ~]# virt-builder --list | wc -l
52
[root@PacktTemplate ~]# virt-builder --list | grep -i centos
centos-6                 x86_64     CentOS 6.6
centos-7.0               x86_64     CentOS 7.0
centos-7.1               x86_64     CentOS 7.1
centos-7.2               aarch64    CentOS 7.2 (aarch64)
centos-7.2               x86_64     CentOS 7.2
centos-7.3               x86_64     CentOS 7.3
centos-7.4               x86_64     CentOS 7.4
centos-7.5               x86_64     CentOS 7.5
centos-7.6               x86_64     CentOS 7.6
centos-7.7               x86_64     CentOS 7.7
centos-8.0               x86_64     CentOS 8.0
[root@PacktTemplate ~]#
```

Figure 8.24 – virt-builder supported OSes, and CentOS distributions

In our test scenario, we need to create a CentOS 8 image as soon as possible, and create a VM out of that image. All of the ways of deploying VMs so far have been based on the idea of installing them from scratch, or cloning, or templating. These are either *start-from-zero* or *deploy-first-template-or-template-second-provision-later* types of mechanisms. What if there's another way?

`virt-builder` provides us with a way of doing just that. By issuing a couple of simple commands, we can import a CentOS 8 image, import it to KVM, and start it. Let's proceed, as follows:

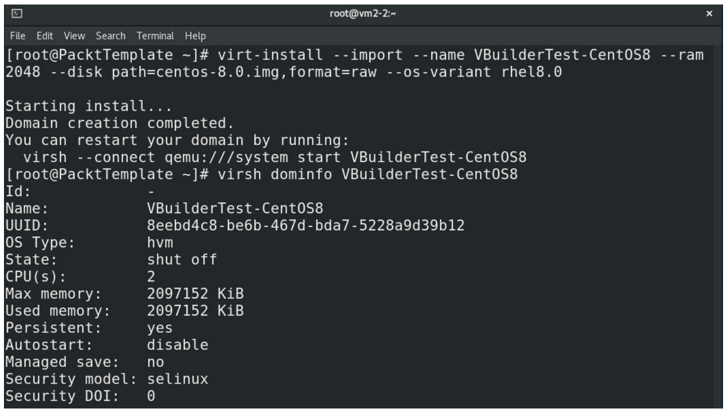
1. First, let's use **`virt-builder`** to download a CentOS 8 image with specified parameters, as follows:



```
[root@PacktTemplate ~]# virt-builder centos-8.0 --size=10G --root-password passw
ord:packt123
[   0.6] Downloading: http://libguestfs.org/download/builder/centos-8.0.xz
######################################################################### 100.0%
[  32.8] Planning how to build this image
[  32.8] Uncompressing
[  53.5] Resizing (using virt-resize) to expand the disk to 10.0G
[ 103.3] Opening the new disk
[ 109.0] Setting a random seed
[ 109.0] Setting passwords
[ 111.1] Finishing off
                   Output file: centos-8.0.img
                   Output size: 10.0G
                 Output format: raw
            Total usable space: 9.3G
                    Free space: 8.1G (86%)
[root@PacktTemplate ~]# ls -al centos-8.0.img
-rw-r--r--. 1 root root 10737418240 Feb 17 15:25 centos-8.0.img
[root@PacktTemplate ~]#
```

Figure 8.25 – Using virt-builder to grab a CentOS 8.0 image and check its size

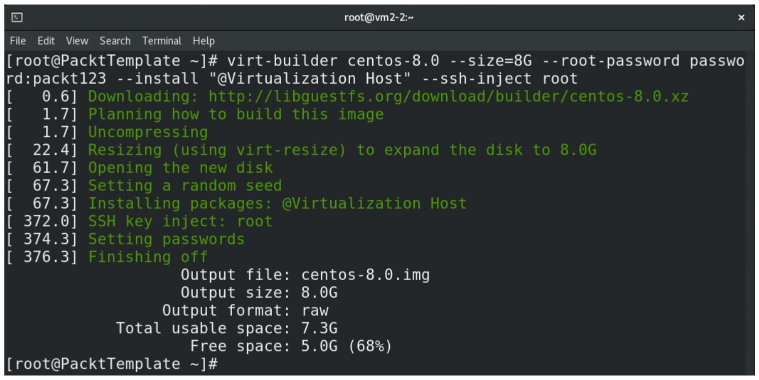
2. A logical next step is to do **virt-install**—so, here we go:



```
[root@PacktTemplate ~]# virt-install --import --name VBuilderTest-CentOS8 --ram
2048 --disk path=centos-8.0.img,format=raw --os-variant rhel8.0

Starting install...
Domain creation completed.
You can restart your domain by running:
  virsh --connect qemu:///system start VBuilderTest-CentOS8
[root@PacktTemplate ~]# virsh dominfo VBuilderTest-CentOS8
Id:             -
Name:           VBuilderTest-CentOS8
UUID:           8eebd4c8-be6b-467d-bda7-5228a9d39b12
OS Type:        hvm
State:          shut off
CPU(s):         2
Max memory:     2097152 KiB
Used memory:    2097152 KiB
Persistent:     yes
Autostart:      disable
Managed save:   no
Security model: selinux
Security DOI:   0
```

Figure 8.26 – New VM configured, deployed, and added to our local KVM hypervisor

3. If this seems cool to you, let's expand on that. Let's say that we want to take a **virt-builder** image, add a **yum** package group called **Virtualization Host** to that image, and, while we're at it, add the root's SSH key. This is what we'd do:



