



Mastering KVM Virtualization

Second Edition

Design expert data center virtualization solutions
with the power of Linux KVM

Vedran Dakic | Humble Devassy Chirammal |
Prasad Mukhedkar | Anil Vettathu



Mastering KVM Virtualization

Second Edition

Design expert data center virtualization solutions
with the power of Linux KVM

Vedran Dakic

Humble Devassy Chirammal

Prasad Mukhedkar

Anil Vettathu

Packt

BIRMINGHAM—MUMBAI

Mastering KVM Virtualization

Second Edition

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Vijin Boricha

Acquisition Editor: Shrilekha Inani

Senior Editor: Arun Nadar

Content Development Editor: Nihar Kapadia

Technical Editor: Soham Amburle

Copy Editor: Safis Editing

Project Coordinator: Neil D'mello

Proofreader: Safis Editing

Indexer: Priyanka Dhadke

Production Designer: Aparna Bhagat

First published: June 2019

Second edition: October 2020

Production reference: 2250920

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83882-871-4

www.packt.com

25 years ago, a colleague suggested that I should write what he called "a Linux book". I liked the idea and I promised I would. Years rolled by, and here I am, a quarter of a century later, acting on a promise. As Steve Jobs once said, 'Ideas without action aren't ideas. They're regrets.'

To my family – my mother, father, and brother, for putting up with me over the course of the last 25 years – which led to writing this book. To my TA, Jasmin, for both helping me to improve and offering insights into various topics covered in this book.

To my son, Luka, for showing me how young people can be both talented and focused, especially when faced with problems that require innovative solutions.

To my partner, Sanja, for driving me on in everything that I do.

Here's to not having any regrets.

– Vedran Dakic



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and, as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customer-care@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Vedran Dakic has a master's in electrical engineering and computing and is an IT trainer, covering system administration, cloud, automatization, and orchestration courses. He is a certified Red Hat, VMware, and Microsoft trainer. He is currently employed as the head of department of operating systems at Algebra University College in Zagreb. As part of his job, he lectures in relation to 3- and 5-year study programs in systems engineering, programming, and multimedia tracks. He also does a lot of consulting and systems integration in relation to his clients' projects – something he has been doing for the past 20 years. His approach is simple – bring real-world experience to all of the courses that he is involved with as this will provide added value for his students and customers.

Humble Devassy Chiramal is a senior software engineer in the Storage Engineering team at Red Hat. He has more than 15 years of IT experience, and his area of expertise is in understanding the full stack in an ecosystem, with emphasis on architecting solutions based on demand. These days he primarily concentrates on Ceph and GlusterFS and its integration to container orchestrator systems like Kubernetes. He has hands-on experience of emerging technologies, such as IaaS and PaaS solutions in Cloud and Containers. In the past, he has worked on intrusion detection systems, Clustering solutions, and Virtualization. As an open source advocate, he is a core contributor to many open source projects like Kubernetes. He actively organizes meetups on Openshift/ Kubernetes, Virtualization, GlusterFS, CentOS. His twitter handle is @hchiram and his website is <https://www.humblec.com>.

This book is dedicated to the loving memory of my parents, C.O. Devassy and Elsy Devassy, whose steady, balanced, and loving guidance has given me the strength and determination to be the person I am today. I would like to thank my wife, Anitha, for standing beside me throughout my career, and for the effort she put into taking care of our son, Heaven, and our daughters, Hail Mariya and Hanna Mariya, while I was writing this book. I would like to thank my brothers, Sible and Fr. Able Chirammal, as well, without whose constant support this book would not have been possible.

Finally, a special thanks to Ulrich Obergfell for being an inspiration that helped me enrich my knowledge in Virtualization.

Prasad Mukhedkar is a specialist cloud solution architect at Red Hat India with over 10 years of experience in helping customers in their journey to Virtualization and Cloud adoption. He is a Red Hat Certified Architect and has extensive experience in designing and implementing high performing cloud infrastructure. His areas of expertise are Red Hat Enterprise Linux 7/8 performance tuning, KVM virtualization, Ansible Automation, and Red Hat OpenStack. He is a huge fan of the Linux "GNU screen" utility.

Anil Vettathu began his association with Linux while in college and began his career as a Linux System Administrator soon after. He is a generalist, with an interest in open source technologies. He has hands-on experience in designing and implementing large scale virtualization environments using open source technologies and has extensive knowledge in libvirt and KVM. These days he primarily works on Red Hat Enterprise Virtualization, containers, and real time performance tuning. Currently, he is working as a Technical Account Manager for Red Hat. His website is <http://anilv.in>.

I'd like to thank my wife, Chandni, for her unconditional support. She took on the pain of looking after our two naughtiest kids, while I enjoyed writing this book. I'd like to thank my parents, Dr. Annieamma and Dr. George Vettathu, for their guidance and for pushing me hard to study something new. Finally, I would like to thank my sister, Dr. Wilma, for her guidance, and my brother, Vimal.

About the reviewer

Ranjith Rajaram is employed as a senior principle technical support engineer at a leading open source Enterprise Linux company. He began his career by providing support to web hosting companies and managing servers remotely. Ranjith has also provided technical support to end customers. Early in his career, he worked on Linux, Unix, and FreeBSD platforms.

For the past 15 years, he has been continuously learning something new. This is what he likes and admires about technical support. As a mark of respect to all his fellow technical support engineers, he has included "developing software is humane, but supporting it is divine" in his email signature.

At his current organization, he is involved in implementing, installing, and troubleshooting Linux environment networks. Aside from this, he is also an active contributor to the Linux container space (Docker, Podman), Kubernetes, and OpenShift.

Apart from this book, he has reviewed the first editions of *Mastering KVM Virtualization* and *Learning RHEL Networking*, both available from Packt.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Section 1: KVM Virtualization Basics

1

Understanding Linux Virtualization

Linux virtualization and how it all started	4	Xen	12
		KVM	13
Types of virtualization	6	What Linux virtualization offers you in the cloud	14
Using the hypervisor/virtual machine manager	9	Summary	15
Type 1 and type 2 hypervisors	10	Questions	16
Open source virtualization projects	11	Further reading	16

2

KVM as a Virtualization Solution

Virtualization as a concept	18	The internal workings of libvirt, QEMU, and KVM	30
Virtualized versus physical environments	18	libvirt	30
Why is virtualization so important?	20	QEMU	38
Hardware requirements for virtualization	21	QEMU – KVM internals	41
Software requirements for virtualization	24	Data structures	42
		Threading models in QEMU	48

KVM	49	Summary	64
Data structures	55	Questions	64
Execution flow of vCPU	59	Further reading	65

Section 2: libvirt and ovirt for Virtual Machine Management

3

Installing KVM Hypervisor, libvirt, and oVirt

Getting acquainted with QEMU and libvirt	70	Automating virtual machine installation	77
Getting acquainted with oVirt	71	Installing oVirt	80
Installing QEMU, libvirt, and oVirt	73	Starting a virtual machine using QEMU and libvirt	83
Installing the first virtual machine in KVM	76	Summary	87
		Questions	87
		Further reading	87

4

Libvirt Networking

Understanding physical and virtual networking	90	Configuring Open vSwitch	105
Virtual networking	91	Other Open vSwitch use cases	113
Libvirt NAT network	93	Understanding and using SR-IOV	114
Libvirt routed network	94	Understanding macvtap	118
Libvirt isolated network	95	Summary	121
Using userspace networking with TAP and TUN devices	101	Questions	121
Implementing Linux bridging	103	Further reading	122

5

Libvirt Storage

Introduction to storage	124	Getting image information	164
Storage pools	126	Attaching a disk using virt-manager	164
Local storage pools	128	Attaching a disk using virsh	166
Libvirt storage pools	130	Creating an ISO image library	167
		Deleting a storage pool	169
NFS storage pool	131	Creating storage volumes	170
iSCSI and SAN storage	136	Creating volumes using the virsh command	171
Storage redundancy and multipathing	146	Deleting a volume using the virsh command	171
Gluster and Ceph as a storage backend for KVM	150	The latest developments in storage – NVMe and NVMeOF	172
Gluster	150	Summary	176
Ceph	155	Questions	176
Virtual disk images and formats and basic KVM storage operations	162	Further reading	177

6

Virtual Display Devices and Protocols

Using virtual machine display devices	180	Using the SPICE display protocol	198
Physical and virtual graphics cards in VDI scenarios	185	Adding a SPICE graphics server	198
GPU PCI passthrough	189	Methods to access a virtual machine console	200
Discussing remote display protocols	193	Getting display portability with noVNC	202
Remote display protocols history	193	Summary	206
Types of remote display protocols	195	Questions	206
Using the VNC display protocol	196	Further reading	207
Why VNC?	197		

7

Virtual Machines: Installation, Configuration, and Life Cycle Management

Creating a new VM using virt-manager	210	Migrating VMs	238
Using virt-manager	210	Benefits of VM migration	239
Using virt-* commands	217	Setting up the environment	240
Creating a new VM using Cockpit	223	Offline migration	243
		Live or online migration	247
Creating a new VM using oVirt	226	Summary	252
Configuring your VM	230	Questions	252
Adding and removing virtual hardware from your VM	236	Further reading	253

8

Creating and Modifying VM Disks, Templates, and Snapshots

Modifying VM images using libguestfs tools	256	Snapshots	285
virt-v2v	257	Working with internal snapshots	286
virt-p2v	259	Managing snapshots using virt-manager	291
guestfish	259	Working with external disk snapshots	292
VM templating	263	Use cases and best practices while using snapshots	305
Working with templates	266	Summary	306
Deploying VMs from a template	275	Questions	306
virt-builder and virt-builder repos	281	Further reading	306
virt-builder repositories	283		

Section 3: Automation, Customization, and Orchestration for KVM VMs

9

Customizing a Virtual Machine with cloud-init

What is the need for virtual machine customization?	312	Using cloud-init modules	322
Understanding cloud-init	314	Examples on how to use a cloud-config script with cloud-init	323
Understanding cloud-init architecture	315	The first deployment	329
Installing and configuring cloud-init at boot time	318	The second deployment	332
Cloud-init images	319	The third deployment	334
Cloud-init data sources	320	Summary	342
Passing metadata and user data to cloud-init	321	Questions	342
		Further reading	343

10

Automated Windows Guest Deployment and Customization

The prerequisites to creating Windows VMs on KVM	346	cloudbase-init customization examples	353
Creating Windows VMs using the virt-install utility	347	Troubleshooting common cloudbase-init customization issues	361
Customizing Windows VMs using cloudbase-init	350	Summary	364
		Questions	364
		Further reading	364

11

Ansible and Scripting for Orchestration and Automation

Understanding Ansible	366	automation and orchestration	399
Automation approaches	367	Orchestrating multi-tier application deployment on KVM VM	406
Introduction to Ansible	369	Learning by example - various examples of using Ansible with KVM	409
Deploying and using AWX	372	Summary	410
Deploying Ansible	382	Questions	410
Provisioning a virtual machine using the kvm_libvirt module	383	Further reading	411
Working with playbooks	387		
Installing KVM	393		
Using Ansible and cloud-init for			

Section 4: Scalability, Monitoring, Performance Tuning, and Troubleshooting

12

Scaling Out KVM with OpenStack

Introduction to OpenStack	416	Additional OpenStack use cases	439
Software-defined networking	418	Creating a Packstack demo environment for OpenStack	441
Understanding VXLAN	420	Provisioning the OpenStack environment	443
Understanding GENEVE	425	Installing OpenStack step by step	445
OpenStack components	426	OpenStack administration	449
Swift	427	Day-to-day administration	457
Nova	430	Identity management	460
Glance	434	Integrating OpenStack with Ansible	462
Horizon	435	Installing an Ansible deployment server	464
Designate	436		
Keystone	436		
Neutron	437		

Configuring the Ansible inventory	466	Summary	467
Running Ansible playbooks	467	Questions	468
		Further reading	468

13

Scaling out KVM with AWS

Introduction to AWS	470	What do we want to do?	484
Approaching the cloud	470	Uploading an image to EC2	498
Multi-cloud	472	Building hybrid KVM clouds with Eucalyptus	507
Shadow IT	473	How do you install it?	509
Market share	474	Using Eucalyptus for AWS control	518
Big infrastructure but no services	474	Summary	521
Pricing	475	Questions	521
Data centers	477	Further reading	522
Placement is the key	478		
AWS services	480		
Preparing and converting virtual machines for AWS	483		

14

Monitoring the KVM Virtualization Platform

Monitoring the KVM virtualization platform	524	Workflow	533
Introduction to the open source ELK solution	526	Configuring data collector and aggregator	536
Elasticsearch	526	Creating charts in Kibana	537
Logstash	527	Creating custom utilization reports	538
Kibana	528	ELK and KVM	549
Setting up and integrating the ELK stack	528	Summary	557
		Questions	557
		Further reading	557

15

Performance Tuning and Optimization for KVM VMs

It's all about design	560	Automatic NUMA balancing	592
General hardware design	561	The numactl command	593
VM design	565	Understanding numad and numastat	594
Tuning the VM CPU and memory performance	565	Virtio device tuning	596
CPU pinning	568	Block I/O tuning	597
Working with memory	572	Network I/O tuning	601
Getting acquainted with KSM	580	How to turn it on	602
Tuning the CPU and memory with NUMA	585	KVM guest time-keeping best practices	604
NUMA memory allocation policies	586	Software-based design	606
Understanding emulatorpin	589	Summary	609
KSM and NUMA	591	Questions	610
		Further reading	610

16

Troubleshooting Guidelines for the KVM Platform

Verifying the KVM service status	614	Troubleshooting Eucalyptus	629
KVM services logging	617	AWS and its verbosity, which doesn't help	637
Enabling debug mode logging	618	Paying attention to details	638
Advanced troubleshooting tools	621	Troubleshooting problems with the ELK stack	639
oVirt	622	Best practices for troubleshooting KVM issues	640
oVirt and KVM storage problems	623	Summary	641
Problems with snapshots and templates - virtual machine customization	624	Questions	641
Problems working with Ansible and OpenStack	627	Further reading	642
Dependencies	628		

Other Books You May Enjoy

Index

Preface

Mastering KVM Virtualization is a book that should get you "from zero to hero" status in the time it takes for you to go through this book. This book is a large collection of everything that KVM has to offer, both for a DevOps and regular system administration audience, and developers. It is our hope that, by going through this book, you'll be able to understand everything about the inner workings of KVM, as well as the more advanced concepts and everything in between. It doesn't matter if you're just barely starting with KVM virtualization or if you're already well on the way – you should find some valuable information on the pages of this book.

Who this book is for

This book is for Linux beginners and professionals alike, as it doesn't necessarily require an advanced knowledge of Linux beforehand. We'll get you there as you go through the book – it's an integral part of the learning process. If you're interested in KVM, OpenStack, the ELK Stack, Eucalyptus, or AWS – we've got you covered.

What this book covers

Chapter 1, Understanding Linux Virtualization, discusses different types of virtualization, hypervisor types, and Linux virtualization concepts (Xen and KVM). In this chapter, we try to explain some basics of Linux virtualization and how it fits into the cloud environment from a high-level perspective.

Chapter 2, KVM as a Virtualization Solution, starts with a discussion of virtualization concepts and the need to virtualize our environments, explains the basic hardware and software aspects of virtualization, and the various approaches to virtualization. In this chapter, we start discussing KVM and libvirt, concepts that we'll use throughout this book.

Chapter 3, Installing KVM Hypervisor, libvirt, and oVirt, expands on *Chapter 2* by introducing some new concepts including oVirt, a GUI that can be used to manage our virtualized Linux infrastructure. We take you through the process of checking whether the hardware used is compatible with KVM, introduce some basic commands for virtual machine deployment, and then move on to explain how we'd use oVirt in the same scenario.

Chapter 4, Libvirt Networking, explains how libvirt interacts with various networking concepts – virtual switches in different modes, how to use CLI tools to manage libvirt networking, TAP and TUN devices, Linux bridging, and Open vSwitch. After that, we discuss more extreme examples of networking by using SR-IOV, a concept that should get us the lowest latency and highest throughput and is used in cases where every single millisecond counts.

Chapter 5, Libvirt Storage, is a big one, as storage concepts are extremely important when building virtualized and cloud environments. We discuss every type of storage that KVM supports – local storage pools, NFS, iSCSI, SAN, Ceph, Gluster, multipathing and redundancy, virtual disk types, and so on. We also offer you a glimpse into the future of storage – with NVMe and NVMeoF being some of the technologies discussed.

Chapter 6, Virtual Display Devices and Protocols, talks about various virtual machine display types, remote protocols including VNC and Spice, as well as NoVNC, which ensures display portability as we can use a virtual machine console inside a web browser by using NoVNC.

Chapter 7, Virtual Machines: Installation, Configuration, and Life Cycle Management, introduces additional ways of deploying and configuring KVM virtual machines, as well as migration processes, which are very important for any kind of production environment.

Chapter 8, Creating and Modifying VM Disks, Templates, and Snapshots, discusses various virtual machine image types, virtual machine templating processes, the use of snapshots, and some of the use cases and best practices while using snapshots. It also serves as an introduction to the next chapter, where we will be using templating and virtual machine disks in a much more streamlined fashion to customize virtual machines post-boot by using `cloud-init` and `cloudbase-init`.

Chapter 9, Customize a Virtual Machine with cloud-init, discusses one of the most fundamental concepts in cloud environments – how to customize a virtual machine image/template post-boot. Cloud-init is used in almost all of the cloud environments to do post-boot Linux virtual machine configuration, and we explain how it works and how to make it work in your environment.

Chapter 10, Automated Windows Guest Deployment and Customization, is a continuation of *Chapter 9*, with a razor-sharp focus on Microsoft Windows virtual machine templating and post-boot customization. For that, we use `cloudbase-init`, a concept that's basically the same as `cloud-init`, but which is suited for Microsoft-based operating systems only.

Chapter 11, Ansible and Scripting for Orchestration and Automation, takes us on the first part of the Ansible journey – deploying AWX and Ansible, and describes how to use these concepts in our KVM-based environments. This is just one of the Ansible usage models that is employed in modern-day IT, as the whole DevOps and infrastructure-as-a-code story gets much more exposure in IT infrastructure all over the world.

Chapter 12, Scaling Out KVM with OpenStack, discusses the process of building cloud environments based on KVM. OpenStack is the standard approach to delivering just that when using KVM. In this chapter, we talk about all of the OpenStack building blocks and services, how to deploy it from scratch, and describe how to use it in production environments.

Chapter 13, Scaling Out KVM with AWS, takes us on a journey toward using public and hybrid cloud concepts by using **Amazon Web Services (AWS)**. Like almost all the other chapters, this is a heavily hands-on chapter that you can also use to get your feet wet in terms of getting to know AWS as a concept, which will be key to deploying a hybrid-cloud infrastructure using Eucalyptus at the end of the chapter.

Chapter 14, Monitoring the KVM Virtualization Platform, introduces a very popular concept of monitoring via the **Elasticsearch, Logstash, Kibana (ELK)** stack. It also takes you through the whole process of setting up and integrating the ELK stack with your KVM infrastructure, all the way through to the end result – using dashboards and UIs to monitor your KVM-based environment.

Chapter 15, Performance Tuning and Optimization for KVM VMs, talks about various approaches to tuning and optimization in KVM-based environments by deconstructing all of the infrastructure design principles and putting them to (correct) use. We cover a number of advanced topics here – NUMA, KSM, CPU and memory performance, CPU pinning, the tuning of VirtIO, and block and network devices.

Chapter 16, Troubleshooting Guidelines for the KVM Platform, starts with the basics – troubleshooting KVM services and logging, and explains various troubleshooting methodologies for KVM and oVirt, Ansible and OpenStack, Eucalyptus, and AWS. These are the real-life problems that we've also encountered in our production environments while writing this book. In this chapter, we basically discuss problems related to every single chapter of this book, including problems associated with snapshots and templating.

To get the most out of this book

We're assuming at least a basic knowledge of Linux and prior experience with installing virtual machines as prerequisites for this book.

Software/hardware covered in the book	OS requirements
CentOS 7	Windows, macOS X, and Linux (any)
CentOS 8	Windows, macOS X, and Linux (any)
Ubuntu 18.04	Windows, macOS X, and Linux (any)

Code in Action

Code in Action videos for this book can be viewed at <https://bit.ly/32IHmDO>.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: http://www.packtpub.com/sites/default/files/downloads/9781838828714_ColorImages.pdf

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "What we need to do is just uncomment the one pipeline that is defined in the configuration file, located in the `/etc/logstash` folder."