```
[root@PacktTemplate ~]# virt-builder centos-8.0 --size=8G --root-password passwo
rd:packt123 --install "@Virtualization Host" --ssh-inject root
[   0.6] Downloading: http://libguestfs.org/download/builder/centos-8.0.xz
[   1.7] Planning how to build this image
[   1.7] Uncompressing
[  22.4] Resizing (using virt-resize) to expand the disk to 8.0G
[  61.7] Opening the new disk
[  67.3] Setting a random seed
[  67.3] Installing packages: @Virtualization Host
[ 372.0] SSH key inject: root
[ 374.3] Setting passwords
[ 376.3] Finishing off
                   Output file: centos-8.0.img
                   Output size: 8.0G
                 Output format: raw
            Total usable space: 7.3G
                    Free space: 5.0G (68%)
[root@PacktTemplate ~]#
```

Figure 8.27 – Adding Virtualization Host

In all reality, this is really, really cool—it makes our life much easier, does quite a bit of work for us, and does it in a pretty simple way, and it works with Microsoft Windows operating systems as well. Also, we can use custom **virt-builder** repositories to download specific VMs that are tailored to our own needs, as we're going to learn next.

virt-builder repositories

Obviously, there are some pre-defined **virt-builder** repositories (http://libguestfs.org/ is one of them), but we can also create our own. If we go to the **/etc/virt-builder/repos.d** directory, we'll see a couple of files there (**libguestfs.conf** and its key, and so on). We can easily create our own additional configuration file that will reflect our local or remote **virt-builder** repository. Let's say that we want to create a local **virt-builder** repository. Let's create a config file called **local.conf** in the **/etc/virt-builder/repos.d** directory, with the following content:

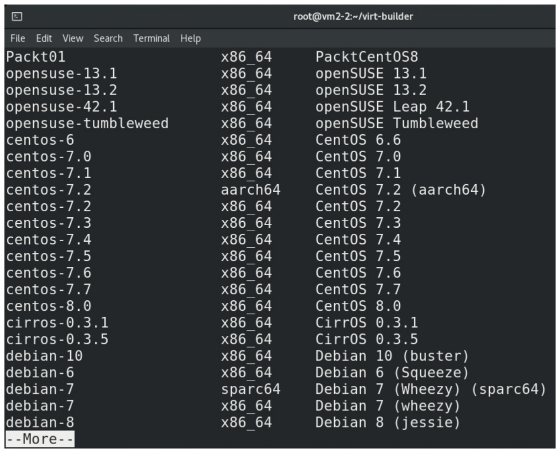
```
[local]
uri=file:///root/virt-builder/index
```

Then, copy or move an image to the **/root/virt-builder** directory (we will use our **centos-8.0.img** file created in the previous step, which we will convert to **xz** format by using the **xz** command), and create a file called **index** in that directory, with the following content:

```
[Packt01]
name=PacktCentOS8
osinfo=centos8.0
arch=x86_64
file=centos-8.0.img.xz
checksum=ccb4d840f5eb77d7d0ffbc4241fb
f4d21fcc1acdd3679
c13174194810b17dc472566f6a29dba3a8992
c1958b4698b6197e6a1689882
b67c1bc4d7de6738e947f
```

```
format=raw
size=8589934592
compressed_size=1220175252
notes=CentOS8 with KVM and SSH
```

A couple of explanations. **`checksum`** was calculated by using the **`sha512sum`** command on the **`centos-8.0.img.xz`**. **`size`** and **`compressed_size`** are real sizes of the original and XZd file. After this, if we issue the **`virt-builder --list |more`** command, we should get something like this:


root@vm2-2:~/virt-builder
File Edit View Search Terminal Help

```
Packt01                  x86_64     PacktCentOS8
opensuse-13.1            x86_64     openSUSE 13.1
opensuse-13.2            x86_64     openSUSE 13.2
opensuse-42.1            x86_64     openSUSE Leap 42.1
opensuse-tumbleweed      x86_64     openSUSE Tumbleweed
centos-6                 x86_64     CentOS 6.6
centos-7.0               x86_64     CentOS 7.0
centos-7.1               x86_64     CentOS 7.1
centos-7.2               aarch64    CentOS 7.2 (aarch64)
centos-7.2               x86_64     CentOS 7.2
centos-7.3               x86_64     CentOS 7.3
centos-7.4               x86_64     CentOS 7.4
centos-7.5               x86_64     CentOS 7.5
centos-7.6               x86_64     CentOS 7.6
centos-7.7               x86_64     CentOS 7.7
centos-8.0               x86_64     CentOS 8.0
cirros-0.3.1             x86_64     CirrOS 0.3.1
cirros-0.3.5             x86_64     CirrOS 0.3.5
debian-10                x86_64     Debian 10 (buster)
debian-6                 x86_64     Debian 6 (Squeeze)
debian-7                 sparc64    Debian 7 (Wheezy) (sparc64)
debian-7                 x86_64     Debian 7 (wheezy)
debian-8                 x86_64     Debian 8 (jessie)
--More--
```

Figure 8.28 – We successfully added an image to our local virt-builder repository

You can clearly see that our **`Packt01`** image is at the top of our list, and we can easily use it to deploy new VMs. By using additional repositories, we can greatly enhance our workflow and reuse our existing VMs and templates to deploy as many VMs as we want to. Imagine what this, combined with **`virt-builder`**'s customization options, does for cloud services on OpenStack, **Amazon Web Services** (**AWS**), and so on.

The next topic on our list is related to snapshots, a hugely valuable and misused VM concept. Sometimes, you have concepts in IT that can be equally good and bad, and snapshots are the usual suspect in that regard. Let's explain what snapshots are all about.

Snapshots

A VM snapshot is a file-based representation of the system state at a particular point in time. The snapshot includes configuration and disk data. With a snapshot, you can revert a VM to a point in time, which means by taking a snapshot of a VM, you preserve its state and can easily revert to it in the future if needed.

Snapshots have many use cases, such as saving a VM's state before a potentially destructive operation. For example, suppose you want to make some changes on your existing web server VM, which is running fine at the moment, but you are not certain if the changes you are planning to make are going to work or will break something. In that case, you can take a snapshot of the VM before doing the intended configuration changes, and if something goes wrong, you can easily revert to the previous working state of the VM by restoring the snapshot.

`libvirt` supports taking live snapshots. You can take a snapshot of a VM while the guest is running. However, if there are any **input/output (I/O)**-intensive applications running on the VM, it is recommended to shut down or suspend the guest first to guarantee a clean snapshot.

There are mainly two classes of snapshots for `libvirt` guests: internal and external; each has its own benefits and limitations, as detailed here:

- **Internal snapshot**: Internal snapshots are based on **qcow2** files. Before-snapshot and after-snapshot bits are stored in a single disk, allowing greater flexibility. **virt-manager** provides a graphical management utility to manage internal snapshots. The following are the limitations of an internal snapshot:
  a) Supported only with the **qcow2** format
  b) VM is paused while taking the snapshot
  c) Doesn't work with LVM storage pools
- **External snapshot**: External snapshots are based on a COW concept. When a snapshot is taken, the original disk image becomes read-only, and a new overlay disk image is created to accommodate guest writes, as illustrated in the following diagram:

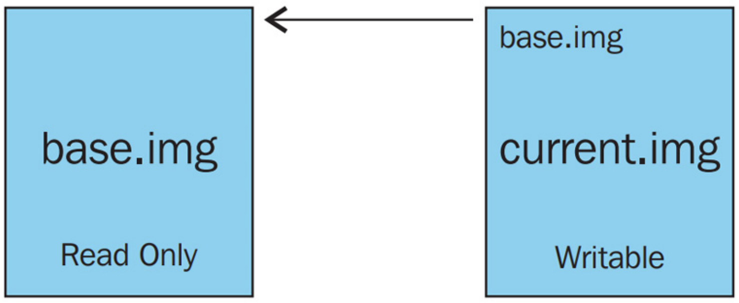


Figure 8.29 – Snapshot concept

The overlay disk image is initially created as **0** bytes in length, and it can grow to the size of the original disk. The overlay disk image is always **qcow2**. However, external snapshots work with any base disk image. You can take external snapshots of raw disk images, **qcow2**, or any other **libvirt**-supported disk image format. However, there is no **graphical user interface** (**GUI**) support available yet for external snapshots, so they are more expensive to manage when compared to internal snapshots.

Working with internal snapshots

In this section, you'll learn how to create, delete, and restore internal snapshots (offline/online) for a VM. You'll also learn how to use **`virt-manager`** to manage internal snapshots.

Internal snapshots work only with **`qcow2`** disk images, so first make sure that the VM for which you want to take a snapshot uses the **`qcow2`** format for the base disk image. If not, convert it to **`qcow2`** format using the **`qemu-img`** command. An internal snapshot is a combination of disk snapshots and the VM memory state—it's a kind of checkpoint to which you can revert easily when needed.

I am using a **`LAMP01`** VM here as an example to demonstrate internal snapshots. The **`LAMP01`** VM is residing on a local filesystem-backed storage pool and has a **`qcow2`** image acting as a virtual disk. The following command lists the snapshot associated with the VM:

```
# virsh snapshot-list LAMP01
Name Creation Time State
-------------------------------------
------------
```

As can be seen, currently, there are no existing snapshots associated with the VM; the **`LAMP01 virsh snapshot-list`** command lists all of the available snapshots for the given VM. The default information includes the snapshot name, creation time, and domain state. There is a lot of other snapshot-related information that can be listed by passing additional options to the **snapshot-list** command.

Creating the first internal snapshot

The easiest and preferred way to create internal snapshots for a VM on a KVM host is through the **virsh** command. **virsh** has a series of options to create and manage snapshots, listed as follows:

- **snapshot-create**: Use XML file to create a snapshot
- **snapshot-create-as**: Use list of arguments to create a snapshot
- **snapshot-current**: Get or set the current snapshot
- **snapshot-delete**: Delete a VM snapshot
- **snapshot-dumpxml**: Dump snapshot configuration in XML format
- **snapshot-edit**: Edit XML for a snapshot
- **snapshot-info**: Get snapshot information
- **snapshot-list**: List VM snapshots
- **snapshot-parent**: Get the snapshot parent name
- **snapshot-revert**: Revert a VM to a specific snapshot

The following is a simple example of creating a snapshot. Running the following command will create an internal snapshot for the **LAMP01** VM:

```
# virsh snapshot-create LAMP01
Domain snapshot 1439949985 created
```

By default, a newly created snapshot gets a unique number as its name. To create a snapshot with a custom name and description, use the

snapshot-create-as command. The difference between these two commands is that the latter one allows configuration parameters to be passed as an argument, whereas the former one does not. It only accepts XML files as the input. We are using **snapshot-create-as** in this chapter as it's more convenient and easy to use.

Creating an internal snapshot with a custom name and description

To create an internal snapshot for the **LAMP01** VM with the name **Snapshot 1** and the description **First snapshot**, type the following command:

```
# virsh snapshot-create-as LAMP01 --
name "Snapshot 1" --description
"First snapshot" --atomic
```

With the **--atomic** option specified, **libvirt** will make sure that no changes happen if the snapshot operation is successful or fails. It's always recommended to use the **--atomic** option to avoid any corruption while taking the snapshot. Now, check the **snapshot-list** output here:

```
# virsh snapshot-list LAMP01
Name Creation Time State
-------------------------------------
--------------
Snapshot1 2020-02-05 09:00:13 +0230
running
```

Our first snapshot is ready to use and we can now use it to revert the VM's state if something goes wrong in the future. This snapshot was taken while the VM was in a running state. The time to complete snapshot creation depends on how much memory the VM has and how actively the guest is modifying that memory at the time.

Note that the VM goes into paused mode while snapshot creation is in progress; therefore, it is always recommended you take the snapshot while the VM is not running. Taking a snapshot from a guest that is shut down ensures data integrity.

Creating multiple snapshots

We can keep creating more snapshots as required. For example, if we create two more snapshots so that we have a total of three, the output of **snapshot-list** will look like this:

```
# virsh snapshot-list LAMP01 --parent
Name Creation Time State Parent
-------------------------------------
------------------------------
Snapshot1 2020-02-05 09:00:13 +0230
running (null)
Snapshot2 2020-02-05 09:00:43 +0230
running Snapshot1
Snapshot3 2020-02-05 09:01:00 +0230
shutoff Snapshot2
```

Here, we used the **--parent** switch, which prints the parent-children relation of snapshots. The first snapshot's parent is **(null)**, which means it was created directly on the disk image, and **Snapshot1** is the parent of **Snapshot2** and **Snapshot2** is the parent of **Snapshot3**. This helps us know the sequence of snapshots. A tree-like view of snapshots can also be obtained using the **--tree** option, as follows:

```
# virsh snapshot-list LAMP01 --tree
Snapshot1
   |
  +- Snapshot2
       |
      +- Snapshot3
```

Now, check the **state** column, which tells us whether the particular snapshot is live or offline. In the preceding example, the first and second snapshots were taken while the VM was running, whereas the third was taken when the VM was shut down.

Restoring to a shutoff-state snapshot will cause the VM to shut down. You can also use the **qemu-img** command utility to get more information about internal snapshots—for example, the snapshot size, snapshot tag, and so on. In the following example output, you can see that the disk named as **LAMP01.qcow2** has three snapshots with different tags. This also shows you when a particular snapshot was taken, with its date and time:

```
# qemu-img info
/var/lib/libvirt/qemu/LAMP01.qcow2
image:
/var/lib/libvirt/qemu/LAMP01.qcow2
file format: qcow2
virtual size: 8.0G (8589934592 bytes)
disk size: 1.6G
cluster_size: 65536
Snapshot list:
ID TAG VM SIZE DATE VM CLOCK
```

```
1 1439951249 220M 2020-02-05 09:57:29
00:09:36.885
2 Snapshot1 204M 2020-02-05 09:00:13
00:01:21.284
3 Snapshot2 204M 2020-02-05 09:00:43
00:01:47.308
4 Snapshot3 0 2020-02-05 09:01:00
00:00:00.000
```

This can also be used to check the integrity of the **qcow2** image using the **check** switch, as follows:

```
# qemu-img check
/var/lib/libvirt/qemu/LAMP01.qcow2
No errors were found on the image.
```

If any corruption occurred in the image, the preceding command will throw an error. A backup from the VM should be immediately taken as soon as an error is detected in the **qcow2** image.

Reverting to internal snapshots

The main purpose of taking snapshots is to revert to a clean/working state of the VM when needed. Let's take an example. Suppose, after taking **Snapshot3** of your VM, you installed an application that messed up the whole configuration of the system. In such a situation, the VM can easily revert to the state it was in when **Snapshot3** was created. To revert to a snapshot, use the **snapshot-revert** command, as follows:

```
# virsh snapshot-revert <vm-name> --
snapshotname "Snapshot1"
```

If you are reverting to a shutdown snapshot, then you will have to start the VM manually. Use the **--running** switch with **virsh snapshot-revert** to get it started automatically.

Deleting internal snapshots

Once you are certain that you no longer need a snapshot, you can—and should—delete it to save space. To delete a snapshot of a VM, use the **snapshot-delete** command. From our previous example, let's remove the second snapshot, as follows:

```
# virsh snapshot-list LAMP01
Name Creation Time State
-------------------------------------
-----------------
Snapshot1 2020-02-05 09:00:13 +0230
running
Snapshot2 2020-02-05 09:00:43 +0230
running
Snapshot3 2020-02-05 09:01:00 +0230
shutoff
Snapshot4 2020-02-18 03:28:36 +0230
shutoff
# virsh snapshot-delete LAMP01
Snapshot 2
Domain snapshot Snapshot2 deleted
# virsh snapshot-list LAMP01
Name Creation Time State
-------------------------------------
-----------------
Snapshot1 2020-02-05 09:00:13 +0230
running
Snapshot3 2020-02-05 09:00:43 +0230
running
```

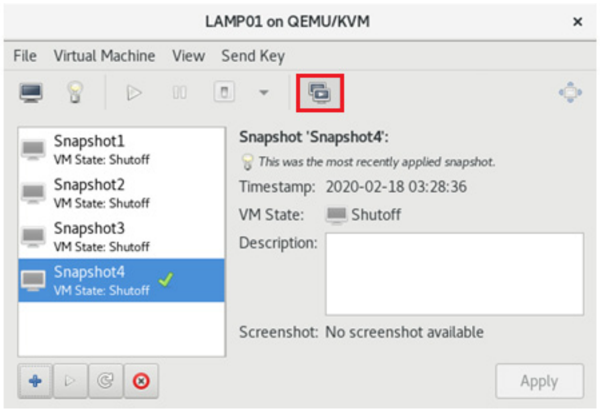
```
Snapshot4 2020-02-05 10:17:00 +0230
shutoff
```

Let's now check how to do these procedures by using **virt-manager**, our GUI utility for VM management.

Managing snapshots using virt-manager

As you might expect, **virt-manager** has a user-interface for creating and managing VM snapshots. At present, it works only with **qcow2** images, but soon, there will be support for raw images as well. Taking a snapshot with **virt-manager** is actually very easy; to get started, open VM Manager and click on the VM for which you would like to take a snapshot.

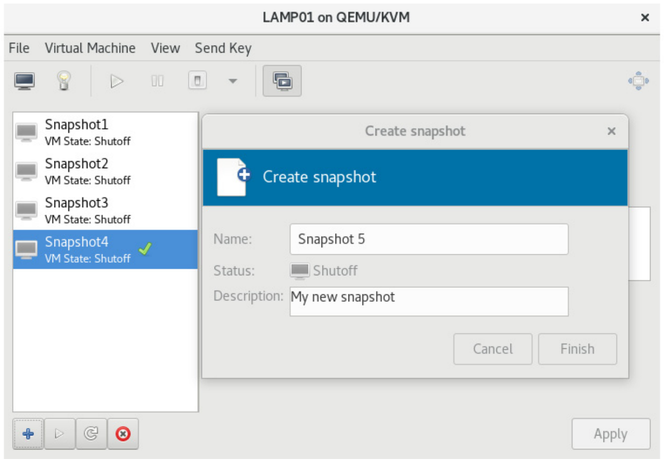
The snapshot user interface button (marked on the following screenshot in red) is present on the toolbar; this button gets activated only when the VM uses a **qcow2** disk:



Snapshot 'Snapshot4':
This was the most recently applied snapshot.
Timestamp: 2020-02-18 03:28:36
VM State: Shutoff
Description:
Screenshot: No screenshot available

Figure 8.30 – Working with snapshots from virt-manager

Then, if we want to take a snapshot, just use the + button, which will open a simple wizard so that we can give the snapshot a name and description, as illustrated in the following screenshot:



Name: Snapshot 5
Status: Shutoff
Description: My new snapshot

Figure 8.31 – Create snapshot wizard

Let's check how to work with external disk snapshots next, a faster and more modern (albeit not as mature) concept for KVM/VM snapshotting. Bear in mind that external snapshots are here to stay as they have much more capability that's really important for modern production environments.

Working with external disk snapshots

You learned about internal snapshots in the previous section. Internal snapshots are pretty simple to create and manage. Now, let's explore external snapshots. External snapshotting is all about **`overlay_image`** and **`backing_file`**. Basically, it turns **`backing_file`** into the read-only state and starts writing on **`overlay_image`**. These two images are described as follows:

- **`backing_file`**: The original disk image of a VM (read-only)
- **`overlay_image`**: The snapshot image (writable)

If something goes wrong, you can simply discard the **overlay_image** image and you are back to the original state.

With external disk snapshots, the **backing_file** image can be any disk image (**raw**; **qcow**; even **vmdk**) unlike internal snapshots, which only support the **qcow2** image format.

Creating an external disk snapshot

We are using a **WS2019SQL-Template** VM here as an example to demonstrate external snapshots. This VM resided in a filesystem storage pool named **vmstore1** and has a raw image acting as a virtual disk. The following code snippet provides details of this VM:

```
# virsh domblklist WS2019SQL-Template
--details
Type Device Target Source
-------------------------------------
-----------
file disk vda
/var/lib/libvirt/images/WS2019SQL-
Template.img
```

Let's see how to create an external snapshot of this VM, as follows:

1. Check if the VM you want to take a snapshot of is running, by executing the following code:

```
# virsh list
Id Name State
-------------------------------------
------
4 WS2019SQL-Template running
```

You can take an external snapshot while a VM is running or when it is shut down. Both live and offline snapshot methods are supported.

2. Create a VM snapshot via **`virsh`**, as follows:

```
# virsh snapshot-create-as
WS2019SQL-Template snapshot1 "My
First Snapshot" --disk-only --
atomic
```

The **`--disk-only`** parameter creates a disk snapshot. This is used for integrity and to avoid any possible corruption.

3. Now, check the **`snapshot-list`** output, as follows:

```
# virsh snapshot-list WS2019SQL-
Template
Name Creation Time State
-----------------------------------
----------------------
snapshot1 2020-02-10 10:21:38 +0230
disk-snapshot
```

4. Now, the snapshot has been taken, but it is only a snapshot of the disk's state; the contents of memory have not been stored, as illustrated in the following screenshot:

```
# virsh snapshot-info WS2019SQL-
Template snapshot1
Name: snapshot1
Domain: WS2019SQL-Template
Current: no
State: disk-snapshot
Location: external <<
Parent: -
Children: 1
Descendants: 1
```

```
Metadata: yes
```

5. Now, list all the block devices associated with the VM once again, as follows:

```
# virsh domblklist WS2019SQL-
Template
Target Source
-----------------------------------
-
vda
/var/lib/libvirt/images/WS2019SQL-
Template.snapshot1
```

Notice that the source got changed after taking the snapshot. Let's gather some more information about this new **image** **/var/lib/libvirt/images/WS2019SQL-Template.snapshot1** snapshot, as follows:

```
# qemu-img info
/var/lib/libvirt/images/WS2019SQL-
Template.snapshot1
image:
/var/lib/libvirt/images/WS2019SQL-
Template.snapshot1
file format: qcow2
virtual size: 19G (20401094656
bytes)
disk size: 1.6M
cluster_size: 65536
backing file:
/var/lib/libvirt/images/WS2019SQL-
Template.img
backing file format: raw
```

Note that the backing file field is pointing to **/var/lib/libvirt/images/WS2019SQL-Template.img**.

6. This indicates that the new **image /var/lib/libvirt/images/WS2019SQL-Template.snapshot1** snapshot is now a read/write snapshot of the original image, **/var/lib/libvirt/images/WS2019SQL-Template.img**; any changes made to **WS2019SQL-Template.snapshot1** will not be reflected in **WS2019SQL-Template.img**.
   Important note
   /var/lib/libvirt/images/WS2019SQL-Template.img *is the backing file (original disk).* **/var/lib/libvirt/images/WS2019SQL-Template.snapshot1** *is the newly created overlay image, where all the writes are now happening.*
7. Now, let's create one more snapshot:

```
# virsh snapshot-create-as
WS2019SQL-Template snapshot2 --
description "Second Snapshot" --
disk-only --atomic
Domain snapshot snapshot2 created
# virsh domblklist WS2019SQL-
Template --details
Type Device Target Source
-----------------------------------
-------------
file disk vda
/snapshot_store/WS2019SQL-
Template.snapshot2
```

Here, we used the **--diskspec** option to create a snapshot in the desired location. The option needs to be formatted in the **disk[,snapshot=type][,driver=type]**

[,file=name] format. This is what the parameters used signify:

- **disk**: The target disk shown in **virsh domblklist <vm_name>**.
- **snapshot**: Internal or external.
- **driver**: **libvirt**.
- **file**: The path of the location where you want to create the resulting snapshot disk. You can use any location; just make sure the appropriate permissions have been set.

Let's create one more snapshot, as follows:

```
# virsh snapshot-create-as WS2019SQL-
Template snapshot3 --description
"Third Snapshot" --disk-only --
quiesce
Domain snapshot snapshot3 created
```

Notice that this time, I added one more option: **--quiesce**. Let's discuss this in the next section.

What is quiesce?

Quiesce is a filesystem freeze (**fsfreeze**/**fsthaw**) mechanism. This puts the guest filesystems into a consistent state. If this step is not taken, anything waiting to be written to disk will not be included in the snapshot. Also, any changes made during the snapshot process may corrupt the image. To work around this, the **qemu-guest** agent needs to be installed on—and running inside—the guest. The snapshot creation will fail with an error, as illustrated here:

```
error: Guest agent is not responding:
Guest agent not available for now
```

Always use this option to be on the safe side while taking a snapshot. Guest tool installation is covered in **Chapter 5**, *Libvirt Storage*; you might want to revisit this and install the guest agent in your VM if it's not already installed.

We have created three snapshots so far. Let's see how they are connected with each other to understand how an external snapshot chain is formed, as follows:

1. List all the snapshots associated with the VM, like this:

```
# virsh snapshot-list WS2019SQL-
Template
Name Creation Time State
-----------------------------------
-----------------------
snapshot1 2020-02-10 10:21:38 +0230
disk-snapshot
snapshot2 2020-02-10 11:51:04 +0230
disk-snapshot
snapshot3 2020-02-10 11:55:23 +0230
disk-snapshot
```

2. Check which is the current active (read/write) disk/snapshot for the VM by running the following code:

```
# virsh domblklist WS2019SQL-
Template
Target Source
-----------------------------------
-------------
vda /snapshot_store/WS2019SQL-
Template.snapshot3
```

3. You can enumerate the backing file chain of the current active (read/write) snapshot using the **`--backing-chain`** option provided with **`qemu-img`**. **`--backing-chain`** will show us the whole tree of parent-child relationships in a disk image chain. Refer to the following code snippet for a further description:

   ```
   # qemu-img info --backing-chain
   /snapshot_store/WS2019SQL-
   Template.snapshot3|grep backing
   backing file:
   /snapshot_store/WS2019SQL-
   Template.snapshot2
   backing file format: qcow2
   backing file:
   /var/lib/libvirt/images/WS2019SQL-
   Template.snapshot1
   backing file format: qcow2
   backing file:
   /var/lib/libvirt/images/WS2019SQL-
   Template.img
   backing file format: raw
   ```

From the preceding details, we can see the chain is formed in the following manner:

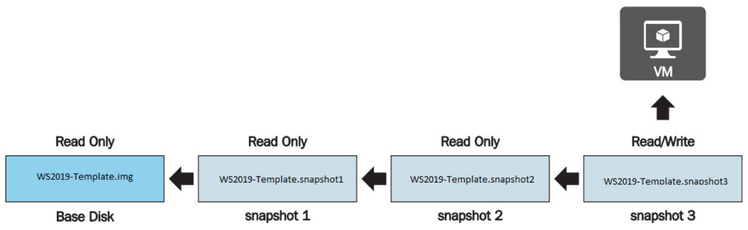


Figure 8.32 – Snapshot chain for our example VM

So, it has to be read as follows: **`snapshot3`** has **`snapshot2`** as its backing file; **`snapshot2`** has **`snapshot1`** as its backing file; and **`snapshot1`** has the base image as its backing file. Currently,

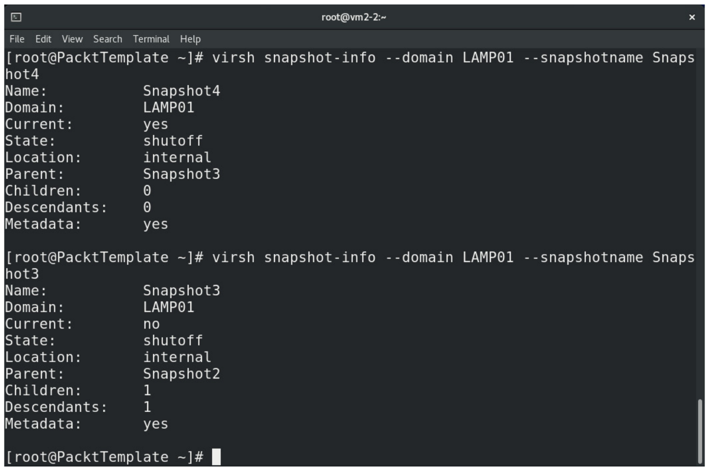
snapshot3 is the current active snapshot, where live guest writes happen.

Reverting to external snapshots

External snapshot support in **libvirt** was incomplete in some older RHEL/CentOS versions, even as recently as RHEL/CentOS 7.5. Snapshots can be created online or offline, and with RHEL/CentOS 8.0 there has been a significant change in terms of how snapshots are treated. For starters, Red Hat recommends using external snapshots now. Furthermore, to quote Red Hat:

Creating or loading a snapshot of a running VM, also referred to as a live snapshot, is not supported in RHEL 8. In addition, note that non-live VM snapshots are deprecated in RHEL 8. Therefore, creating or loading a snapshot of a shut-down VM is supported, but Red Hat recommends not using it.

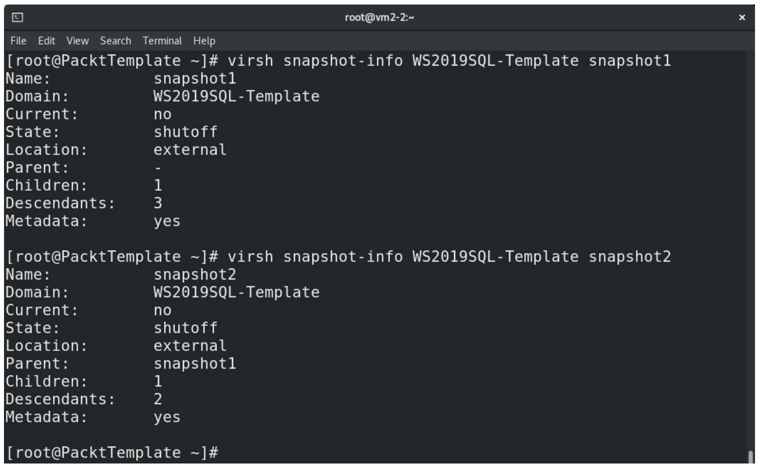
A caveat to this is the fact that **virt-manager** still doesn't support external snapshots, as evident by the following screenshot and the fact that when we created these snapshots just a couple of pages ago, we never got an option to select external snapshot as the snapshot type:



```
[root@PacktTemplate ~]# virsh snapshot-info --domain LAMP01 --snapshotname Snaps
hot4
Name:           Snapshot4
Domain:         LAMP01
Current:        yes
State:          shutoff
Location:       internal
Parent:         Snapshot3
Children:       0
Descendants:    0
Metadata:       yes

[root@PacktTemplate ~]# virsh snapshot-info --domain LAMP01 --snapshotname Snaps
hot3
Name:           Snapshot3
Domain:         LAMP01
Current:        no
State:          shutoff
Location:       internal
Parent:         Snapshot2
Children:       1
Descendants:    1
Metadata:       yes

[root@PacktTemplate ~]#
```

Figure 8.33 – All snapshots made from virt-manager and libvirt commands without additional options are internal snapshots

Now, we also worked with the **`WS2019SQL-Template`** VM and created *external* snapshots on it, so the situation is different. Let's check it, as follows:



```
[root@PacktTemplate ~]# virsh snapshot-info WS2019SQL-Template snapshot1
Name:           snapshot1
Domain:         WS2019SQL-Template
Current:        no
State:          shutoff
Location:       external
Parent:         -
Children:       1
Descendants:    3
Metadata:       yes

[root@PacktTemplate ~]# virsh snapshot-info WS2019SQL-Template snapshot2
Name:           snapshot2
Domain:         WS2019SQL-Template
Current:        no
State:          shutoff
Location:       external
Parent:         snapshot1
Children:       1
Descendants:    2
Metadata:       yes