A block of code is set as follows:

```
<memoryBacking>
  <locked/>
</memoryBacking>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
POWER TTWU_QUEUE NO_FORCE_SD_OVERLAP RT_RUNTIME_SHARE NO_LB_MIN
NUMA
NUMA_FAVOUR_HIGHER NO_NUMA_RESIST_LOWER
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "After you push the **Refresh** button, new data should appear on the page."

Tips or important notes
Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Section 1: KVM Virtualization Basics

Part 1 provides you with an insight into the prevailing technologies in Linux virtualization and its advantages over other virtualization solutions. We will discuss the important data structures and the internal implementation of libvirt, QEMU, and KVM.

This part of the book comprises the following chapters:

- *Chapter 1, Understanding Linux Virtualization*
- *Chapter 2, KVM as a Virtualization Solution*

1

Understanding Linux Virtualization

Virtualization is the technology that started a big technology shift toward IT consolidation, which provides more efficient use of resources and the cloud as a more integrated, automated, and orchestrated version of virtualization with a focus on not only virtual machines but also additional services. There are a total of 16 chapters in this book, all of which have been lined up to cover all the important aspects of Kernel-based Virtual Machine (KVM) virtualization. We will start with basic KVM topics such as the history of virtualization concepts and Linux virtualization and then move on and look at advanced topics in KVM such as automation, orchestration, virtual networking, storage, and troubleshooting. This chapter will provide you with an insight into the prevailing technologies in Linux virtualization and their advantages over others.

In this chapter, we will cover the following topics:

- Linux virtualization and its basic concepts
- Types of virtualization
- Hypervisor/VMM
- Open source virtualization projects
- What Linux virtualization offers you in the cloud

Linux virtualization and how it all started

Virtualization is a concept that creates virtualized resources and maps them to physical resources. This process can be done using specific hardware functionality (partitioning, via some kind of partition controller) or software functionality (hypervisor). So, as an example, if you have a physical PC-based server with 16 cores running a hypervisor, you can easily create one or more virtual machines with two cores each and start them up. Limits regarding how many virtual machines you can start is something that's vendor-based. For example, if you're running Red Hat Enterprise Virtualization v4.x (a KVM-based bare-metal hypervisor), you can use up to 768 logical CPU cores or threads (you can read more information about this at <https://access.redhat.com/articles/906543>). In any case, hypervisor is going to be the *go-to guy* that's going to try to manage that as efficiently as possible so that all of the virtual machine workloads get as much time on the CPU as possible.

I vividly remember writing my first article about virtualization in 2004. AMD just came out with its first consumer 64-bit CPUs in 2003 (Athlon 64, Opteron) and it just threw me for a loop a bit. Intel was still a bit hesitant to introduce a 64-bit CPU – a lack of a 64-bit Microsoft Windows OS might have had something to do with that as well. Linux was already out with 64-bit support, but it was a dawn of many new things to come to the PC-based market. Virtualization as such wasn't something revolutionary as an idea since other companies already had non-x86 products that could do virtualization for decades (for example, IBM CP-40 and its S/360-40, from 1967). But it sure was a new idea for a PC market, which was in a weird phase with many things happening at the same time. Switching to 64-bit CPUs with multi-core CPUs appearing on the market, then switching from DDR1 to DDR2, and then from PCI/ISA/AGP to PCI Express, as you might imagine, was a challenging time.

Specifically, I remember thinking about the possibilities – how cool it would be to run an OS, and then another couple of OSes on top of that. Working in the publishing industry, you might imagine how many advantages that would offer to anyone's workflow, and I remember really getting excited about it.

15 or so years of development later, we now have a competitive market in terms of virtualization solutions – Red Hat with KVM, Microsoft with Hyper-V, VMware with ESXi, Oracle with Oracle VM, and Google and other key players duking it out for users and market dominance. This led to the development of various cloud solutions such as EC2, AWS, Office 365, Azure, vCloud Director, and vRealize Automation for various types of cloud services. All in all, it was a very productive 15 years for IT, wouldn't you say?

But, going back to October 2003, with all of the changes that were happening in the IT industry, there was one that was really important for this book and virtualization for Linux in general: the introduction of the first open source Hypervisor for x86 architecture, called **Xen**. It supports various CPU architectures (Itanium, x86, x86_64, and ARM), and it can run various OSes – Windows, Linux, Solaris, and some flavors of BSD – and it's still alive and kicking as a virtualization solution of choice for some vendors, such as Citrix (XenServer) and Oracle (Oracle VM). We'll get into more technical details about Xen a little bit later in this chapter.

The biggest corporate player in the open source market, Red Hat, included Xen virtualization in initial releases of its Red Hat Enterprise Linux 5, which was released in 2007. But Xen and Red Hat weren't exactly a match made in heaven and although Red Hat shipped Xen with its Red Hat Enterprise Linux 5 distribution, Red Hat switched to **KVM** in Red Hat Enterprise Linux 6 in 2010, which was – at the time – a very risky move. Actually, the whole process of migrating from Xen to KVM began in the previous version, with 5.3/5.4 releases, both of which came out in 2009. To put things into context, KVM was a pretty young project back then, just a couple of years old. But there were more than a few valid reasons why that happened, varying from *Xen is not in the mainline kernel, KVM is*, to political reasons (Red Hat wanted more influence over Xen development, and that influence was fading with time).

Technically speaking, KVM uses a different, modular approach that transforms Linux kernels into fully functional hypervisors for supported CPU architectures. When we say *supported CPU architectures*, we're talking about the basic requirement for KVM virtualization – CPUs need to support hardware virtualization extensions, known as AMD-V or Intel VT. To make things a bit easier, let's just say that you're really going to have to try very hard to find a modern CPU that doesn't support these extensions. For example, if you're using an Intel CPU on your server or desktop PC, the first CPUs that supported hardware virtualization extensions date all the way back to 2006 (Xeon LV) and 2008 (Core i7 920). Again, we'll get into more technical details about KVM and provide a comparison between KVM and Xen a little bit later in this chapter and in the next.

Types of virtualization

There are various types of virtualization solutions, all of which are aimed at different use cases and are dependent on the fact that we're virtualizing a different piece of the hardware or software stack, that is, *what* you're virtualizing. It's also worth noting that there are different types of virtualization in terms of *how* you're virtualizing – by partitioning, full virtualization, paravirtualization, hybrid virtualization, or container-based virtualization.

So, let's first cover the five different types of virtualization in today's IT based on *what* you're virtualizing:

- **Desktop virtualization (Virtual Desktop Infrastructure (VDI)):** This is used by a lot of enterprise companies and offers huge advantages for a lot of scenarios because of the fact that users aren't dependent on a specific device that they're using to access their desktop system. They can connect from a mobile phone, tablet, or a computer, and they can usually connect to their virtualized desktop from anywhere as if they're sitting at their workplace and using a hardware computer. Benefits include easier, centralized management and monitoring, much more simplified update workflows (you can update the base image for hundreds of virtual machines in a VDI solution and re-link that to hundreds of virtual machines during maintenance hours), simplified deployment processes (no more physical installations on desktops, workstations, or laptops, as well as the possibility of centralized application management), and easier management of compliance and security-related options.
- **Server virtualization:** This is used by a vast majority of IT companies today. It offers good consolidation of server virtual machines versus physical servers, while offering many other operational advantages over regular, physical servers – easier to backup, more energy efficient, more freedom in terms of moving workloads from server to server, and more.
- **Application virtualization:** This is usually implemented using some kind of streaming/remote protocol technology such as Microsoft App-V, or some solution that can package applications into volumes that can be mounted to the virtual machine and profiled for consistent settings and delivery options, such as VMware App Volumes.

- **Network virtualization** (and a more broader, cloud-based concept called **Software-Defined Networking (SDN)**): This is a technology that creates virtual networks that are independent of the physical networking devices, such as switches. On a much bigger scale, SDN is an extension of the network virtualization idea that can span across multiple sites, locations, or data centers. In terms of the concept of SDN, entire network configuration is done in software, without you necessarily needing a specific physical networking configuration. The biggest advantage of network virtualization is how easy it is for you to manage complex networks that span multiple locations without having to do massive, physical network reconfiguration for all the physical devices on the network data path. This concept will be explained in *Chapter 4, libvirt Networking*, and *Chapter 12, Scaling Out KVM with OpenStack*.
- **Storage virtualization** (and a newer concept **Software-Defined Storage (SDS)**): This is a technology that creates virtual storage devices out of pooled, physical storage devices that we can centrally manage as a single storage device. This means that we're creating some sort of abstraction layer that's going to isolate the internal functionality of storage devices from computers, applications, and other types of resources. SDS, as an extension of that, *decouples* the storage software stack from the hardware it's running on by abstracting control and management planes from the underlying hardware, as well as offering different types of storage resources to virtual machines and applications (block, file, and object-based resources).

If you take a look at these virtualization solutions and scale them up massively (hint: the cloud), that's when you realize that you're going to need various tools and solutions to *effectively* manage the ever-growing infrastructure, hence the development of various automatization and orchestration tools. Some of these tools will be covered later in this book, such as Ansible in *Chapter 11, Ansible for Orchestration and Automation*. For the time being, let's just say that you just can't manage an environment that contains thousands of virtual machines by relying on standard utilities only (scripts, commands, and even GUI tools). You're definitely going to need a more programmatic, API-driven approach that's tightly integrated with the virtualization solution, hence the development of OpenStack, OpenShift, Ansible, and the **Elasticsearch, Logstash, Kibana (ELK)** stack, which we'll cover in *Chapter 14, Monitoring the KVM Virtualization Platform Using the ELK Stack*.

If we're talking about *how* we're virtualizing a virtual machine as an object, there are different types of virtualization:

- **Partitioning:** This is a type of virtualization in which a CPU is divided into different parts, and each part works as an individual system. This type of virtualization solution isolates a server into partitions, each of which can run a separate OS (for example, **IBM Logical Partitions (LPARs)**).
- **Full virtualization:** In full virtualization, a virtual machine is used to simulate regular hardware while not being aware of the fact that it's virtualized. This is done for compatibility reasons – we don't have to modify the guest OS that we're going to run in a virtual machine. We can use a software- and hardware-based approach for this.

Software-based: Uses binary translation to virtualize the execution of sensitive instruction sets while emulating hardware using software, which increases overhead and impacts scalability.

Hardware-based: Removes binary translation from the equation while interfacing with a CPU's virtualization features (AMD-V, Intel VT), which, in turn, means that instruction sets are being executed directly on the host CPU. This is what KVM does (as well as other popular hypervisors, such as ESXi, Hyper-V, and Xen).

- **Paravirtualization:** This is a type of virtualization in which the guest OS understands the fact that it's being virtualized and needs to be modified, along with its drivers, so that it can run on top of the virtualization solution. At the same time, it doesn't need CPU virtualization extensions to be able to run a virtual machine. For example, Xen can work as a paravirtualized solution.
- **Hybrid virtualization:** This is a type of virtualization that uses full virtualization and paravirtualization's biggest virtues – the fact that the guest OS can be run unmodified (full), and the fact that we can insert additional paravirtualized drivers into the virtual machine to work with some specific aspects of virtual machine work (most often, I/O-intensive memory workloads). Xen and ESXi can also work in hybrid virtualization mode.

- **Container-based virtualization:** This is a type of application virtualization that uses containers. A container is an object that packages an application and all its dependencies so that the application can be scaled out and rapidly deployed without needing a virtual machine or a hypervisor. Keep in mind that there are technologies that can operate as both a hypervisor and a container host at the same time. Some examples of this type of technology include Docker and Podman (a replacement for Docker in Red Hat Enterprise Linux 8).

Next, we're going to learn how to use hypervisors.

Using the hypervisor/virtual machine manager

As its name suggests, the **Virtual Machine Manager (VMM)** or hypervisor is a piece of software that is responsible for monitoring and controlling virtual machines or guest OSes. The hypervisor/VMM is responsible for ensuring different virtualization management tasks, such as providing virtual hardware, virtual machine life cycle management, migrating virtual machines, allocating resources in real time, defining policies for virtual machine management, and so on. The VMM/hypervisor is also responsible for efficiently controlling physical platform resources, such as memory translation and I/O mapping. One of the main advantages of virtualization software is its capability to run multiple guests operating on the same physical system or hardware. These multiple guest systems can be on the same OS or different ones. For example, there can be multiple Linux guest systems running as guests on the same physical system. The VMM is responsible for allocating the resources requested by these guest OSes. The system hardware, such as the processor, memory, and so on, must be allocated to these guest OSes according to their configuration, and the VMM can take care of this task. Due to this, the VMM is a critical component in a virtualization environment.

In terms of types, we can categorize hypervisors as either type 1 or type 2.

Type 1 and type 2 hypervisors

Hypervisors are mainly categorized as either type 1 or type 2 hypervisors, based on where they reside in the system or, in other terms, whether the underlying OS is present in the system or not. But there is no clear or standard definition of type 1 and type 2 hypervisors. If the VMM/hypervisor runs directly on top of the hardware, it's generally considered to be a type 1 hypervisor. If there is an OS present, and if the VMM/hypervisor operates as a separate layer, it will be considered as a type 2 hypervisor. Once again, this concept is open to debate and there is no standard definition for this. A type 1 hypervisor directly interacts with the system hardware; it does not need any host OS. You can directly install it on a bare-metal system and make it ready to host virtual machines. Type 1 hypervisors are also called **bare-metal**, **embedded**, or **native hypervisors**. oVirt-node, VMware ESXi/vSphere, and **Red Hat Enterprise Virtualization Hypervisor (RHEV-H)** are examples of a type 1 Linux hypervisor. The following diagram provides an illustration of the type 1 hypervisor design concept:



Figure 1.1 – Type 1 hypervisor design

Here are the advantages of type 1 hypervisors:

- Easy to install and configure
- Small in size; optimized to give most of the physical resources to the hosted guest (virtual machines)
- Generates less overhead as it comes with only the applications needed to run virtual machines
- More secure, because problems in one guest system do not affect the other guest systems running on the hypervisor

However, a type 1 hypervisor doesn't favor customization. Generally, there will be some restrictions when you try to install any third-party applications or drivers on it.

On the other hand, a type 2 hypervisor resides on top of the OS, allowing you to do numerous customizations. Type 2 hypervisors are also known as hosted hypervisors that are dependent on the host OS for their operations. The main advantage of type 2 hypervisors is the wide range of hardware support, because the underlying host OS controls hardware access. The following diagram provides an illustration of the type 2 hypervisor design concept:

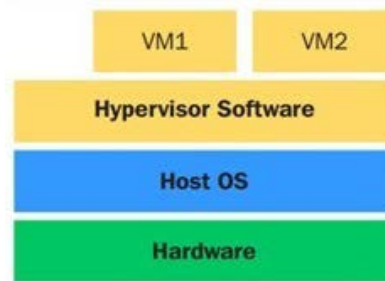


Figure 1.2 – Type 2 hypervisor design

When do we use type 1 versus type 2 hypervisors? It primarily depends on whether we already have an OS running on a server where we want to deploy virtual machines. For example, if we're already running a Linux desktop on our workstation, we're probably not going to format our workstation and install a hypervisor – it just wouldn't make any sense. That's a good example of a type 2 hypervisor use case. Well-known type 2 hypervisors include VMware Player, Workstation, Fusion, and Oracle VirtualBox. On the other hand, if we're specifically aiming to create a server that we're going to use to host virtual machines, then that's type 1 hypervisor territory.

Open source virtualization projects

The following table is a list of open source virtualization projects in Linux:

Project	Virtualization-type	Project-URL
KVM (Kernel-based Virtual Machine)	Full-virtualization	http://www.linux-kvm.org/
VirtualBox	Full-virtualization	https://www.virtualbox.org/
Xen	Full-and-paravirtualization	http://www.xenproject.org/
Lguest	Paravirtualization	http://lguest.ozlabs.org/
UML (User-Mode Linux)	First	http://user-mode-linux.sourceforge.net/
Linux-VServer	First	http://www.linuxvirtualserver.org/

Figure 1.3 – Open source virtualization projects in Linux

In the upcoming sections, we will discuss Xen and KVM, which are the leading open source virtualization solutions in Linux.

Xen

Xen originated at the University of Cambridge as a research project. The first public release of Xen was in 2003. Later, the leader of this project at the University of Cambridge, Ian Pratt, co-founded a company called XenSource with Simon Crosby (also from the University of Cambridge). This company started to develop the project in an open source fashion. On 15 April 2013, the Xen project was moved to the Linux Foundation as a collaborative project. The Linux Foundation launched a new trademark for the Xen Project to differentiate the project from any commercial use of the older Xen trademark. More details about this can be found at <https://xenproject.org/>.

The Xen hypervisor has been ported to a number of processor families, such as Intel IA-32/64, x86_64, PowerPC, ARM, MIPS, and so on.

The core concept of Xen has four main building blocks:

- **Xen hypervisor:** The integral part of Xen that handles intercommunication between the physical hardware and virtual machine(s). It handles all interrupts, times, CPU and memory requests, and hardware interaction.
- **Dom0:** Xen's control domain, which controls a virtual machine's environment. The main part of it is called QEMU, a piece of software that emulates a regular computer system by doing binary translation to emulate a CPU.
- **Management utilities:** Command-line utilities and GUI utilities that we use to manage the overall Xen environment.
- **Virtual machines** (unprivileged domains, DomU): Guests that we're running on Xen.

As shown in the following diagram, Dom0 is a completely separate entity that controls the other virtual machines, while all the other are happily stacked next to each other using system resources provided by the hypervisor:

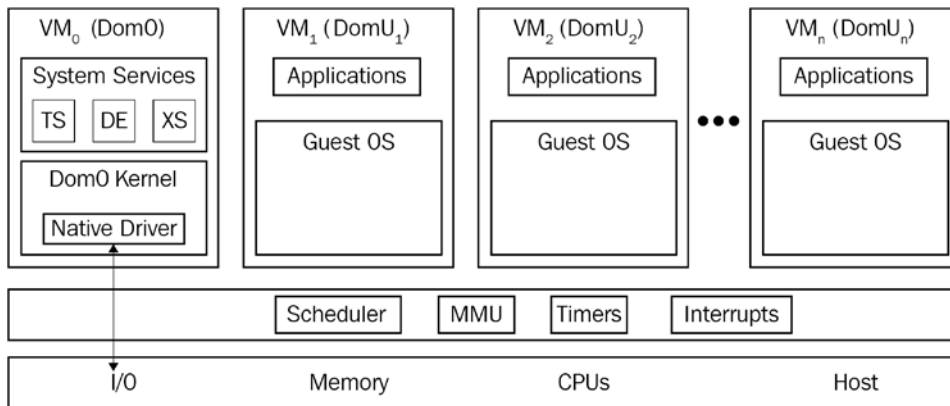


Figure 1.4 – Xen

Some management tools that we're going to mention a bit later in this book are actually capable of working with Xen virtual machines as well. For example, the `virsh` command can be easily used to connect to and manage Xen hosts. On the other hand, oVirt was designed around KVM virtualization and that would definitely not be the preferred solution to manage your Xen-based environment.

KVM

KVM represents the latest generation of open source virtualization. The goal of the project was to create a modern hypervisor that builds on the experience of previous generations of technologies and leverages the modern hardware available today (VT-x, AMD-V, and so on).

KVM simply turns the Linux kernel into a hypervisor when you install the KVM kernel module. However, as the standard Linux kernel is the hypervisor, it benefits from the changes that were made to the standard kernel (memory support, scheduler, and so on). Optimizations for these Linux components, such as the scheduler in the 3.1 kernel, improvement to nested virtualization in 4.20+ kernels, new features for mitigation of Spectre attacks, support for AMD Secure Encrypted Virtualization, Intel iGPU passthrough in 4/5.x kernels, and so on benefit both the hypervisor (the host OS) and the Linux guest OSes. For I/O emulations, KVM uses a userland software, QEMU; this is a userland program that does hardware emulation.

QEMU emulates the processor and a long list of peripheral devices such as the disk, network, VGA, PCI, USB, serial/parallel ports, and so on to build a complete piece of virtual hardware that the guest OS can be installed on. This emulation is powered by KVM.

What Linux virtualization offers you in the cloud

The cloud is *the buzzword* that's been a part of almost all IT-related discussions in the past 10 or so years. If we take a look at the history of cloud, we'll probably realize the fact that Amazon was the first key player in the cloud market, with the release of **Amazon Web Services (AWS)** and **Amazon Elastic Compute Cloud (EC2)** in 2006. Google Cloud Platform was released in 2008, and Microsoft Azure was released in 2010. In terms of the **Infrastructure-as-a-Service (IaaS)** cloud models, these are the biggest IaaS cloud providers now, although there are others (IBM Cloud, VMware Cloud on AWS, Oracle Cloud, and Alibaba Cloud, to name a few). If you go through this list, you'll soon realize that most of these cloud platforms are based on Linux (just as an example, Amazon uses Xen and KVM, while Google Cloud uses KVM virtualization).

Currently, there are three main open source cloud projects that use Linux virtualization to build IaaS solutions for the private and/or hybrid cloud:

- **OpenStack:** A fully open source cloud OS that consists of several open source sub projects that provide all the building blocks to create an IaaS cloud. KVM (Linux virtualization) is the most used (and best-supported) hypervisor in OpenStack deployments. It's governed by the vendor-agnostic OpenStack Foundation. How to build an OpenStack cloud using KVM will be explained in detail in *Chapter 12, Scaling out KVM with OpenStack*
- **CloudStack** This is another open source **Apache Software Foundation (ASF)**-controlled cloud project used to build and manage highly scalable multitenant IaaS clouds and is fully compatible with EC2/S3 APIs. Although it supports all top-level Linux hypervisors, most CloudStack users choose Xen as it is tightly integrated with CloudStack.
- **Eucalyptus:** This is an AWS-compatible private cloud software for organizations to use in order to reduce their public cloud cost and regain control over security and performance. It supports both Xen and KVM as a computing resources provider.

There are other important questions to consider when discussing OpenStack beyond the technical bits and pieces that we've discussed so far in this chapter. One of the most important concepts in IT today is actually being able to run an environment (purely virtualized one, or a cloud environment) that includes various types of solutions (such as virtualization solutions) by using some kind of management layer that's capable of working with different solutions at the same time. Let's take OpenStack as an example of this. If you go through the OpenStack documentation, you'll soon realize that OpenStack supports 10+ different virtualization solutions, including the following:

- KVM
- Xen (via libvirt)
- LXC (Linux containers)
- Microsoft Hyper-V
- VMware ESXi
- Citrix XenServer
- **User Mode Linux (UML)**
- PowerVM (IBM Power 5-9 platform)
- Virtuozzo (hyperconverged solution that can use virtual machines, storage, and containers)
- z/VM (virtualization solution for IBM Z and IBM LinuxONE servers)

That brings us to the multi-cloud environments that could span different CPU architectures, different hypervisors, and other technologies such as hypervisors – all under the same management toolset. This is just one thing that you can do with OpenStack. We'll get back to the subject of OpenStack later in this book, specifically in *Chapter 12, Scaling Out KVM with OpenStack*.

Summary

In this chapter, we covered the basics of virtualization and its different types. Keeping in mind the importance of virtualization in today's large-scale IT world is beneficial as it's good to know how these concepts can be tied together to create a bigger picture – large, virtualized environments and cloud environments. Cloud-based technologies will be covered later in much greater detail – treat what we've mentioned so far as a starter; the main course is still to come. But the next chapter belongs to the main star of our book – the KVM hypervisor and its related utilities.

Questions

1. Which types of hypervisors exist?
2. What are containers?
3. What is container-based virtualization?
4. What is OpenStack?

Further reading

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

- What is KVM?: <https://www.redhat.com/en/topics/virtualization/what-is-kvm>
- KVM hypervisors: https://www.linux-kvm.org/page/Main_Page
- OpenStack Platform: <https://www.openstack.org>
- Xen Project: <https://xenproject.org/>

2

KVM as a Virtualization Solution

In this chapter, we will discuss virtualization as a concept and its implementation via libvirt, **Quick Emulator (QEMU)**, and KVM. Realistically, if we want to explain how virtualization works and why KVM virtualization is such a fundamental part of 21st-century IT, we must start by explaining the technical background of multi-core CPUs and virtualization; and that's impossible to do without delving deep into the theory of CPUs and OSes so that we can get to what we're really after – what hypervisors are and how virtualization actually works.

In this chapter, we will cover the following topics:

- Virtualization as a concept
- The internal workings of libvirt, QEMU, and KVM
- How all these communicate with each other to provide virtualization

Virtualization as a concept

Virtualization is a computing approach that decouples hardware from software. It provides a better, more efficient, and programmatic approach to resource splitting and sharing between various workloads – virtual machines running OSes, and applications on top of them.

If we were to compare traditional, physical computing of the past with virtualization, we can say that by virtualizing, we get the possibility to run multiple guest OSes (multiple virtual servers) on the same piece of hardware (same physical server). If we're using a type 1 hypervisor (explained in *Chapter 1, Understanding Linux Virtualization*), this means that the hypervisor is going to be in charge of letting the virtual servers access physical hardware. This is because there is more than one virtual server using the same hardware as the other virtual servers on the same physical server. This is usually supported by some kind of scheduling algorithm that's implemented programmatically in hypervisors so that we can get more efficiency from the same physical server.

Virtualized versus physical environments

Let's try to visualize these two approaches – physical and virtual. In a physical server, we're installing an OS right on top of the server hardware and running applications on top of that OS. The following diagram shows us how this approach works:

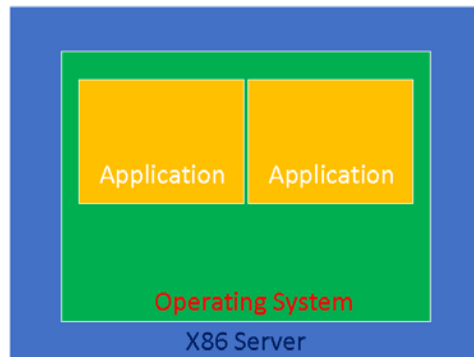


Figure 2.1 – Physical server

In a virtualized world, we're running a hypervisor (such as KVM), and virtual machines on top of that hypervisor. Inside these virtual machines, we're running the same OS and application, just like in the physical server. The virtualized approach is shown in the following diagram:

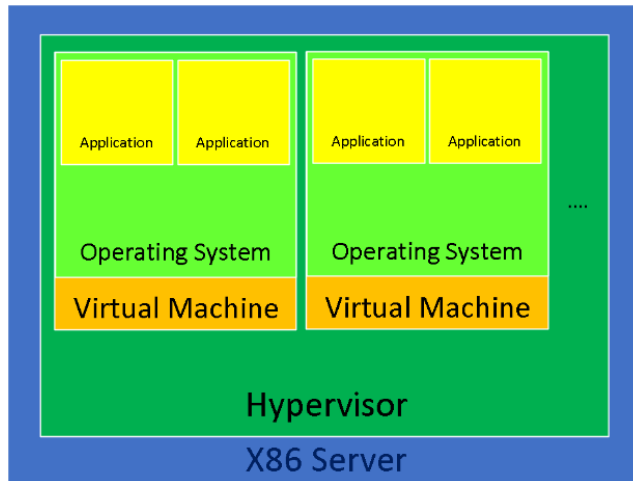


Figure 2.2 – Hypervisor and two virtual machines

There are still various scenarios in which the physical approach is going to be needed. For example, there are still thousands of applications on physical servers all over the world because these servers *can't* be virtualized. There are different reasons *why* they can't be virtualized. For example, the most common reason is actually the simplest reason – maybe these applications are being run on an OS that's not on the supported OS list by the virtualization software vendor. That can mean that you can't virtualize that OS/application combination because that OS doesn't support some virtualized hardware, most commonly a network or a storage adapter. The same general idea applies to the cloud as well – moving things to the cloud isn't always the best idea, as we will describe later in this book.

Why is virtualization so important?

A lot of applications that we run today don't scale up well (adding more CPU, memory, or other resources) – they just aren't programmed that way or can't be seriously parallelized. That means that if an application can't use all the resources at its disposal, a server is going to have a lot of *slack space* – and this time, we're not talking about disk slack space; we're actually referring to *compute* slack space, so slack space at the CPU and memory levels. This means that we're underutilizing the capabilities of the server that we paid for – with the intention for it to be used fully, not partially.

There are other reasons why efficiency and programmatic approaches are so important. The fact of the matter is that beyond their war of press releases in the 2003–2005 timeframe when it was all about the CPU frequency bragging rights (which *equals* CPU speed), Intel and AMD hit a wall in terms of the development of the *single-core* CPU as a concept. They just couldn't cram as many additional elements on the CPU (be it for execution or the cache) and/or bump the single core's speed without seriously compromising the way CPUs were being fed with electrical current. This meant that, at the end of the day, this approach compromised the reliability of the CPU and the whole system that it was running. If you want to learn more about that, we suggest that you look for articles about Intel's NetBurst architecture CPUs (for example, the Prescott core) and their younger brother, Pentium D (the Smithfield core), which was basically two Prescott cores glued together so that the end result was a dual-core CPU. A *very, very hot* dual-core CPU.

A couple of generations before that, there were other techniques that Intel and AMD tried and tested in terms of the *let's have multiple execution units per system* principle. For example, we had Intel Pentium Pro dual-socket systems and AMD Opteron dual- and quad-socket systems. We'll come back to these later in this book when we start discussing some very important aspects of virtualization (for example, **Non-Unified Memory Access (NUMA)**).

So, whichever way you look at it, when PC CPUs started getting multiple cores in 2005 (AMD being the first to the market with a server multi-core CPU, and Intel being the first to the market with a desktop multi-core CPU), it was the only rational way to go forward. These cores were smaller, more efficient (drawing less power), and were generally a better long-term approach. Of course, that meant that OSes and applications had to be reworked heavily if companies such as Microsoft and Oracle wanted to use their applications and reap the benefits of a multi-core server.

In conclusion, for PC-based servers, looking from the CPU perspective, switching to multi-core CPUs was an opportune moment to start working toward virtualization as the concept that we know and love today.

In parallel with these developments, CPUs got other additions – for example, additional CPU registers that can handle specific types of operations. A lot of people heard about instruction sets such as MMX, SSE, SSE2, SSE3, SSE4.x, AVX, AVX2, AES, and so on. These are all very important today as well because they give us a possibility of *offloading* certain instruction types to a specific CPU register. This means that these instructions don't have to be run on a CPU as a general serial device, which executes these tasks slower. Instead, these instructions can be sent to a specific CPU register that's specialized for these instructions. Think of it as having separate mini accelerators on a CPU die that could run some pieces of the software stack without hogging the general CPU pipeline. One of these additions was **Virtual Machine Extensions (VMX)** for Intel, or **AMD Virtualization (AMD-V)**, both of which enable us to have full, hardware-based virtualization support for their respective platforms.

Hardware requirements for virtualization

After the introduction of software-based virtualization on PCs, a lot of development was made, both on the hardware and software sides. The end result – as we mentioned in the previous chapter – was a CPU that had an awful lot more features and power. This led to a big push toward hardware-assisted virtualization, which – on paper – looked like the faster and more advanced way to go. Just as an example, there were a whole bunch of CPUs that didn't support hardware-assisted virtualization in the 2003–2006 timeframe, such as the Intel Pentium 4, Pentium D, the initial AMD Athlons, Turions, Durons, and so on. It took both Intel and AMD until 2006 to have hardware-assisted virtualization as a feature that's more widely available on their respective CPUs. Furthermore, it took some time to have 64-bit CPUs, and there was little or no interest in running hardware-assisted virtualization on 32-bit architectures. The primary reason for this was the fact that you couldn't allocate more than 4 GB of memory, which severely limited the scope of using virtualization as a concept.

Keeping all of this in mind, these are the requirements that we have to comply with today so that we can run modern-day hypervisors with full hardware-assisted virtualization support:

- **Second-Level Address Translation, Rapid Virtualization Indexing, Extended Page Tables (SLAT/RVI/EPT) support:** This is the CPU technology that a hypervisor uses so that it can have a map of virtual-to-physical memory addresses. Virtual machines operate in a virtual memory space that can be scattered all over the physical memory, so by using an additional map such as SLAT/EPT, (implemented via an additional **Translation Lookaside Buffer**, or **TLB**), you're reducing latency for memory access. If we didn't have a technology like this, we'd have to have physical memory access to the computer memory's physical addresses, which would be messy, insecure, and latency-prone. To avoid any confusion, EPT is Intel's name for SLAT technology in their CPUs (AMD uses RVI terminology, while Intel uses EPT terminology).
- **Intel VT or AMD-V support:** If an Intel CPU has VT (or an AMD CPU has AMD-V), that means that it supports hardware virtualization extensions and full virtualization.
- **Long mode support**, which means that the CPU has 64-bit support. Without a 64-bit architecture, virtualization would be basically useless because you'd have only 4 GB of memory to give to virtual machines (which is a limitation of the 32-bit architecture). By using a 64-bit architecture, we can allocate much more memory (depending on the CPU that we're using), which means more opportunities to feed virtual machines with memory, without which the whole virtualization concept wouldn't make any sense in the 21st-century IT space.
- **The possibility of having Input/Output Memory Management Unit (IOMMU) virtualization (such as AMD-Vi, Intel VT-d, and stage 2 tables on ARM)**, which means that we allow virtual machines to access peripheral hardware directly (graphics cards, storage controllers, network devices, and so on). This functionality must be enabled both on the CPU and motherboard chipset/firmware side.
- **The possibility to do Single Root Input Output Virtualization (SR-IOV)**, which allows us to directly forward a PCI Express device (for example, an Ethernet port) to multiple virtual machines. The key aspect of SR-IOV is its ability to share one physical device with multiple virtual machines via functionality called **Virtual Functions (VFs)**. This functionality requires hardware and driver support.

- **The possibility to do PCI passthrough**, which means we can take a PCI Express connected card (for example, a video card) connected to a server motherboard and present it to a virtual machine as if that card was directly connected to the virtual machine via functionality called **Physical Functions (PFs)**. This means bypassing various hypervisor levels that the connection would ordinarily take place through.
- **Trusted Platform Module (TPM) support**, which is usually implemented as an additional motherboard chip. Using TPM can have a lot of advantages in terms of security because it can be used to provide cryptographic support (that is, to create, save, and secure the use of cryptographic keys). There was quite a bit of buzz in the Linux world around the use of TPM with KVM virtualization, which led to Intel's open sourcing of the TPM2 stack in the summer of 2018.

When discussing SR-IOV and PCI passthrough, make sure that you take note of the core functionalities, called PF and VF. These two keywords will make it easier to remember *where* (on a physical or virtual level) and *how* (directly or via a hypervisor) devices are forwarded to their respective virtual machines. These capabilities are very important for the enterprise space and quite a few specific scenarios. Just as an example, there's literally no way to have a **virtual desktop infrastructure (VDI)** solution with workstation-grade virtual machines that you can use to run AutoCAD and similar applications without these capabilities. This is because integrated graphics on CPUs are just too slow to do that properly. That's when you start adding GPUs to your servers – so that you can use a hypervisor to forward the *whole* GPU or *parts* of it to a virtual machine or multiple virtual machines.

In terms of system memory, there are also various subjects to consider. AMD started integrating memory controllers into CPUs in Athlon 64, which was years before Intel did that (Intel did that first with the Nehalem CPU core, which was introduced in 2008). Integrating a memory controller into a CPU meant that your system had less latency when CPU accessed memory for memory I/O operations. Before this, the memory controller was integrated into what was called a NorthBridge chip, which was a separate chip on a system motherboard that was in charge of all fast buses and memory. But that means additional latency, especially when you try to scale out that principle to multi-socket, multi-core CPUs. Also, with the introduction of Athlon 64 on Socket 939, AMD switched to a dual-channel memory architecture, which is now a familiar theme in the desktop and server market. Triple and quad-channel memory controllers are de facto standards in servers. Some of the latest Intel Xeon CPUs support six-channel memory controllers, and AMD EPYC CPUs support eight-channel memory controllers as well. This has huge implications for the overall memory bandwidth and latency, which – in turn – has huge implications for the speed of memory-sensitive applications, both on physical and virtual servers.

Why is this important? The more channels you have and the lower the latency is, the more bandwidth you have from CPU to memory. And that is very, very desirable for a lot of workloads in today's IT space (for example, databases).

Software requirements for virtualization

Now that we've covered the basic hardware aspects of virtualization, let's move on to the software aspect of virtualization. To do that, we must cover some jargon in computer science. That being said, let's start with something called protection rings. In computer science, various hierarchical protection domains/privileged rings exist. These are the mechanisms that protect data or faults based on the security that's enforced when accessing the resources in a computer system. These protection domains contribute to the security of a computer system. By imagining these protection rings as instruction zones, we can represent them via the following diagram:

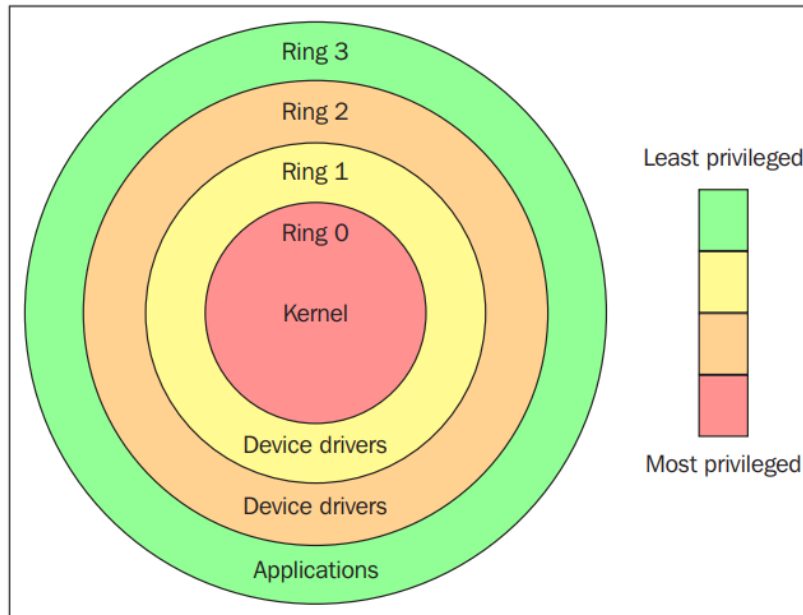


Figure 2.3 – Protection rings (source: https://en.wikipedia.org/wiki/Protection_ring)

As shown in the preceding diagram, the protection rings are numbered from the most privileged to the least privileged. Ring 0 is the level with the most privilege and interacts directly with physical hardware, such as the CPU and memory. The resources, such as memory, I/O ports, and CPU instructions, are protected via these privileged rings. Rings 1 and 2 are mostly unused. Most general-purpose systems use only two rings, even if the hardware they run on provides more CPU modes than that. The two main CPU modes are the kernel mode and the user mode, which are also related to the way processes are executed. You can read more about it at this link: https://access.redhat.com/sites/default/files/attachments/processstates_20120831.pdf. From an OS's point of view, ring 0 is called the kernel mode/supervisor mode and ring 3 is the user mode. As you may have assumed, applications run in ring 3.

OSes such as Linux and Windows use supervisor/kernel and user mode. This mode can do almost nothing to the outside world without calling on the kernel or without its help due to its restricted access to memory, CPU, and I/O ports. The kernels can run in privileged mode, which means that they can run on ring 0. To perform specialized functions, the user-mode code (all the applications that run in ring 3) must perform a system call to the supervisor mode or even to the kernel space, where the trusted code of the OS will perform the needed task and return the execution back to the userspace. In short, the OS runs in ring 0 in a normal environment. It needs the most privileged level to do resource management and provide access to the hardware. The following diagram explains this:

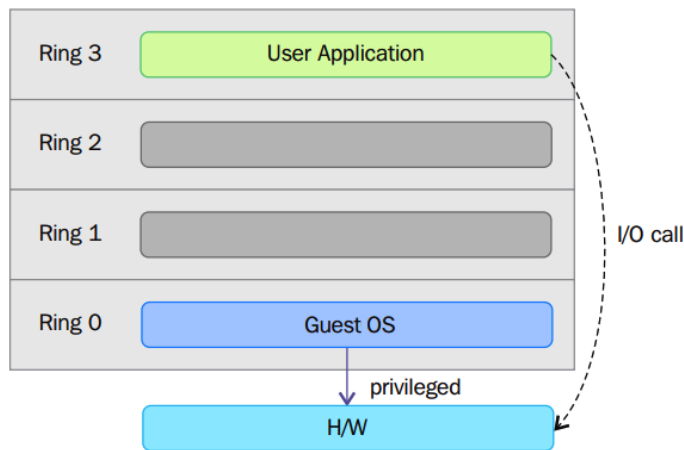


Figure 2.4 – System call to supervisor mode

The rings above 0 run instructions in a processor mode called unprotected. The hypervisor/**Virtual Machine Monitor (VMM)** needs to access the memory, CPU, and I/O devices of the host. Since only the code running in ring 0 is allowed to perform these operations, it needs to run in the most privileged ring, which is ring 0, and has to be placed next to the kernel. Without specific hardware virtualization support, the hypervisor or VMM runs in ring 0; this basically blocks the virtual machine's OS in ring 0. So, the virtual machine's OS must reside in ring 1. An OS installed in a virtual machine is also expected to access all the resources as it's unaware of the virtualization layer; to achieve this, it has to run in ring 0, similar to the VMM. Due to the fact that only one kernel can run in ring 0 at a time, the guest OSes have to run in another ring with fewer privileges or have to be modified to run in user mode.

This has resulted in the introduction of a couple of virtualization methods called full virtualization and paravirtualization, which we mentioned earlier. Now, let's try to explain them in a more technical way.

Full virtualization

In full virtualization, privileged instructions are emulated to overcome the limitations that arise from the guest OS running in ring 1 and the VMM running in ring 0. Full virtualization was implemented in first-generation x86 VMMs. It relies on techniques such as binary translation to trap and virtualize the execution of certain sensitive and non-virtualizable instructions. This being said, in binary translation, some system calls are interpreted and dynamically rewritten. The following diagram depicts how the guest OS accesses the host computer hardware through ring 1 for privileged instructions and how unprivileged instructions are executed without the involvement of ring 1:

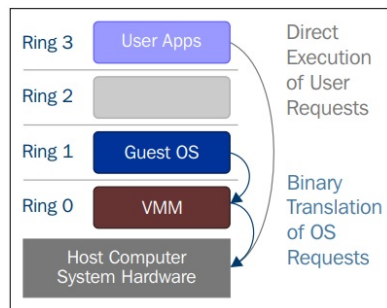


Figure 2.5 – Binary translation

With this approach, the critical instructions are discovered (statically or dynamically at runtime) and replaced with traps in the VMM that are to be emulated in software. A binary translation can incur a large performance overhead in comparison to a virtual machine running on natively virtualized architectures. This can be seen in the following diagram:

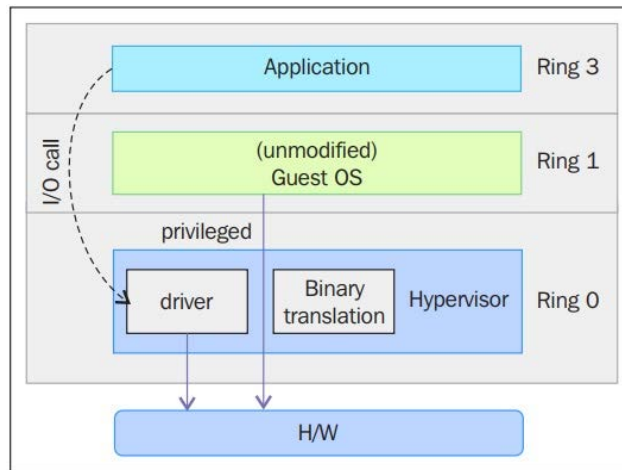


Figure 2.6 – Full virtualization

However, as shown in the preceding diagram, when we use full virtualization, we can use the unmodified guest OSes. This means that we don't have to alter the guest kernel so that it runs on a VMM. When the guest kernel executes privileged operations, the VMM provides the CPU emulation to handle and modify the protected CPU operations. However, as we mentioned earlier, this causes performance overhead compared to the other mode of virtualization, called paravirtualization.

Paravirtualization

In paravirtualization, the guest OS needs to be modified to allow those instructions to access ring 0. In other words, the OS needs to be modified to communicate between the VMM/hypervisor and the guest through the *backend* (hypercalls) path:

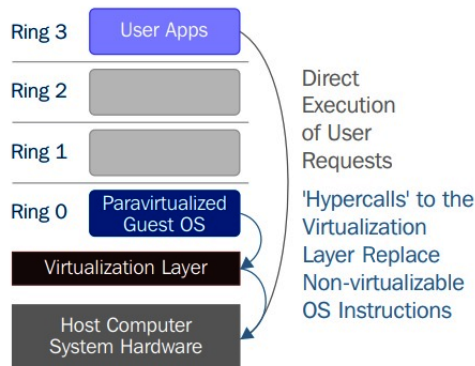


Figure 2.7 – Paravirtualization

Paravirtualization (<https://en.wikipedia.org/wiki/Paravirtualization>) is a technique in which the hypervisor provides an API, and the OS of the guest virtual machine calls that API, which requires host OS modifications. Privileged instruction calls are exchanged with the API functions provided by the VMM. In this case, the modified guest OS can run in ring 0.

As you can see, under this technique, the guest kernel is modified to run on the VMM. In other words, the guest kernel knows that it's been virtualized. The privileged instructions/operations that are supposed to run in ring 0 have been replaced with calls known as hypercalls, which talk to the VMM. These hypercalls invoke the VMM so that it performs the task on behalf of the guest kernel. Since the guest kernel can communicate directly with the VMM via hypercalls, this technique results in greater performance compared to full virtualization. However, this requires a specialized guest kernel that is aware of paravirtualization and comes with needed software support.

The concepts of paravirtualization and full virtualization used to be a common way to do virtualization but not in the best possible, manageable way. That's where hardware-assisted virtualization comes into play, as we will describe in the following section.

Hardware-assisted virtualization

Intel and AMD realized that full virtualization and paravirtualization are the major challenges of virtualization on the x86 architecture (since the scope of this book is limited to x86 architecture, we will mainly discuss the evolution of this architecture here) due to the performance overhead and complexity of designing and maintaining the solution. Intel and AMD independently created new processor extensions of the x86 architecture, called Intel VT-x and AMD-V, respectively. On the Itanium architecture, hardware-assisted virtualization is known as VT-i. Hardware-assisted virtualization is a platform virtualization method designed to efficiently use full virtualization with the hardware capabilities. Various vendors call this technology by different names, including accelerated virtualization, hardware virtual machine, and native virtualization.

For better support for virtualization, Intel and AMD introduced **Virtualization Technology (VT)** and **Secure Virtual Machine (SVM)**, respectively, as extensions of the IA-32 instruction set. These extensions allow the VMM/hypervisor to run a guest OS that expects to run in kernel mode, in lower privileged rings. Hardware-assisted virtualization not only proposes new instructions but also introduces a new privileged access level, called ring -1, where the hypervisor/VMM can run. Hence, guest virtual machines can run in ring 0. With hardware-assisted virtualization, the OS has direct access to resources without any emulation or OS modification. The hypervisor or VMM can now run at the newly introduced privilege level, ring -1, with the guest OSes running on ring 0. Also, with hardware-assisted virtualization, the VMM/hypervisor is relaxed and needs to perform less work compared to the other techniques mentioned, which reduces the performance overhead. This capability to run directly in ring -1 can be described with the following diagram:

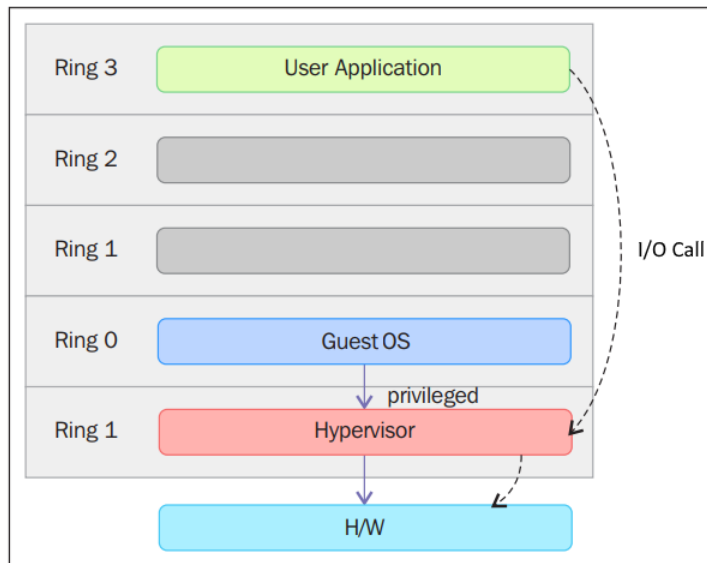


Figure 2.8 – Hardware-assisted virtualization

In simple terms, this virtualization-aware hardware provides us with support to build the VMM and also ensures the isolation of a guest OS. This helps us achieve better performance and avoid the complexity of designing a virtualization solution. Modern virtualization techniques make use of this feature to provide virtualization. One example is KVM, which we are going to discuss in detail throughout this book.

Now that we've covered the hardware and software aspects of virtualization, let's see how all of this applies to KVM as a virtualization technology.

The internal workings of libvirt, QEMU, and KVM

The interaction of libvirt, QEMU, and KVM is something that gives us the full virtualization capabilities that are covered in this book. They are the most important pieces in the Linux virtualization puzzle, as each has a role to play. Let's describe what they do and how they interact with each other.

libvirt

When working with KVM, you're most likely to first interface with its main **Application Programming Interface (API)**, called libvirt (<https://libvirt.org>). But libvirt has other functionalities – it's also a daemon and a management tool for different hypervisors, some of which we mentioned earlier. One of the most common tools used to interface with libvirt is called virt-manager (<http://virt-manager.org>), a Gnome-based graphical utility that you can use to manage various aspects of your local and remote hypervisors, if you so choose. libvirt's CLI utility is called virsh. Keep in mind that you can manage remote hypervisors via libvirt, so you're not restricted to a local hypervisor only. That's why virt-manager has an additional parameter called `--connect`. libvirt is also part of various other KVM management tools, such as oVirt (<http://www.ovirt.org>), which we will discuss in the next chapter.

The goal of the libvirt library is to provide a common and stable layer for managing virtual machines running on a hypervisor. In short, as a management layer, it is responsible for providing the API that performs management tasks such as virtual machine provision, creation, modification, monitoring, control, migration, and so on. In Linux, you will have noticed that some of the processes are daemonized. The libvirt process is also daemonized, and it is called `libvirtd`. As with any other daemon process, `libvirtd` provides services to its clients upon request. Let's try to understand what exactly happens when a libvirt client such as `virsh` or `virt-manager` requests a service from `libvirtd`. Based on the connection URI (discussed in the following section) that's passed by the client, `libvirtd` opens a connection to the hypervisor. This is how the client's `virsh` or `virt-manager` asks `libvirtd` to start talking to the hypervisor. In the scope of this book, we are aiming to look at KVM virtualization technology. So, it would be better to think about it in terms of a QEMU/KVM hypervisor instead of discussing some other hypervisor communication from `libvirtd`. You may be a bit confused when you see QEMU/KVM as the underlying hypervisor name instead of either QEMU or KVM. But don't worry – all will become clear in due course. The connection between QEMU and KVM will be discussed in the following chapters. For now, just know that there is a hypervisor that uses both the QEMU and KVM technologies.

Connecting to a remote system via virsh

A simple command-line example of a `virsh` binary for a remote connection would be as follows:

```
virsh --connect qemu+ssh://root@remoteserver.yourdomain.com/
system list --all
```

Let's take a look at the source code now. We can get the libvirt source code from the libvirt Git repository:

```
[root@kvmsource]# yum -y install git-core
```

```
[root@kvmsource]# git clone git://libvirt.org/libvirt.git
```

Once you clone the repo, you can see the following hierarchy of files in the repo:

```
[root@kvmsource qemu]# ls
accel          dump          meson_options.txt  qga
audio         exec.c       migration         qobject
authz        exec-vary.c  module-common.c  qom
backends     fsdev       monitor         README.rst
block        gdbstub.c   nbd             replay
block.c      gdb-xml     net            replication.c
blockdev.c   gitdm.config os-posix.c     replication.h
blockdev-nbd.c  gitdm.config os-win32.c     roms
blockjob.c   hmp-commands.hx  pc-bios       rules.mak
bootdevice.c hmp-commands-info.hx  plugins      scripts
bsd-user     hw          po             scsi
capstone     include     python         slirp
Changelog    io          qapi          softmmu
chardev      iothread.c  qdev-monitor.c storage-daemon
CODING_STYLE.rst  job.c      qemu-bridge-helper.c  stubs
configure     job-qmp.c  qemu-edid.c     target
contrib       Kconfig    qemu-img.c      tcg
COPYING       Kconfig.host  qemu-img-cmds.hx  tests
COPYING.LIB   libdecnumber  qemu-io.c       thunk.c
cpus-common.c LICENSE     qemu-io-cmds.c  tools
crypto        linux-headers  qemu-keymap.c   tpm.c
default-configs  linux-user   qemu-nbd.c      trace
device_tree.c  MAINTAINERS  qemu-nsi        trace-events
disas         Makefile     qemu-options.h  ui
disas.c       Makefile.objs  qemu-options.hx  util
dma-helpers.c  memory_ldst.c.inc  qemu-options-wrapper.h  VERSION
docs          meson        qemu.sasl       version.rc
dtd           meson.build  qemu-seccomp.c  version.texi.in
```

Figure 2.9 – QEMU source content, downloaded via Git

libvirt code is based on the C programming language; however, libvirt has language bindings in different languages, such as C#, Java, OCaml, Perl, PHP, Python, Ruby, and so on. For more details on these bindings, please refer to <https://libvirt.org/bindings.html>. The main (and few) directories in the source code are `docs`, `daemon`, `src`, and so on. The libvirt project is well documented and the documentation is available in the source code repo and also at <http://libvirt.org>.

libvirt uses a *driver-based architecture*, which enables libvirt to communicate with various external hypervisors. This means that libvirt has internal drivers that are used to interface with other hypervisors and solutions, such as LXC, Xen, QEMU, VirtualBox, Microsoft Hyper-V, bhyve (BSD hypervisor), IBM PowerVM, OpenVZ (open Virtuozzo container-based solution), and others, as shown in the following diagram:

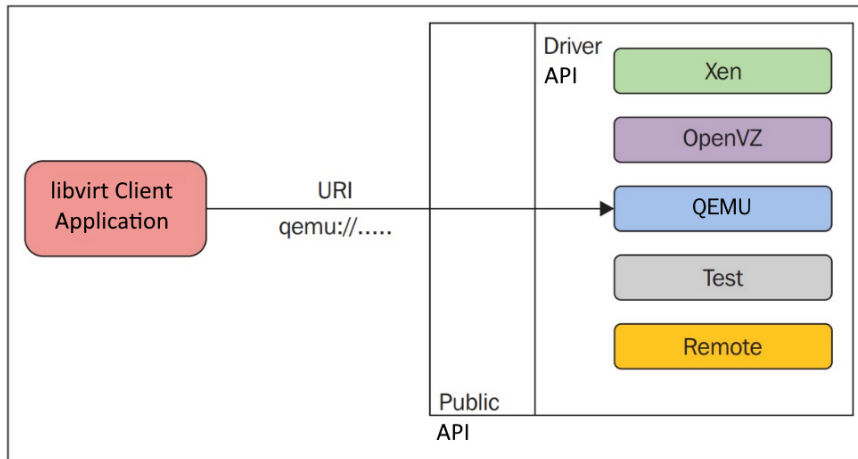


Figure 2.10 – Driver-based architecture

The ability to connect to various virtualization solutions gets us much more usability out of the `virsh` command. This might come in very handy in mixed environments, such as if you're connecting to both KVM and XEN hypervisors from the same system.

As in the preceding figure, there is a **public API** that is exposed to the outside world. Depending on the connection URI (for example, `virsh --connect QEMU://xxxx/system`) passed by the clients, when initializing the library, this public API uses internal drivers in the background. Yes, there are different categories of driver implementations in libvirt. For example, there are `hypervisor`, `interface`, `network`, `nodeDevice`, `nwfilter`, `secret`, `storage`, and so on. Refer to `driver.h` inside the libvirt source code to learn about the driver data structures and other functions associated with the different drivers.

Take the following example:

```
struct _virConnectDriver {
    virHypervisorDriverPtr hypervisorDriver;
    virInterfaceDriverPtr interfaceDriver;
    virNetworkDriverPtr networkDriver;
    virNodeDeviceDriverPtr nodeDeviceDriver;
```

```
virNWFilterDriverPtr nwfilterDriver;  
virSecretDriverPtr secretDriver;  
virStorageDriverPtr storageDriver;  
};
```

The `struct` fields are self-explanatory and convey which type of driver is represented by each of the field members. As you might have assumed, one of the important or main drivers is the hypervisor driver, which is the driver implementation of different hypervisors supported by libvirt. The drivers are categorized as **primary** and **secondary** drivers. The hypervisor driver is an example of a primary driver. The following list gives us some idea about the hypervisors supported by libvirt. In other words, hypervisor-level driver implementations exist for the following hypervisors (check the `README` and the libvirt source code):

- `bhyve`: The BSD hypervisor
- `esx`/: VMware ESX and GSX support using vSphere API over SOAP
- `hyperv`/: Microsoft Hyper-V support using WinRM
- `lxc`/: Linux native containers
- `openvz`/: OpenVZ containers using CLI tools
- `phyp`/: IBM Power Hypervisor using CLI tools over SSH
- `qemu`/: QEMU/KVM using the QEMU CLI/monitor
- `remote`/: Generic libvirt native RPC client
- `test`/: A *mock* driver for testing
- `uml`/: User-mode Linux
- `vbox`/: VirtualBox using the native API
- `vmware`/: VMware Workstation and Player using the `vmrun` tool
- `xen`/: Xen using hypercalls, XenD SEXPR, and XenStore
- `xenapi`: Xen using `libxenserver`

Previously, we mentioned that there are secondary-level drivers as well. Not all, but some secondary drivers (see the following) are shared by several hypervisors. That said, currently, these secondary drivers are used by hypervisors such as the LXC, OpenVZ, QEMU, UML, and Xen drivers. The ESX, Hyper-V, Power Hypervisor, Remote, Test, and VirtualBox drivers all implement secondary drivers directly.

Examples of secondary-level drivers include the following:

- `cpu/`: CPU feature management
- `interface/`: Host network interface management
- `network/`: Virtual NAT networking
- `nwfilter/`: Network traffic filtering rules
- `node_device/`: Host device enumeration
- `secret/`: Secret management
- `security/`: Mandatory access control drivers
- `storage/`: Storage management drivers

libvirt is heavily involved in regular management operations, such as the creating and managing of virtual machines (guest domains). Additional secondary drivers are consumed to perform these operations, such as interface setup, firewall rules, storage management, and general provisioning of APIs. The following is from <https://libvirt.org/api.html>:

"OnDevice the application obtains a virConnectPtr connection to the hypervisor it can then use to manage the hypervisor's available domains and related virtualization resources, such as storage and networking. All those are exposed as first class objects and connected to the hypervisor connection (and the node or cluster where it is available)."

The following figure shows the five main objects exported by the API and the connections between them:

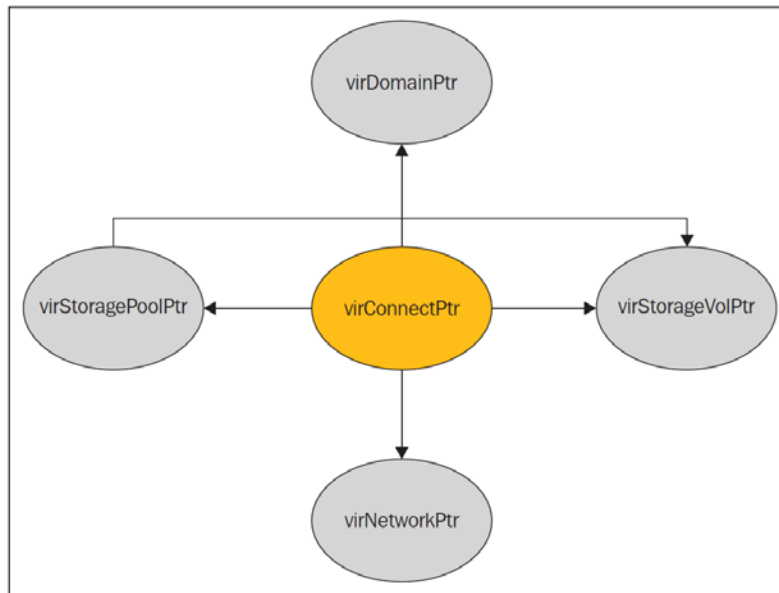
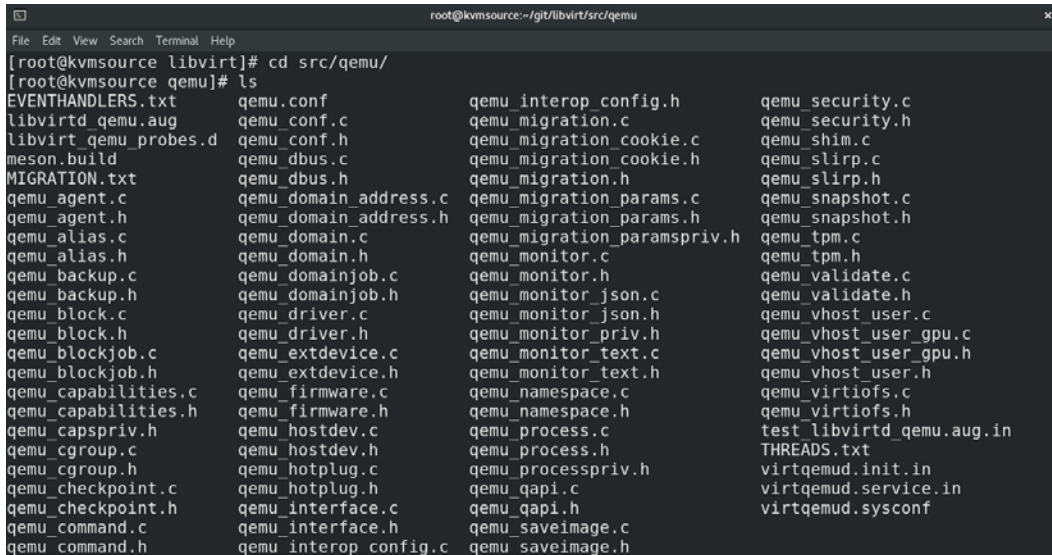


Figure 2.11 – Exported API objects and their communication

Let's give some details about the main objects available in the libvirt code. Most functions inside libvirt make use of these objects for their operations:

- `virConnectPtr`: As we discussed earlier, libvirt has to connect to a hypervisor and act. The connection to the hypervisor has been represented as this object. This object is one of the core objects in libvirt's API.
- `virDomainPtr`: Virtual machines or guest systems are generally referred to as domains in libvirt code. `virDomainPtr` represents an object to an active/defined domain/virtual machine.
- `virStorageVolPtr`: There are different storage volumes, exposed to the domains/guest systems. `virStorageVolPtr` generally represents one of the volumes.
- `virStoragePoolPtr`: The exported storage volumes are part of one of the storage pools. This object represents one of the storage pools.
- `virNetworkPtr`: In libvirt, we can define different networks. A single virtual network (active/defined status) is represented by the `virNetworkPtr` object.

You should now have some idea about the internal structure of libvirt implementations; this can be expanded further:



```

root@kvmsource:~/git/libvirt/src/qemu
File Edit View Search Terminal Help
[root@kvmsource libvirt]# cd src/qemu/
[root@kvmsource qemu]# ls
EVENTHANDLERS.txt      qemu.conf              qemu_interop_config.h  qemu_security.c
libvirtd_qemu.aug      qemu_conf.c            qemu_migration.c       qemu_security.h
libvirt_qemu_probes.d  qemu_conf.h            qemu_migration_cookie.c  qemu_shim.c
meson.build            qemu_dbus.c            qemu_migration_cookie.h  qemu_slirp.c
MIGRATION.txt         qemu_dbus.h            qemu_migration.h        qemu_slirp.h
qemu_agent.c           qemu_domain_address.c  qemu_migration_params.c  qemu_snapshot.c
qemu_agent.h           qemu_domain_address.h  qemu_migration_params.h  qemu_snapshot.h
qemu_alias.c           qemu_domain.c           qemu_migration_paramspriv.h  qemu_tpm.c
qemu_alias.h           qemu_domain.h           qemu_monitor.c          qemu_tpm.h
qemu_backup.c          qemu_domainjob.c       qemu_monitor.h          qemu_validate.c
qemu_backup.h          qemu_domainjob.h       qemu_monitor_json.c     qemu_validate.h
qemu_block.c           qemu_driver.c           qemu_monitor_json.h     qemu_vhost_user.c
qemu_block.h           qemu_driver.h           qemu_monitor_priv.h     qemu_vhost_user_gpu.c
qemu_blockjob.c        qemu_extdevice.c        qemu_monitor_text.c     qemu_vhost_user_gpu.h
qemu_blockjob.h        qemu_extdevice.h        qemu_monitor_text.h     qemu_vhost_user.h
qemu_capabilities.c    qemu_firmware.c         qemu_namespace.c        qemu_virtiofs.c
qemu_capabilities.h    qemu_firmware.h         qemu_namespace.h        qemu_virtiofs.h
qemu_capspriv.h        qemu_hostdev.c          qemu_process.c           test_libvirtd_qemu.aug.in
qemu_cgroup.c          qemu_hostdev.h          qemu_process.h           THREADS.txt
qemu_cgroup.h          qemu_hotplug.c          qemu_processpriv.h      virtqemud.init.in
qemu_checkpoint.c      qemu_hotplug.h          qemu_qapi.c              virtqemud.service.in
qemu_checkpoint.h      qemu_interface.c        qemu_qapi.h              virtqemud.sysconf
qemu_command.c         qemu_interface.h        qemu_saveimage.c
qemu_command.h         qemu_interop_config.c   qemu_saveimage.h

```

Figure 2.12 – libvirt source code

Our area of interest is QEMU/KVM. So, let's explore it further. Inside the `src` directory of the libvirt source code repository, there is a directory for QEMU hypervisor driver implementation code. Pay some attention to the source files, such as `qemu_driver.c`, which carries core driver methods for managing QEMU guests.

See the following example:

```

static virDrvOpenStatus qemuConnectOpen(virConnectPtr conn,
                                         virConnectAuthPtr auth
ATTRIBUTE_UNUSED,
                                         unsigned int flags)

```

libvirt makes use of different driver codes to probe the underlying hypervisor/emulator. In the context of this book, the component of libvirt responsible for finding out the QEMU/KVM presence is the QEMU driver code. This driver probes for the `qemu-kvm` binary and the `/dev/kvm` device node to confirm that the KVM fully virtualized hardware-accelerated guests are available. If these are not available, the possibility of a QEMU emulator (without KVM) is verified with the presence of binaries such as `qemu`, `qemu-system-x86_64`, `qemu-system-mips`, `qemu-system-microblaze`, and so on.

The validation can be seen in `qemu_capabilities.c`:

```

from (qemu_capabilities.c)

static int virQEMUCapsInitGuest ( .., .., virArch hostarch,
virArch guestarch)
{
...
binary = virQEMUCapsFindBinaryForArch (hostarch, guestarch);
...
native_kvm = (hostarch == guestarch);
x86_32on64_kvm = (hostarch == VIR_ARCH_X86_64 && guestarch ==
VIR_ARCH_I686);
...
if (native_kvm || x86_32on64_kvm || arm_32on64_kvm || ppc64_
kvm) {

    const char *kvmbins[] = {
        "/usr/libexec/qemu-kvm", /* RHEL */
        "qemu-kvm", /* Fedora */
        "kvm", /* Debian/Ubuntu */    ...};
    ...
    kvmbin = virFindFileInPath(kvmbins[i]);
    ...
    virQEMUCapsInitGuestFromBinary (caps, binary, qemubinCaps,
kvmbin, kvmbinCaps, guestarch);
    ...
}

```

Then, KVM enablement is performed as shown in the following code snippet:

```

int virQEMUCapsInitGuestFromBinary(..., *binary, qemubinCaps,
*kvmbin, kvmbinCaps, guestarch)
{
.....
    if (virFileExists("/dev/kvm") && (virQEMUCapsGet(qemubinCaps,
QEMU_CAPS_KVM) ||
        virQEMUCapsGet(qemubinCaps, QEMU_CAPS_ENABLE_KVM) ||
kvmbin))
        haskvm = true;

```

Basically, libvirt's QEMU driver is looking for different binaries in different distributions and different paths – for example, `qemu-kvm` in RHEL/Fedora. Also, it finds a suitable QEMU binary based on the architecture combination of both host and guest. If both the QEMU binary and KVM are found, then KVM is fully virtualized and hardware-accelerated guests will be available. It's also libvirt's responsibility to form the entire command-line argument for the QEMU-KVM process. Finally, after forming the entire command-line (`qemu_``command.c`) arguments and inputs, libvirt calls `exec()` to create a QEMU-KVM process:

```
util/vircommand.c
static int virExec(virCommandPtr cmd) {
...
if (cmd->env)
    execve(binary, cmd->args, cmd->env);
else
    execv(binary, cmd->args);
```

In KVMland, there is a misconception that libvirt directly uses the device file (`/dev/kvm`) exposed by KVM kernel modules, and instructs KVM to do the virtualization via the different `ioctl()` function calls available with KVM. This is indeed a misconception! As mentioned earlier, libvirt spawns the QEMU-KVM process and QEMU talks to the KVM kernel modules. In short, QEMU talks to KVM via different `ioctl()` to the `/dev/kvm` device file exposed by the KVM kernel module. To create a virtual machine (for example, `virsh create`), all libvirt does is spawn a QEMU process, which in turn creates the virtual machine. Please note that a separate QEMU-KVM process is launched for each virtual machine by `libvirtd`. Properties of virtual machines (the number of CPUs, memory size, I/O device configuration, and so on) are defined in separate XML files that are located in the `/etc/libvirt/qemu` directory. These XML files contain all of the necessary settings that QEMU-KVM processes need to start running virtual machines. libvirt clients issue requests via the `AF_UNIX` socket `/var/run/libvirt/libvirt-sock` that `libvirtd` is listening on.

The next topic on our list is QEMU – what it is, how it works, and how it interacts with KVM.

QEMU

QEMU was written by Fabrice Bellard (creator of FFmpeg). It's a free piece of software and mainly licensed under GNU's **General Public License (GPL)**. QEMU is a generic and open source machine emulator and virtualizer. When used as a machine emulator, QEMU can run OSes and programs made for one machine (such as an ARM board) on a different machine (such as your own PC).

By using dynamic translation, it achieves very good performance (see <https://www.qemu.org/>). Let me rephrase the preceding paragraph and give a more specific explanation. QEMU is actually a hosted hypervisor/VMM that performs hardware virtualization. Are you confused? If so, don't worry. You will get a better picture by the end of this chapter, especially when you go through each of the interrelated components and correlate the entire path used here to perform virtualization. QEMU can act as an emulator or virtualizer.

QEMU as an emulator

In the previous chapter, we discussed binary translation. When QEMU operates as an emulator, it is capable of running OSes/programs made for one machine type on a different machine type. How is this possible? It just uses binary translation methods. In this mode, QEMU emulates CPUs through dynamic binary translation techniques and provides a set of device models. Thus, it is enabled to run different unmodified guest OSes with different architectures. Binary translation is needed here because the guest code has to be executed in the host CPU. The binary translator that does this job is known as a **Tiny Code Generator (TCG)**; it's a **Just-In-Time (JIT)** compiler. It transforms the binary code written for a given processor into another form of binary code (such as ARM in X86), as shown in the following diagram (TCG information from Wikipedia at https://en.wikipedia.org/wiki/QEMU#Tiny_Code_Generator):

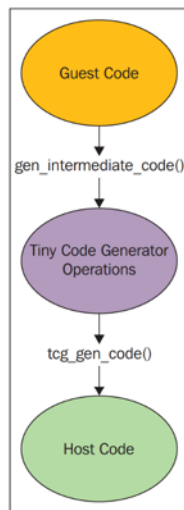


Figure 2.13 – TCG in QEMU

By using this approach, QEMU can sacrifice a bit of execution speed for much broader compatibility. Keeping in mind that most environments nowadays are based around different OSes, this seems like a sensible trade-off.

QEMU as a virtualizer

This is the mode where QEMU executes the guest code directly on the host CPU, thus achieving native performance. For example, when working under Xen/KVM hypervisors, QEMU can operate in this mode. If KVM is the underlying hypervisor, QEMU can virtualize embedded guests such as Power PC, S390, x86, and so on. In short, QEMU is capable of running without KVM using the aforementioned binary translation method. This execution will be slower compared to the hardware-accelerated virtualization enabled by KVM. In any mode, either as a virtualizer or emulator, QEMU *not only* emulates the processor; it also emulates different peripherals, such as disks, networks, VGA, PCI, serial and parallel ports, USB, and so on. Apart from this I/O device emulation, when working with KVM, QEMU-KVM creates and initializes virtual machines. As shown in the following diagram, it also initializes different POSIX threads for each **virtual CPU (vCPU)** of a guest. It also provides a framework that's used to emulate the virtual machine's physical address space within the user-mode address space of QEMU-KVM:

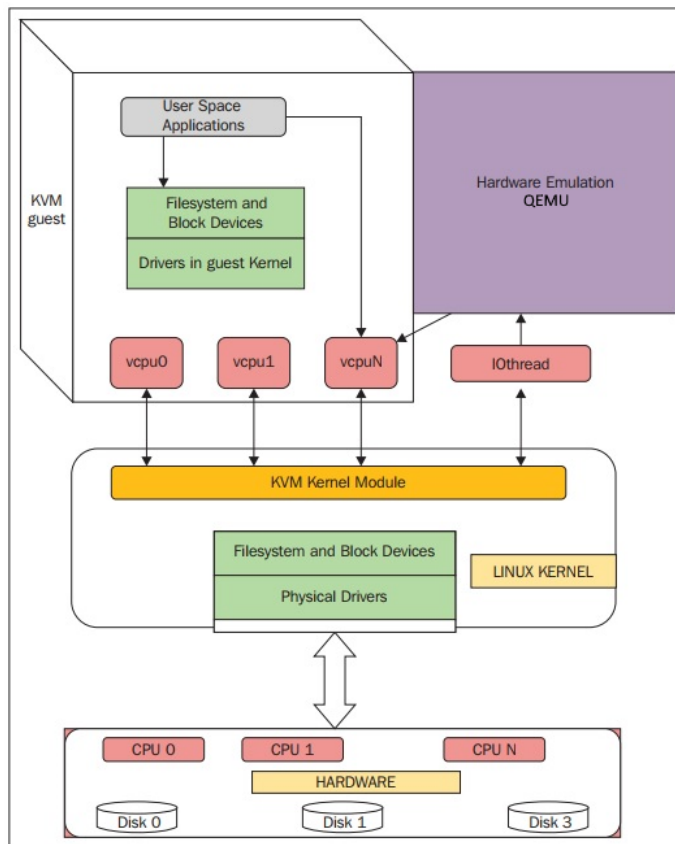


Figure 2.14 – QEMU as a virtualizer

To execute the guest code in the physical CPU, QEMU makes use of POSIX threads. That being said, the guest vCPUs are executed in the host kernel as POSIX threads. This itself brings lots of advantages, as these are just some processes for the host kernel at a high-level view. From another angle, the user-space part of the KVM hypervisor is provided by QEMU. QEMU runs the guest code via the KVM kernel module. When working with KVM, QEMU also does I/O emulation, I/O device setup, live migration, and so on.

QEMU opens the device file (`/dev/kvm`) that's exposed by the KVM kernel module and executes `ioctl()` function calls on it. Please refer to the next section on KVM to find out more about these `ioctl()` function calls. To conclude, KVM makes use of QEMU to become a complete hypervisor. KVM is an accelerator or enabler of the hardware virtualization extensions (VMX or SVM) provided by the processor so that they're tightly coupled with the CPU architecture. Indirectly, this conveys that virtual systems must also use the same architecture to make use of hardware virtualization extensions/capabilities. Once it is enabled, it will definitely give better performance than other techniques, such as binary translation.

Our next step is to check how QEMU fits into the whole KVM story.

QEMU – KVM internals

Before we start looking into QEMU internals, let's clone the QEMU Git repository:

```
# git clone git://git.qemu-project.org/qemu.git
```


Once it's cloned, you can see a hierarchy of files inside the repo, as shown in the following screenshot:

```
[root@kvmsource qemu]# ls
accel                dump                meson_options.txt   qga
audio               exec.c             migration           qobject
authz              exec-vary.c       module-common.c    qom
backends           fpu               monitor            README.rst
block              fsdev             nbd                replay
block.c            gdbstub.c         net                replication.c
blockdev.c         gdb-xml           os-posix.c         replication.h
blockdev-nbd.c    gitdm.config      os-win32.c         roms
blockjob.c         hmp-commands.hx  pc-bios            rules.mak
bootdevice.c      hmp-commands-info.hx  plugins           scripts
bsd-user          hw                po                 scsi
capstone          include           python             slirp
Changelog         io                qapi              softmmu
chardev           iothread.c       qdev-monitor.c    storage-daemon
CODING_STYLE.rst  job.c            qemu-bridge-helper.c  stubs
configure         job-qmp.c        qemu-edid.c        target
contrib           Kconfig          qemu-img.c         tcg
COPYING           Kconfig.host     qemu-img-cmds.hx   tests
COPYING.LIB      libdecnumber     qemu-io.c          thunk.c
cpus-common.c    LICENSE          qemu-io-cmds.c    tools
crypto           linux-headers   qemu-keymap.c      tpm.c
default-configs  linux-user       qemu-nbd.c         trace
device_tree.c   MAINTAINERS     qemu.nsi           trace-events
disas            Makefile         qemu-options.h     ui
disas.c         Makefile.objs   qemu-options.hx    util
dma-helpers.c   memory_ldst.c.inc  qemu-options-wrapper.h  VERSION
docs            meson            qemu.sasl          version.rc
dtc             meson.build     qemu-seccomp.c    version.texi.in
```

Figure 2.15 – QEMU source code

Some important data structures and `ioctl()` function calls make up the QEMU userspace and KVM kernel space. Some of the important data structures are `KVMState`, `CPU{X86}State`, `MachineState`, and so on. Before we further explore the internals, I would like to point out that covering them in detail is beyond the scope of this book; however, I will give enough pointers to understand what is happening under the hood and give additional references for further explanation.

Data structures

In this section, we will discuss some of the important data structures of QEMU. The `KVMState` structure contains important file descriptors of virtual machine representation in QEMU. For example, it contains the virtual machine file descriptor, as shown in the following code:

```
struct KVMState      ( kvm-all.c )
{
    ...
    int fd;
```

```

int vmfd;
int coalesced_mmio;
struct kvm_coalesced_mmio_ring *coalesced_mmio_ring; ...}

```

QEMU-KVM maintains a list of `CPUX86State` structures, one structure for each vCPU. The content of general-purpose registers (as well as RSP and RIP) is part of `CPUX86State`:

```

struct CPUState {
....
int nr_cores;
int nr_threads;
...
int kvm_fd;
...
struct KVMState *kvm_state;
struct kvm_run *kvm_run
}

```

Also, `CPUX86State` looks into the standard registers for exception and interrupt handling:

```

typedef struct CPUX86State ( target/i386/cpu.h )
{
/* standard registers */
target_ulong regs[CPU_NB_REGS];
...
uint64_t system_time_msr;
uint64_t wall_clock_msr;
.....
/* exception/interrupt handling */
int error_code;
int exception_is_int;
....
}

```

Various `ioctl()` function calls exist: `kvm_ioctl()`, `kvm_vm_ioctl()`, `kvm_vcpu_ioctl()`, `kvm_device_ioctl()`, and so on. For function definitions, please visit `KVM-all.c` inside the QEMU source code repo. These `ioctl()` function calls fundamentally map to the system KVM, virtual machine, and vCPU levels. These `ioctl()` function calls are analogous to the `ioctl()` function calls categorized by KVM. We will discuss this when we dig further into KVM internals. To get access to these `ioctl()` function calls exposed by the KVM kernel module, QEMU-KVM has to open `/dev/kvm`, and the resulting file descriptor is stored in `KVMState->fd`:

- `kvm_ioctl()`: These `ioctl()` function calls mainly execute on the `KVMState->fd` parameter, where `KVMState->fd` carries the file descriptor obtained by opening `/dev/kvm` – as in the following example:

```
kvm_ioctl(s, KVM_CHECK_EXTENSION, extension);  
kvm_ioctl(s, KVM_CREATE_VM, type);
```

- `kvm_vm_ioctl()`: These `ioctl()` function calls mainly execute on the `KVMState->vmfd` parameter – as in the following example:

```
kvm_vm_ioctl(s, KVM_CREATE_VCPU, (void *)vcpu_id);  
kvm_vm_ioctl(s, KVM_SET_USER_MEMORY_REGION, &mem);
```

- `kvm_vcpu_ioctl()`: These `ioctl()` function calls mainly execute on the `CPUPState->kvm_fd` parameter, which is a vCPU file descriptor for KVM – as in the following example:

```
kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
```

- `kvm_device_ioctl()`: These `ioctl()` function calls mainly execute on the device `fd` parameter – as in the following example:

```
kvm_device_ioctl(dev_fd, KVM_HAS_DEVICE_ATTR, &attribute)  
? 0 : 1;
```

`kvm-all.c` is one of the important source files when considering QEMU KVM communication.

Now, let's move on and see how a virtual machine and vCPUs are created and initialized by QEMU in the context of KVM virtualization.

`kvm_init()` is the function that opens the KVM device file, as shown in the following code, and it also fills `fd [1]` and `vmfd [2]` of `KVMState`:

```
static int kvm_init(MachineState *ms)
{
    ....
    KVMState *s;
        s = KVM_STATE(ms->accelerator);
        ...
        s->vmfd = -1;
        s->fd = qemu_open("/dev/kvm", O_RDWR);    ----> [1]
        ..
        do {
            ret = kvm_ioctl(s, KVM_CREATE_VM, type); --->[2]
        } while (ret == -EINTR);
        s->vmfd = ret;
    ....
        ret = kvm_arch_init(ms, s);    ---> ( target-i386/kvm.c: )
    .....
}

```

As you can see in the preceding code, the `ioctl()` function call with the `KVM_CREATE_VM` argument will return `vmfd`. Once QEMU has `fd` and `vmfd`, one more file descriptor has to be filled, which is just `kvm_fd` or `vcpu fd`. Let's see how this is filled by QEMU:

```
main() ->
    -> cpu_init(cpu_model);    [#define cpu_
init(cpu_model) CPU(cpu_x86_init(cpu_model)) ]
    ->cpu_x86_create()
    ->qemu_init_vcpu
        ->qemu_kvm_start_vcpu()
        ->qemu_thread_create
    ->qemu_kvm_cpu_thread_fn()
        -> kvm_init_vcpu(CPUState *cpu)
int kvm_init_vcpu(CPUState *cpu)
{

```

```
KVMState *s = kvm_state;
...
    ret = kvm_vm_ioctl(s, KVM_CREATE_VCPU, (void *)kvm_
arch_vcpu_id(cpu));
    cpu->kvm_fd = ret;    --->    [vCPU fd]
..
    mmap_size = kvm_ioctl(s, KVM_GET_VCPU_MMAP_SIZE, 0);
    cpu->kvm_run = mmap(NULL, mmap_size, PROT_READ | PROT_WRITE,
MAP_SHARED,    cpu->kvm_fd, 0);    [3]
...
    ret = kvm_arch_init_vcpu(cpu);    [target-i386/kvm.c]
    ....
}
```

Some of the memory pages are shared between the QEMU-KVM process and the KVM kernel modules. You can see such a mapping in the `kvm_init_vcpu()` function. That said, two host memory pages per vCPU make a channel for communication between the QEMU user-space process and the KVM kernel modules: `kvm_run` and `pio_data`. Also understand that, during the execution of these `ioctl()` function calls that return the preceding `fds`, the Linux kernel allocates a file structure and related anonymous nodes. We will discuss the kernel part later when discussing KVM.

We have seen that vCPUs are `posix` threads created by QEMU-KVM. To run guest code, these vCPU threads execute an `ioctl()` function call with `KVM_RUN` as its argument, as shown in the following code:

```
int kvm_cpu_exec(CPUState *cpu) {
    struct kvm_run *run = cpu->kvm_run;
    ..
    run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
    ...
}
```

The same function, `kvm_cpu_exec()`, also defines the actions that need to be taken when the control comes back to the QEMU-KVM userspace from KVM with `VM_EXIT`. Even though we will discuss later on how KVM and QEMU communicate with each other to perform an operation on behalf of the guest, let me touch upon this here. KVM is an enabler of hardware extensions provided by vendors such as Intel and AMD with their virtualization extensions such as SVM and VMX. These extensions are used by KVM to directly execute the guest code on host CPUs. However, if there is an event – for example, as part of an operation, guest kernel code access hardware device register, which is emulated by the QEMU – then KVM has to exit back to QEMU and pass control. Then, QEMU can emulate the outcome of the operation. There are different exit reasons, as shown in the following code:

```
switch (run->exit_reason) {
    case KVM_EXIT_IO:
        DPRINTF("handle_io\n");

        case KVM_EXIT_MMIO:
            DPRINTF("handle_mmio\n");

        case KVM_EXIT_IRQ_WINDOW_OPEN:
            DPRINTF("irq_window_open\n");

        case KVM_EXIT_SHUTDOWN:
            DPRINTF("shutdown\n");

        case KVM_EXIT_UNKNOWN:
            ...
        case KVM_EXIT_INTERNAL_ERROR:
            ...

        case KVM_EXIT_SYSTEM_EVENT:
            switch (run->system_event.type) {
                case KVM_SYSTEM_EVENT_SHUTDOWN:
                case KVM_SYSTEM_EVENT_RESET:
            }
        case KVM_SYSTEM_EVENT_CRASH:
```

Now that we know about QEMU-KVM internals, let's discuss the threading models in QEMU.

Threading models in QEMU

QEMU-KVM is a multithreaded, event-driven (with a big lock) application. The important threads are as follows:

- Main thread
- Worker threads for the virtual disk I/O backend
- One thread for each vCPU

For each and every virtual machine, there is a QEMU process running in the host system. If the guest system is shut down, this process will be destroyed/exited. Apart from vCPU threads, there are dedicated I/O threads running a select (2) event loop to process I/O, such as network packets and disk I/O completion. I/O threads are also spawned by QEMU. In short, the situation will look like this:

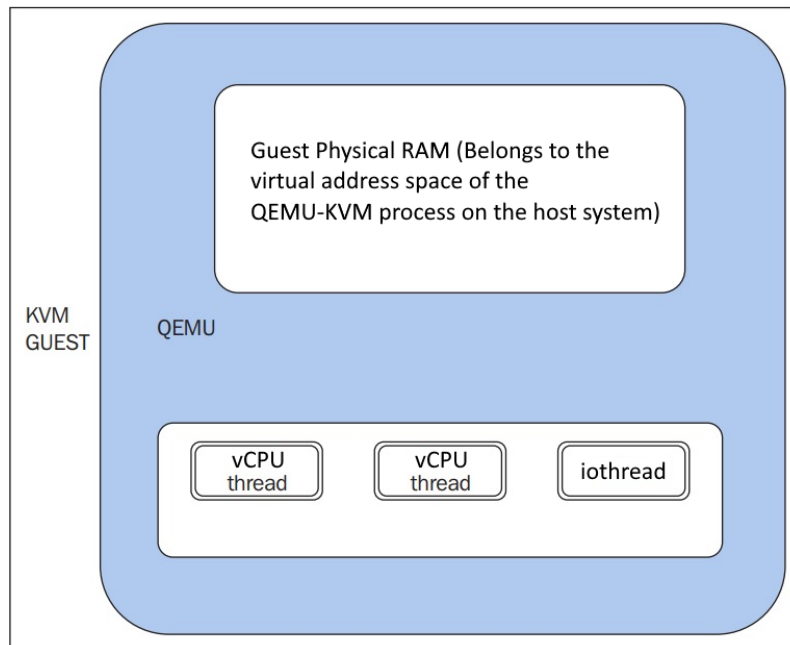


Figure 2.16 – KVM guest

Before we discuss this further, there is always a question about the physical memory of guest systems: where is it located? Here is the deal: the guest RAM is assigned inside the QEMU process's virtual address space, as shown in the preceding figure. That said, the physical RAM of the guest is inside the QEMU process address space.

Important note

More details about threading can be fetched from the threading model at blog.vmsplICE.net/2011/03/qemu-internals-overall-architectre-and-html?m=1.

The event loop thread is also called `iothread`. Event loops are used for timers, file descriptor monitoring, and so on. `main_loop_wait()` is the QEMU main event loop thread. This main event loop thread is responsible for main loop services, including file descriptor callbacks, bottom halves, and timers (defined in `qemu-timer.h`). Bottom halves are similar to timers that execute immediately but have lower overhead, and scheduling them is wait-free, thread-safe, and signal-safe.

Before we leave the QEMU code base, I would like to point out that there are mainly two parts to device codes. For example, the directory `block` contains the host side of the block device code, and `hw/block/` contains the code for device emulation.

KVM

There is a common kernel module called `kvm.ko` and also hardware-based kernel modules such as `kvm-intel.ko` (Intel-based systems) and `kvm-amd.ko` (AMD-based systems). Accordingly, KVM will load the `kvm-intel.ko` (if the `vmx` flag is present) or `kvm-amd.ko` (if the `svm` flag is present) modules. This turns the Linux kernel into a hypervisor, thus achieving virtualization.

KVM exposes a device file called `/dev/kvm` to applications so that they can make use of the `ioctl()` function calls system calls provided. QEMU makes use of this device file to talk with KVM and create, initialize, and manage the kernel-mode context of virtual machines.

Previously, we mentioned that the QEMU-KVM userspace hosts the virtual machine's physical address space within the user-mode address space of QEMU/KVM, which includes memory-mapped I/O. KVM helps us achieve that. There are more things that can be achieved with the help of KVM. The following are some examples:

- Emulation of certain I/O devices; for example, (via *MMIO*) the per-CPU local APIC and the system-wide IOAPIC.
- Emulation of certain *privileged* (R/W of system registers CR0, CR3, and CR4) instructions.

- The facilitation to run guest code via `VMENTRY` and handling *intercepted events* at `VMEXIT`.
- *Injecting* events, such as virtual interrupts and page faults, into the flow of the execution of the virtual machine and so on. This is also achieved with the help of KVM.

KVM is not a full hypervisor; however, with the help of QEMU and emulators (a slightly modified QEMU for I/O device emulation and BIOS), it can become one. KVM needs hardware virtualization-capable processors to operate. Using these capabilities, KVM turns the standard Linux kernel into a hypervisor. When KVM runs virtual machines, every virtual machine is a normal Linux process, which can obviously be scheduled to run on a CPU by the host kernel, as with any other process present in the host kernel. In *Chapter 1, Understanding Linux Virtualization*, we discussed different CPU modes of execution. As you may recall, there is mainly a user mode and a kernel/supervisor mode. KVM is a virtualization feature in the Linux kernel that lets a program such as QEMU safely execute guest code directly on the host CPU. This is only possible when the target architecture is supported by the host CPU.

However, KVM introduced one more mode called guest mode. In a nutshell, guest mode allows us to execute guest system code. It can either run the guest user or the kernel code. With the support of virtualization-aware hardware, KVM virtualizes the process states, memory management, and so on.

Virtualization from a CPU perspective

With its hardware virtualization capabilities, the processor manages the processor states by using **Virtual Machine Control Structure (VMCS)** and **Virtual Machine Control Block (VMCB)** for the host and guest OSes, and it also manages the I/O and interrupts on behalf of the virtualized OS. That being said, with the introduction of this type of hardware, tasks such as CPU instruction interception, register read/write support, memory management support (**Extended Page Tables (EPTs)** and **Nested Paging Table (NPT)**), interrupt handling support (APICv), IOMMU, and so on came into the picture. KVM uses the standard Linux scheduler, memory management, and other services. In short, what KVM does is help the userspace program make use of hardware virtualization capabilities. Here, you can treat QEMU as a userspace program as it's well integrated for different use cases. When I say *hardware-accelerated virtualization*, I am mainly referring to Intel VT-X and AMD-Vs SVM. Introducing virtualization technology processors brought about an extra instruction set called **VMX**.

With Intel's VT-X, the VMM runs in *VMX root operation mode*, while the guests (which are unmodified OSes) run in *VMX non-root operation mode*. This VMX brings additional virtualization-specific instructions to the CPU, such as `VMPTRLD`, `VMPTRST`, `VMCLEAR`, `VMREAD`, `VMWRITE`, `VMCALL`, `VMLAUNCH`, `VMRESUME`, `VMXOFF`, and `VMXON`. The **virtualization mode (VMX)** is turned on by `VMXON` and can be disabled by `VMXOFF`. To execute the guest code, we have to use `VMLAUNCH/VMRESUME` instructions and leave `VMEXIT`. But wait, leave what? It's a transition from non-root operation to root operation. Obviously, when we do this transition, some information needs to be saved so that it can be fetched later. Intel provides a structure to facilitate this transition called `VMCS`; it handles much of the virtualization management functionality. For example, in the case of `VMEXIT`, the exit reason will be recorded inside this structure. Now, how do we read or write from this structure? `VMREAD` and `VMWRITE` instructions are used to read or write to the respective fields.

Previously, we discussed SLAT/EPT/AMD-Vi. Without EPT, the hypervisor must exit the virtual machine to perform address translations, which reduces performance. As we noticed in Intel's virtualization-based processors' operating modes, AMD's SVM also has a couple of operating modes, which are nothing but host mode and guest mode. As you may have assumed, the hypervisor runs in host mode and the guests run in guest mode. Obviously, when in guest mode, some instructions can cause `VMEXIT` exceptions, which are handled in a manner that is specific to the way guest mode is entered. There should be an equivalent structure of `VMCS` here, and it is called `VMCB`; as discussed earlier, it contains the reason for `VMEXIT`. AMD added eight new instruction opcodes to support SVMs. For example, the `VMRUN` instruction starts the operation of a guest OS, the `VMLOAD` instruction loads the processor state from the `VMCB`, and the `VMSAVE` instruction saves the processor state to the `VMCB`. That's why AMD introduced nested paging, which is similar to EPT in Intel.

When we discussed hardware virtualization extensions, we touched upon `VMCS` and `VMCB`. These are important data structures when we think about hardware-accelerated virtualization. These control blocks especially help in `VMEXIT` scenarios. Not every operation can be allowed for guests; at the same time, it's also difficult if the hypervisor does everything on behalf of the guest. Virtual machine control structures, such as `VMCS` or `VMCB`, control this behavior. Some operations are allowed for guests, such as changing some bits in shadowed control registers, but others are not. This clearly provides fine-grained control over what guests are allowed to do and not do. `VMCS` control structures also provide control over interrupt delivery and exceptions. Previously, we said the exit reason of `VMEXIT` is recorded inside the `VMCS`; it also contains some data about it. For example, if write access to a control register caused the exit, information about the source and destination registers is recorded there.

Please take note of the VMCS or VMCB store guest configuration specifics, such as machine control bits and processor register settings. I suggest that you examine the structure definitions from the source. These data structures are also used by the hypervisor to define events to monitor while the guest is executing. These events can be intercepted. Note that these structures are in the host memory. At the time of using VMEXIT, the guest state is saved in VMCS. As mentioned earlier, the VMREAD instruction reads the specified field from the VMCS, while the VMWRITE instruction writes the specified field to the VMCS. Also, note that there is one VMCS or VMCB per vCPU. These control structures are part of the host memory. The vCPU state is recorded in these control structures.

KVM APIs

As mentioned earlier, there are three main types of `ioctl()` function calls. The kernel docs says the following (you can check it at <https://www.kernel.org/doc/Documentation/virtual/kvm/api.txt>):

Three sets of `ioctl` make up the KVM API. The KVM API is a set of `ioctls` that are issued to control various aspects of a virtual machine. These `ioctls` belong to three classes:

- System `ioctls`: These query and set global attributes, which affect the whole KVM subsystem. In addition, a system `ioctl` is used to create virtual machines.*
- Device `ioctls`: Used for device control, executed from the same context that spawned the VM creation.*
- VM `ioctls`: These query and set attributes that affect an entire virtual machine—for example, memory layout. In addition, a VM `ioctl` is used to create virtual CPUs (vCPUs). It runs VM `ioctls` from the same process (address space) that was used to create the VM.*
- vCPU `ioctls`: These query and set attributes that control the operation of a single virtual CPU. They run vCPU `ioctls` from the same thread that was used to create the vCPU.*

To find out more about the `ioctl()` function calls exposed by KVM and the `ioctl()` function calls that belong to a particular group of `fd`, please refer to `KVM.h`.

See the following example:

```

/* ioctls for /dev/kvm fds: */
#define KVM_GET_API_VERSION      _IO(KVMIO, 0x00)
#define KVM_CREATE_VM           _IO(KVMIO, 0x01) /* returns a
VM fd */
...

/* ioctls for VM fds */
#define KVM_SET_MEMORY_REGION    _IOW(KVMIO, 0x40, struct kvm_
memory_region)
#define KVM_CREATE_VCPU         _IO(KVMIO, 0x41)
...