[root@PacktTemplate ~]#
```

Figure 8.34 – WS2019SQL-Template has external snapshots

The next step that we could take is to revert to a previous state—for example, **`snapshot3`**. We can easily do that from the shell, by using the **`virsh snapshot-revert`** command, as follows:

```
# virsh snapshot-revert WS2019SQL-
Template --snapshotname "snapshot3"
```

```
error: unsupported configuration:
revert to external snapshot not
supported yet
```

Does that mean that, once an external disk snapshot is taken for a VM, there is no way to revert to that snapshot? No—it's not like that; you can definitely revert to a snapshot but there is no **`libvirt`** support to accomplish this. You will have to revert manually by manipulating the domain XML file.

Take as an example a **`WS2019SQL-Template`** VM that has three snapshots associated with it, as follows:

```
virsh snapshot-list WS2019SQL-
Template
Name Creation Time State
-------------------------------------
----------------------
snapshot1 2020-02-10 10:21:38 +0230
disk-snapshot
snapshot2 2020-02-10 11:51:04 +0230
disk-snapshot
snapshot3 2020-02-10 11:55:23 +0230
disk-snapshot
```

Suppose you want to revert to **`snapshot2`**. The solution is to shut down the VM (yes—a shutdown/power-off is mandatory) and edit its XML file to point to the **`snapshot2`** disk image as the boot image, as follows:

1. Locate the disk image associated with **`snapshot2`**. We need the absolute path of the image. You can simply look into the storage pool and get the path, but the best option is to

check the snapshot XML file. How? Get help from the **virsh** command, as follows:

```
# virsh snapshot-dumpxml WS2019SQL-
Template --snapshotname snapshot2 |
grep
'source file' | head -1
<source
file='/snapshot_store/WS2019SQL-
Template.snapshot2'/>
```

2. **/snapshot_store/WS2019SQL-Template.snapshot2** is the file associated with **snapshot2**. Verify that it's intact and properly connected to the **backing_file**, as follows:

```
# qemu-img check
/snapshot_store/WS2019SQL-
Template.snapshot2
No errors were found on the image.
46/311296 = 0.01% allocated, 32.61%
fragmented, 0.00% compressed
clusters
Image end offset: 3670016
```

If checking against the image produces no errors, this means **backing_file** is correctly pointing to the **snapshot1** disk. All good. If an error is detected in the **qcow2** image, use the **-r leaks/all** parameter. It may help repair the inconsistencies, but this isn't guaranteed. Check this excerpt from the **qemu-img** man page:

3. The -r switch with qemu-img tries to repair any inconsistencies that are found
4. During the check. -r leaks repairs only cluster leaks, whereas –r all fixes all

5. Kinds of errors, with a higher risk of choosing the wrong fix or hiding
6. Corruption that has already occurred.

   Let's check the information about this snap-shot, as follows:

   ```
   # qemu-img info
   /snapshot_store/WS2019SQL-
   Template.snapshot2 | grep backing
   backing file:
   /var/lib/libvirt/images/WS2019SQL-
   Template.snapshot1
   backing file format: qcow2
   ```
7. It is time to manipulate the XML file. You can remove the currently attached disk from the VM and **add /snapshot_store/WS2019SQL-Template.snapshot2**. Alternatively, edit the VM's XML file by hand and modify the disk path. One of the better options is to use the **virt-xml** command, as follows:

   ```
   # virt-xml WS2019SQL-Template --
   remove-device --disk target=vda
   # virt-xml --add-device --disk
   /snapshot_store/WS2019SQL-
   Template.snapshot2,fo
   rmat=qcow2,bus=virtio
   ```

   This should add **WS2019SQL-Template.snapshot2** as the boot disk for the VM; you can verify that by executing the fol-lowing command:

   ```
   # virsh domblklist WS2019SQL-
   Template
   Target Source
   -----------------------------------
   -------------
   ```

```
vda /snapshot_store/WS2019SQL-
Template.snapshot2
```

There are many options to manipulate a VM XML file with the **virt-xml** command. Refer to its man page to get acquainted with it. It can also be used in scripts.

8. Start the VM, and you are back to the state when **snapshot2** was taken. Similarly, you can revert to **snapshot1** or the base image when required.

The next topic on our list is about deleting external disk snapshot which—as we mentioned—is a bit complicated. Let's check how we can do that next.

Deleting external disk snapshots

Deleting external snapshots is somewhat tricky. An external snapshot cannot be deleted directly, unlike an internal snapshot. It first needs to be manually merged with the base layer or toward the active layer; only then can you remove it. There are two live block operations available for merging online snapshots, as follows:

- **blockcommit**: Merges data with the base layer. Using this merging mechanism, you can merge overlay images into backing files. This is the fastest method of snapshot merging because overlay images are likely to be smaller than backing images.
- **blockpull**: Merges data toward the active layer. Using this merging mechanism, you can merge data from **backing_file** to overlay im-

ages. The resulting file will always be in **qcow2** format.

Next, we are going to read about merging external snapshots using **blockcommit**.

Merging external snapshots using blockcommit

We created a new VM named **VM1**, which has a base image (raw) called **vm1.img** with a chain of four external snapshots. **/var/lib/libvirt/images/vm1.snap4** is the active snapshot image where live writes happen; the rest are in read-only mode. Our target is to remove all the snapshots associated with this VM, as follows:

1. List the current active disk image in use, like this:
   ```
   # virsh domblklist VM1
   Target Source
   ----------------------------
   hda
   /var/lib/libvirt/images/vm1.snap4
   ```
   Here, we can verify that **the /var/lib/libvirt/images/vm1.snap4** image is the currently active image on which all writes are occurring.
2. Now, enumerate the backing file chain of **/var/lib/libvirt/images/vm1.snap4**, as follows:
   ```
   # qemu-img info --backing-chain
   /var/lib/libvirt/images/vm1.snap4 |
   grep backing
   backing file:
   /var/lib/libvirt/images/vm1.snap3
   ```

```
backing file format: qcow2
backing file:
/var/lib/libvirt/images/vm1.snap2
backing file format: qcow2
backing file:
/var/lib/libvirt/images/vm1.snap1
backing file format: qcow2
backing file:
/var/lib/libvirt/images/vm1.img
backing file format: raw
```

3. Time to merge all the snapshot images into the base image, like this:

```
# virsh blockcommit VM1 hda --
verbose --pivot --active
Block Commit: [100 %]
Successfully pivoted
4. Now, check the current active
block device in use:
# virsh domblklist VM1
Target Source
--------------------------
hda /var/lib/libvirt/images/vm1.img
```

Notice that now, the current active block device is the base image and all writes are switched to it, which means we successfully merged the snapshot images into the base image. But the **snapshot-list** output in the following code snippet shows that there are still snapshots associated with the VM:

```
# virsh snapshot-list VM1
Name Creation Time State
-------------------------------------
---------------
```

```
snap1 2020-02-12 09:10:56 +0230
shutoff
snap2 2020-02-12 09:11:03 +0230
shutoff
snap3 2020-02-12 09:11:09 +0230
shutoff
snap4 2020-02-12 09:11:17 +0230
shutoff
```

If you want to get rid of this, you will need to remove the appropriate metadata and delete the snapshot images. As mentioned earlier, **libvirt** does not have complete support for external snapshots. Currently, it can just merge the images, but no support is available for automatically removing snapshot metadata and overlaying image files. This has to be done manually. To remove snapshot metadata, run the following code:

```
# virsh snapshot-delete VM1 snap1 --
children --metadata
# virsh snapshot-list VM1
Name Creation Time State
```

In this example, we learned how to merge external snapshots by using the **blockcommit** method. Let's learn how to merge external snapshot using the **blockpull** method next.

Merging external snapshots using blockpull

We created a new VM named **VM2**, which has a base image (raw) called **vm2.img** with only one external snapshot. The snapshot disk is the active image where live writes happen and the base image is in read-only mode. Our target is to

remove snapshots associated with this VM.
Proceed as follows:

1. List the current active disk image in use, like this:
   ```
   # virsh domblklist VM2
   Target Source
   ----------------------------
   hda
   /var/lib/libvirt/images/vm2.snap1
   ```
   Here, we can verify that the **/var/lib/libvirt/images/vm2.snap1** image is the currently active image on which all writes are occurring.
2. Now, enumerate the backing file chain of **/var/lib/libvirt/imagesvar/lib/libvirt/images/vm2.snap1**, as follows:
   ```
   # qemu-img info --backing-chain
   /var/lib/libvirt/images/vm2.snap1 |
   grep backing
   backing file:
   /var/lib/libvirt/images/vm1.img
   backing file format: raw
   ```
3. Merge the base image into the snapshot image (base to overlay image merging), like this:
   ```
   # virsh blockpull VM2 --path
   /var/lib/libvirt/images/vm2.snap1 -
   -wait --verbose
   Block Pull: [100 %]
   Pull complete
   ```
   Now, check the size of **/var/lib/libvirt/images/vm2.snap1**. It got considerably larger because we pulled the **base_image** and merged it into the snapshot image to get a single file.

4. Now, you can remove the **base_image** and snapshot metadata, as follows:

```
# virsh snapshot-delete VM2 snap1 -
-metadata
```

We ran the merge and snapshot deletion tasks while the VM is in the running state, without any downtime. **blockcommit** and **blockpull** can also be used to remove a specific snapshot from the snapshot chain. See the man page for **virsh** to get more information and try it yourself. You will also find some additional links in the *Further reading* section of this chapter, so make sure that you go through them.

Use cases and best practices while using snapshots

We mentioned that there's a big love-hate relationship in the IT world with regard to snapshots. Let's discuss the reasons and some common-sense best practices when using snapshots, as follows:

- When you take a VM snapshot, you are creating new delta copy of the VM disk, **qemu2**, or a raw file, and then you are writing to that delta. So, the more data you write, the longer it's going to take to commit and consolidate it back into the parent. Yes—you will eventually need to commit snapshots, but it is not recommended you go into production with a snapshot attached to the VM.
- Snapshots are not backups; they are just a picture of a state, taken at a specific point in time,

to which you can revert when required. Therefore, do not rely on it as a direct backup process. For that, you should implement a backup infrastructure and strategy.

- Don't keep a VM with a snapshot associated with it for long time. As soon as you verify that reverting to the state at the time a snapshot was taken is no longer required, merge and delete the snapshot immediately.
- Use external snapshots whenever possible. The chances of corruption are much lower in external snapshots when compared to internal snapshots.
- Limit the snapshot count. Taking several snapshots in a row without any cleanup can hit VM and host performance, as **`qemu`** will have to trawl through each image in the snapshot chain to read a new file from **`base_image`**.
- Have Guest Agent installed in the VM before taking snapshots. Certain operations in the snapshot process can be improved through support from within the guest.
- Always use the **`--quiesce`** and **`--atomic`** options while taking snapshots.

If you're using these best practices, we are comfortable recommending using snapshots for your benefit. They will make your life much easier and give you a point you can come back to, without all the problems and hoopla that comes with them.

Summary

In this chapter, you learned how to work with **`libguestfs`** utilities to modify VM disks, create templates, and manage snapshots. We also looked into **`virt-builder`** and various provisioning methodologies for our VMs, as these are some of the most common scenarios used in the real world. We will learn even more about the concept of deploying VMs in large numbers (hint: cloud services) in the next chapter, which is all about **`cloud-init`**.

Questions

1. Why would we need to modify VM disks?
2. How can we convert a VM to KVM?
3. Why do we use VM templates?
4. How do we create a Linux-based template?
5. How do we create a Microsoft Windows-based template?
6. Which cloning mechanisms for deploying from template do you know of? What are the differences between them?
7. Why do we use **`virt-builder`**?
8. Why do we use snapshots?
9. What are the best practices of using snapshots?

Further reading

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

- **`libguesfs`** documentation: http://libguestfs.org/

- **`virt-builder`**: http://libguestfs.org/virt-builder.1.html
- Managing snapshots: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/sect-managing_guests_with_the_virtual_machine_manager_virt_manager-managing_snapshots
- Generate VM Images with **`virt-builder`**: http://www.admin-magazine.com/Articles/Generate-VM-Images-with-virt-builder
- QEMU snapshot documentation: http://wiki.qemu.org/Features/Snapshots
- **`libvirt`**—Snapshot XML format: https://libvirt.org/formatsnapshot.html