/* ioctls for vcpu fds */
#define KVM_RUN                  _IO(KVMIO, 0x80)
#define KVM_GET_REGS             _IOR(KVMIO, 0x81, struct kvm_
regs)
#define KVM_SET_REGS            _IOW(KVMIO, 0x82, struct kvm_
regs)

```

Let's now discuss anonymous inodes and file structures.

Anonymous inodes and file structures

Previously, when we discussed QEMU, we said the Linux kernel allocates file structures and sets its `f_ops` and anonymous inodes. Let's look at the `kvm_main.c` file:

```

static struct file_operations kvm_chardev_ops = {
    .unlocked_ioctl = kvm_dev_ioctl,
    .llseek        = noop_llseek,
    KVM_COMPAT(kvm_dev_ioctl),
};

kvm_dev_ioctl ()
    switch (ioctl) {
        case KVM_GET_API_VERSION:
            if (arg)
                goto out;

```

```
        r = KVM_API_VERSION;
        break;
    case KVM_CREATE_VM:
        r = kvm_dev_ioctl_create_vm(arg);
        break;
    case KVM_CHECK_EXTENSION:
        r = kvm_vm_ioctl_check_extension_generic(NULL,
arg);
        break;
    case KVM_GET_VCPU_MMAP_SIZE:
        . . . .
}
}
```

Like `kvm_chardev_fops`, there are `kvm_vm_fops` and `kvm_vcpu_fops`:

```
static struct file_operations kvm_vm_fops = {
    .release          = kvm_vm_release,
    .unlocked_ioctl  = kvm_vm_ioctl,
    . . . .
    .llseek          = noop_llseek,
};

static struct file_operations kvm_vcpu_fops = {
    .release          = kvm_vcpu_release,
    .unlocked_ioctl  = kvm_vcpu_ioctl,
    . . . .
    .mmap            = kvm_vcpu_mmap,
    .llseek          = noop_llseek,
};
```

An inode allocation may be seen as follows:

```
anon_inode_getfd(name, &kvm_vcpu_fops, vcpu, O_RDWR | O_
CLOEXEC);
```

Let's have a look at the data structures now.

Data structures

From the perspective of the KVM kernel modules, each virtual machine is represented by a `kvm` structure:

```
include/linux/kvm_host.h :
...
struct kvm {
    ...
    struct mm_struct *mm; /* userspace tied to this vm */
    ...
    struct kvm_vcpu *vcpus[KVM_MAX_VCPUS];
    ....
    struct kvm_io_bus __rcu *buses[KVM_NR_BUSES];
    ...
    struct kvm_coalesced_mmio_ring *coalesced_mmio_ring;
    ....
}
```

As you can see in the preceding code, the `kvm` structure contains an array of pointers to `kvm_vcpu` structures, which are the counterparts of the `CPUX86State` structures in the QEMU-KVM userspace. A `kvm_vcpu` structure consists of a common part and an x86 architecture-specific part, which includes the register content:

```
struct kvm_vcpu {
    ...
    struct kvm *kvm;
    int cpu;
    ....
    int vcpu_id;
    ....
    struct kvm_run *run;
    ....
    struct kvm_vcpu_arch arch;
    ...
}
```

The x86 architecture-specific part of the `kvm_vcpu` structure contains fields to which the guest register state can be saved after a virtual machine exit and from which the guest register state can be loaded before a virtual machine entry:

```
arch/x86/include/asm/kvm_host.h
struct kvm_vcpu_arch {
    ..
    unsigned long regs[NR_VCPU_REGS];
    unsigned long cr0;
    unsigned long cr0_guest_owned_bits;
    ....
    struct kvm_lapic *apic; /* kernel irqchip context */
    ..
    struct kvm_mmu mmu;
    ..
    struct kvm_pio_request pio;
    void *pio_data;
    ..
    /* emulate context */
    struct x86_emulate_ctxt emulate_ctxt;
    ...
    int (*complete_userspace_io)(struct kvm_vcpu *vcpu);
    ....
}
```

As you can see in the preceding code, `kvm_vcpu` has an associated `kvm_run` structure used for the communication (with `pio_data`) between the QEMU userspace and the KVM kernel module, as mentioned earlier. For example, in the context of `VMEXIT`, to satisfy the emulation of virtual hardware access, KVM has to return to the QEMU userspace process; KVM stores the information in the `kvm_run` structure for QEMU to fetch it:

```
/include/uapi/linux/kvm.h:
/* for KVM_RUN, returned by mmap(vcpu_fd, offset=0) */
struct kvm_run {
    /* in */
    ...
    /* out */
```

```

...
    /* in (pre_kvm_run), out (post_kvm_run) */
...
    union {
        /* KVM_EXIT_UNKNOWN */
        ...
        /* KVM_EXIT_FAIL_ENTRY */
        ...
        /* KVM_EXIT_EXCEPTION */
        ...
        /* KVM_EXIT_IO */
        struct {
#define KVM_EXIT_IO_IN  0
#define KVM_EXIT_IO_OUT 1
        ...
        } io;
        ...
    }

```

The `kvm_run` struct is an important data structure; as you can see in the preceding code, `union` contains many exit reasons, such as `KVM_EXIT_FAIL_ENTRY`, `KVM_EXIT_IO`, and so on.

When we discussed hardware virtualization extensions, we touched upon VMCS and VMCB. These are important data structures when we think about hardware-accelerated virtualization. These control blocks especially help in `VMEXIT` scenarios. Not every operation can be allowed for guests; at the same time, it's also difficult if the hypervisor does everything on behalf of the guest. Virtual machine control structures, such as VMCS or VMCB, control the behavior. Some operations are allowed for guests, such as changing some bits in shadowed control registers, but others are not. This clearly provides fine-grained control over what guests are allowed to do and not do. VMCS control structures also provide control over interrupt delivery and exceptions. Previously, we said the exit reason of `VMEXIT` is recorded inside the VMCS; it also contains some data about it. For example, if write access to a control register caused the exit, information about the source and destination registers is recorded there.

Let's look at some of the important data structures before we dive into the vCPU execution flow.

The Intel-specific implementation is in `vmx.c` and the AMD-specific implementation is in `svm.c`, depending on the hardware we have. As you can see, the following `kvm_vcpu` is part of `vcpu_vmx`. The `kvm_vcpu` structure is mainly categorized as a common part and an architecture-specific part. The common part contains the data that is common to all supported architectures and is architecture-specific – for example, the x86 architecture-specific (guest's saved general-purpose registers) part contains the data that is specific to a particular architecture. As discussed earlier, `kvm_vCPUs`, `kvm_run`, and `pio_data` are shared with the userspace.

The `vcpu_vmx` and `vcpu_svm` structures (mentioned next) have a `kvm_vcpu` structure, which consists of an x86-architecture-specific part (`struct 'kvm_vcpu_arch'`) and a common part and also, it points to the `vmcs` and `vmcb` structures accordingly. Let's check the Intel (`vmx`) structure first:

```
vcpu_vmx structure
struct vcpu_vmx {
    struct kvm_vcpu    *vcpu;
    ...
    struct loaded_vmcs vmcs01;
    struct loaded_vmcs *loaded_vmcs;
    ...
}
```

Similarly, let's check the AMD (`svm`) structure next:

```
vcpu_svm structure
struct vcpu_svm {
    struct kvm_vcpu *vcpu;
    ...
    struct vmcb *vmcb;
    ...
}
```

The `vcpu_vmx` or `vcpu_svm` structures are allocated by the following code path:

```
kvm_arch_vcpu_create()
->kvm_x86_ops->vcpu_create
    ->vcpu_create() [.vcpu_create = svm_create_
vcpu, .vcpu_create = vmx_create_vcpu,]
```

Please note that the VMCS or VMCB store guest configuration specifics such as machine control bits and processor register settings. I would suggest you examine the structure definitions from the source. These data structures are also used by the hypervisor to define events to monitor while the guest is executing. These events can be intercepted and these structures are in the host memory. At the time of `VMEXIT`, the guest state is saved in VMCS. As mentioned earlier, the `VMREAD` instruction reads a field from the VMCS, whereas the `VMWRITE` instruction writes the field to it. Also, note that there is one VMCS or VMCB per vCPU. These control structures are part of the host memory. The vCPU state is recorded in these control structures.

Execution flow of vCPU

Finally, we are into the vCPU execution flow, which helps us put everything together and understand what happens under the hood.

I hope you didn't forget that the QEMU creates a POSIX thread for a vCPU of the guest and `ioctl()`, which is responsible for running a CPU and has `KVM_RUN` arg (`#define KVM_RUN _IO(KVMIO, 0x80)`). The vCPU thread executes `ioctl(..., KVM_RUN, ...)` to run the guest code. As these are POSIX threads, the Linux kernel can schedule these threads as with any other process/thread in the system.

Let's see how it all works:

```
Qemu-kvm User Space:
kvm_init_vcpu ()
    kvm_arch_init_vcpu()
        qemu_init_vcpu()
            qemu_kvm_start_vcpu()
                qemu_kvm_cpu_thread_fn()
                    while (1) {
                        if (cpu_can_run(cpu)) {
                            r = kvm_cpu_exec(cpu);
                        }
                    }

kvm_cpu_exec (CPUState *cpu)
->     run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
```

According to the underlying architecture and hardware, different structures are initialized by the KVM kernel modules and one among them is `vmx_x86_ops/svm_x86_ops` (owned by either the `kvm-intel` or `kvm-amd` module). It defines different operations that need to be performed when the vCPU is in context. KVM makes use of the `kvm_x86_ops` vector to point either of these vectors according to the KVM module (`kvm-intel` or `kvm-amd`) loaded for the hardware. The `run` pointer defines the function that needs to be executed when the guest vCPU run is in action, and `handle_exit` defines the actions needed to be performed at the time of `VMEXIT`. Let's check the Intel (`vmx`) structure for that:

```
static struct kvm_x86_ops vmx_x86_ops = {  
    ...  
    .vcpu_create = vmx_create_vcpu,  
    .run = vmx_vcpu_run,  
    .handle_exit = vmx_handle_exit,  
    ...  
}
```

Now, let's see the AMD (`svm`) structure for that:

```
static struct kvm_x86_ops svm_x86_ops = {  
    .vcpu_create = svm_create_vcpu,  
    .run = svm_vcpu_run,  
    .handle_exit = handle_exit,  
    ..  
}
```

The `run` pointer points to `vmx_vcpu_run` or `svm_vcpu_run` accordingly. The `svm_vcpu_run` or `vmx_vcpu_run` functions do the job of saving KVM host registers, loading guest OS registers, and `SVM_VMLOAD` instructions. We walked through the QEMU KVM userspace code execution at the time of `vcpu_run`, once it enters the kernel via `syscall`. Then, following the file operations structures, it calls `kvm_vcpu_ioctl()`; this defines the action to be taken according to the `ioctl()` function call it defines:

```
static long kvm_vcpu_ioctl(struct file *file,  
                           unsigned int ioctl, unsigned long arg)  
{  
  
    switch (ioctl) {
```

```

        case KVM_RUN:
        ...
            kvm_arch_vcpu_ioctl_run(vcpu, vcpu->run);
            ->vcpu_load
                -> vmx_vcpu_load
                    ->vcpu_run(vcpu);
            ->vcpu_enter_guest
                ->vmx_vcpu_run
        ...
    }

```

We will go through `vcpu_run()` to understand how it reaches `vmx_vcpu_run` or `svm_vcpu_run`:

```

static int vcpu_run(struct kvm_vcpu *vcpu) {
    ...

    for (;;) {
        if (kvm_vcpu_running(vcpu)) {
            r = vcpu_enter_guest(vcpu);
        } else {
            r = vcpu_block(kvm, vcpu);
        }
    }
}

```

Once it's in `vcpu_enter_guest()`, you can see some of the important calls happening when it enters guest mode in KVM:

```

static int vcpu_enter_guest(struct kvm_vcpu *vcpu) {
    ...
    kvm_x86_ops.prepare_guest_switch(vcpu);
    vcpu->mode = IN_GUEST_MODE;
    __kvm_guest_enter();
    kvm_x86_ops->run(vcpu);
        [vmx_vcpu_run or svm_vcpu_run ]
    vcpu->mode = OUTSIDE_GUEST_MODE;
    kvm_guest_exit();
}

```

```
    r = kvm_x86_ops->handle_exit(vcpu);  
                                     [vmx_handle_exit or handle_exit ]  
    ...  
}
```

You can see a high-level picture of `VMENTRY` and `VMEXIT` from the `vcpu_enter_guest()` function. That said, `VMENTRY` (`[vmx_vcpu_run` or `svm_vcpu_run]`) is just a guest OS executing in the CPU; different intercepted events can occur at this stage, causing `VMEXIT`. If this happens, any `vmx_handle_exit` or `handle_exit` function call will start looking into this exit cause. We have already discussed the reasons for `VMEXIT` in previous sections. Once there is `VMEXIT`, the exit reason is analyzed and action is taken accordingly.

`vmx_handle_exit()` is the function responsible for handling the exit reason:

```
static int vmx_handle_exit(struct kvm_vcpu *vcpu, , fastpath_t  
exit_fastpath)  
{  
    ... }  
static int (*const kvm_vmx_exit_handlers[])(struct kvm_vcpu  
*vcpu) = {  
    [EXIT_REASON_EXCEPTION_NMI]          = handle_exception,  
    [EXIT_REASON_EXTERNAL_INTERRUPT]     = handle_external_  
interrupt,  
    [EXIT_REASON_TRIPLE_FAULT]          = handle_triple_  
fault,  
    [EXIT_REASON_IO_INSTRUCTION]        = handle_io,  
    [EXIT_REASON_CR_ACCESS]             = handle_cr,  
    [EXIT_REASON_VMCALL]                = handle_vmcall,  
    [EXIT_REASON_VMCLEAR]               = handle_vmclear,  
    [EXIT_REASON_VMLAUNCH]              = handle_vmlaunch,  
    ...  
}
```

`kvm_vmx_exit_handlers []` is the table of virtual machine exit handlers, indexed by `exit_reason`. Similar to Intel, the `svm` code has `handle_exit()`:

```
static int handle_exit(struct kvm_vcpu *vcpu, fastpath_t exit_
fastpath)
{
    struct vcpu_svm *svm = to_svm(vcpu);
    struct kvm_run *kvm_run = vcpu->run;
    u32 exit_code = svm->vmcb->control.exit_code;
    ...
    return svm_exit_handlers[exit_code](svm);
}
```

`handle_exit()` has the `svm_exit_handler` array, as shown in the following section.

If needed, KVM has to fall back to the userspace (QEMU) to perform the emulation as some of the instructions have to be performed on the QEMU emulated devices. For example, to emulate I/O port access, the control goes to the userspace (QEMU):

```
kvm-all.c:
static int (*const svm_exit_handlers[])(struct vcpu_svm *svm) =
{
    [SVM_EXIT_READ_CR0] = cr_interception,
    [SVM_EXIT_READ_CR3] = cr_interception,
    [SVM_EXIT_READ_CR4] = cr_interception,
    ...
}

switch (run->exit_reason) {
    case KVM_EXIT_IO:
        DPRINTF("handle_io\n");
        /* Called outside BQL */
        kvm_handle_io(run->io.port, attrs,
                    (uint8_t *)run + run->io.data_
offset,
                    run->io.direction,
                    run->io.size,
```

```
run->io.count);  
ret = 0;  
break;
```

This chapter was a bit source code-heavy. Sometimes, digging in and checking the source code is just about the only way to understand how something works. Hopefully, this chapter managed to do just that.

Summary

In this chapter, we covered the inner workings of KVM and its main partners in Linux virtualization – libvirt and QEMU. We discussed various types of virtualization – binary translation, full, paravirtualization, and hardware-assisted virtualization. We checked a bit of kernel, QEMU, and libvirt source code to learn about their interaction *from inside*. This gave us the necessary technical know-how to understand the topics that will follow in this book – everything ranging from how to create virtual machines and virtual networks to scaling the virtualization idea to a cloud concept. Understanding these concepts will also make it much easier for you to understand the key goal of virtualization from an enterprise company's perspective – how to properly design a physical and virtual infrastructure, which will slowly but surely be introduced as a concept throughout this book. Now that we've covered the basics about how virtualization works, it's time to move on to a more practical subject – how to deploy the KVM hypervisor, management tools, and oVirt. We'll do this in the next chapter.

Questions

1. What is paravirtualization?
2. What is full virtualization?
3. What is hardware-assisted virtualization?
4. What is the primary goal of libvirt?
5. What does KVM do? What about QEMU?

Further reading

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

- Binary translation: <https://pdfs.semanticscholar.org/d6a5/1a7e73f747b309ef5d44b98318065d5267cf.pdf>
- Virtualization basics: <http://dsc.soic.indiana.edu/publications/virtualization.pdf>
- KVM: <https://www.redhat.com/en/topics/virtualization/what-is-kvm>
- QEMU: <https://www.qemu.org/>
- Understanding full virtualization, paravirtualization, and hardware assist: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf

Section 2: libvirt and ovirt for Virtual Machine Management

In this part of the book, you will get a complete understanding of how to install, configure, and manage a KVM hypervisor using libvirt. You will get advanced knowledge of KVM infrastructure components, such as networking, storage, and virtual hardware configuration. As part of the learning process, you will also get a thorough knowledge of virtual machine life cycle management and virtual machine migration techniques, as well as virtual machine disk management. At the end of part 2, you will be well acquainted with the libvirt command-line management tool `virsh` and the GUI tool `virt-manager`.

This part of the book comprises the following chapters:

- *Chapter 3, Installing KVM Hypervisor, libvirt, and ovirt*
- *Chapter 4, Libvirt Networking*
- *Chapter 5, Libvirt Storage*
- *Chapter 6, Virtual Display Devices and Protocols*
- *Chapter 7, Virtual Machines Installation, Configuration, and Life Cycle Management*
- *Chapter 8, Creating and Modifying VM Disks, Templates, and Snapshots*

3

Installing KVM Hypervisor, libvirt, and oVirt

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

In this chapter, we will cover the following topics:

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

Let's get started!

Getting acquainted with QEMU and libvirt

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

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

When we add libvirt as a management platform on top of QEMU, we get access to some cool new utilities such as the `virsh` command, which we can use to do virtual machine administration, virtual network administration, and a whole lot more. Some of the utilities that we're going to discuss later on in this book (for example, oVirt) use libvirt as a standardized set of libraries and utilities to make their GUI-magic possible – basically, they use libvirt as an API. There are other commands that we get access to for a variety of purposes. For example, we're going to use a command called `virt-host-validate` to check whether our server is compatible with KVM or not.

Getting acquainted with oVirt

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

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

This is where the oVirt project comes in. oVirt is an open source platform for the management of our KVM environment. It's a GUI-based tool that has a lot of moving parts in the background – the engine runs on a Java-based WildFly server (what used to be known as JBoss), the frontend uses a GWT toolkit, and so on. But all of them are there to make one thing possible – for us to manage a KVM-based environment from a centralized, web-based administration console.

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

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

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

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

Obviously, we need to deploy an oVirt agent and related utilities to our hosts, which are going to be the main part of our environment and a place where we will host everything – virtual machines, templates, virtual networks, and so on. For that purpose, oVirt uses a specific agent-based mechanism, via an agent called **vdsm**. This is an agent that we will deploy to our CentOS 8 hosts so that we can add them to oVirt's inventory, which, in turn, means that we can manage them by using the oVirt engine GUI. Vdsm is a Python-based agent that the oVirt engine uses so that it can directly communicate with a KVM host, and vdsmd can then talk to the locally installed libvirt engine to do all the necessary operations. It's also used for configuration purposes as hosts need to be configured to be used in the oVirt environment in order to configure virtual networks, storage management and access, and so on. Also, vdsmd has **Memory Overcommitment Manager (MOM)** integration so that it can efficiently manage memory on our virtualization hosts.

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

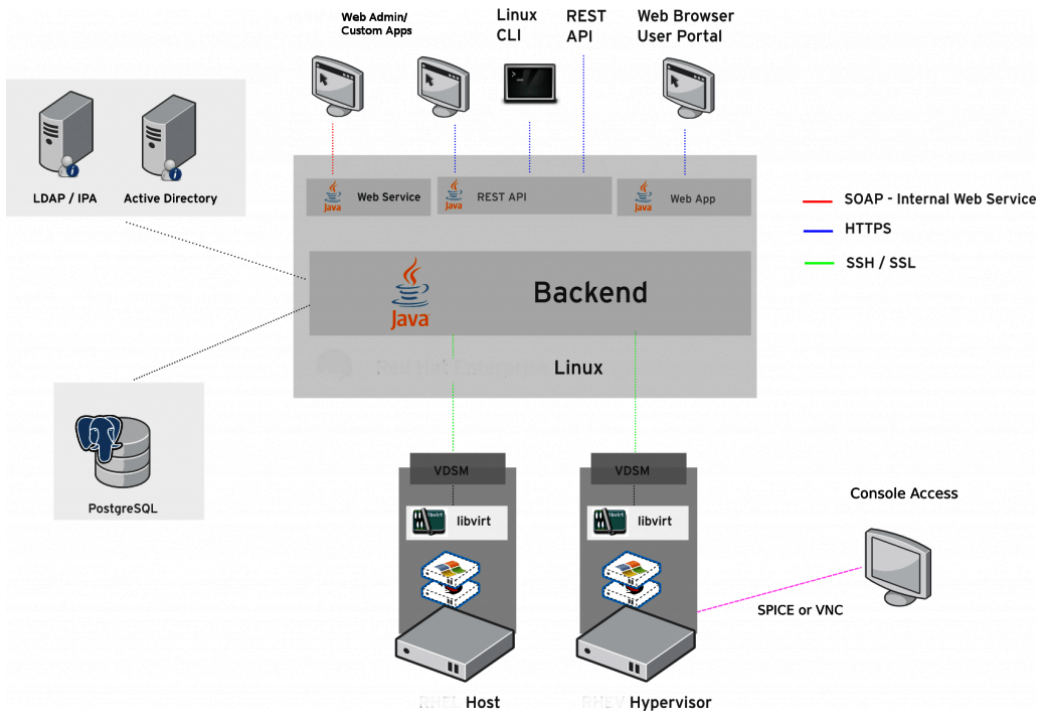


Figure 3.1 – The oVirt architecture (source: <http://ovirt.org>)

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

Installing QEMU, libvirt, and oVirt

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

- We're going to use CentOS 8 for everything in this book (apart from some bits and pieces that only support CentOS 7 as the last supported version at the time of writing).
- Our default installation profile is always going to be **Server with GUI**, with the premise being that we're going to cover both GUI and text-mode utilities to do almost everything that we're going to do in this book.
- Everything that we need to install on top of our default *Server with GUI* installation is going to be installed manually so that we have a complete, step-by-step guide for everything that we do.

- All the examples that we're going to cover in this book can be installed on a single physical server with 16 physical cores and 64 GB of memory. If you modify some numbers (number of cores assigned to virtual machines, amount of memory assigned to some virtual machines, and so on), you could do this with a 6-core laptop and 16 GB of memory, provided that you're not running all the virtual machines all the time. If you shut the virtual machines down after you've completed this chapter and start the necessary ones in the next chapter, you'll be fine with that. In our case, we used a HP ProLiant DL380p Gen8, an easy-to-find, second-hand server – and a quite cheap one at that.

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

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

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

```
intel_iommu=on
```

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

```
systemctl reboot
```

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

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

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

```
virt-host-validate
```

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

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

Figure 3.2 – virt-host-validate output

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

```
[root@packtVM01 ~]# virsh net-list
Name                State      Autostart   Persistent
-----
default             active    yes         yes

[root@packtVM01 ~]# virsh list
Id      Name                               State
-----
[root@packtVM01 ~]# █
```

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

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

Installing the first virtual machine in KVM

We can now start using our KVM virtualization server for its primary purpose – to run virtual machines. Let's start by deploying a virtual machine on our host. For this purpose, we copied a CentOS 8.0 ISO file to our local folder called `/var/lib/libvirt/images`, which we're going to use to create our first virtual machine. We can do that from the command line by using the following command:

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

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

```
osinfo-query os
```

The `--network` parameter is related to our default virtual bridge (we mentioned this earlier). We definitely want our virtual machine to be network-connected, so we picked this parameter to make sure that it's network-connected out of the box.

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

If all of this seems a bit like *manual labor* to you, we feel your pain. Imagine having to deploy dozens of virtual machines and clicking on all these settings. We're not in the 19th century anymore, so there must be an easier way to do this.

Automating virtual machine installation

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

For our purpose, we're going to use a kickstart file that's available on the network (via the web server), but we're going to edit it a little bit so that it's usable, and leave it on our network where `virt-install` can use it.

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

The simplest way to do that in CentOS 7 was to deploy a utility called `system-config-kickstart`, which is not available anymore in CentOS 8. There's a replacement online utility at <https://access.redhat.com/labs/kickstartconfig/> called Kickstart Generator, but you need to have a Red Hat Customer Portal account for that one. So, if you don't have that, you're stuck with text-editing an existing kickstart file. It's not very difficult, but it might take a bit of effort. The most important setting that we need to configure correctly is related to the *location* that we're going to install our virtual machine from – on a network or from a local directory (as we did in our first `virt-install` example, by using a CentOS ISO from local disk). If we're going to use an ISO file locally stored on the server, then it's an easy configuration. First, we're going to deploy the Apache web server so that we can host our kickstart file online (which will come in handy later). So, we need the following commands:

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

Before we start the deployment process, use the `vi` editor (or any other editor you prefer) to edit the first configuration line in our kickstart file (`/var/www/html/ks.cfg`), which says something like `ignoredisk --only-use=sda`, to `ignoredisk --only-use=vda`. This is because virtual KVM machines don't use `sd*` naming for devices, but `vd` naming. This makes it easier for any administrator to figure out if they are administering a physical or a virtual server after connecting to it.

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

```
ip addr show virbr0
```

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

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

Important note

Please take note of the parameter that we changed. Here, we must use the `--location` parameter, not the `--cdrom` parameter, as we're injecting a kickstart configuration into the boot process (it's mandatory to do it this way).

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

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

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

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

If we're using a GUI server installation, we can use GUI utilities to administer our KVM server. One of these utilities is called **Virtual Machine Manager**, and it's a graphical utility that enables you to do pretty much everything you need for your basic administration needs: manipulate virtual networks and virtual machines, open a virtual machine console, and so on. This utility is accessible from GNOME desktop – you can use the Windows search key on your desktop and type in `virtual`, click on **Virtual Machine Manager**, and start using it. This is what Virtual Machine Manager looks like:

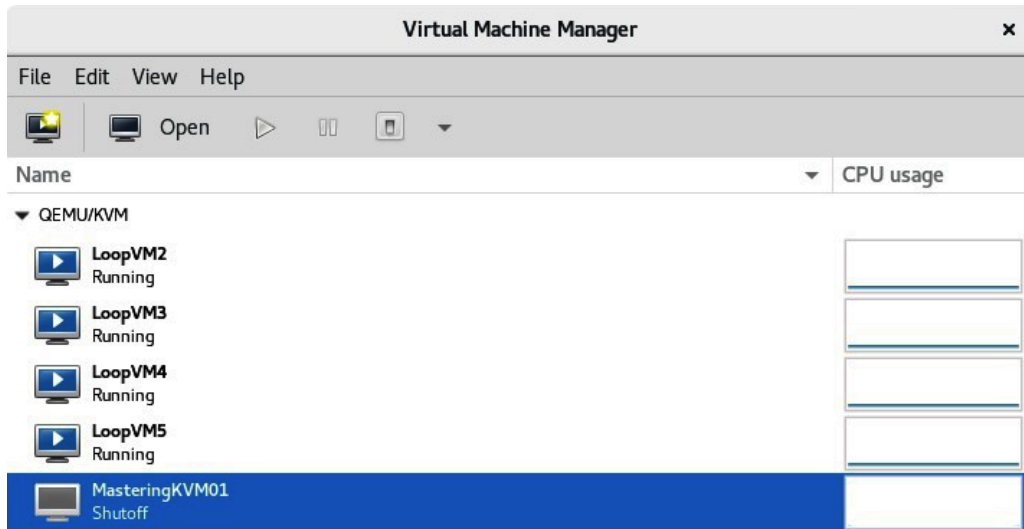


Figure 3.4 – Virtual Machine Manager

Now that we've covered the basic command-line utilities (`virsh` and `virt-install`) and have a very simple-to-use GUI application (Virtual Machine Manager), let's move away from that perspective a bit and think about what we said about oVirt and managing a lot of hosts, virtual machines, networks, and storage devices. So, now, let's discuss how to install oVirt, which we will then use to manage our KVM-based environments in a much more centralized fashion.

Installing oVirt

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

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

First, let's deal with oVirt engine deployment. Deployment is simple enough, and people usually use one virtual machine for this purpose. Keeping in mind that CentOS 8 is not supported for oVirt, in our CentOS 8 virtual machine, we need to punch in a couple of commands:

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

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

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

Figure 3.5 – oVirt configuration settings

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

```

--== SUMMARY ==--
[ INFO ] Restarting httpd
Please use the user 'admin@internal' and password specified in order to login
Web access is enabled at:
http://oVirt:80/ovirt-engine
https://oVirt:443/ovirt-engine
Internal CA 71:CC:F5:A2:23:0A:9E:63:0C:CC:AF:A3:96:80:75:C3:56:F7:9F:93
SSH fingerprint: SHA256:4ZcwR2epbLKKHDva5Ww+TiEonlj5sCdRyezPdZPXaMk
Web access for grafana is enabled at:
https://oVirt/ovirt-engine-grafana/
Please run the following command on the engine machine kvmsource, for SSO to work:
systemctl restart ovirt-engine
--== END OF SUMMARY ==--

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

```

Figure 3.6 – oVirt engine setup summary

Now, we should be able to log into our oVirt engine by using a web browser and pointing it to the URL mentioned in the installation summary. During the installation process, we're asked to provide a password for the `admin@internal` user – this is the oVirt administrative user that we're going to use to manage our environment. The oVirt web interface is simple enough to use, and for the time being, we just need to log into the Administration Portal (a link is directly available on the oVirt engine web GUI before you try to log in). After logging in, we should be greeted with the oVirt GUI:

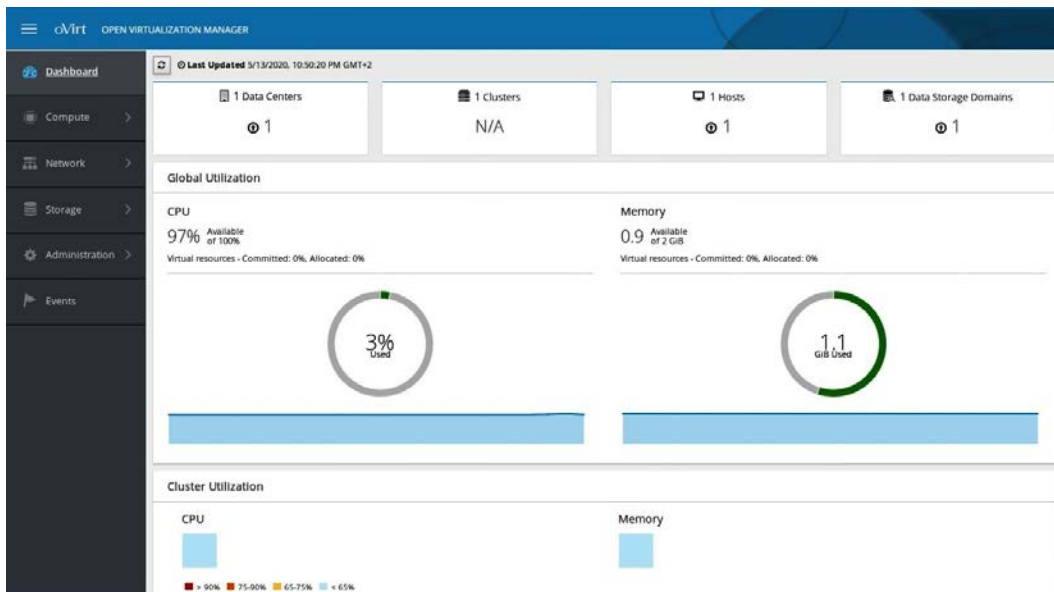


Figure 3.7 – oVirt Engine Administration Portal

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

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

We will deal with many more oVirt-related operations in *Chapter 7, Virtual Machine – Installation, Configuration, and Life Cycle Management*, which is all about oVirt. But for the time being, let's keep the oVirt engine up and running so that we can come back to it later and use it for all of our day-to-day operations in our KVM-based virtualized environment.

Starting a virtual machine using QEMU and libvirt

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

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

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

Figure 3.8 – Using the `virsh start` command

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

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

Figure 3.9 – Using the `virsh list` command

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

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

Figure 3.10 – Using the `virsh shutdown` command

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

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

Figure 3.11 – Using the `virsh destroy` command

If we want to completely remove a virtual machine (for example, `MasteringKVM02`), you'd normally shut it down first (gracefully or forcefully) and then use the `virsh undefine` command:

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

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

Figure 3.12 – Using the `virsh destroy` and `undefine` commands

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

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

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

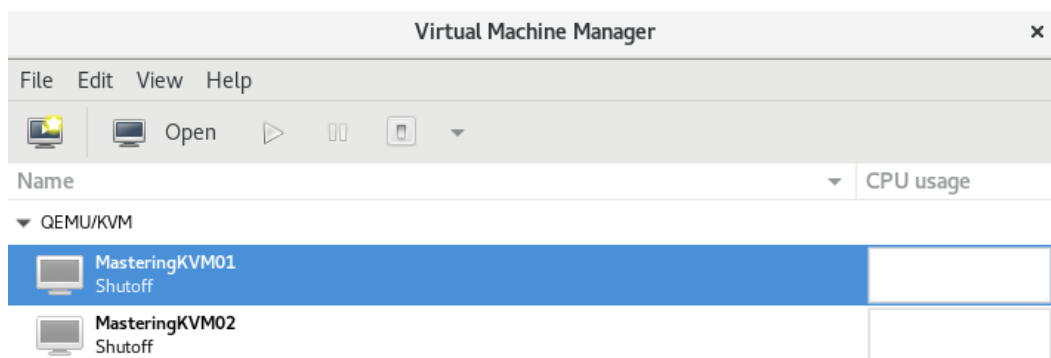


Figure 3.13 – The `virt-manager` GUI – we can see the list of registered virtual machines and start managing them

If we want to do our regular operations on a virtual machine – start, restart, shut down, turn off – we just need to right-click it and select that option from the menu. For all the operations to become visible, first, we must start a virtual machine; otherwise, only four actions are usable out of the available seven – **Run**, **Clone**, **Delete**, and **Open**. The **Pause**, **Shut Down** sub-menu, and **Migrate** options will be grayed-out as they can only be used on a virtual machine that's powered on. So, after we – for example – start `MasteringKVM01`, the list of available options is going to get quite a bit bigger:

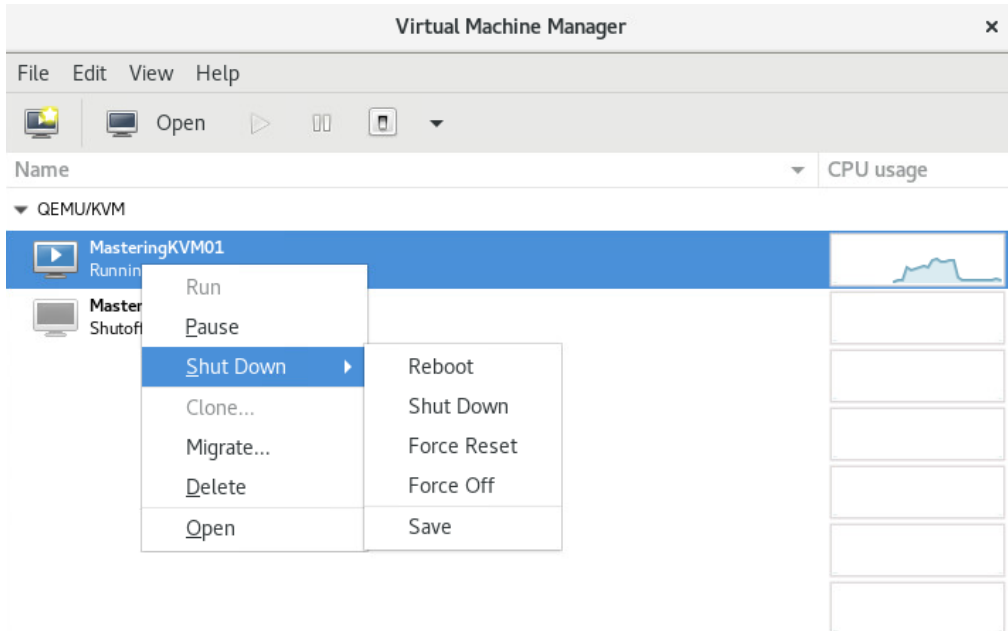


Figure 3.14 – The virt-manager options – after powering the virtual machine on, we can now use many more options

We will use `virt-manager` for various operations throughout this book, so make sure that you familiarize yourself with it. It is going to make our administrative jobs quite a bit easier in many situations.

Summary

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

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

Questions

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

Further reading

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

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

4

Libvirt Networking

Understanding how virtual networking works is really essential for virtualization. It would be very hard to justify the costs associated with a scenario in which we didn't have virtual networking. Just imagine having multiple virtual machines on a virtualization host and buying network cards so that every single one of those virtual machines can have their own dedicated, physical network port. By implementing virtual networking, we're also consolidating networking in a much more manageable way, both from an administration and cost perspective.

This chapter provides you with an insight into the overall concept of virtualized networking and Linux-based networking concepts. We will also discuss physical and virtual networking concepts, try to compare them, and find similarities and differences between them. Also covered in this chapter is the concept of virtual switching for a per-host concept and spanned-across-hosts concept, as well as some more advanced topics. These topics include single-root input/output virtualization, which allows for a much more direct approach to hardware for certain scenarios. We will come back to some of the networking concepts later in this book as we start discussing cloud overlay networks. This is because the basic networking concepts aren't scalable enough for large cloud environments.

In this chapter, we will cover the following topics:

- Understanding physical and virtual networking
- Using TAP/TUN
- Implementing Linux bridging

- Configuring Open vSwitch
- Understanding and configuring SR-IOV
- Understanding macvtap
- Let's get started!

Understanding physical and virtual networking

Let's think about networking for a second. This is a subject that most system administrators nowadays understand pretty well. This might not up to the level many of us think we do, but still – if we were to try to find an area of system administration where we'd find the biggest common level of knowledge, it would be networking.

So, what's the problem with that?

Actually, nothing much. If we really understand physical networking, virtual networking is going to be a piece of cake for us. Spoiler alert: *it's the same thing*. If we don't, it's going to be exposed rather quickly, because there's no way around it. And the problems are going to get bigger and bigger as time goes by because environments evolve and – usually – grow. The bigger they are, the more problems they're going to create, and the more time you're going to spend in debugging mode.

That being said, if you have a firm grasp of VMware or Microsoft-based virtual networking purely at a technological level, you're in the clear here as all of these concepts are very similar.

With that out of the way, what's the whole hoopla about virtual networking? It's actually about understanding where things happen, how, and why. This is because, physically speaking, virtual networking is literally the same as physical networking. Logically speaking, there are some differences that relate more to the *topology* of things than to the principle or engineering side of things. And that's what usually throws people off a little bit – the fact that there are some weird, software-based objects that do the same job as the physical objects that most of us have grown used to managing via our favorite CLI-based or GUI-based utilities.

First, let's introduce the basic building block of virtualized networking – a virtual switch. A virtual switch is basically a software-based Layer 2 switch that you use to do two things:

- Hook up your virtual machines to it.
- Use its uplinks to connect them to physical server cards so that you can hook these physical network cards to a physical switch.

So, let's deal with why we need these virtual switches from the virtual machine perspective. As we mentioned earlier, we use a virtual switch to connect virtual machines to it. Why? Well, if we didn't have some kind of software object that sits in-between our physical network card and our virtual machine, we'd have a big problem – we could only connect virtual machines for which we have physical network ports to our physical network, and that would be intolerable. First, it goes against some of the basic principles of virtualization, such as efficiency and consolidation, and secondly, it would cost a lot. Imagine having 20 virtual machines on your server. This means that, without a virtual switch, you'd have to have at least 20 physical network ports to connect to the physical network. On top of that, you'd actually use 20 physical ports on your physical switch as well, which would be a disaster.

So, by introducing a virtual switch between a virtual machine and a physical network port, we're solving two problems at the same time – we're reducing the number of physical network adapters that we need per server, and we're reducing the number of physical switch ports that we need to use to connect our virtual machines to the network. We can actually argue that we're solving a third problem as well – efficiency – as there are many scenarios where one physical network card can handle being an uplink for 20 virtual machines connected to a virtual switch. Specifically, there are large parts of our environments that don't consume a lot of network traffic and for those scenarios, virtual networking is just amazingly efficient.

Virtual networking

Now, in order for that virtual switch to be able to connect to something on a virtual machine, we have to have an object to connect to – and that object is called a virtual network interface card, often referred to as a vNIC. Every time you configure a virtual machine with a virtual network card, you're giving it the ability to connect to a virtual switch that uses a physical network card as an uplink to a physical switch.

Of course, there are some potential drawbacks to this approach. For example, if you have 50 virtual machines connected to the same virtual switch that uses the same physical network card as an uplink and that uplink fails (due to a network card issue, cable issue, switch port issue, or switch issue), your 50 virtual machines won't have access to the physical network. How do we get around this problem? By implementing a better design and following the basic design principles that we'd use on a physical network as well. Specifically, we'd use more than one physical uplink to the same virtual switch.

Linux has *a lot* of different types of networking interfaces, something like 20 different types, some of which are as follows:

- **Bridge:** Layer 2 interface for (virtual machine) networking.
- **Bond:** For combining network interfaces to a single interface (for balancing and failover reasons) into one logical interface.
- **Team:** Different to bonding, teaming doesn't create one logical interface, but can still do balancing and failover.
- **MACVLAN:** Creates multiple MAC addresses on a single physical interface (creates subinterfaces) on Layer 2.
- **IPVLAN:** Unlike MACVLAN, IPVLAN uses the same MAC address and multiplexes on Layer 3.
- **MACVTAP/IPVTAP:** Newer drivers that should simplify virtual networking by combining TUN, TAP, and bridge as a single module.
- **VXLAN:** A commonly used cloud overlay network concept that we will describe in detail in *Chapter 12, Scaling Out KVM with OpenStack*.
- **VETH:** A virtual Ethernet interface that can be used in a variety of ways for local tunneling.
- **IPOIB:** IP over Infiniband. As Infiniband gains traction in HPC/low latency networks, this type of networking is also supported by the Linux kernel.

There are a whole host of others. Then, on top of these network interface types, there are some 10 types of tunneling interfaces, some of which are as follows:

- **GRETAP, GRE:** Generic Routing Encapsulation protocols for encapsulating Layer 2 and Layer 3 protocols, respectively.
- **GENEVE:** A convergence protocol for cloud overlay networking that's meant to fuse VXLAN, GRE, and others into one. This is why it's supported in Open vSwitch, VMware NSX, and other products.
- **IPIP:** IP over IP tunnel for connecting internal IPv4 subnets via a public network.
- **SIT:** Simple Internet Translation for interconnecting isolated IPv6 networks over IPv4.
- **ip6tnl:** IPv4/6 tunnel over IPv6 tunnel interface.
- **IP6GRE, IP6GRETAP,** and others.

Getting your head around all of them is quite a complex and tedious process, so, in this book, we're only going to focus on the types of interfaces that are really important to us for virtualization and (later in this book) the cloud. This is why we will discuss VXLAN and GENEVE overlay networks in *Chapter 12, Scaling Out KVM with OpenStack*, as we need to have a firm grip on **Software-Defined Networking (SDN)** as well.

So, specifically, as part of this chapter, we're going to cover TAP/TUN, bridging, Open vSwitch, and macvtap interfaces as these are fundamentally the most important networking concepts for KVM virtualization.

But before we dig deep into that, let's explain a couple of basic virtual network concepts that apply to KVM/libvirt networking and other virtualization products (for example, VMware's hosted virtualization products such as Workstation or Player use the same concept). When you start configuring libvirt networking, you can choose between three basic types: NAT, routed, and isolated. Let's discuss what these networking modes do.

Libvirt NAT network

In a NAT libvirt network (and just to make sure that we mention this, the *default* network is configured like this), our virtual machine is behind a libvirt switch in NAT mode. Think of your *I have an internet connection @home scenario* – that's exactly what most of us have: our own *private* network behind a public IP address. This means that our device for accessing the internet (for example, DSL modem) connects to the public network (internet) and gets a public IP address as a part of that process. On our side of the network, we have our own subnet (for example, 192.168.0.0/24 or something like that) for all the devices that we want to connect to the internet.

Now, let's convert that into a virtualized network example. In our virtual machine scenario, this means that our virtual machine can communicate with anything that's connected to the physical network via host's IP address, but not the other way around. For something to communicate to our virtual machine behind a NAT'd switch, our virtual machine has to initiate that communication (or we have to set up some kind of port forwarding, but that's beside the point).

The following diagram might explain what we're talking about a bit better:

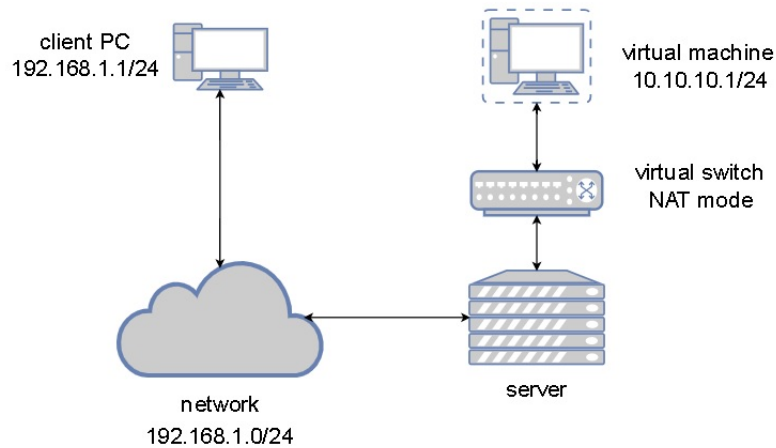


Figure 4.1 – libvirt networking in NAT mode

From the virtual machine perspective, it's happily sitting in a completely separate network segment (hence the 192.168.122.210 and 220 IP addresses) and using a virtual network switch as its gateway to access external networks. It doesn't have to be concerned with any kind of additional routing as that's one of the reasons why we use NAT – to simplify endpoint routing.

Libvirt routed network

The second network type is a routed network, which basically means that our virtual machine is directly connected to the physical network via a virtual switch. This means that our virtual machine is in the same Layer 2/3 network as the physical host. This type of network connection is used very often as, oftentimes, there is no need to have a separate NAT network to access your virtual machines in your environments. In a way, it just makes everything more complicated, especially because you have to configure routing to be aware of the NAT network that you're using for your virtual machines. When using routed mode, the virtual machine sits *in the same* network segment as the next physical device. The following diagram tells a thousand words about routed networks:

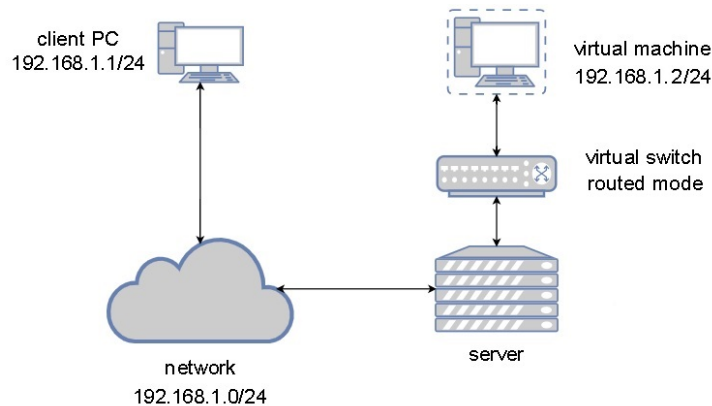


Figure 4.2 – libvirt networking in routed mode

Now that we've covered the two most commonly used types of virtual machine networking scenarios, it's time for the third one, which will seem a bit obscure. If we configure a virtual switch without any *uplinks* (which means it has no physical network cards attached to it), then that virtual switch can't send traffic to the physical network at all. All that's left is communication within the limits of that switch itself, hence the name *isolated*. Let's create that elusive isolated network now.

Libvirt isolated network

In this scenario, virtual machines attached to the same isolated switch can communicate with each other, but they cannot communicate with anything outside the host that they're running on. We used the word *obscure* to describe this scenario earlier, but it really isn't – in some ways, it's actually an ideal way of *isolating* specific types of traffic so that it doesn't even get to the physical network.

Think of it this way – let's say that you have a virtual machine that hosts a web server, for example, running WordPress. You create two virtual switches: one running routed network (direct connection to the physical network) and another that's isolated. Then, you can configure your WordPress virtual machine with two virtual network cards, with the first one connected to the routed virtual switch and the second one connected to the isolated virtual switch. WordPress needs a database, so you create another virtual machine and configure it to use an internal virtual switch only. Then, you use that isolated virtual switch to *isolate* traffic between the web server and the database server so that WordPress connects to the database server via that switch. What did you get by configuring your virtual machine infrastructure like this? You have a two-tier application, and the most important part of that web application (database) is inaccessible from the outside world. Doesn't seem like that bad of an idea, right?

Isolated virtual networks are used in many other security-related scenarios, but this is just an example scenario that we can easily identify with.

Let's describe our isolated network with a diagram:

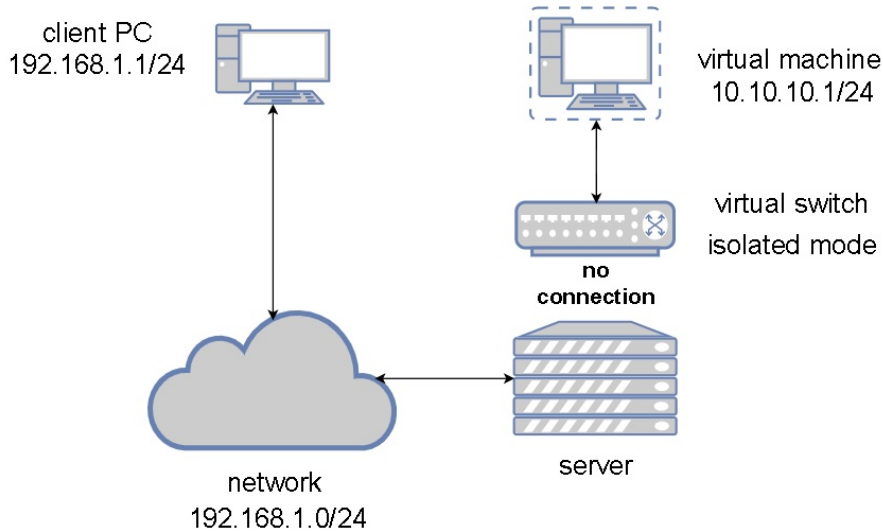


Figure 4.3 – libvirt networking in isolated mode

The previous chapter (*Chapter 3, Installing KVM Hypervisor, libvirt, and ovirt*) of this book mentioned the *default* network, and we said that we're going to talk about that a bit later. This seems like an opportune moment to do so because now, we have more than enough information to describe what the default network configuration is.

When we install all the necessary KVM libraries and utilities like we did in *Chapter 3, Installing KVM Hypervisor, libvirt, and oVirt*, a default virtual switch gets configured out of the box. The reason for this is simple – it's more user-friendly to pre-configure something so that users can just start creating virtual machines and connecting them to the default network than expect users to configure that as well. VMware's vSphere hypervisor does the same thing (the default switch is called vSwitch0), and Hyper-V asks us during the deployment process to configure the first virtual switch (which we can actually skip and configure later). So, this is just a well-known, standardized, established scenario that enables us to start creating our virtual machines faster.

The default virtual switch works in NAT mode with the DHCP server active, and again, there's a simple reason for that – guest operating systems are, by default pre-configured with DHCP networking configuration, which means that the virtual machine that we just created is going to poll the network for necessary IP configuration. This way, the VM gets all the necessary network configuration and we can start using it right away.

The following diagram shows what the default KVM network does:

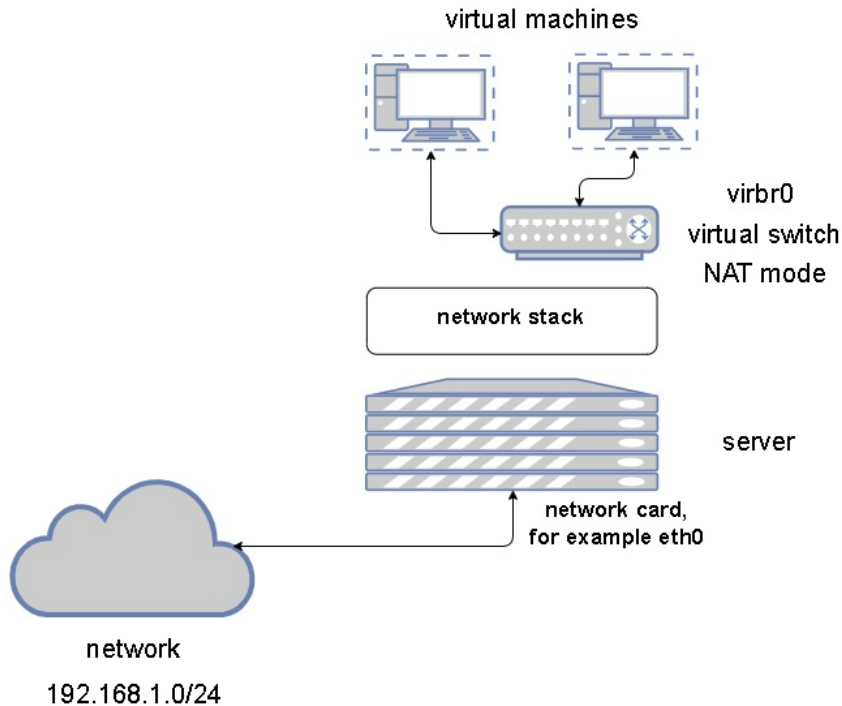


Figure 4.4 – libvirt default network in NAT mode

Now, let's learn how to configure these types of virtual networking concepts from the shell and from the GUI. We will treat this procedure as a procedure that needs to be done sequentially:

1. Let's start by exporting the default network configuration to XML so that we can use it as a template to create a new network:

```
[root@packtVM01 ~]# virsh net-dumpxml default > default.xml
[root@packtVM01 ~]# cat default.xml
<network>
  <name>default</name>
  <uuid>bb3fed90-ced1-45ce-ba3e-13c9ec64ff42</uuid>
  <forward mode='nat'>
    <nat>
      <port start='1024' end='65535' />
    </nat>
  </forward>
  <bridge name='virbr0' stp='on' delay='0' />
  <mac address='52:54:00:35:55:2d' />
  <ip address='192.168.122.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.122.2' end='192.168.122.254' />
    </dhcp>
  </ip>
</network>
```

Figure 4.5 – Exporting the default virtual network configuration

2. Now, let's copy that file to a new file called `packtnat.xml`, edit it, and then use it to create a new NAT virtual network. Before we do that, however, we need to generate two things – a new object UUID (for our new network) and a unique MAC address. A new UUID can be generated from the shell by using the `uuidgen` command, but generating a MAC address is a bit trickier. So, we can use the standard Red Hat-proposed method available on the Red Hat website: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/virtualization_administration_guide/sect-virtualization-tips_and_tricks-generating_a_new_unique_mac_address. By using the first snippet of code available at that URL, create a new MAC address (for example, `00:16:3e:27:21:c1`).

By using `yum` command, install `python2`:

```
yum -y install python2
```

Make sure that you change the XML file so that it reflects the fact that we are configuring a new bridge (`virbr1`). Now, we can complete the configuration of our new virtual machine network XML file:

```
<network>
  <name>packtnat</name>
  <uuid>1f9e3fa3-859b-4bab-9598-d01d32336156</uuid>
  <forward mode='nat'>
    <nat>
      <port start='1024' end='65535' />
    </nat>
  </forward>
  <bridge name='virbr1' stp='on' delay='0' />
  <mac address='00:16:3e:27:21:c1' />
  <ip address='192.168.123.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.123.2' end='192.168.123.254' />
    </dhcp>
  </ip>
</network>
```

Figure 4.6 – New NAT network configuration

The next step is importing this configuration.

3. We can now use the `virsh` command to import that configuration and create our new virtual network, start that network and make it available permanently, and check if everything loaded correctly:

```
virsh net-define packtnat.xml
virsh net-start packtnat
virsh net-autostart packtnat
virsh net-list
```

Given that we didn't delete our default virtual network, the last command should give us the following output:

```
[root@packtVM01 ~]# virsh net-list
Name                State      Autostart  Persistent
-----
default             active    yes        yes
packtnat            active    yes        yes
```

Figure 4.7 – Using `virsh net-list` to check which virtual networks we have on the KVM host

Now, let's create two more virtual networks – a bridged network and an isolated network. Again, let's use files as templates to create both of these networks. Keep in mind that, in order to be able to create a bridged network, we are going to need a physical network adapter, so we need to have an available physical adapter in the server for that purpose. On our server, that interface is called `ens224`, while the interface called `ens192` is being used by the default libvirt network. So, let's create two configuration files called `packtro.xml` (for our routed network) and `packtiso.xml` (for our isolated network):

```
<network>
  <name>packtro</name>
  <uuid>3cac2e7a-a3fd-4b25-8717-450e665b7103</uuid>
  <forward dev='ens224' mode='route'>
    <interface dev='ens224' />
  </forward>
  <bridge name='virbr2' stp='on' delay='0' />
  <mac address='52:54:00:fe:9f:ec' />
  <domain name='packt' />
  <ip address='192.168.2.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.2.100' end='192.168.2.105' />
    </dhcp>
  </ip>
</network>
```

Figure 4.8 – libvirt routed network definition

In this specific configuration, we're using `ens224` as an uplink to the routed virtual network, which would use the same subnet (`192.168.2.0/24`) as the physical network that `ens224` is connected to:

```
<network>
  <name>packtiso</name>
  <uuid>2b92b03d-acb4-4a23-b205-2095c6a27bd4</uuid>
  <bridge name='virbr3' stp='on' delay='0'>
    <mac address='00:16:3e:0b:5d:85' />
  </bridge>
  <domain name='packtiso' />
  <ip address='192.168.3.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.3.128' end='192.168.3.254' />
    </dhcp>
  </ip>
</network>
```

Figure 4.9 – libvirt isolated network definition

Just to cover our bases, we could have easily configured all of this by using the Virtual Machine Manager GUI, as that application has a wizard for creating virtual networks as well. But when we're talking about larger environments, importing XML is a much simpler process, even when we forget about the fact that a lot of KVM virtualization hosts don't have a GUI installed at all.

So far, we've discussed virtual networking from an overall host-level. However, there's also a different approach to the subject – using a virtual machine as an object to which we can add a virtual network card and connect it to a virtual network. We can use `virsh` for that purpose. So, just as an example, we can connect our virtual machine called `MasteringKVM01` to an isolated virtual network:

```
virsh attach-interface --domain MasteringKVM01 --source
isolated --type network --model virtio --config --live
```

There are other concepts that allow virtual machine connectivity to a physical network, and some of them we will discuss later in this chapter (such as SR-IOV). However, now that we've covered the basic approaches to connecting virtual machines to a physical network by using a virtual switch/bridge, we need to get a bit more technical. The thing is, there are more concepts involved in connecting a virtual machine to a virtual switch, such as TAP and TUN, which we will be covering in the following section.

Using userspace networking with TAP and TUN devices

In *Chapter 1, Understanding Linux Virtualization*, we used the `virt-host-validate` command to do some pre-flight checks in terms of the host's preparedness for KVM virtualization. As a part of that process, some of the checks include checking if the following devices exist:

- `/dev/kvm`: The KVM drivers create a `/dev/kvm` character device on the host to facilitate direct hardware access for virtual machines. Not having this device means that the VMs won't be able to access physical hardware, although it's enabled in the BIOS and this will reduce the VM's performance significantly.
- `/dev/vhost-net`: The `/dev/vhost-net` character device will be created on the host. This device serves as the interface for configuring the `vhost-net` instance. Not having this device significantly reduces the virtual machine's network performance.
- `/dev/net/tun`: This is another character special device used for creating TUN/TAP devices to facilitate network connectivity for a virtual machine. The TUN/TAP device will be explained in detail in future chapters. For now, just understand that having a character device is important for KVM virtualization to work properly.

Let's focus on the last device, the TUN device, which is usually accompanied by a TAP device.

So far, all the concepts that we've covered include some kind of connectivity to a physical network card, with isolated virtual networks being an exception. But even an isolated virtual network is just a virtual network for our virtual machines. What happens when we have a situation where we need our communication to happen in the user space, such as between applications running on a server? It would be useless to patch them through some kind of virtual switch concept, or a regular bridge, as that would just bring additional overhead. This is where TUN/TAP devices come in, providing packet flow for user space programs. Easily enough, an application can open `/dev/net/tun` and use an `ioctl()` function to register a network device in the kernel, which, in turn, presents itself as a `tunXX` or `tapXX` device. When the application closes the file, the network devices and routes created by it disappear (as described in the kernel `tuntap.txt` documentation). So, it's just a type of virtual network interface for the Linux operating system supported by the Linux kernel – you can add an IP address and routes to it so that traffic from your application can route through it, and not via a regular network device.

TUN emulates an L3 device by creating a communication tunnel, something like a point-to-point tunnel. It gets activated when the `tuntap` driver gets configured in `tun` mode. When you activate it, any data that you receive from a descriptor (the application that configured it) will be data in the form of regular IP packages (as the most commonly used case). Also, when you send data, it gets written to the TUN device as regular IP packages. This type of interface is sometimes used in testing, development, and debugging for simulation purposes.

The TAP interface basically emulates an L2 Ethernet device. It gets activated when the `tuntap` driver gets configured in `tap` mode. When you activate it, unlike what happens with the TUN interface (Layer 3), you get Layer 2 raw Ethernet packages, including ARP/RARP packages and everything else. Basically, we're talking about a virtualized Layer 2 Ethernet connection.

These concepts (especially TAP) are usable on `libvirt/QEMU` as well because by using these types of configurations, we can create connections from the host to a virtual machine – without the `libvirt` bridge/switch, just as an example. We can actually configure all of the necessary details for the TUN/TAP interface and then start deploying virtual machines that are hooked up directly to those interfaces by using `kvm-qemu` options. So, it's a rather interesting concept that has its place in the virtualization world as well. This is especially interesting when we start creating Linux bridges.

Implementing Linux bridging

Let's create a bridge and then add a TAP device to it. Before we do that, we must make sure the bridge module is loaded into the kernel. Let's get started:

1. If it is not loaded, use `modprobe bridge` to load the module:

```
# lsmod | grep bridge
```

Run the following command to create a bridge called `tester`:

```
# brctl addbr tester
```

Let's see if the bridge has been created:

```
# brctl show
```

```
bridge name bridge id STP enabled interfaces
```

```
tester 8000.460a80dd627d no
```

The `# brctl show` command will list all the available bridges on the server, along with some basic information, such as the ID of the bridge, **Spanning Tree Protocol (STP)** status, and the interfaces attached to it. Here, the `tester` bridge does not have any interfaces attached to its virtual ports.

2. A Linux bridge will also be shown as a network device. To see the network details of the bridge `tester`, use the `ip` command:

```
# ip link show tester
```

```
6: tester: <BROADCAST,MULTICAST>mtu 1500 qdiscnoop state  
DOWN mode
```

```
DEFAULT group default link/ether 26:84:f2:f8:09:e0  
brdff:ff:ff:ff:ff:ff
```

You can also use `ifconfig` to check and configure the network settings for a Linux bridge; `ifconfig` is relatively easy to read and understand but not as feature-rich as the `ip` command:

```
# ifconfig tester
```

```
tester: flags=4098<BROADCAST,MULTICAST>mtu 1500
```

```
ether26:84:f2:f8:09:e0txqueuelen 1000 (Ethernet)
```

```
RX packets 0 bytes 0 (0.0 B)
```

```
RX errors 0 dropped 0 overruns 0 frame 0
```

```
TX packets 0 bytes 0 (0.0 B)
```

```
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

The Linux bridge `tester` is now ready. Let's create and add a TAP device to it.

3. First, check if the TUN/TAP device module is loaded into the kernel. If not, you already know the drill:

```
# lsmod | grep tun
tun 28672 1
```

Run the following command to create a tap device named `vm-vnic`:

```
# ip tuntap add dev vm-vnic mode tap
# ip link show vm-vnic
7: vm-vnic: <BROADCAST,MULTICAST>mtu 1500 qdiscnoop state
DOWN
mode DEFAULT group default qlen 500 link/ether
46:0a:80:dd:62:7d
brdff:ff:ff:ff:ff:ff
```

We now have a bridge named `tester` and a tap device named `vm-vnic`. Let's add `vm-vnic` to `tester`:

```
# brctl addif tester vm-vnic
# brctl show
bridge name bridge id STP enabled interfaces
tester 8000.460a80dd627d no vm-vnic
```

Here, you can see that `vm-vnic` is an interface that was added to the `tester` bridge. Now, `vm-vnic` can act as the interface between your virtual machine and the `tester` bridge, which, in turn, enables the virtual machine to communicate with other virtual machines that are added to this bridge:

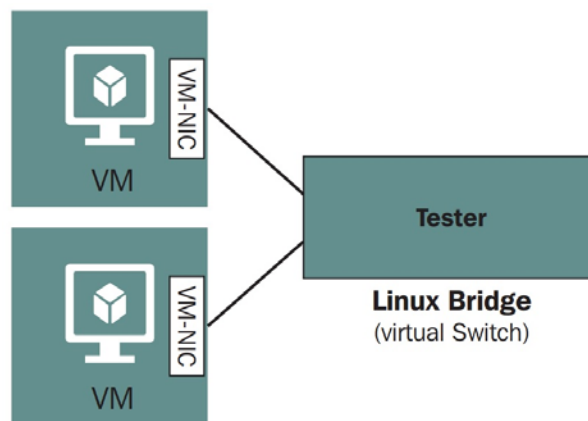


Figure 4.10 – Virtual machines connected to a virtual switch (bridge)

You might also need to remove all the objects and configurations that were created in the previous procedure. Let's do this step by step via the command line:

1. First, we need to remove the `vm-vnic` tap device from the `tester` bridge:

```
# brctl delif tester vm-vnic
# brctl show tester
bridge name bridge id STP enabled interfaces
tester 8000.460a80dd627d no
```

Once the `vm-vnic` has been removed from the bridge, remove the tap device using the `ip` command:

```
# ip tuntap del dev vm-vnic mode tap
```

2. Then, remove the `tester` bridge:

```
# brctl delbr tester
```

These are the same steps that `libvirt` carried out in the backend while enabling or disabling networking for a virtual machine. We want you to understand this procedure thoroughly before moving ahead. Now that we've covered Linux bridging, it's time to move on to a more advanced concept called Open vSwitch.

Configuring Open vSwitch

Imagine for a second that you're working for a small company that has three to four KVM hosts, a couple of network-attached storage devices to host their 15 virtual machines, and that you've been employed by the company from the very start. So, you've seen it all – the company buying some servers, network switches, cables, and storage devices, and you were a part of a small team of people that built that environment. After 2 years of that process, you're aware of the fact that everything works, it's simple to maintain, and doesn't give you an awful lot of grief.

Now, imagine the life of a friend of yours working for a bigger enterprise company that has 400 KVM hosts and close to 2,000 virtual machines to manage, doing the same job as you're doing in a comfy chair of your office in your small company.

Do you think that your friend can manage his or her environment by using the very same tools that you're using? XML files for network switch configuration, deploying servers from a bootable USB drive, manually configuring everything, and having the time to do so? Does that seem like a possibility to you?

There are two basic problems in this second situation:

- **The scale of the environment:** This one is more obvious. Because of the environment size, you need some kind of concept that's going to be managed centrally, instead of on a host-per-host level, such as the virtual switches we've discussed so far.
- **Company policies:** These usually dictate some kind of compliance that comes from configuration standardization as much as possible. Now, we can agree that we could script some configuration updates via Ansible, Puppet, or something like that, but what's the use? We're going to have to create new config files, new procedures, and new workbooks every single time we need to introduce a change to KVM networking. And big companies frown upon that.

So, what we need is a centralized networking object that can span across multiple hosts and offer configuration consistency. In this context, configuration consistency offers us a huge advantage – every change that we introduce in this type of object will be replicated to all the hosts that are members of this centralized networking object. In other words, what we need is **Open vSwitch (OVS)**. For those who are more versed in VMware-based networking, we can use an approximate metaphor – Open vSwitch is for KVM-based environments similar to what vSphere Distributed Switch is for VMware-based environments.

In terms of technology, OVS supports the following:

- VLAN isolation (IEEE 802.1Q)
- Traffic filtering
- NIC bonding with or without LACP
- Various overlay networks – VXLAN, GENEVE, GRE, STT, and so on
- 802.1ag support
- Netflow, sFlow, and so on
- (R)SPAN
- OpenFlow
- OVSDB
- Traffic queuing and shaping
- Linux, FreeBSD, NetBSD, Windows, and Citrix support (and a host of others)

Now that we've listed some of the supported technologies, let's discuss the way in which Open vSwitch works.

First, let's talk about the Open vSwitch architecture. The implementation of Open vSwitch is broken down into two parts: the Open vSwitch kernel module (the data plane) and the user space tools (the control pane). Since the incoming data packets must be processed as fast as possible, the data plane of Open vSwitch was pushed to the kernel space:

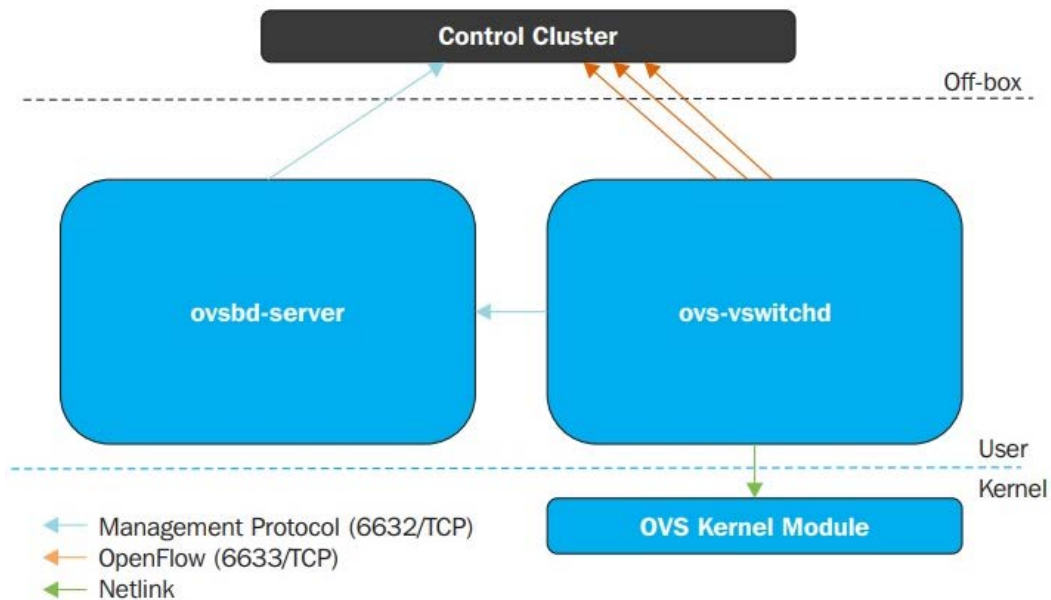


Figure 4.11 – Open vSwitch architecture

The data path (OVS kernel module) uses the netlink socket to interact with the vswitchd daemon, which implements and manages any number of OVS switches on the local system.

Open vSwitch doesn't have a specific SDN controller that it uses for management purposes, in a similar fashion to VMware's vSphere distributed switch and NSX, which have vCenter and various NSX components to manage their capabilities. In OVS, the point is to use someone else's SDN controller, which then interacts with ovs-vswitchd using the OpenFlow protocol. The ovsdb-server maintains the switch table database and external clients can talk to the ovsdb-server using JSON-RPC; JSON is the data format. The ovsdb database currently contains around 13 tables and this database is persistent across restarts.

Open vSwitch works in two modes: normal and flow mode. This chapter will primarily concentrate on how to bring up a KVM VM connected to Open vSwitch's bridge in standalone/normal mode and will give a brief introduction to flow mode using the OpenDaylight controller:

- **Normal Mode:** Switching and forwarding are handled by OVS bridge. In this mode OVS acts as an L2 learning switch. This mode is specifically useful when configuring several overlay networks for your target rather than manipulating the switch's flow.
- **Flow Mode:** In flow mode, the Open vSwitch bridge flow table is used to decide on which port the receiving packets should be forwarded to. All the flows are managed by an external SDN controller. Adding or removing the control flow requires using an SDN controller that's managing the bridge or using the `ctl` command. This mode allows a greater level of abstraction and automation; the SDN controller exposes the REST API. Our applications can make use of this API to directly manipulate the bridge's flows to meet network needs.

Let's move on to the practical aspect and learn how to install Open vSwitch on CentOS 8:

1. The first thing that we must do is tell our system to use the appropriate repositories. In this case, we need to enable the repositories called `epel` and `centos-release-openstack-train`. We can do that by using a couple of `yum` commands:

```
yum -y install epel-release
```

```
yum -y install centos-release-openstack-train
```

2. The next step will be installing `openvswitch` from Red Hat's repository:

```
dnf install openvswitch -y
```

3. After the installation process, we need to check if everything is working by starting and enabling the Open vSwitch service and running the `ovs-vsctl -V` command:

```
systemctl start openvswitch
```

```
systemctl enable openvswitch
```

```
ovs-vsctl -V
```

The last command should throw you some output specifying the version of Open vSwitch and its DB schema. In our case, it's Open vSwitch 2.11.0 and DB schema 7.16.1.

4. Now that we've successfully installed and started Open vSwitch, it's time to configure it. Let's choose a deployment scenario in which we're going to use Open vSwitch as a new virtual switch for our virtual machines. In our server, we have another physical interface called `ens256`, which we're going to use as an uplink for our Open vSwitch virtual switch. We're also going to clear `ens256` configuration, configure an IP address for our OVS, and start the OVS by using the following commands:

```
ovs-vsctl add-br ovs-br0
ip addr flush dev ens256
ip addr add 10.10.10.1/24 dev ovs-br0
ovs-vsctl add-port ovs-br0 ens256
ip link set dev ovs-br0 up
```

5. Now that everything has been configured but not persistently, we need to make the configuration persistent. This means configuring some network interface configuration files. So, go to `/etc/sysconfig/network-scripts` and create two files. Call one of them `ifcfg-ens256` (for our uplink interface):

```
DEVICE=ens256
TYPE=OVSPort
DEVICETYPE=ovs
OVS_BRIDGE=ovs-br0
ONBOOT=yes
```

Call the other file `ifcfg-ovs-br0` (for our OVS):

```
DEVICE=ovs-br0
DEVICETYPE=ovs
TYPE=OVSBridge
BOOTPROTO=static
IPADDR=10.10.10.1
NETMASK=255.255.255.0
GATEWAY=10.10.10.254
ONBOOT=yes
```

- We didn't configure all of this just for show, so we need to make sure that our KVM virtual machines are also able to use it. This means – again – that we need to create a KVM virtual network that's going to use OVS. Luckily, we've dealt with KVM virtual network XML files before (check the *Libvirt isolated network* section), so this one isn't going to be a problem. Let's call our network `packtovs` and its corresponding XML file `packtovs.xml`. It should contain the following content:

```
<network>
<name>packtovs</name>
<forward mode='bridge' />
<bridge name='ovs-br0' />
<virtualport type='openvswitch' />
</network>
```

So, now, we can perform our usual operations when we have a virtual network definition in an XML file, which is to define, start, and autostart the network:

```
virsh net-define packtovs.xml
virsh net-start packtovs
virsh net-autostart packtovs
```

If we left everything as it was when we created our virtual networks, the output from `virsh net-list` should look something like this:

Name	State	Autostart	Persistent
default	active	yes	yes
packtiso	active	yes	yes
packtnat	active	yes	yes
packtovs	active	yes	yes
packtro	active	yes	yes

Figure 4.12 – Successful OVS configuration, and OVS+KVM configuration

So, all that's left now is to hook up a VM to our newly defined OVS-based network called `packtovs` and we're home free. Alternatively, we could just create a new one and pre-connect it to that specific interface using the knowledge we gained in *Chapter 3, Installing KVM Hypervisor, libvirt, and oVirt*. So, let's issue the following command, which has just two changed parameters (`--name` and `--network`):

```
virt-install --virt-type=kvm --name MasteringKVM03 --vcpus
2 --ram 4096 --os-variant=rhel8.0 --cdrom=/var/lib/libvirt/
images/CentOS-8-x86_64-1905-dvd1.iso --network network:packtovs
--graphics vnc --disk size=16
```

After the virtual machine installation completes, we're connected to the OVS-based `packtovs` virtual network, and our virtual machine can use it. Let's say that additional configuration is needed and that we got a request to tag traffic coming from this virtual machine with `VLAN ID 5`. Start your virtual machine and use the following set of commands:

```
ovs-vsctl list-ports ovs-br0
ens256
vnet0
```

This command tells us that we're using the `ens256` port as an uplink and that our virtual machine, `MasteringKVM03`, is using the virtual `vnet0` network port. We can apply VLAN tagging to that port by using the following command:

```
ovs-vsctl set port vnet0 tag=5
```

We need to take note of some additional commands related to OVS administration and management since this is done via the CLI. So, here are some commonly used OVS CLI administration commands:

- `#ovs-vsctl show`: A very handy and frequently used command. It tells us what the current running configuration of the switch is.
- `#ovs-vsctl list-br`: Lists bridges that were configured on Open vSwitch.
- `#ovs-vsctl list-ports <bridge>`: Shows the names of all the ports on `BRIDGE`.
- `#ovs-vsctl list interface <bridge>`: Shows the names of all the interfaces on `BRIDGE`.
- `#ovs-vsctl add-br <bridge>`: Creates a bridge in the switch database.
- `#ovs-vsctl add-port <bridge> : <interface>`: Binds an interface (physical or virtual) to the Open vSwitch bridge.
- `#ovs-ofctl` and `ovs-dpctl`: These two commands are used for administering and monitoring flow entries. You learned that OVS manages two kinds of flows: OpenFlows and Datapath. The first is managed in the control plane, while the second one is a kernel-based flow.
- `#ovs-ofctl`: This speaks to the OpenFlow module, whereas `ovs-dpctl` speaks to the Kernel module.

The following examples are the most used options for each of these commands:

- `#ovs-ofctl show <BRIDGE>`: Shows brief information about the switch, including the port number to port name mapping.
- `#ovs-ofctl dump-flows <Bridge>`: Examines OpenFlow tables.
- `#ovs-dpctl show`: Prints basic information about all the logical datapaths, referred to as *bridges*, present on the switch.
- `#ovs-dpctl dump-flows`: It shows the flow cached in datapath.
- `ovs-appctl`: This command offers a way to send commands to a running Open vSwitch and gathers information that is not directly exposed to the `ovs-ofctl` command. This is the Swiss Army knife of OpenFlow troubleshooting.
- `#ovs-appctl bridge/dumpflows
`: Examines flow tables and offers direct connectivity for VMs on the same hosts.
- `#ovs-appctl fdb/show
`: Lists MAC/VLAN pairs learned.

Also, you can always use the `ovs-vsctl show` command to get information about the configuration of your OVS switch:

```
[root@packtVM01 ~]# ovs-vsctl show
c64c1381-af64-4b51-8db7-f62e37319a06
  Bridge "ovs-br0"
    Port "ovs-br0"
      Interface "ovs-br0"
        type: internal
    Port "vnet0"
      tag: 0
      Interface "vnet0"
    Port "ens256"
      Interface "ens256"
  ovs version: "2.11.0"
```

Figure 4.13 – ovs-vsctl show output

We are going to come back to the subject of Open vSwitch in *Chapter 12, Scaling Out KVM with OpenStack*, as we go deeper into our discussion about spanning Open vSwitch across multiple hosts, especially while keeping in mind the fact that we want to be able to span our cloud overlay networks (based on GENEVE, VXLAN, GRE, or similar protocols) across multiple hosts and sites.

Other Open vSwitch use cases

As you might imagine, Open vSwitch isn't just a handy concept for libvirt or OpenStack – it can be used for a variety of other scenarios as well. Let's describe one of them as it might be important for people looking into VMware NSX or NSX-T integration.

Let's just describe a few basic terms and relationships here. VMware's NSX is an SDN-based technology that can be used for a variety of use cases:

- Connecting data centers and extending cloud overlay networks across data center boundaries.
- A variety of disaster recover scenarios. NSX can be a big help for disaster recover, for multi-site environments, and for integration with a variety of external services and devices that can be a part of the scenario (Palo Alto PANs).
- Consistent micro-segmentation, across sites, done *the right way* on the virtual machine network card level.
- For security purposes, varying from different types of supported VPN technologies to connect sites and end users, to distributed firewalls, guest introspection options (antivirus and anti-malware), network introspection options (IDS/IPS), and more.
- For load balancing, up to Layer 7, with SSL offload, session persistence, high availability, application rules, and more.

Yes, VMware's take on SDN (NSX) and Open vSwitch seem like *competing technologies* on the market, but realistically, there are loads of clients who want to use both. This is where VMware's integration with OpenStack and NSX's integration with Linux-based KVM hosts (by using Open vSwitch and additional agents) comes in really handy. Just to further explain these points – there are things that NSX does that take *extensive* usage of Open vSwitch-based technologies – hardware VTEP integration via Open vSwitch Database, extending GENEVE networks to KVM hosts by using Open vSwitch/NSX integration, and much more.

Imagine that you're working for a service provider – a cloud service provider, an ISP; basically, any type of company that has large networks with a lot of network segmentation. There are loads of service providers using VMware's vCloud Director to provide cloud services to end users and companies. However, because of market needs, these environments often need to be extended to include AWS (for additional infrastructure growth scenarios via the public cloud) or OpenStack (to create hybrid cloud scenarios). If we didn't have a possibility to have interoperability between these solutions, there would be no way to use both of these offerings at the same time. But from a networking perspective, the network background for that is NSX or NSX-T (which actually *uses* Open vSwitch).

It's been clear for years that the future is all about multi-cloud environments, and these types of integrations will bring in more customers; they will want to take advantage of these options in their cloud service design. Future developments will also most probably include (and already partially include) integration with Docker, Kubernetes, and/or OpenShift to be able to manage containers in the same environment.

There are also some more extreme examples of using hardware – in our example, we are talking about network cards on a PCI Express bus – in a *partitioned* way. For the time being, our explanation of this concept, called SR-IOV, is going to be limited to network cards, but we will expand on the same concept in *Chapter 6, Virtual Display Devices and Protocols*, when we start talking about partitioning GPUs for use in virtual machines. So, let's discuss a practical example of using SR-IOV on an Intel network card that supports it.

Understanding and using SR-IOV

The SR-IOV concept is something that we already mentioned in *Chapter 2, KVM as a Virtualization Solution*. By utilizing SR-IOV, we can *partition* PCI resources (for example, network cards) into virtual PCI functions and inject them into a virtual machine. If we're using this concept for network cards, we're usually doing this with a single purpose – so that we can avoid using the operating system kernel and network stack while accessing a network interface card from our virtual machine. In order for us to be able to do this, we need to have hardware support, so we need to check if our network card actually supports it. On a physical server, we could use the `lspci` command to extract attribute information about our PCI devices and then `grep` out *Single Root I/O Virtualization* as a string to try to see if we have a device that's compatible. Here's an example from our server:

```
root@storage3:~# lspci -s 24:00 -vvv | grep -i Single
Capabilities: [160 v1] Single Root I/O Virtualization (SR-IOV)
Capabilities: [160 v1] Single Root I/O Virtualization (SR-IOV)
```

Figure 4.14 – Checking if our system is SR-IOV compatible

Important Note

Be careful when configuring SR-IOV. You need to have a server that supports it, a device that supports it, and you must make sure that you turn on SR-IOV functionality in BIOS. Then, you need to keep in mind that there are servers that only have specific slots assigned for SR-IOV. The server that we used (HP Proliant DL380p G8) has three PCI-Express slots assigned to CPU1, but SR-IOV worked only in slot #1. When we connected our card to slot #2 or #3, we got a BIOS message that SR-IOV will not work in that slot and that we should move our card to a slot that supports SR-IOV. So, please, make sure that you read the documentation of your server thoroughly and connect a SR-IOV compatible device to a correct PCI-Express slot.

In this specific case, it's an Intel 10 Gigabit network adapter with two ports, which we could use to do SR-IOV. The procedure isn't all that difficult, and it requires us to complete the following steps:

1. Unbind from the previous module.
2. Register it to the `vfio-pci` module, which is available in the Linux kernel stack.
3. Configure a guest that's going to use it.

So, what you would do is unload the module that the network card is currently using by using `modprobe -r`. Then, you would load it again, but by assigning an additional parameter. On our specific server, the Intel dual-port adapter that we're using (X540-AT2) was assigned to the `ens1f0` and `ens1f1` network devices. So, let's use `ens1f0` as an example for SR-IOV configuration at boot time:

1. The first thing that we need to do (as a general concept) is find out which kernel module our network card is using. To do that, we need to issue the following command:

```
ethtool -i ens1f0 | grep ^driver
```

In our case, this is the output that we got:

```
driver: ixgbe
```

We need to find additional available options for that module. To do that, we can use the `modinfo` command (we're only interested in the `parm` part of the output):

```
modinfo ixgbe
```

```
....
```

```
Parm:      max_vfs (Maximum number of virtual functions
to allocate per physical function - default is zero and
maximum value is 63.
```

For example, we're using the `ixgbe` module here, and we can do the following:

```
modprobe -r ixgbe
```

```
modprobe ixgbe max_vfs=4
```

2. Then, we can use the `modprobe` system to make these changes permanent across reboots by creating a file in `/etc/modprobe.d` called (for example) `ixgbe.conf` and adding the following line to it:

```
options ixgbe max_vfs=4
```

This would give us up to four virtual functions that we can use inside our virtual machines. Now, the next issue that we need to solve is how to boot our server with SR-IOV active at boot time. There are quite a few steps involved here, so, let's get started:

1. We need to add the `iommu` and `vfs` parameters to the default kernel boot line and the default kernel configuration. So, first, open `/etc/default/grub` and edit the `GRUB_CMDLINE_LINUX` line and add `intel_iommu=on` (or `amd_iommu=on` if you're using an AMD system) and `ixgbe.max_vfs=4` to it.
2. We need to reconfigure `grub` to use this change, so we need to use the following command:

```
grub2-mkconfig -o /boot/grub2/grub.cfg
```

3. Sometimes, even that isn't enough, so we need to configure the necessary kernel parameters, such as the maximum number of virtual functions and the `iommu` parameter to be used on the server. That leads us to the following command:

```
grubby --update-kernel=ALL --args="intel_iommu=on ixgbe.max_vfs=4"
```

After reboot, we should be able to see our virtual functions. Type in the following command:

```
lspci -nn | grep "Virtual Function"
```

We should get an output that looks like this:

```
root@PacktPhy01 ~]# lspci -nn | grep "Virtual Function"
04:10.0 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.1 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.2 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.3 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.4 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.5 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.6 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.7 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
```

Figure 4.15 – Checking for virtual function visibility

We should be able to see these virtual functions from `libvirt`, and we can check that via the `virsh` command. Let's try this (we're using `grep 04` because our device IDs start with 04, which is visible from the preceding image; we'll shrink the output to important entries only):

```
virsh nodedev-list | grep 04
```

```
.....
```

```
pci_0000_04_00_0
```

```
pci_0000_04_00_1
```

```
pci_0000_04_10_0
```

```
pci_0000_04_10_1
pci_0000_04_10_2
pci_0000_04_10_3
pci_0000_04_10_4
pci_0000_04_10_5
pci_0000_04_10_6
pci_0000_04_10_7
```

The first two devices are our physical functions. The remaining eight devices (two ports times four functions) are our virtual devices (from `pci_0000_04_10_0` to `pci_0000_04_10_7`). Now, let's dump that device's information by using the `virsh nodedev-dumpxml pci_0000_04_10_0` command:

```
[root@PacktPhy01 ~]# virsh nodedev-dumpxml pci_0000_04_10_0
<device>
  <name>pci_0000_04_10_0</name>
  <path>/sys/devices/pci0000:00/0000:00:03.0/0000:04:10.0</path>
  <parent>pci_0000_00_03_0</parent>
  <driver>
    <name>ixgbevf</name>
  </driver>
  <capability type='pci'>
    <domain>0</domain>
    <bus>4</bus>
    <slot>16</slot>
    <function>0</function>
    <product id='0x1515'>X540 Ethernet Controller Virtual Function</product>
    <vendor id='0x8086'>Intel Corporation</vendor>
    <capability type='phys_function'>
      <address domain='0x0000' bus='0x04' slot='0x00' function='0x0' />
    </capability>
    <iommuGroup number='60'>
      <address domain='0x0000' bus='0x04' slot='0x10' function='0x0' />
    </iommuGroup>
    <numa node='0' />
    <pci-express>
      <link validity='cap' port='0' speed='5' width='8' />
      <link validity='sta' width='0' />
    </pci-express>
  </capability>
</device>
```

Figure 4.16 – Virtual function information from the perspective of `virsh`

So, if we have a running virtual machine that we'd like to reconfigure to use this, we'd have to create an XML file with definition that looks something like this (let's call it `packtsriov.xml`):

```
<interface type='hostdev' managed='yes' >
  <source>
```

```
<address type='pci' domain='0x0000' bus='0x04' slot='0x10'  
function='0x0'>  
</address>  
</source>  
</interface>
```

Of course, the domain, bus, slot, and function need to point exactly to our VF. Then, we can use the `virsh` command to attach that device to our virtual machine (for example, `MasteringKVM03`):

```
virsh attach-device MasteringKVM03 packtsriov.xml --config
```

When we use `virsh dumpxml`, we should now see a part of the output that starts with `<driver name='vfio' />`, along with all the information that we configured in the previous step (address type, domain, bus, slot, function). Our virtual machine should have no problems using this virtual function as a network card.

Now, it's time to cover another concept that's very much useful in KVM networking: `macvtap`. It's a newer driver that should simplify our virtualized networking by completely removing `tun/tap` and `bridge` drivers with a single module.

Understanding macvtap

This module works like a combination of the `tap` and `macvlan` modules. We already explained what the `tap` module does. The `macvlan` module enables us to create virtual networks that are pinned to a physical network interface (usually, we call this interface a *lower* interface or device). Combining `tap` and `macvlan` enables us to choose between four different modes of operation, called **Virtual Ethernet Port Aggregator (VEPA)**, `bridge`, `private`, and `passthru`.

If we're using the `VEPA` mode (default mode), the physical switch has to support `VEPA` by supporting `hairpin` mode (also called `reflective relay`). When a *lower* device receives data from a `VEPA` mode `macvlan`, this traffic is always sent out to the upstream device, which means that traffic is always going through an external switch. The advantage of this mode is the fact that network traffic between virtual machines becomes visible on the external network, which can be useful for a variety of reasons. You can check how network flow works in the following sequence of diagrams:

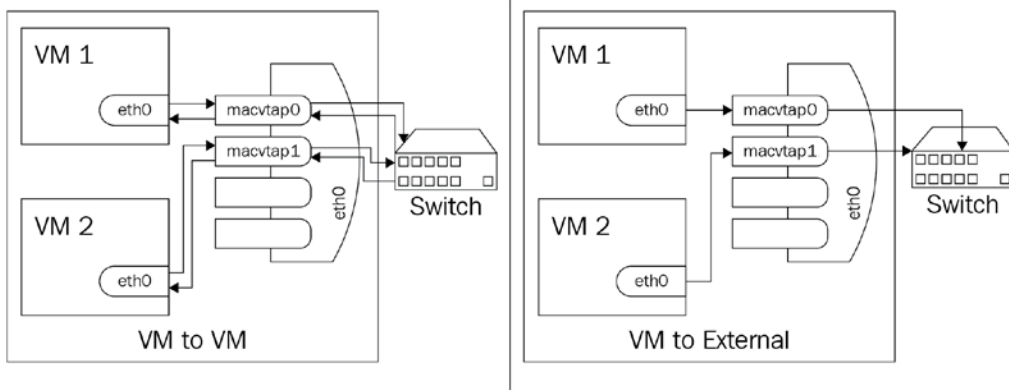


Figure 4.17 – macvtap VEPA mode, where traffic is forced to the external network

In private mode, it's similar to VEPA in that everything goes to an external switch, but unlike VEPA, traffic only gets delivered if it's sent via an external router or switch. You can use this mode if you want to isolate virtual machines connected to the endpoints from one another, but not from the external network. If this sounds very much like a private VLAN scenario, you're completely correct:

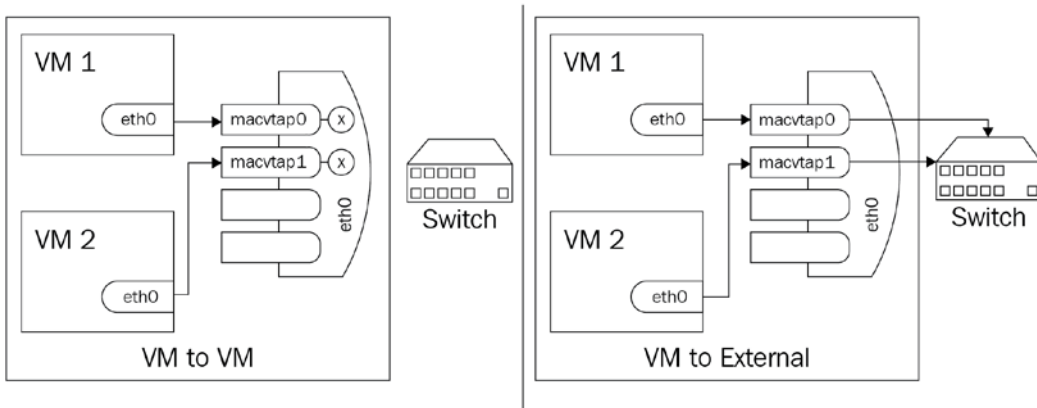


Figure 4.18 – macvtap in private mode, using it for internal network isolation

In bridge mode, data received on your macvlan that's supposed to go to another macvlan on the same lower device is sent directly to the target, not externally, and then routed back. This is very similar to what VMware NSX does when communication is supposed to happen between virtual machines on different VXLAN networks, but on the same host:

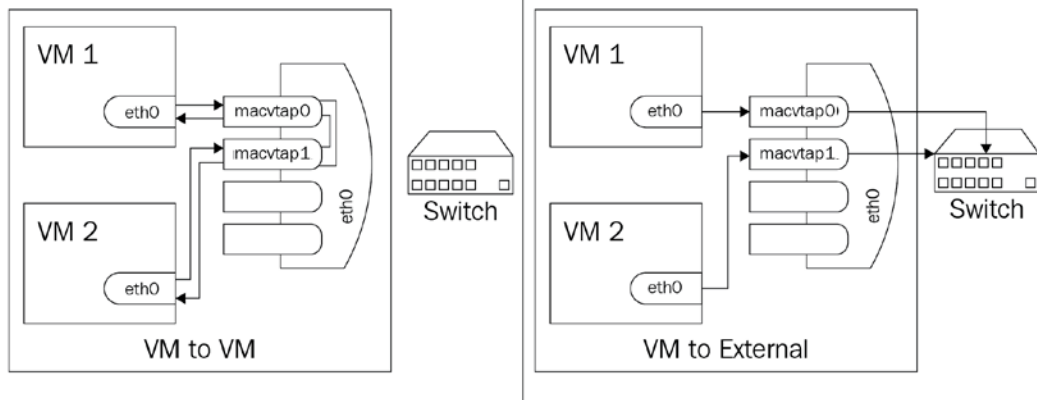


Figure 4.19 – macvtap in bridge mode, providing a kind of internal routing

In passthrough mode, we're basically talking about the SR-IOV scenario, where we're using a VF or a physical device directly to the macvtap interface. The key difference is that a single network interface can only be passed to a single guest (1:1 relationship):

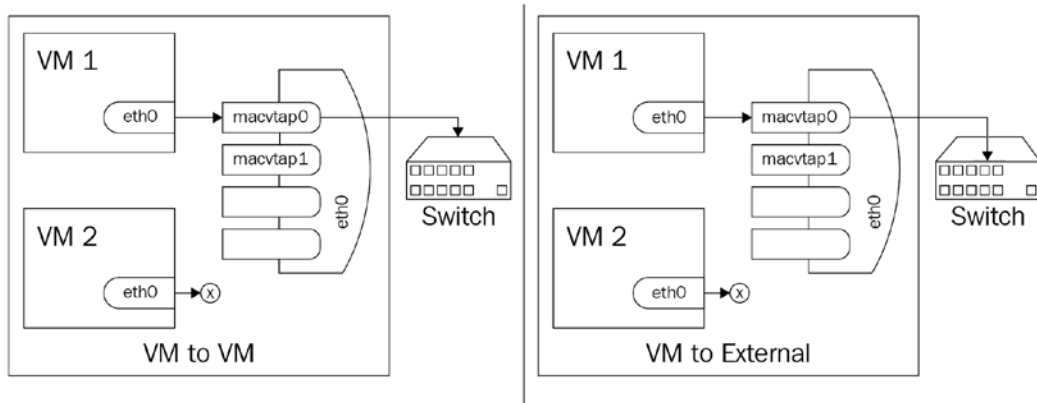


Figure 4.20 – macvtap in passthrough mode

In *Chapter 12, Scaling Out KVM with OpenStack* and *Chapter 13, Scaling Out KVM with AWS*, we'll describe why virtualized and *overlay* networking (VXLAN, GRE, GENEVE) is even more important for cloud networking as we extend our local KVM-based environment to the cloud either via OpenStack or AWS.

Summary

In this chapter, we covered the basics of virtualized networking in KVM and explained why virtualized networking is such a huge part of virtualization. We went knee-deep into configuration files and their options as this will be the preferred method for administration in larger environments, especially when talking about virtualized networks.

Pay close attention to all the configuration steps that we discussed through this chapter, especially the part related to using `virsh` commands to manipulate network configuration and to configure Open vSwitch and SR-IOV. SR-IOV-based concepts are heavily used in latency-sensitive environments to provide networking services with the lowest possible overhead and latency, which is why this principle is very important for various enterprise environments related to the financial and banking sector.

Now that we've covered all the necessary networking scenarios (some of which will be revisited later in this book), it's time to start thinking about the next big topic of the virtualized world. We've already talked about CPU and memory, as well as networks, which means we're left with the fourth pillar of virtualization: storage. We will tackle that subject in the next chapter.

Questions

1. Why is it important that virtual switches accept connectivity from multiple virtual machines at the same time?
2. How does a virtual switch work in NAT mode?
3. How does a virtual switch work in routed mode?
4. What is Open vSwitch and for what purpose can we use it in virtualized and cloud environments?
5. Describe the differences between TAP and TUN interfaces.

Further reading

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

- Libvirt networking: <https://wiki.libvirt.org/page/VirtualNetworking>
- Network XML format: <https://libvirt.org/formatnetwork.html>
- Open vSwitch: <https://www.openvswitch.org/>
- Open vSwitch and libvirt: <http://docs.openvswitch.org/en/latest/howto/libvirt/>
- Open vSwitch Cheat Sheet: <https://adhioutlined.github.io/virtual/Openvswitch-Cheat-Sheet/>

5

Libvirt Storage

This chapter provides you with an insight into the way that KVM uses storage. Specifically, we will cover both storage that's internal to the host where we're running virtual machines and *shared storage*. Don't let the terminology confuse you here – in virtualization and cloud technologies, the term *shared storage* means storage space that multiple hypervisors can have access to. As we will explain a bit later, the three most common ways of achieving this are by using block-level, share-level, or object-level storage. We will use NFS as an example of share-level storage, and **Internet Small Computer System Interface (iSCSI)** and **Fiber Channel (FC)** as examples of block-level storage. In terms of object-based storage, we will use Ceph. GlusterFS is also commonly used nowadays, so we'll make sure that we cover that, too. To wrap everything up in an easy-to-use and easy-to-manage box, we will discuss some open source projects that might help you while practicing with and creating testing environments.

In this chapter, we will cover the following topics:

- Introduction to storage
- Storage pools
- NFS storage
- iSCSI and SAN storage
- Storage redundancy and multipathing

- Gluster and Ceph as a storage backend for KVM
- Virtual disk images and formats and basic KVM storage operations
- The latest developments in storage – NVMe and NVMeOF

Introduction to storage

Unlike networking, which is something that most IT people have at least a basic understanding of, storage tends to be quite different. In short, yes, it tends to be a bit more complex. There are loads of parameters involved, different technologies, and...let's be honest, loads of different types of configuration options and people enforcing them. And a *lot* of questions. Here are some of them:

- Should we configure one NFS share per storage device or two?
- Should we create one iSCSI target per storage device or two?
- Should we create one FC target or two?
- How many **Logical Unit Numbers (LUNs)** per target?
- What kind of cluster size should we use?
- How should we carry out multipathing?
- Should we use block-level or share-level storage?
- Should we use block-level or object-level storage?
- Which technology or solution should we choose?
- How should we configure caching?
- How should we configure zoning or masking?
- How many switches should we use?
- Should we use some kind of clustering technology on a storage level?

As you can see, the questions just keep piling up, and we've barely touched the surface, because there are also questions about which filesystem to use, which physical controller we will use to access storage, and what type of cabling—it just becomes a big mashup of variables that has many potential answers. What makes it worse is the fact that many of those answers can be correct—not just one of them.

Let's get the basic-level mathematics out of the way. In an enterprise-level environment, shared storage is usually *the most expensive* part of the environment and can also have *the most significant negative impact* on virtual machine performance, while at the same time being *the most oversubscribed resource* in that environment. Let's think about this for a second—every powered-on virtual machine is constantly going to hammer our storage device with I/O operations. If we have 500 virtual machines running on a single storage device, aren't we asking a bit too much from that storage device?

At the same time, some kind of shared storage concept is a key pillar of virtualized environments. The basic principle is very simple – there are loads of advanced functionalities that will work so much better with shared storage. Also, many operations are much faster if shared storage is available. Even more so, there are so many simple options for high availability when we don't have our virtual machines stored in the same place where they are being executed.

As a bonus, we can easily avoid **Single Point Of Failure (SPOF)** scenarios if we design our shared storage environment correctly. In an enterprise-level environment, avoiding SPOF is one of the key design principles. But when we start adding switches and adapters and controllers to the *to buy* list, our managers' or clients' heads usually starts to hurt. We talk about performance and risk management, while they talk about price. We talk about the fact that their databases and applications need to be properly fed in terms of I/O and bandwidth, and they feel that you can produce that out of thin air. Just wave your magic wand and there we are: unlimited storage performance.

But the best, and our all-time favorite, apples-to-oranges comparison that your clients are surely going to try to enforce on you goes something like this... "*the shiny new 1 TB NVMe SSD in my laptop has more than 1,000 times more IOPS and more than 5 times more performance than your \$50,000 storage device, while costing 100 times less! You have no idea what you're doing!*"

If you've been there, we feel for you. Rarely will you see so many discussions and fights about a piece of hardware in a box. But it's such an essential piece of hardware in a box that it's a good fight to have. So, let's explain some key concepts that libvirt uses in terms of storage access and how to work with it. Then, let's use our knowledge to extract as much performance as possible out of our storage system and libvirt using it.

In this chapter, we're basically going to cover almost all of these storage types via installation and configuration examples. Each and every one of these has its own use case, but generally, it's going to be up to you to choose what you're going to use.

So, let's start our journey through these supported protocols and learn how to configure them. After we cover storage pools, we are going to discuss NFS, a typical share-level protocol for virtual machine storage. Then, we're going to move to block-level protocols such as iSCSI and FC. Then, we will move to redundancy and multipathing to increase the availability and bandwidth of our storage devices. We're also going to cover various use cases for not-so-common filesystems (such as Ceph, Gluster, and GFS) for KVM virtualization. We're also going to discuss the new developments that are de facto trends right now.

Storage pools

When you first start using storage devices—even if they're cheaper boxes—you're faced with some choices. They will ask you to do a bit of configuration—select the RAID level, configure hot-spares, SSD caching...it's a process. The same process applies to a situation in which you're building a data center from scratch or extending an existing one. You have to configure the storage to be able to use it.

Hypervisors are a bit *picky* when it comes to storage, as there are storage types that they support and storage types that they don't support. For example, Microsoft's Hyper-V supports SMB shares for virtual machine storage, but it doesn't really support NFS storage for virtual machine storage. VMware's vSphere Hypervisor supports NFS, but it doesn't support SMB. The reason is simple—a company developing a hypervisor chooses and qualifies technologies that its hypervisor is going to support. Then, it's up to various HBA/controller vendors (Intel, Mellanox, QLogic, and so on) to develop drivers for that hypervisor, and it's up to storage vendor to decide which types of storage protocols they're going to support on their storage device.

From a CentOS perspective, there are many different storage pool types that are supported. Here are some of them:

- **Logical Volume Manager (LVM)**-based storage pools
- Directory-based storage pools
- Partition-based storage pools
- GlusterFS-based storage pools
- iSCSI-based storage pools
- Disk-based storage pools
- HBA-based storage pools, which use SCSI devices

From the perspective of libvirt, a storage pool can be a directory, a storage device, or a file that libvirt manages. That leads us to 10+ different storage pool types, as you're going to see in the next section. From a virtual machine perspective, libvirt manages virtual machine storage, which virtual machines use so that they have the capacity to store data.

oVirt, on the other hand, sees things a bit differently, as it has its own service that works with libvirt to provide centralized storage management from a data center perspective. *Data center perspective* might seem like a term that's a bit odd. But think about it—a datacenter is some kind of *higher-level* object in which you can see all of your resources. A data center uses *storage* and *hypervisors* to provide us with all of the services that we need in virtualization—virtual machines, virtual networks, storage domains, and so on. Basically, from a data center perspective, you can see what's happening on all of your hosts that are members of that datacenter. However, from a host level, you can't see what's happening on another host. It's a hierarchy that's completely logical from both a management and a security perspective.

oVirt can centrally manage these different types of storage pools (and the list can get bigger or smaller as the years go by):

- **Network File System (NFS)**
- **Parallel NFS (pNFS)**
- iSCSI
- FC
- Local storage (attached directly to KVM hosts)
- GlusterFS exports
- POSIX-compliant file systems

Let's take care of some terminology first:

- **Brtfs** is a type of filesystem that supports snapshots, RAID and LVM-like functionality, compression, defragmentation, online resizing, and many other advanced features. It was deprecated after it was discovered that its RAID5/6 can easily lead to a loss of data.
- **ZFS** is a type of filesystem that supports everything that Brtfs does, plus read and write caching.

CentOS has a new way of dealing with storage pools. Although still in technology preview state, it's worth going through the complete configuration via this new tool, called **Stratis**. Basically, a couple of years ago, Red Hat finally deprecated the idea of pushing Btrfs for future releases and started working on Stratis. If you've ever used ZFS, that's where this is probably going—an easy-to-manage, ZFS-like, volume-managing set of utilities that Red Hat can stand behind in their future releases. Also, just like ZFS, a Stratis-based pool can use cache; so, if you have an SSD that you'd like to dedicate to pool cache, you can actually do that, as well. If you have been expecting Red Hat to support ZFS, there's a fundamental Red Hat policy that stands in the way. Specifically, ZFS is not a part of the Linux kernel, mostly because of licensing reasons. Red Hat has a policy for these situations—if it's not a part of the kernel (upstream), then they don't provide nor support it. As it stands, that's not going to happen anytime soon. These policies are also reflected in CentOS.

Local storage pools

On the other hand, Stratis is available right now. We're going to use it to manage our local storage by creating storage pools. Creating a pool requires us to set up partitions or disks beforehand. After we create a pool, we can create a volume on top of it. We only have to be very careful about one thing—although Stratis can manage XFS filesystems, we shouldn't make changes to Stratis-managed XFS filesystems directly from the filesystem level. For example, do not reconfigure or reformat a Stratis-based XFS filesystem directly from XFS-based commands because you'll create havoc on your system.

Stratis supports various different types of block storage devices:

- Hard disks and SSDs
- iSCSI LUNs
- LVM
- LUKS
- MD RAID
- A device mapper multipath
- NVMe devices

Let's start from scratch and install Stratis so that we can use it. Let's use the following command:

```
yum -y install stratisd stratis-cli
systemctl enable --now stratisd
```

The first command installs the Stratis service and the corresponding command-line utilities. The second one will start and enable the Stratis service.

Now, we are going to go through a complete example of how to use Stratis to configure your storage devices. We're going to cover an example of this layered approach. So, what we are going to do is as follows:

- Create a software RAID10 + spare by using MD RAID.
- Create a Stratis pool out of that MD RAID device.
- Add a cache device to the pool to use Stratis' cache capability.
- Create a Stratis filesystem and mount it on our local server.

The premise here is simple—the software RAID10+ spare via MD RAID is going to approximate the regular production approach, in which you'd have some kind of a hardware RAID controller presenting a single block device to the system. We're going to add a cache device to the pool to verify the caching functionality, as this is something that we would most probably do if we were using ZFS, as well. Then, we are going to create a filesystem on top of that pool and mount it to a local directory with the help of the following commands:

```
mdadm --create /dev/md0 --verbose --level=10 --raid-devices=4 /  
dev/sdb /dev/sdc /dev/sdd /dev/sde --spare-devices=1 /dev/sdf2  
stratis pool create PacktStratisPool01 /dev/md0  
stratis pool add-cache PacktStratisPool01 /dev/sdg  
stratis pool add-cache PacktStratisPool01 /dev/sdg  
stratis fs create PackStratisPool01 PacktStratisXFS01  
mkdir /mnt/packtStratisXFS01  
mount /stratis/PacktStratisPool01/PacktStratisXFS01 /mnt/  
packtStratisXFS01
```

This mounted filesystem is XFS-formatted. We could then easily use this filesystem via NFS export, which is exactly what we're going to do in the NFS storage lesson. But for now, this was just an example of how to create a pool by using Stratis.

We've covered some basics of local storage pools, which brings us closer to our next subject, which is how to use pools from a libvirt perspective. So, that will be our next topic.

Libvirt storage pools

Libvirt manages its own storage pools, which is done with one thing in mind—to provide different pools for virtual machine disks and related data. Keeping in mind that libvirt uses what the underlying operating system supports, it's no wonder that it supports loads of different storage pool types. A picture is worth a thousand words, so here's a screenshot of creating a libvirt storage pool from virt-manager:

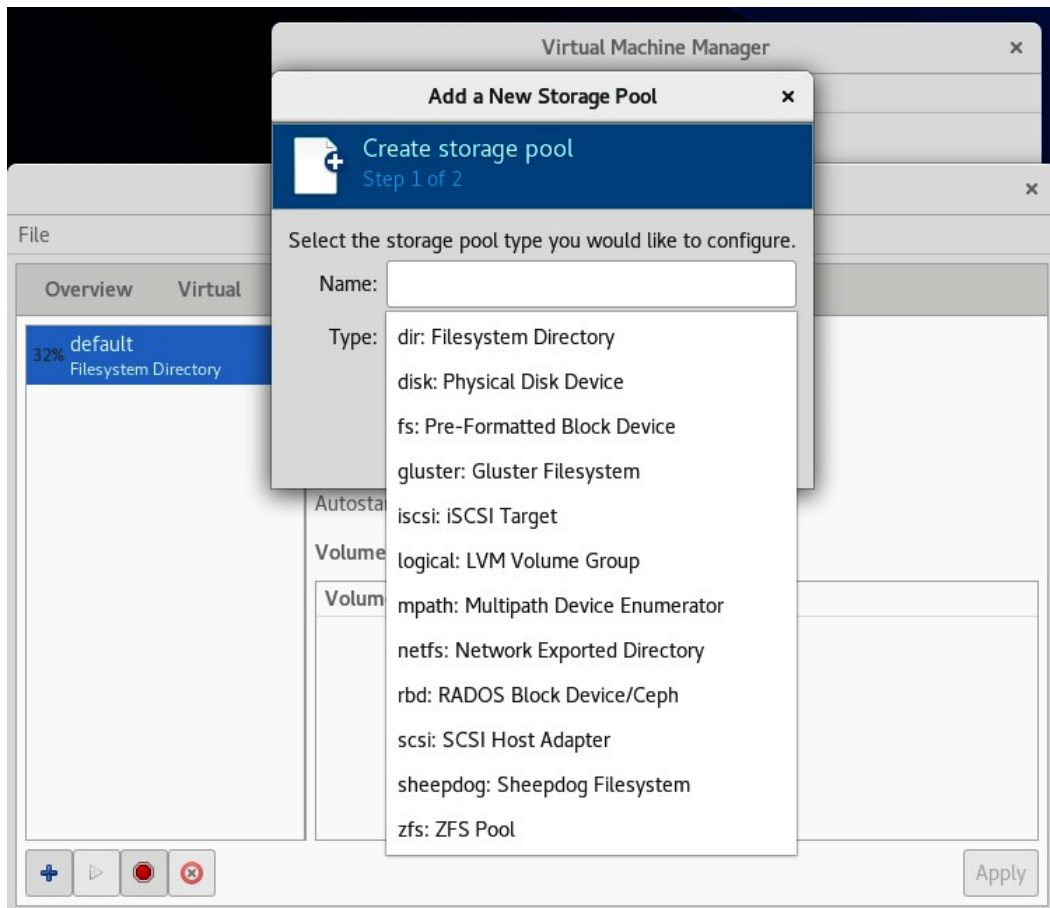


Figure 5.1 – Different storage pool types supported by libvirt

Out of the box, libvirt already has a predefined default storage pool, which is a directory storage pool on the local server. This default pool is located in the `/var/lib/libvirt/images` directory. This represents our default location where we'll save all the data from locally installed virtual machines.

We're going to create various different types of storage pools in the following sections—an NFS-based pool, an iSCSI and FC pool, and Gluster and Ceph pools: the whole nine yards. We're also going to explain when to use each and every one of them as there will be different usage models involved.

NFS storage pool

As a protocol, NFS has been around since the mid-80s. It was originally developed by Sun Microsystems as a protocol for sharing files, which is what it's been used for up to this day. Actually, it's still being developed, which is quite surprising for a technology that's so *old*. For example, NFS version 4.2 came out in 2016. In this version, NFS received a very big update, such as the following:

- **Server-side copy:** A feature that significantly enhances the speed of cloning operations between NFS servers by carrying out cloning directly between NFS servers
- **Sparse files and space reservation:** Features that enhance the way NFS works with files that have unallocated blocks, while keeping an eye on capacity so that we can guarantee space availability when we need to write data
- **Application data block support:** A feature that helps applications that work with files as block devices (disks)
- Better pNFS implementation

There are other bits and pieces that were enhanced in v4.2, but for now, this is more than enough. You can find even more information about this in IETF's RFC 7862 document (<https://tools.ietf.org/html/rfc7862>). We're going to focus our attention on the implementation of NFS v4.2 specifically, as it's the best that NFS currently has to offer. It also happens to be the default NFS version that CentOS 8 supports.

The first thing that we have to do is install the necessary packages. We're going to achieve that by using the following commands:

```
yum -y install nfs-utils
systemctl enable --now nfs-server
```

The first command installs the necessary utilities to run the NFS server. The second one is going to start it and permanently enable it so that the NFS service is available after reboot.

Our next task is to configure what we're going to share via the NFS server. For that, we need to *export* a directory and make it available to our clients over the network. NFS uses a configuration file, `/etc/exports`, for that purpose. Let's say that we want to create a directory called `/exports`, and then share it to our clients in the `192.168.159.0/255.255.255.0` network, and we want to allow them to write data on that share. Our `/etc/exports` file should look like this:

```
/mnt/packtStratisXFS01 192.168.159.0/24 (rw)
exportfs -r
```

These configuration options tell our NFS server which directory to export (`/exports`), to which clients (`192.168.159.0/24`), and what options to use (`rw` means read-write).

Some other available options include the following:

- `ro`: Read-only mode.
- `sync`: Synchronous I/O operations.
- `root_squash`: All I/O operations from UID 0 and GID 0 are mapped to configurable anonymous UIDs and GIDs (the `anonuid` and `anongid` options).
- `all_squash`: All I/O operations from any UIDs and GIDs are mapped to anonymous UIDs and GIDs (`anonuid` and `anongid` options).
- `no_root_squash`: All I/O operations from UID 0 and GID 0 are mapped to UID 0 and GID 0.

If you need to apply multiple options to the exported directory, you add them with a comma between them, as follows:

```
/mnt/packtStratisXFS01 192.168.159.0/24 (rw, sync, root_squash)
```

You can use fully qualified domain names or short hostnames (if they're resolvable by DNS or any other mechanism). Also, if you don't like using prefixes (`24`), you can use regular netmasks, as follows:

```
/mnt/packtStratisXFS01 192.168.159.0/255.255.255.0 (rw, root_squash)
```

Now that we have configured the NFS server, let's see how we're going to configure libvirt to use that server as a storage pool. As always, there are a couple of ways to do this. We could just create an XML file with the pool definition and import it to our KVM host by using the `virsh pool-define --file` command. Here's an example of that configuration file:

```
<pool type='netfs'>
  <name>NFSpool1</name>
  <source>
    <host name='192.168.159.144' />
    <dir path='/mnt/packtStratisXFS01' />
    <format type='auto' />
  </source>
  <target>
    <path>/var/lib/libvirt/images/NFSpool1</path>
    <permissions>
      <mode>0755</mode>
      <owner>0</owner>
      <group>0</group>
      <label>system_u:object_r:nfs_t:s0</label>
    </permissions>
  </target>
</pool>
```

Figure 5.2 – Example XML configuration file for NFS pool

Let's explain these configuration options:

- `pool type: netfs` means that we are going to use an NFS file share.
- `name`: The pool name, as libvirt uses pools as named objects, just like virtual networks.
- `host`: The address of the NFS server that we are connecting to.
- `dir path`: The NFS export path that we configured on the NFS server via `/etc/exports`.
- `path`: The local directory on our KVM host where that NFS share is going to be mounted to.
- `permissions`: The permissions used for mounting this filesystem.
- `owner` and `group`: The UID and GID used for mounting purposes (that's why we exported the folder earlier with the `no_root_squash` option).
- `label`: The SELinux label for this folder—we're going to discuss this in *Chapter 16, Troubleshooting Guideline for the KVM Platform*.

If we wanted, we could've easily done the same thing via the Virtual Machine Manager GUI. First, we would have to select the correct type (the NFS pool) and give it a name:

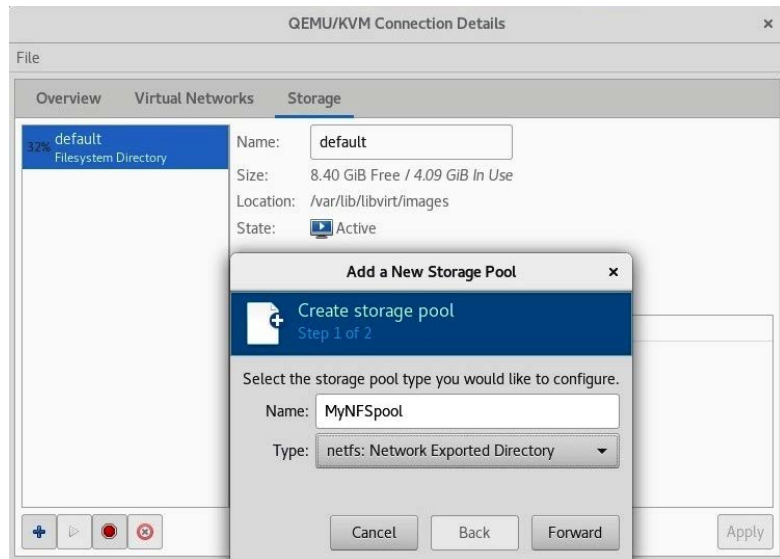


Figure 5.3 – Selecting the NFS pool type and giving it a name

After we click **Forward**, we can move to the final configuration step, where we need to tell the wizard which server we're mounting our NFS share from:



Figure 5.4 – Configuring NFS server options

When we finish typing in these configuration options (**Host Name** and **Source Path**), we can press **Finish**, which will mean exiting the wizard. Also, our previous configuration screen, which only contained the **default** storage pool, now has our newly configured pool listed as well:

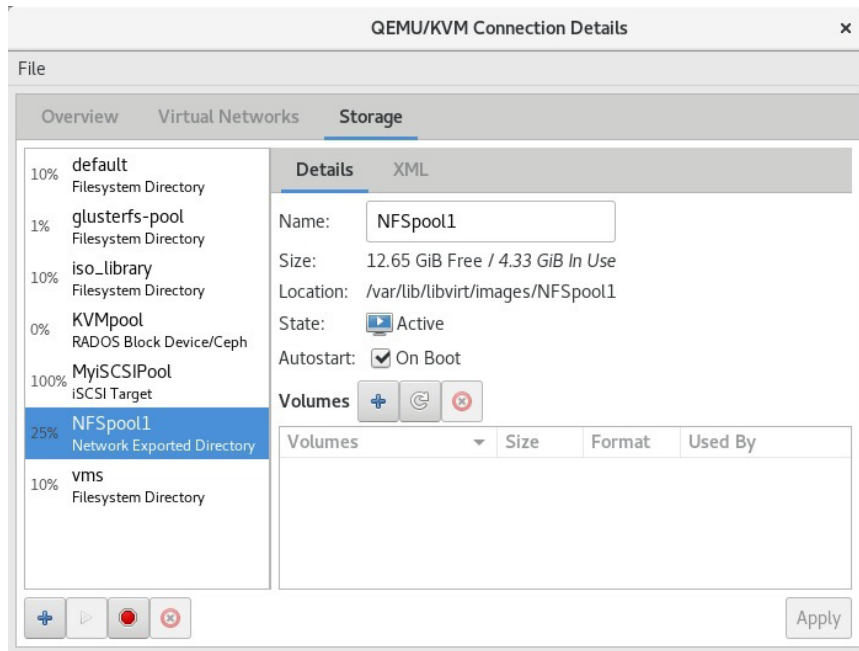


Figure 5.5 – Newly configured NFS pool visible on the list

When would we use NFS-based storage pools in libvirt, and for what? Basically, we can use them nicely for anything related to the storage of installation images—ISO files, virtual floppy disk files, virtual machine files, and so on.

Please remember that even though it seemed that NFS is almost gone from enterprise environments just a while ago, NFS is still around. Actually, with the introduction of NFS 4.1, 4.2, and pNFS, its future on the market actually looks even better than a couple of years ago. It's such a familiar protocol with a very long history, and it's still quite competitive in many scenarios. If you're familiar with VMware virtualization technology, VMware introduced a technology called Virtual Volumes in ESXi 6.0. This is an object-based storage technology that can use both block- and NFS-based protocols for its basis, which is a really compelling use case for some scenarios. But for now, let's move on to block-level technologies, such as iSCSI and FC.

iSCSI and SAN storage

Using iSCSI for virtual machine storage has long been the regular thing to do. Even if you take into account the fact that iSCSI isn't the most efficient way to approach storage, it's still so widely accepted that you'll find it everywhere. Efficiency is compromised for two reasons:

- iSCSI encapsulates SCSI commands into regular IP packages, which means segmentation and overhead as IP packages have a pretty large header, which means less efficiency.
- Even worse, it's TCP-based, which means that there are sequence numbers and retransmissions, which can lead to queueing and latency, and the bigger the environment is, the more you usually feel these effects affect your virtual machine performance.

That being said, the fact that it's based on an Ethernet stack makes it easier to deploy iSCSI-based solutions, while at the same time offering some unique challenges. For example, sometimes it's difficult to explain to a customer that using the same network switch(es) for virtual machine traffic and iSCSI traffic is not the best idea. What makes it even worse is the fact that clients are sometimes so blinded by their desire to save money that they don't understand that they're working against their own best interest. Especially when it comes to network bandwidth. Most of us have been there, trying to work with clients' questions such as *"but we already have a Gigabit Ethernet switch, why would you need anything faster than that?"*

The fact of the matter is, with iSCSI's intricacies, more is just – more. The more speed you have on the disk/cache/controller side and the more bandwidth you have on the networking side, the more chance you have of creating a storage system that's faster. All of that can have a big impact on our virtual machine performance. As you'll see in the *Storage redundancy and multipathing* section, you can actually build a very good storage system yourself—both for iSCSI and FC. This might come in real handy when you try to create some kind of a testing lab/environment to play with as you develop your KVM virtualization skills. You can apply that knowledge to other virtualized environments, as well.

The iSCSI and FC architectures are very similar—they both need a target (an iSCSI target and an FC target) and an initiator (an iSCSI initiator and an FC initiator). In this terminology, the target is a *server* component, and the initiator is a *client* component. To put it simply, the initiator connects to a target to get access to block storage that's presented via that target. Then, we can use the initiator's identity to *limit* what the initiator is able to see on the target. This is where the terminology starts to get a bit different when comparing iSCSI and FC.

In iSCSI, the initiator's identity can be defined by four different properties. They are as follows:

- **iSCSI Qualified Name (IQN):** This is a unique name that all initiators and targets have in iSCSI communication. We can compare this to a MAC or IP address in regular Ethernet-based networks. You can think of it this way—an IQN is for iSCSI what a MAC or IP address is for Ethernet-based networks.
- **IP address:** Every initiator will have a different IP address that it uses to connect to the target.
- **MAC address:** Every initiator has a different MAC address on Layer 2.
- **Fully Qualified Domain Name (FQDN):** This represents the name of the server as it's resolved by a DNS service.

From the iSCSI target perspective—depending on its implementation—you can use any one of these properties to create a configuration that's going to tell the iSCSI target which IQNs, IP addresses, MAC addresses, or FQDNs can be used to connect to it. This is what's called *masking*, as we can *mask* what an initiator can *see* on the iSCSI target by using these identities and pairing them with LUNs. LUNs are just raw, block capacities that we export via an iSCSI target toward initiators. LUNs are *indexed*, or *numbered*, usually from 0 onward. Every LUN number represents a different storage capacity that an initiator can connect to.

For example, we can have an iSCSI target with three different LUNs—LUN0 with 20 GB, LUN1 with 40 GB, and LUN2 with 60 GB. These will all be hosted on the same storage system's iSCSI target. We can then configure the iSCSI target to accept an IQN to see all the LUNs, another IQN to only see LUN1, and another IQN to only see LUN1 and LUN2. This is actually what we are going to configure right now.

Let's start by configuring the iSCSI target service. For that, we need to install the `targetcli` package, and configure the service (called `target`) to run:

```
yum -y install targetcli
systemctl enable --now target
```

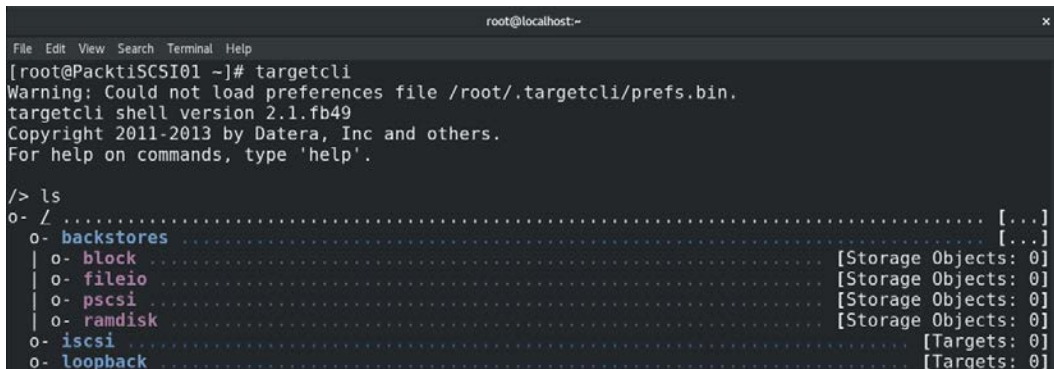

Be careful about the firewall configuration; you might need to configure it to allow connectivity on port 3260/tcp, which is the port that the iSCSI target portal uses. So, if your firewall has started, type in the following command:

```
firewall-cmd --permanent --add-port=3260/tcp ; firewall-cmd
--reload
```

There are three possibilities for iSCSI on Linux in terms of what storage backend to use. We could use a regular filesystem (such as XFS), a block device (a hard drive), or LVM. So, that's exactly what we're going to do. Our scenario is going to be as follows:

- LUN0 (20 GB): XFS-based filesystem, on the /dev/sdb device
- LUN1 (40 GB): Hard drive, on the /dev/sdc device
- LUN2 (60 GB): LVM, on the /dev/sdd device

So, after we install the necessary packages and configure the target service and firewall, we should start with configuring our iSCSI target. We'll just start the `targetcli` command and check the state, which should be a blank slate as we're just beginning the process:

A terminal window titled 'root@localhost:-' showing the execution of the 'targetcli' command. The prompt is '[root@PacktiSCSI01 ~]# targetcli'. The output shows a warning about a missing preferences file, followed by the targetcli shell version (2.1.fb49) and copyright information (2011-2013 by Datera, Inc and others). The user enters the command '/> ls' and the output shows a tree structure of storage backends: 'backstores' (with sub-items 'block', 'fileio', 'pscsi', 'ramdisk'), 'iscsi', and 'loopback'. Each item shows its current state, such as '[Storage Objects: 0]' or '[Targets: 0]'.

```
File Edit View Search Terminal Help
root@localhost:-
[root@PacktiSCSI01 ~]# targetcli
Warning: Could not load preferences file /root/.targetcli/prefs.bin.
targetcli shell version 2.1.fb49
Copyright 2011-2013 by Datera, Inc and others.
For help on commands, type 'help'.

/> ls
0- / ..... [..]
  0- backstores ..... [..]
    | 0- block ..... [Storage Objects: 0]
    | 0- fileio ..... [Storage Objects: 0]
    | 0- pscsi ..... [Storage Objects: 0]
    | 0- ramdisk ..... [Storage Objects: 0]
  0- iscsi ..... [Targets: 0]
  0- loopback ..... [Targets: 0]
```

Figure 5.6 – The targetcli starting point – empty configuration

Let's start with the step-by-step procedure:

1. So, let's configure the XFS-based filesystem and configure the LUN0 file image to be saved there. First, we need to partition the disk (in our case, `/dev/sdb`):

```

root@localhost:~
File Edit View Search Terminal Help
[root@PacktiSCSI01 ~]# fdisk /dev/sdb

Welcome to fdisk (util-linux 2.32.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Command (m for help): n
Partition type
  p   primary (0 primary, 0 extended, 4 free)
  e   extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-41943039, default 2048):
Last sector, +sectors or +size{K,M,G,T,P} (2048-41943039, default 41943039):

Created a new partition 1 of type 'Linux' and of size 20 GiB.

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.

```

Figure 5.7 – Partitioning `/dev/sdb` for the XFS filesystem

2. The next step is to format this partition, create and use a directory called `/LUN0` to mount this filesystem, and serve our LUN0 image, which we're going to configure in the next steps:

```

root@localhost:~
File Edit View Search Terminal Help
[root@PacktiSCSI01 ~]# mkfs.xfs /dev/sdb1 ; mkdir /LUN0 ; mount /dev/sdb1 /LUN0
meta-data=/dev/sdb1          isize=512    agcount=4, agsize=1310656 blks
=                               sectsz=512   attr=2, projid32bit=1
=                               crc=1       finobt=1, sparse=1, rmapbt=0
=                               reflink=1
data            =             bsize=4096  blocks=5242624, imaxpct=25
=                               sunit=0
naming         =version 2     bsize=4096  ascii-ci=0, ftype=1
log            =internal log  bsize=4096  blocks=2560, version=2
=                               sectsz=512   sunit=0 blks, lazy-count=1
realtime      =none         extsz=4096  blocks=0, rtextents=0

```

Figure 5.8 – Formatting the XFS filesystem, creating a directory, and mounting it to that directory

- The next step is configuring `targetcli` so that it creates LUN0 and assigns an image file for LUN0, which will be saved in the `/LUN0` directory. First, we need to start the `targetcli` command:

```

root@localhost:~
File Edit View Search Terminal Help
[root@PacktiSCSI01 ~]# targetcli
targetcli shell version 2.1.fb49
Copyright 2011-2013 by Datera, Inc and others.
For help on commands, type 'help'.

/> /iscsi
/iscsi> create
Created target iqn.2003-01.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdabb1.
Created TPG 1.
Global pref auto add default portal=true
Created default portal listening on all IPs (0.0.0.0), port 3260.
/iscsi> /backstores
/backstores> /backstores/fileio create LUN0 /LUN0/LUN0.img 20000M write_back=false
Created fileio LUN0 with size 20971520000
/backstores> ls
o- backstores ..... [Storage Objects: 0]
o- block ..... [Storage Objects: 1]
o- fileio ..... [Storage Objects: 1]
  o- LUN0 ..... [LUN0/LUN0.img (19.5GiB) write-thru deactivated]
    o- alua ..... [ALUA Groups: 1]
      o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
o- pscsi ..... [Storage Objects: 0]
o- ramdisk ..... [Storage Objects: 0]

```

Figure 5.9 – Creating an iSCSI target, LUN0, and hosting it as a file

- Next, let's configure a block device-based LUN backend— LUN2—which is going to use `/dev/sdc1` (create the partition using the previous example) and check the current state:

```

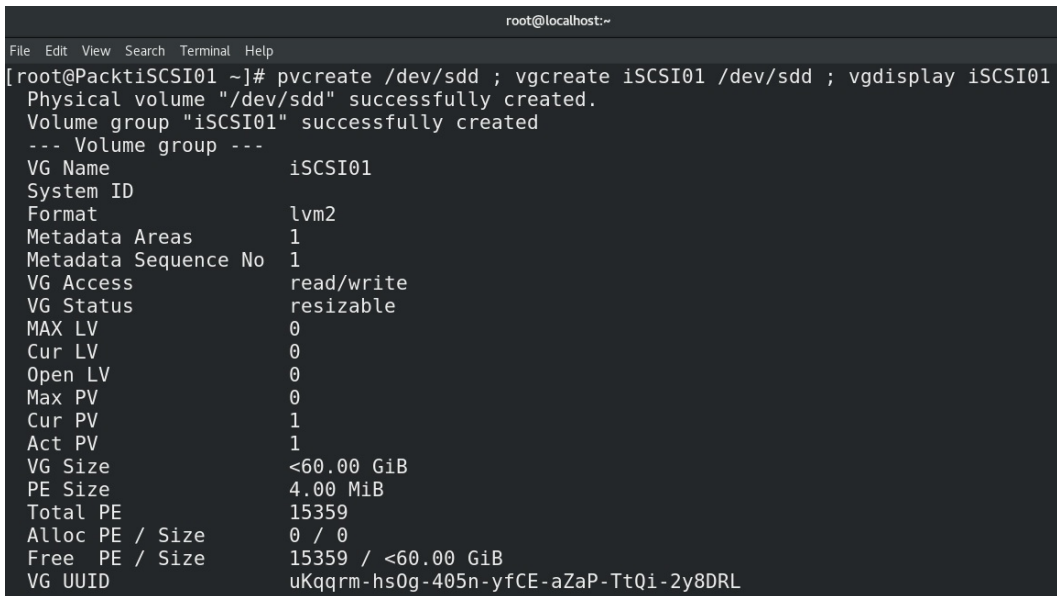
root@localhost:~
File Edit View Search Terminal Help
/backstores> /backstores/block create name=LUN1 dev=/dev/sdc1
Created block storage object LUN1 using /dev/sdc1.
/backstores> cd /
/> ls
o- / ..... [Storage Objects: 1]
o- backstores ..... [Storage Objects: 1]
  o- block ..... [Storage Objects: 1]
    o- LUN1 ..... [/dev/sdc1(40.0GiB) write-thru deactivated]
      o- alua ..... [ALUA Groups: 1]
        o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
o- fileio ..... [Storage Objects: 1]
  o- LUN0 ..... [LUN0/LUN0.img (19.5GiB) write-thru deactivated]
    o- alua ..... [ALUA Groups: 1]
      o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
o- pscsi ..... [Storage Objects: 0]
o- ramdisk ..... [Storage Objects: 0]
o- iscsi ..... [Targets: 1]
  o- iqn.2003-01.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdabb1 ..... [TPGs: 1]
    o- tpg1 ..... [no-gen-acls, no-auth]
      o- acls ..... [ACLs: 0]
        o- luns ..... [LUNs: 0]
          o- portals ..... [Portals: 1]
            o- 0.0.0.0:3260 ..... [OK]
o- loopback ..... [Targets: 0]

```

Figure 5.10 – Creating LUN1, hosting it directly from a block device

So, LUN0 and LUN1 and their respective backends are now configured. Let's finish things off by configuring LVM:

1. First, we are going to prepare the physical volume for LVM, create a volume group out of that volume, and display all the information about that volume group so that we can see how much space we have for LUN2:

A terminal window with a dark background and light text. The title bar reads 'root@localhost:~'. The menu bar includes 'File Edit View Search Terminal Help'. The prompt is '[root@PacktiISCSI01 ~]#'. The user enters the command 'pvcreate /dev/sdd ; vgcreate iSCSI01 /dev/sdd ; vgdisplay iSCSI01'. The output shows two successful creation messages, followed by a separator '--- Volume group ---' and a detailed list of VG properties for 'iSCSI01'.

```
root@localhost:~
File Edit View Search Terminal Help
[root@PacktiISCSI01 ~]# pvcreate /dev/sdd ; vgcreate iSCSI01 /dev/sdd ; vgdisplay iSCSI01
Physical volume "/dev/sdd" successfully created.
Volume group "iSCSI01" successfully created
--- Volume group ---
VG Name                iSCSI01
System ID
Format                 lvm2
Metadata Areas        1
Metadata Sequence No  1
VG Access              read/write
VG Status              resizable
MAX LV                 0
Cur LV                0
Open LV                0
Max PV                 0
Cur PV                1
Act PV                 1
VG Size                <60.00 GiB
PE Size                4.00 MiB
Total PE               15359
Alloc PE / Size        0 / 0
Free PE / Size         15359 / <60.00 GiB
VG UUID                uKqqrM-hs0g-405n-yfCE-aZaP-TtQi-2y8DRL
```

Figure 5.11 – Configuring the physical volume for LVM, building a volume group, and displaying information about that volume group

- The next step is to actually create the logical volume, which is going to be our block storage device backend for LUN2 in the iSCSI target. We can see from the `vgdisplay` output that we have 15,359 4 MB blocks available, so let's use that to create our logical volume, called LUN2. Go to `targetcli` and configure the necessary settings for LUN2:

```

root@localhost:~# /> /backstores/block create name=LUN2 dev=/dev/iSCSI01/LUN2
Created block storage object LUN2 using /dev/iSCSI01/LUN2.
root@localhost:~# /> cd /
root@localhost:~# /> ls
0- /
  o- backstores
    o- block
      o- LUN1
      | o- alua
      |   o- default_tg_pt_gp
      o- LUN2
      | o- alua
      |   o- default_tg_pt_gp
    o- fileio
    o- LUN0
      | o- alua
      |   o- default_tg_pt_gp
    o- pscsi
    o- ramdisk
  o- iscsi
    o- iqn.2003-01.org.linux-iscsi.packticlscsi01.x8664:sn.7b3c2efdbb11
      o- tpg1
      | o- acls
      | o- luns
      | o- portals
      |   o- 0.0.0.0:3260
    o- loopback
  />

```

Figure 5.12 – Configuring LUN2 with the LVM backend

- Let's stop here for a second and switch to the KVM host (the iSCSI initiator) configuration. First, we need to install the iSCSI initiator, which is part of a package called `iscsi-initiator-utils`. So, let's use the `yum` command to install that:

```
yum -y install iscsi-initiator-utils
```

- Next, we need to configure the IQN of our initiator. We usually want this name to be reminiscent of the hostname, so, seeing that our host's FQDN is `PacktStratis01`, we'll use that to configure the IQN. To do that, we need to edit the `/etc/iscsi/initiatorname.iscsi` file and configure the `InitiatorName` option. For example, let's set it to `iqn.2019-12.com.packt:PacktStratis01`. The content of the `/etc/iscsi/initiatorname.iscsi` file should be as follows:

```
InitiatorName=iqn.2019-12.com.packt:PacktStratis01
```

- Now that this is configured, let's go back to the iSCSI target and create an **Access Control List (ACL)**. The ACL is going to allow our KVM host's initiator to connect to the iSCSI target portal:

```
root@localhost:~# cd /iscsi/iqn.2003-01.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdbb11/tpg1/
/iscsi/iqn.20...2efdbb11/tpg1> ls
o- tpg1 ..... [no-gen-acls, no-auth]
o- acls ..... [ACLs: 0]
o- luns ..... [LUNs: 0]
o- portals ..... [Portals: 1]
o- 0.0.0.0:3260 ..... [OK]
/iscsi/iqn.20...2efdbb11/tpg1> cd acls
/iscsi/iqn.20...b11/tpg1/acls> create iqn.2019-12.com.packt:PacktStratis01
Created Node ACL for iqn.2019-12.com.packt:packtstratis01
/iscsi/iqn.20...b11/tpg1/acls>
```

Figure 5.13 – Creating an ACL so that the KVM host's initiator can connect to the iSCSI target

- Next, we need to publish our pre-created file-based and block-based devices to the iSCSI target LUNs. So, we need to do this:

```
/iscsi/iqn.20...b11/tpg1/acls> cd ../luns
/iscsi/iqn.20...b11/tpg1/luns> create /backstores/fileio/LUN0
Created LUN 0.
Created LUN 0->0 mapping in node ACL iqn.2019-12.com.packt:packtstratis01
/iscsi/iqn.20...b11/tpg1/luns> create /backstores/block/LUN1
Created LUN 1.
Created LUN 1->1 mapping in node ACL iqn.2019-12.com.packt:packtstratis01
/iscsi/iqn.20...b11/tpg1/luns> create /backstores/block/LUN2
Created LUN 2.
Created LUN 2->2 mapping in node ACL iqn.2019-12.com.packt:packtstratis01
```

Figure 5.14 – Adding our file-based and block-based devices to the iSCSI target LUNs 0, 1, and 2

The end result should look like this:

```

root@localhost:~# cd /
root@localhost:~# ls
o- /
o- backstores ..... [..]
  o- block ..... [Storage Objects: 2]
    o- LUN1 ..... [/dev/sdc1(40.0GiB) write-thru activated]
      o- alua ..... [ALUA Groups: 1]
        o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
    o- LUN2 ..... [/dev/iscsi01/LUN2 (60.0GiB) write-thru activated]
      o- alua ..... [ALUA Groups: 1]
        o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
  o- fileio ..... [Storage Objects: 1]
    o- LUN0 ..... [/LUN0/LUN0.img (19.5GiB) write-thru activated]
      o- alua ..... [ALUA Groups: 1]
        o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
  o- pscsi ..... [Storage Objects: 0]
  o- ramdisk ..... [Storage Objects: 0]
o- iscsi ..... [Targets: 1]
  o- iqn.2003-01.org.linux-iscsi.packttiscsi01.x8664:sn.7b3c2efdbb11 ..... [TPGs: 1]
    o- tpg1 ..... [no-gen-acls, no-auth]
      o- acls ..... [ACLs: 1]
        o- iqn.2019-12.com.packt:packtstratis01 ..... [Mapped LUNs: 3]
          o- mapped_lun0 ..... [lun0 fileio/LUN0 (rw)]
          o- mapped_lun1 ..... [lun1 block/LUN1 (rw)]
          o- mapped_lun2 ..... [lun2 block/LUN2 (rw)]
        o- luns ..... [LUNs: 3]
          o- lun0 ..... [fileio/LUN0 (/LUN0/LUN0.img) (default tg_pt_gp)]
          o- lun1 ..... [block/LUN1 (/dev/sdc1) (default tg_pt_gp)]
          o- lun2 ..... [block/LUN2 (/dev/iscsi01/LUN2) (default tg_pt_gp)]
      o- portals ..... [Portals: 1]
        o- 0.0.0.0:3260 ..... [OK]
o- loopback ..... [Targets: 0]

```

Figure 5.15 – The end result

At this point, everything is configured. We need to go back to our KVM host and define a storage pool that will use these LUNs. The easiest way to do that would be to use an XML configuration file for the pool. So, here's our sample configuration XML file; we'll call it `iscsiPool.xml`:

```

<pool type='iscsi'>
  <name>MyiSCSIpool</name>
  <source>
    <host name='192.168.159.145' />
    <device path='iqn.2003-01.org.linux-iscsi.packttiscsi01.x8664:sn.7b3c2efdbb11' />
  </source>
  <initiator>
    <iqn name='iqn.2019-12.com.packt:PacktStratis01' />
  </initiator>
  <target>

```

```

    <path>/dev/disk/by-path</path>
  </target>
</pool>

```

Let's explain the file step by step:

- `pool type= 'iscsi'`: We're telling libvirt that this is an iSCSI pool.
- `name`: The pool name.
- `host name`: The IP address of the iSCSI target.
- `device path`: The IQN of the iSCSI target.
- The IQN name in the initiator section: The IQN of the initiator.
- `target path`: The location where iSCSI target's LUNs will be mounted.

Now, all that's left for us to do is to define, start, and autostart our new iSCSI-backed KVM storage pool:

```

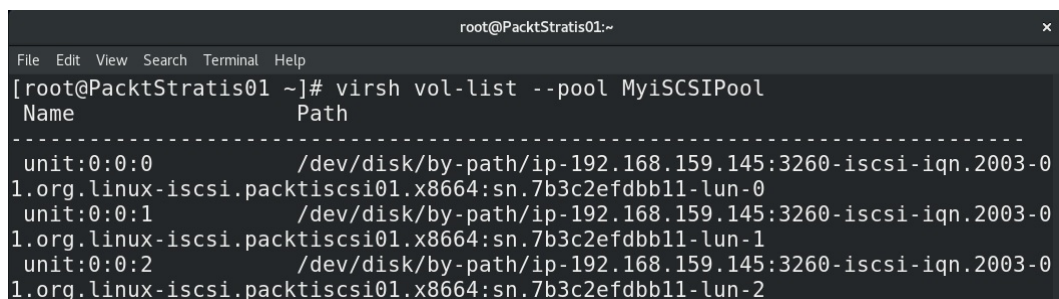
virsh pool-define --file iSCSIpool.xml
virsh pool-start --pool MyiSCSIpool
virsh pool-autostart --pool MyiSCSIpool

```

The target path part of the configuration can be easily checked via `virsh`. If we type the following command into the KVM host, we will get the list of available LUNs from the `MyiSCSIpool` pool that we just configured:

```
virsh vol-list --pool MyiSCSIpool
```

We get the following result for this command:



```

root@PacktStratis01:~
File Edit View Search Terminal Help
[root@PacktStratis01 ~]# virsh vol-list --pool MyiSCSIpool
Name                               Path
-----
unit:0:0:0                         /dev/disk/by-path/ip-192.168.159.145:3260-iscsi-iqn.2003-0
1.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdbb11-lun-0
unit:0:0:1                         /dev/disk/by-path/ip-192.168.159.145:3260-iscsi-iqn.2003-0
1.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdbb11-lun-1
unit:0:0:2                         /dev/disk/by-path/ip-192.168.159.145:3260-iscsi-iqn.2003-0
1.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdbb11-lun-2

```

Figure 5.16 – Runtime names for our iSCSI pool LUNs

If this output reminds you a bit of the VMware vSphere Hypervisor storage runtime names, you are definitely on the right track. We will be able to use these storage pools in *Chapter 7, Virtual Machine – Installation, Configuration, and Life-Cycle Management*, when we start deploying our virtual machines.

Storage redundancy and multipathing

Redundancy is one of the keywords of IT, where any single component failure could mean big problems for a company or its customers. The general design principle of avoiding SPOF is something that we should always stick to. At the end of the day, no network adapter, cable, switch, router, or storage controller is going to work forever. So, calculating redundancy into our designs helps our IT environment during its normal life cycle.

At the same time, redundancy can be combined with multipathing to also ensure higher throughput. For example, when we connect our physical host to FC storage with two controllers with four FC ports each, we can use four paths (if the storage is active-passive) or eight paths (if it's active-active) to the same LUN(s) exported from this storage device to a host. This gives us multiple additional options for LUN access, on top of the fact that it gives us more availability, even in the case of failure.

Getting a regular KVM host to do, for example, iSCSI multipathing is quite a bit complex. There are multiple configuration issues and blank spots in terms of documentation, and supportability of such a configuration is questionable. However, there are products that use KVM that support it out of the box, such as oVirt (which we covered before) and **Red Hat Enterprise Virtualization Hypervisor (RHEV-H)**. So, let's use oVirt for this example on iSCSI.

Before you do this, make sure that you have done the following:

- Your Hypervisor host is added to the oVirt inventory.
- Your Hypervisor host has two additional network cards, independent of the management network.
- The iSCSI storage has two additional network cards in the same L2 networks as the two additional hypervisor network cards.
- The iSCSI storage is configured so that it has at least a target and a LUN already configured in a way that will enable the hypervisor host to connect to it.

So, as we're doing this in oVirt, there are a couple of things that we need to do. First, from a networking perspective, it would be a good idea to create some storage networks. In our case, we're going to assign two networks for iSCSI, and we will call them `iSCSI01` and `iSCSI02`. We need to open the oVirt administration panel, hover over **Network**, and select **Networks** from the menu. This will open a pop-up window for the **New Logical Network** wizard. So, we just need to name the network `iSCSI01` (for the first one), uncheck the **VM network** checkbox (as this isn't a virtual machine network), and go to the **Cluster** tab, where we deselect the **Require all** checkbox. Repeat the whole process again for the `iSCSI02` network:

The screenshot shows the 'New Logical Network' wizard with the following configuration:

- General** tab selected.
- Data Center: Primary
- Name: iSCSI01
- Description: (empty)
- Comment: (empty)
- Network Parameters**
 - Network Label: (empty)
 - Enable VLAN tagging:
 - VM network: (with a green 'vm' icon)
 - MTU: Default (1500), Custom
 - Host Network QoS: [Unlimited]

Buttons: OK, Cancel

Figure 5.17 – Configuring networks for iSCSI bond

The next step is assigning these networks to host network adapters. Go to `compute/hosts`, double-click on the host that you added to oVirt's inventory, select the **Network interfaces** tab, and click on the **Setup Host Networks** icon in the top-right corner. In that UI, drag and drop `iSCSI01` on the second network interface and `iSCSI02` on the third network interface. The first network interface is already taken by the oVirt management network. It should look something like this:

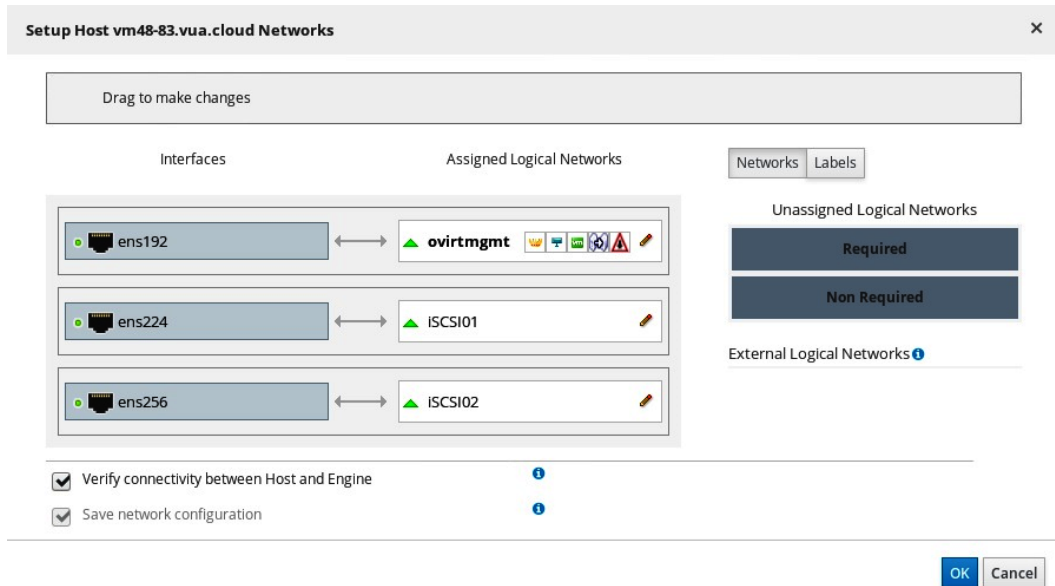


Figure 5.18 – Assigning virtual networks to the hypervisor's physical adapters

Before you close the window down, make sure that you click on the *pencil* sign on both `iSCSI01` and `iSCSI02` to set up IP addresses for these two virtual networks. Assign network configuration that can connect you to your iSCSI storage on the same or different subnets:

Edit iSCSI Bond
✕

Name

Description

Logical Networks

	Name	Description
<input checked="" type="checkbox"/>	iSCSI01	
<input checked="" type="checkbox"/>	iSCSI02	

Storage Targets

	IQN	Address	Port
<input checked="" type="checkbox"/>	iqn.2003-01.org.linux-iscsi.vm49-199.x...	192.168.2.11	3260
<input type="checkbox"/>			

OK
Cancel

Figure 5.19 – Creating an iSCSI bond on the data center level

There you go, you have just configured an iSCSI bond. The last part of our configuration is enabling it. Again, in the oVirt GUI, go to **Compute | Data Centers**, select your datacenter with a double-click, and go to the **iSCSI Multipathing** tab:

The screenshot shows the oVirt Open Virtualization Manager interface. The breadcrumb navigation is **Compute > Data Centers > Primary**. The **iSCSI Multipathing** tab is highlighted in red. Below the navigation are buttons for **Attach Data**, **Attach ISO**, **Attach Export**, **Detach**, **Activate**, and **Maintenance**. A table displays the configuration:

	Domain Name	Domain Type	Status	Free Space	Used Space	Total Space	Description
▶	primary	Data (Master)	Active	15 GiB	6 GiB	21 GiB	

Figure 5.20 – Configuring iSCSI multipathing on the data center level

Click on the **Add** button at the top-right side and go through the wizard. Specifically, select both the **iSCSI01** and **iSCSI02** networks in the top part of the pop-up window, and the iSCSI target on the lower side.

Now that we have covered the basics of storage pools, NFS, and iSCSI, we can move on to a standard open source way of deploying storage infrastructure, which would be to use Gluster and/or Ceph.

Gluster and Ceph as a storage backend for KVM

There are other advanced types of filesystems that can be used as the libvirt storage backend. So, let's now discuss two of them—Gluster and Ceph. Later, we'll also check how libvirt works with GFS2.

Gluster

Gluster is a distributed filesystem that's often used for high-availability scenarios. Its main advantages over other filesystems are the fact that it's scalable, it can use replication and snapshots, it can work on any server, and it's usable as a basis for shared storage—for example, via NFS and SMB. It was developed by a company called Gluster Inc., which was acquired by RedHat in 2011. However, unlike Ceph, it's a *file* storage service, while Ceph offers *block* and *object*-based storage. Object-based storage for block-based devices means direct, binary storage, directly to a LUN. There are no filesystems involved, which theoretically means less overhead as there's no filesystem, filesystem tables, and other constructs that might slow the I/O process down.

Let's first configure Gluster to show its use case with libvirt. In production, that means installing at least three Gluster servers so that we can make high availability possible. Gluster configuration is really straightforward, and in our example, we are going to create three CentOS 7 machines that we will use to host the Gluster filesystem. Then, we will mount that filesystem on our hypervisor host and use it as a local directory. We can use GlusterFS directly from libvirt, but the implementation is just not as refined as using it via the gluster client service, mounting it as a local directory, and using it directly as a directory pool in libvirt.

Our configuration will look like this:

Hostname	gluster1	gluster2	gluster3
IP	192.168.159.147/24	192.168.159.148/24	192.168.159.149/24
Gluster disk	/dev/sdb (10GB)	/dev/sdb (10GB)	/dev/sdb (10GB)
Gluster local directory	/gluster/bricks/1	/gluster/bricks/1	/gluster/bricks/1

Figure 5.21 – Basic settings for our Gluster cluster

So, let's put that into production. We have to issue a large sequence of commands on all of the servers before we configure Gluster and expose it to our KVM host. Let's start with `gluster1`. First, we are going to do a system-wide update and reboot to prepare the core operating system for Gluster installation. Type the following commands into all three CentOS 7 servers:

```
yum -y install epel-release*
yum -y install centos-release-gluster7.noarch
yum -y update
yum -y install glusterfs-server
systemctl reboot
```

Then, we can start deploying the necessary repositories and packages, format disks, configure the firewall, and so on. Type the following commands into all the servers:

```
mkfs.xfs /dev/sdb
mkdir /gluster/bricks/1 -p
echo '/dev/sdb /gluster/bricks/1 xfs defaults 0 0' >> /etc/fstab
mount -a
mkdir /gluster/bricks/1/brick
systemctl disable firewalld
systemctl stop firewalld
systemctl start glusterd
systemctl enable glusterd
```

We need to do a bit of networking configuration as well. It would be good if these three servers can *resolve* each other, which means either configuring a DNS server or adding a couple of lines to our `/etc/hosts` file. Let's do the latter. Add the following lines to your `/etc/hosts` file:

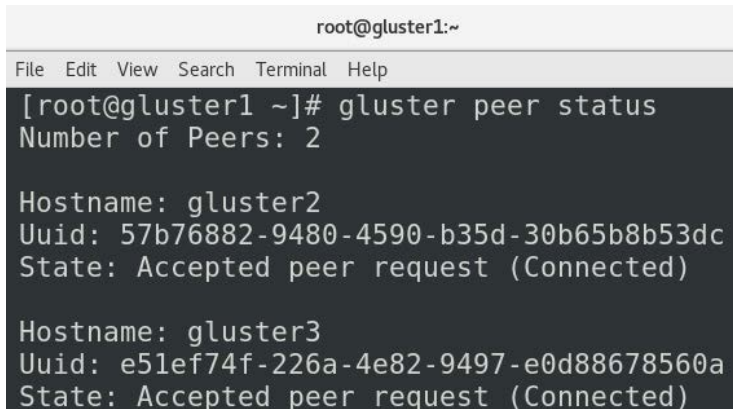
```
192.168.159.147 gluster1
192.168.159.148 gluster2
192.168.159.149 gluster3
```

For the next part of the configuration, we can just log in to the first server and use it as the de facto management server for our Gluster infrastructure. Type in the following commands:

```
gluster peer probe gluster1
gluster peer probe gluster2
```

```
gluster peer probe gluster3
gluster peer status
```

The first three commands should get you the `peer probe: success` status. The third one should return an output similar to this:



```
root@gluster1:~
File Edit View Search Terminal Help
[root@gluster1 ~]# gluster peer status
Number of Peers: 2

Hostname: gluster2
Uuid: 57b76882-9480-4590-b35d-30b65b8b53dc
State: Accepted peer request (Connected)

Hostname: gluster3
Uuid: e51ef74f-226a-4e82-9497-e0d88678560a
State: Accepted peer request (Connected)
```

Figure 5.22 – Confirmation that the Gluster servers peered successfully

Now that this part of the configuration is done, we can create a Gluster-distributed filesystem. We can do this by typing the following sequence of commands:

```
gluster volume create kvmgluster replica 3 \ gluster1:/gluster/
bricks/1/brick gluster2:/gluster/bricks/1/brick \ gluster3:/
gluster/bricks/1/brick
gluster volume start kvmgluster
gluster volume set kvmgluster auth.allow 192.168.159.0/24
gluster volume set kvmgluster allow-insecure on
gluster volume set kvmgluster storage.owner-uid 107
gluster volume set kvmgluster storage.owner-gid 107
```

Then, we could mount Gluster as an NFS directory for testing purposes. For example, we can create a distributed namespace called `kvmgluster` to all of the member hosts (`gluster1`, `gluster2`, and `gluster3`). We can do this by using the following commands:

```
echo 'localhost:/kvmgluster /mnt glusterfs \ defaults,_
netdev,backupvolfile-server=localhost 0 0' >> /etc/fstab
mount.glusterfs localhost:/kvmgluster /mnt
```

The Gluster part is now ready, so we need to go back to our KVM host and mount the Gluster filesystem to it by typing in the following commands:

```
wget \ https://download.gluster.org/pub/gluster/glusterfs/6/  
LATEST/CentOS/glusterfs-rhel8.repo -P /etc/yum.repos.d  
yum install glusterfs glusterfs-fuse attr -y  
mount -t glusterfs -o context="system_u:object_r:virt_  
image_t:s0" \ gluster1:/kvmgluster /var/lib/libvirt/images/  
GlusterFS
```

We have to pay close attention to Gluster releases on the server and client, which is why we downloaded the Gluster repository information for CentOS 8 (we're using it on the KVM server) and installed the necessary Gluster client packages. That enabled us to mount the filesystem with the last command.

Now that we've finished our configuration, we just need to add this directory as a libvirt storage pool. Let's do that by using an XML file with the storage pool definition, which contains the following entries:

```
<pool type='dir'>  
  <name>glusterfs-pool</name>  
  <target>  
    <path>/var/lib/libvirt/images/GlusterFS</path>  
    <permissions>  
      <mode>0755</mode>  
      <owner>107</owner>  
      <group>107</group>  
      <label>system_u:object_r:virt_image_t:s0</label>  
    </permissions>  
  </target>  
</pool>
```

Let's say that we saved this file in the current directory, and that the file is called `gluster.xml`. We can import and start it in libvirt by using the following `virsh` commands:

```
virsh pool-define --file gluster.xml  
virsh pool-start --pool glusterfs-pool  
virsh pool-autostart --pool glusterfs-pool
```


We should mount this pool automatically on boot so that libvirt can use it. Therefore, we need to add the following line to `/etc/fstab`:

```
gluster1:/kvmgluster          /var/lib/libvirt/images/GlusterFS \
glusterfs defaults,_netdev 0 0
```

Using a directory-based approach enables us to avoid two problems that libvirt (and its GUI interface, `virt-manager`) has with Gluster storage pools:

- We can use Gluster's failover capability, which will be managed automatically by the Gluster utilities that we installed directly, as libvirt doesn't support them yet.
- We will avoid creating virtual machine disks *manually*, which is another limitation of libvirt's implementation of Gluster support, while directory-based storage pools support it without any issues.

It seems weird that we're mentioning *failover*, as it seems as though we didn't configure it as a part of any of the previous steps. Actually, we have. When we issued the last mount command, we used Gluster's built-in modules to establish connectivity to the *first* Gluster server. That, in turn, means that after this connection, we got all of the details about the whole Gluster pool, which we configured so that it's hosted on three servers. If any kind of failure happens—which we can easily simulate—this connection will continue working. We can simulate this scenario by turning off any of the Gluster servers, for example—`gluster1`. You'll see that the local directory where we mounted Gluster directory still works, even though `gluster1` is down. Let's see that in action (the default timeout period is 42 seconds):

```
root@PacktStratis01:/var/lib/libvirt/images/GlusterFS
File Edit View Search Terminal Help
[root@PacktStratis01 GlusterFS]# ping gluster1
PING gluster1 (192.168.159.147) 56(84) bytes of data:
From PacktStratis01 (192.168.159.143) icmp_seq=1 Destination Host Unreachable
From PacktStratis01 (192.168.159.143) icmp_seq=2 Destination Host Unreachable
From PacktStratis01 (192.168.159.143) icmp_seq=3 Destination Host Unreachable
^C
--- gluster1 ping statistics ---
 3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 75ms
pipe 3
[root@PacktStratis01 GlusterFS]# cat FailoverFile
This is my file content!
```

Figure 5.23 – Gluster failover working; the first node is down, but we're still able to get our files

If we want to be more aggressive, we can shorten this timeout period to—for example—2 seconds by issuing the following command on any of our Gluster servers:

```
gluster volume set kvmgluster network.ping-timeout number
```

The number part is in seconds, and by assigning it a lower number, we can directly influence how aggressive the failover process is.

So, now that everything is configured, we can start using the Gluster pool to deploy virtual machines, which we will discuss further in *Chapter 7, Virtual Machine – Installation, Configuration, and Life-Cycle Management*.

Seeing as Gluster is a file-based backend that can be used for libvirt, it's only natural to describe how to use an advanced block-level and object-level storage backend. That's where Ceph comes in, so let's work on that now.

Ceph

Ceph can act as file-, block-, and object-based storage. But for the most part, we're usually using it as either block- or object-based storage. Again, this is a piece of open source software that's designed to work on any server (or a virtual machine). In its core, Ceph runs an algorithm called **Controlled Replication Under Scalable Hashing (CRUSH)**. This algorithm tries to distribute data across object devices in a pseudo-random manner, and in Ceph, it's managed by a cluster map (a CRUSH map). We can easily scale Ceph out by adding more nodes, which will redistribute data in a minimum fashion to ensure as small amount of replication as possible.

An internal Ceph component called **Reliable Autonomic Distributed Object Store (RADOS)** is used for snapshots, replication, and thin provisioning. It's an open source project that was developed by the University of California.

Architecture-wise, Ceph has three main services:

- **ceph-mon** : Used for cluster monitoring, CRUSH maps, and **Object Storage Daemon (OSD)** maps.
- **ceph-osd**: This handles actual data storage, replication, and recovery. It requires at least two nodes; we'll use three for clustering reasons.
- **ceph-mds**: Metadata server, used when Ceph needs filesystem access.

In accordance with best practices, make sure that you always design your Ceph environments with the key principles in mind—all of the data nodes need to have the same configuration. That means the same amount of memory, the same storage controllers (don't use RAID controllers, just plain HBAs without RAID firmware if possible), the same disks, and so on. That's the only way to ensure a constant level of Ceph performance in your environments.

One very important aspect of Ceph is data placement and how placement groups work. Placement groups offer us a chance to split the objects that we create and place them in OSDs in an optimal fashion. Translation: the bigger the number of placement groups we configure, the better balance we're going to get.

So, let's configure Ceph from scratch. We're going to follow the best practices again and deploy Ceph by using five servers—one for administration, one for monitoring, and three OSDs.

Our configuration will look like this:

hostname	ceph-admin	ceph-monitor	ceph-osd1	ceph-osd2	ceph-osd3
IP/24	192.168.159.150	192.168.159.151	192.168.159.152	192.168.159.153	192.168.159.154
Ceph disk	/dev/sdb	/dev/sdb	/dev/sdb	/dev/sdb	/dev/sdb

Figure 5.24 – Basic Ceph configuration for our infrastructure

Make sure that these hosts can resolve each other via DNS or `/etc/hosts`, and that you configure all of them to use the same NTP source. Make sure that you update all of the hosts by using the following:

```
yum -y update; reboot
```

Also, make sure that you type the following commands into all of the hosts as the *root* user. Let's start by deploying packages, creating an admin user, and giving them rights to `sudo`:

```
rpm -Uvh http://download.ceph.com/rpm-jewel/el7/noarch/ceph-release-1-1.el7.noarch.rpm
```

```
yum -y install ceph-deploy ceph ceph-radosgw
```

```
useradd cephadmin
```

```
echo "cephadmin:ceph123" | chpasswd
```

```
echo "cephadmin ALL = (root) NOPASSWD:ALL" | sudo tee /etc/sudoers.d/cephadmin
```

```
chmod 0440 /etc/sudoers.d/cephadmin
```

Disabling SELinux will make our life easier for this demonstration, as will getting rid of the firewall:

```
sed -i 's/SELINUX=enforcing/SELINUX=disabled/g' /etc/selinux/config
```

```
systemctl stop firewalld
```

```
systemctl disable firewalld
systemctl mask firewalld
```

Let's add hostnames to `/etc/hosts` so that administration is easier for us:

```
echo "192.168.159.150 ceph-admin" >> /etc/hosts
echo "192.168.159.151 ceph-monitor" >> /etc/hosts
echo "192.168.159.152 ceph-osd1" >> /etc/hosts
echo "192.168.159.153 ceph-osd2" >> /etc/hosts
echo "192.168.159.154 ceph-osd3" >> /etc/hosts
```

Change the last `echo` part to suit your environment—hostnames and IP addresses. We're just using this as an example from our environment. The next step is making sure that we can use our admin host to connect to all of the hosts. The easiest way to do that is by using SSH keys. So, on `ceph-admin`, log in as `root` and type in the `ssh-keygen` command, and then press the *Enter* key all the way through. It should look something like this:

```
root@gluster1:~
File Edit View Search Terminal Help
[root@ceph-admin ~]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:pLaJdR+PXYxFHSAr4P1mtSUZjB0uy0cMMcDDkFne/Ro root@ceph-admin
The key's randomart image is:
+---[RSA 2048]-----+
|. *+ ..+ 0+000.|
|0..+ 0 = .0000 .|
|. . 0.+ 0.+..|
| 0. * .++|
|+ESo.*..0|
| + ++.== .|
|. 0. 00.0|
|. |
+-----[SHA256]-----+
```

Figure 5.25 – Generating an SSH key for root for Ceph setup purposes

We also need to copy this key to all of the hosts. So, again, on `ceph-admin`, use `ssh-copy-id` to copy the keys to all of the hosts:

```
ssh-copy-id cephadmin@ceph-admin
ssh-copy-id cephadmin@ceph-monitor
ssh-copy-id cephadmin@ceph-osd1
ssh-copy-id cephadmin@ceph-osd2
ssh-copy-id cephadmin@ceph-osd3
```

Accept all of the keys when SSH asks you, and use `ceph123` as the password, which we selected in one of the earlier steps. After all of this is done, there's one last step that we need to do on `ceph-admin` before we start deploying Ceph—we have to configure SSH to use the `cephadmin` user as a default user to log in to all of the hosts. We will do this by going to the `.ssh` directory as root on `ceph-admin`, and creating a file called `config` with the following content:

```
Host ceph-admin
    Hostname ceph-admin
    User cephadmin

Host ceph-monitor
    Hostname ceph-monitor
    User cephadmin

Host ceph-osd1
    Hostname ceph-osd1
    User cephadmin

Host ceph-osd2
    Hostname ceph-osd2
    User cephadmin

Host ceph-osd3
    Hostname ceph-osd3
    User cephadmin
```

That was a long pre-configuration, wasn't it? Now it's time to actually start deploying Ceph. The first step is to configure `ceph-monitor`. So, on `ceph-admin`, type in the following commands:

```
cd /root
mkdir cluster
cd cluster
ceph-deploy new ceph-monitor
```

Because of the fact that we selected a configuration in which we have three OSDs, we need to configure Ceph so that it uses these additional two hosts. So, in the `cluster` directory, edit the file called `ceph.conf` and add the following two lines at the end:

```
public network = 192.168.159.0/24
osd pool default size = 2
```

This will make sure that we can only use our example network (`192.168.159.0/24`) for Ceph, and that we have two additional OSDs on top of the original one.

Now that everything's ready, we have to issue a sequence of commands to configure Ceph. So, again, on `ceph-admin`, type in the following commands:

```
ceph-deploy install ceph-admin ceph-monitor ceph-osd1 ceph-osd2
ceph-osd3
ceph-deploy mon create-initial
ceph-deploy gatherkeys ceph-monitor
ceph-deploy disk list ceph-osd1 ceph-osd2 ceph-osd3
ceph-deploy disk zap ceph-osd1:/dev/sdb ceph-osd2:/dev/sdb
ceph-osd3:/dev/sdb
ceph-deploy osd prepare ceph-osd1:/dev/sdb ceph-osd2:/dev/sdb
ceph-osd3:/dev/sdb
ceph-deploy osd activate ceph-osd1:/dev/sdb1 ceph-osd2:/dev/
sdb1 ceph-osd3:/dev/sdb1
```

Let's describe these commands one by one:

- The first command starts the actual deployment process—for the admin, monitor, and OSD nodes, with the installation of all the necessary packages.
- The second and third commands configure the monitor host so that it's ready to accept external connections.

- The two disk commands are all about disk preparation—Ceph will clear the disks that we assigned to it (`/dev/sdb` per OSD host) and create two partitions on them, one for Ceph data and one for the Ceph journal.
- The last two commands prepare these filesystems for use and activate Ceph. If at any time your `ceph-deploy` script stops, check your DNS and `/etc/hosts` and `firewalld` configuration, as that's where the problems usually are.

We need to expose Ceph to our KVM host, which means that we have to do a bit of extra configuration. We're going to expose Ceph as an object pool to our KVM host, so we need to create a pool. Let's call it `KVMpool`. Connect to `ceph-admin`, and issue the following commands:

```
ceph osd pool create KVMpool 128 128
```

This command will create a pool called `KVMpool`, with 128 placement groups.

The next step involves approaching Ceph from a security perspective. We don't want anyone connecting to this pool, so we're going to create a key for authentication to Ceph, which we're going to use on the KVM host for authentication purposes. We do that by typing the following command:

```
ceph auth get-or-create client.KVMpool mon 'allow r' osd 'allow rwx pool=KVMpool'
```

It's going to throw us a status message, something like this:

```
key = AQB9p8RdqS09CBAA1DHsiZJbehb7ZBffhfmFJQ==
```

We can then switch to the KVM host, where we need to do two things:

- Define a secret—an object that's going to link libvirt to a Ceph user—and by doing that, we're going to create a secret object with its **Universally Unique Identifier (UUID)**.
- Use that secret's UUID to link it to the Ceph key when we define the Ceph storage pool.

The easiest way to do these two steps would be by using two XML configuration files for libvirt. So, let's create those two files. Let's call the first one, `secret.xml`, and here are its contents:

```
<secret ephemeral='no' private='no'>  
<usage type='ceph'>
```

```
<name>client.KVMpool secret</name>
</usage>
</secret>
```

Make sure that you save and import this XML file by typing in the following command:

```
virsh secret-define --file secret.xml
```

After you press the *Enter* key, this command is going to throw out a UUID. Please copy and paste that UUID someplace safe, as we're going to need it for the pool XML file. In our environment, this first `virsh` command threw out the following output:

```
Secret 95b1ed29-16aa-4e95-9917-c2cd4f3b2791 created
```

We need to assign a value to this secret so that when `libvirt` tries to use this secret, it knows which *password* to use. That's actually the password that we created on the Ceph level, when we used `ceph auth get-create`, which threw us the key. So, now that we have both the secret UUID and the Ceph key, we can combine them to create a complete authentication object. On the KVM host, we need to type in the following command:

```
virsh secret-set-value 95b1ed29-16aa-4e95-9917-c2cd4f3b2791
AQB9p8RdqS09CBAA1DHsiZJbehb7ZBfhfmFJQ==
```

Now, we can create the Ceph pool file. Let's call the config file `ceph.xml`, and here are its contents:

```
<pool type="rbd">
  <source>
    <name>KVMpool</name>
    <host name='192.168.159.151' port='6789' />
    <auth username='KVMpool' type='ceph'>
      <secret uuid='95b1ed29-16aa-4e95-9917-c2cd4f3b2791' />
    </auth>
  </source>
</pool>
```


So, the UUID from the previous step was used in this file to reference which secret (identity) is going to be used for Ceph pool access. Now we need to do the standard procedure—import the pool, start it, and autostart it—if we want to use it permanently (after the KVM host reboot). So, let's do that with the following sequence of commands on the KVM host:

```
virsh pool-define --file ceph.xml
virsh pool-start KVMpool
virsh pool-autostart KVMpool
virsh pool-list --details
```

The last command should produce an output similar to this:

```
[root@PacktStratis01 ~]# virsh pool-list --details
Name          State    Autostart Persistent Capacity Allocation Available
-----
default       running yes         yes         12.49 GiB  4.26 GiB  8.22 GiB
glusterfs-pool running yes         yes         9.99 GiB  134.61 MiB 9.86 GiB
KVMpool       running yes         yes         14.97 GiB  0.00 B    14.65 GiB
MyiSCSIpool   running yes         yes         119.53 GiB 119.53 GiB 0.00 B
MyNFSpool     running yes         yes         12.49 GiB  4.13 GiB  8.35 GiB
```

Figure 5.26 – Checking the state of our pools; the Ceph pool is configured and ready to be used

Now that the Ceph object pool is available for our KVM host, we could install a virtual machine on it. We're going to work on that – again – in *Chapter 7, Virtual Machine – Installation, Configuration, and Life-Cycle Management*.

Virtual disk images and formats and basic KVM storage operations

Disk images are standard files stored on the host's filesystem. They are large and act as virtualized hard drives for guests. You can create such files using the `dd` command, as shown:

```
# dd if=/dev/zero of=/vms/dbvm_disk2.img bs=1G count=10
```

Here is the translation of this command for you:

Duplicate data (`dd`) from the input file (`if`) of `/dev/zero` (virtually limitless supply of zeros) into the output file (`of`) of `/vms/dbvm_disk2.img` (disk image) using blocks of 1 G size (`bs` = block size) and repeat this (`count`) just once (10).

Important note:

`dd` is known to be a resource-hungry command. It may cause I/O problems on the host system, so it's good to first check the available free memory and I/O state of the host system, and only then run it. If the system is already loaded, lower the block size to MB and increase the count to match the size of the file you wanted (use `bs=1M, count=10000` instead of `bs=1G, count=10`).

`/vms/dbvm_disk2.img` is the result of the preceding command. The image now has 10 GB preallocated and ready to use with guests either as the boot disk or second disk. Similarly, you can also create thin-provisioned disk images. Preallocated and thin-provisioned (sparse) are disk allocation methods, or you may also call it the format:

- **Preallocated:** A preallocated virtual disk allocates the space right away at the time of creation. This usually means faster write speeds than a thin-provisioned virtual disk.
- **Thin-provisioned:** In this method, space will be allocated for the volume as needed—for example, if you create a 10 GB virtual disk (disk image) with sparse allocation. Initially, it would just take a couple of MB of space from your storage and grow as it receives write from the virtual machine up to 10 GB size. This allows storage over-commitment, which means *faking* the available capacity from a storage perspective. Furthermore, this can lead to problems later, when storage space gets filled. To create a thin-provisioned disk, use the `seek` option with the `dd` command, as shown in the following command:

```
dd if=/dev/zero of=/vms/dbvm_disk2_seek.img bs=1G seek=10  
count=0
```

Each comes with its own advantages and disadvantages. If you are looking for I/O performance, go for a preallocated format, but if you have a non-IO-intensive load, choose thin-provisioned.

Now, you might be wondering how you can identify what disk allocation method a certain virtual disk uses. There is a good utility for finding this out: `qemu-img`. This command allows you to read the metadata of a virtual image. It also supports creating a new disk and performing low-level format conversion.

Getting image information

The `info` parameter of the `qemu-img` command displays information about a disk image, including the absolute path of the image, the file format, and the virtual and disk size. By looking at the virtual disk size from a QEMU perspective and comparing that to the image file size on the disk, you can easily identify what disk allocation policy is in use. As an example, let's look at two of the disk images we created:

```
# qemu-img info /vms/dbvm_disk2.img
image: /vms/dbvm_disk2.img
file format: raw
virtual size: 10G (10737418240 bytes)
disk size: 10G
# qemu-img info /vms/dbvm_disk2_seek.img
image: /vms/dbvm_disk2_seek.img
file format: raw
virtual size: 10G (10737418240 bytes)
disk size: 10M
```

See the `disk size` line of both the disks. It's showing 10G for `/vms/dbvm_disk2.img`, whereas for `/vms/dbvm_disk2_seek.img`, it's showing 10M MiB. This difference is because the second disk uses a thin-provisioning format. The virtual size is what guests see and the disk size is what space the disk reserved on the host. If both the sizes are the same, it means the disk is preallocated. A difference means that the disk uses the thin-provisioning format. Now, let's attach the disk image to a virtual machine; you can attach it using `virt-manager` or the CLI alternative, `virsh`.

Attaching a disk using virt-manager

Start `virt-manager` from the host system's graphical desktop environment. It can also be started remotely using SSH, as demonstrated in the following command:

```
ssh -X host's address
[remotehost]# virt-manager
```

So, let's use the Virtual Machine Manager to attach the disk to the virtual machine:

1. In the Virtual Machine Manager main window, select the virtual machine to which you want to add the secondary disk.
2. Go to the virtual hardware details window and click on the **Add Hardware** button located at the bottom-left side of the dialog box.

3. In **Add New Virtual Hardware**, select **Storage** and select the **Create a disk image for the virtual machine** button and virtual disk size, as in the following screenshot:

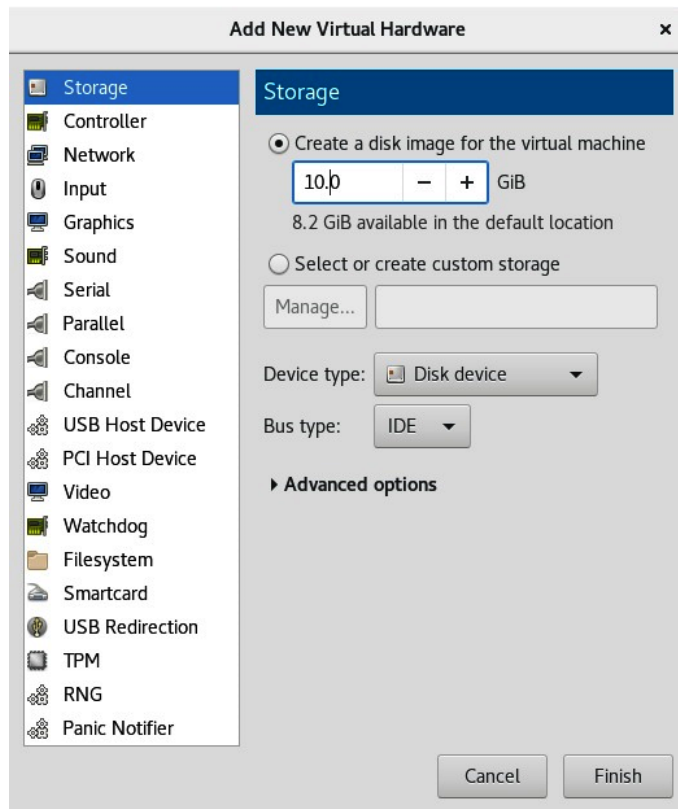


Figure 5.27 – Adding a virtual disk in virt-manager

4. If you want to attach the previously created `dbvm_disk2.img` image, choose **Select** or create custom storage, click on **Manage**, and either browse and point to the `dbvm_disk2.img` file from the `/vms` directory or find it in the local storage pool, then select it and click **Finish**.

Important note:

Here, we used a disk image, but you are free to use any storage device that is present on the host system, such as a LUN, an entire physical disk (`/dev/sdb`) or disk partition (`/dev/sdb1`), or LVM logical volume. We could have used any of the previously configured storage pools for storing this image either as a file or object or directly to a block device.

5. Clicking on the **Finish** button will attach the selected disk image (file) as a second disk to the virtual machine using the default configuration. The same operation can be quickly performed using the `virsh` command.

Using `virt-manager` to create a virtual disk was easy enough—just a couple of clicks of a mouse and a bit of typing. Now, let's see how we can do that via the command line—namely, by using `virsh`.

Attaching a disk using `virsh`

`virsh` is a very powerful command-line alternative to `virt-manager`. You can perform an action in a second that would take minutes to perform through a graphical interface such as `virt-manager`. It provides an `attach-disk` option to attach a new disk device to a virtual machine. There are lots of switches provided with `attach-disk`:

```
attach-disk domain source target [[[--live] [--config] |
[--current]] | [--persistent]] [--targetbusbus] [--driver
driver] [--subdriversubdriver] [--iothreadiothread] [--cache
cache] [--type type] [--mode mode] [--sourcetypesourcetype]
[--serial serial] [--wwnwwn] [--rawio] [--address address]
[--multifunction] [--print-xml]
```

However, in a normal scenario, the following are sufficient to perform hot-add disk attachment to a virtual machine:

```
# virsh attach-disk CentOS8 /vms/dbvm_disk2.img vdb --live
--config
```

Here, `CentOS8` is the virtual machine to which a disk attachment is executed. Then, there is the path of the disk image. `vdb` is the target disk name that would be visible inside the guest operating system. `--live` means performing the action while the virtual machine is running, and `--config` means attaching it persistently across reboot. Not adding a `--config` switch will keep the disk attached only until reboot.

Important note:

Hot plugging support: The `acpiphp` kernel module should be loaded in a Linux guest operating system in order to recognize a hot-added disk; `acpiphp` provides legacy hot plugging support, whereas `pciehp` provides native hot plugging support. `pciehp` is dependent on `acpiphp`. Loading `acpiphp` will automatically load `pciehp` as a dependency.

You can use the `virsh domblklist <vm_name>` command to quickly identify how many vDisks are attached to a virtual machine. Here is an example:

```
# virsh domblklist CentOS8 --details
Type Device Target Source
-----
file disk vda /var/lib/libvirt/images/fedora21.qcow2
file disk vdb /vms/dbvm_disk2_seek.img
```

This clearly indicates that the two vDisks connected to the virtual machine are both file images. They are visible to the guest operating system as `vda` and `vdb`, respectively, and in the last column of the disk images path on the host system.

Next, we are going to see how to create an ISO library.

Creating an ISO image library

Although a guest operating system on the virtual machine can be installed from physical media by carrying out a passthrough the host's CD/DVD drive to the virtual machine, it's not the most efficient way. Reading from a DVD drive is slow compared to reading ISO from a hard disk, so a better way is to store ISO files (or logical CDs) used to install operating systems and applications for the virtual machines in a file-based storage pool and create an ISO image library.

To create an ISO image library, you can either use `virt-manager` or a `virsh` command. Let's see how to create an ISO image library using the `virsh` command:

1. First, create a directory on the host system to store the `.iso` images:

```
# mkdir /iso
```

2. Set the correct permissions. It should be owned by a root user with permission set to 700. If SELinux is in enforcing mode, the following context needs to be set:

```
# chmod 700 /iso
# semanage fcontext -a -t virt_image_t "/iso(/.*)?"
```

3. Define the ISO image library using the `virsh` command, as shown in the following code block:

```
# virsh pool-define-as iso_library dir - - - - "/iso"
# virsh pool-build iso_library
# virsh pool-start iso_library
```

In the preceding example, we used the name `iso_library` to demonstrate how to create a storage pool that will hold ISO images, but you are free to use any name you wish.

4. Verify that the pool (ISO image library) was created:

```
# virsh pool-info iso_library
Name: iso_library
UUID: 959309c8-846d-41dd-80db-7a6e204f320e
State: running
Persistent: yes
Autostart: no
Capacity: 49.09 GiB
Allocation: 8.45 GiB
Available: 40.64 GiB
```

5. Now you can copy or move the `.iso` images to the `/iso_lib` directory.
6. Upon copying the `.iso` files into the `/iso_lib` directory, refresh the pool and then check its contents:

```
# virsh pool-refresh iso_library
Pool iso_library refreshed
# virsh vol-list iso_library
Name Path
-----
-----
-----
CentOS8-Everything.iso /iso/CentOS8-Everything.iso
CentOS7-Everything.iso /iso/CentOS7-Everything.iso
RHEL8.iso /iso/RHEL8.iso
Win8.iso /iso/Win8.iso
```

7. This will list all the ISO images stored in the directory, along with their path. These ISO images can now be used directly with a virtual machine for guest operating system installation, software installation, or upgrades.

Creating an ISO image library is the de facto norm in today's enterprises. It's better to have a centralized place where all your ISO images are, and it makes it easier to implement some kind of synchronization method (for example, `rsync`) if you need to synchronize across different locations.

Deleting a storage pool

Deleting a storage pool is fairly easy. Please note that deleting a storage domain will not remove any file/block devices. It just disconnects the storage from virt-manager. The file/block device has to be removed manually.

We can delete a storage pool via virt-manager or by using the `virsh` command. Let's first check how to do it via virt-manager:

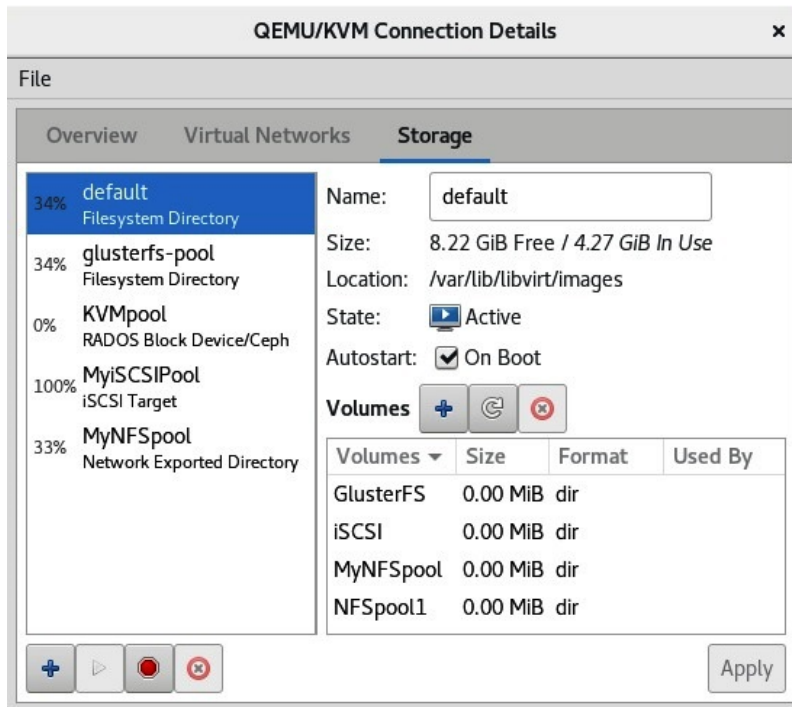


Figure 5.28 – Deleting a pool

First, select the red stop button to stop the pool, and then click on the red circle with an X to delete the pool.

If you want to use `virsh`, it's even simpler. Let's say that we want to delete the storage pool called `MyNFSPool` in the previous screenshot. Just type in the following commands:

```
virsh pool-destroy MyNFSPool
virsh pool-undefine MyNFSPool
```

The next logical step after creating a storage pool is to create a storage volume. From a logical standpoint, the storage volume slices a storage pool into smaller parts. Let's learn how to do that now.

Creating storage volumes

Storage volumes are created on top of storage pools and attached as virtual disks to virtual machines. In order to create a storage volume, start the Storage Management console, navigate to virt-manager, then click **Edit | Connection Details | Storage** and select the storage pool where you want to create a new volume. Click on the create new volume button (+):

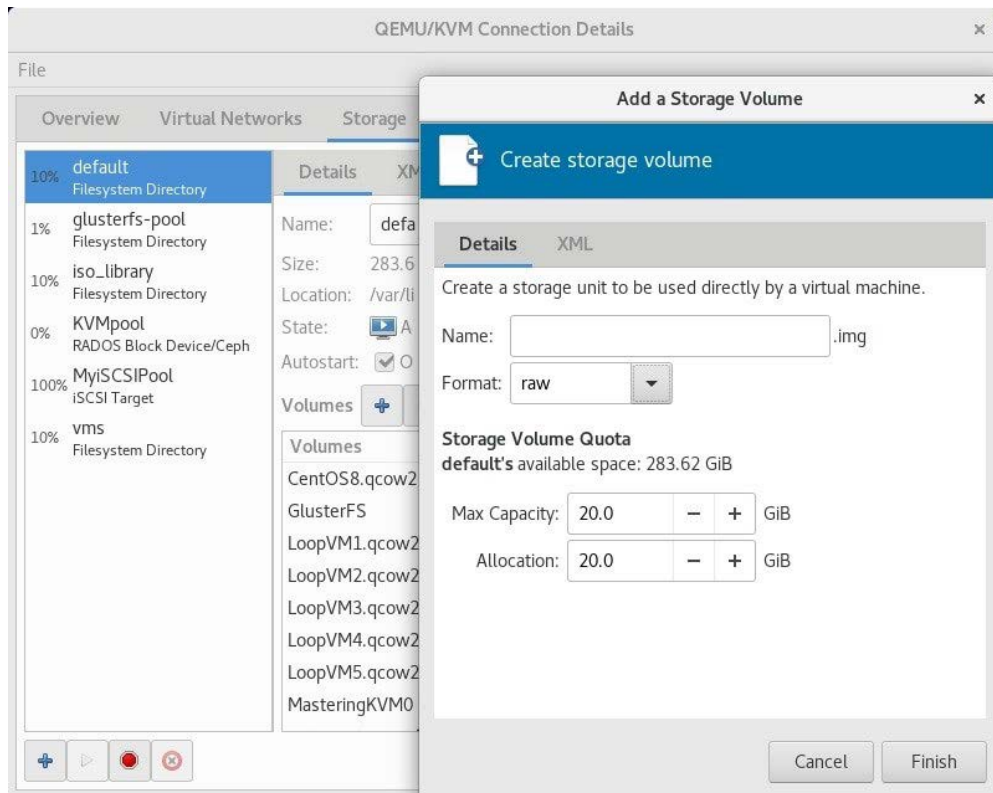


Figure 5.29 – Creating a storage volume for the virtual machine

Next, provide the name of the new volume, choose the disk allocation format for it, and click on the **Finish** button to build the volume and get it ready to attach to a virtual machine. You can attach it using the usual virt-manager or the `virsh` command. There are several disk formats that are supported by libvirt (`raw`, `cow`, `qcow`, `qcow2`, `qed`, and `vmrk`). Use the disk format that suits your environment and set the proper size in the **Max Capacity** and **Allocation** fields to decide whether you wish to go with preallocated disk allocation or thin-provisioned. If you keep the disk size the same in **Max Capacity** and **Allocation**, it will be preallocated rather than thin-provisioned. Note that the `qcow2` format does not support the thick disk allocation method.

In *Chapter 8, Creating and Modifying VM Disks, Templates, and Snapshots*, all the disk formats are explained in detail. For now, just understand that `qcow2` is a specially designed disk format for KVM virtualization. It supports the advanced features needed for creating internal snapshots.

Creating volumes using the `virsh` command

The syntax to create a volume using the `virsh` command is as follows:

```
# virsh vol-create-as dedicated_storage vm_vol1 10G
```

Here, `dedicated_storage` is the storage pool, `vm_vol1` is the volume name, and 10 GB is the size:

```
# virsh vol-info --pool dedicated_storage vm_vol1
Name: vm_vol1
Type: file
Capacity: 1.00 GiB
Allocation: 1.00 GiB
```

The `virsh` command and arguments to create a storage volume are almost the same regardless of the type of storage pool it is created on. Just enter the appropriate input for a `--pool` switch. Now, let's see how to delete a volume using the `virsh` command.

Deleting a volume using the `virsh` command

The syntax to delete a volume using the `virsh` command is as follows:

```
# virsh vol-delete dedicated_storage vm_vol2
```

Executing this command will remove the `vm_vol2` volume from the `dedicated_storage` storage pool.

The next step in our storage journey is about looking a bit into the future as all of the concepts that we mentioned in this chapter have been well known for years, some even for decades. The world of storage is changing and moving into new and interesting directions, so let's discuss that for a bit next.

The latest developments in storage – NVMe and NVMeOF

In the past 20 or so years, by far the biggest disruption in the storage world in terms of technology has been the introduction of **Solid State Drives (SSDs)**. Now, we know that a lot of people have gotten quite used to having them in their computers—laptops, workstations, whichever type of device we use. But again, we're discussing storage for virtualization, and enterprise storage concepts overall, and that means that our regular SATA SSDs aren't going to make the cut. Although a lot of people use them in mid-range storage devices and/or handmade storage devices that host ZFS pools (for cache), some of these concepts have a life of their own in the latest generations of storage devices. These devices are fundamentally changing the way technology is working and redoing parts of modern IT history in terms of which protocols are used, how fast they are, how much lower latencies they have, and how they approach storage tiering—tiering being a concept that differentiates different storage devices or their storage pools based on a capability, usually speed.

Let's briefly explain what we're discussing here by using an example of where the storage world is heading. Along with that, the storage world is taking the virtualization, cloud, and HPC world along for the ride, so these concepts are not outlandish. They already exist, in readily available storage devices that you can buy today.

The introduction of SSDs brought a significant change in the way we access our storage devices. It's all about performance and latency, and older concepts such as **Advanced Host Controller Interface (AHCI)**, which we're still actively using with many SSDs on the market today, are just not good enough to handle the performance that SSDs have. AHCI is a standard way in which a regular hard disk (mechanical disk or regular spindle) talks via software to SATA devices. However, the key part of that is *hard disk*, which means cylinders, heads sectors—things that SSDs just don't have, as they don't spin around and don't need that kind of paradigm. That meant that another standard had to be created so that we can use SSDs in a more native fashion. That's what **Non-Volatile Memory Express (NVMe)** is all about—bridging the gap between what SSDs are capable of doing and what they can actually do, without using translations from SATA to AHCI to PCI Express (and so on).

The fast development pace of SSDs and the integration of NVMe made huge advancements in enterprise storage possible. That means that new controllers, new software, and completely new architectures had to be invented to support this paradigm shift. As more and more storage devices integrate NVMe for various purposes—primarily for caching, then for storage capacity as well—it's becoming clear that there are other problems that need to be solved as well. The first of which is the way in which we're going to connect storage devices offering such a tremendous amount of capability to our virtualized, cloud, or HPC environments.

In the past 10 or so years, many people argued that FC is going to disappear from the market, and a lot of companies hedged their bets on different standards—iSCSI, iSCSI over RDMA, NFS over RDMA, and so on. The reasoning behind that seemed solid enough:

- FC is expensive—it requires separate physical switches, separate cabling, and separate controllers, all of which cost a lot of money.
- There's licensing involved—when you buy, for example, a Brocade switch that has 40 FC ports, that doesn't mean that you can use all of them out of the box, as there are licenses to get more ports (8-port, 16-port, and so on).
- FC storage devices are expensive and often require more expensive disks (with FC connectors).
- Configuring FC requires extensive knowledge and/or training, as you can't simply go and configure a stack of FC switches for an enterprise-level company without knowing the concepts, and the CLI from the switch vendor, on top of knowing what that enterprise's needs are.
- The ability of FC as a protocol to speed up its development to reach new speeds has been really bad. In simple terms, during the time it took FC to go from 8 Gbit/s to 32 Gbit/s, Ethernet went from 1 Gbit/s to 25, 40, 50, and 100 Gbit/s bandwidth. There's already talk about 400 Gbit/s Ethernet, and there are the first devices that support that standard as well. That usually makes customers concerned as higher numbers mean better throughput, at least in most people's minds.

But what's happening on the market *now* tells us a completely different story—not just that FC is back, but that it's back with a mission. The enterprise storage companies have embraced that and started introducing storage devices with *insane* levels of performance (with the aid of NVMe SSDs, as a first phase). That performance needs to be transferred to our virtualized, cloud, and HPC environments, and that requires the best possible protocol, in terms of lowest latency, its design, and the quality and reliability, and FC has all of that.

That leads to the second phase, where NVMe SSDs aren't just being used as cache devices, but as capacity devices as well.

Take note of the fact that, right now, there's a big fight brewing on the storage memory/storage interconnects market. There are multiple different standards trying to compete with Intel's **Quick Path Interconnect (QPI)**, a technology that's been used in Intel CPUs for more than a decade. If this is a subject that's interesting to you, there is a link at the end of this chapter, in the *Further reading* section, where you can find more information. Essentially, QPI is a point-to-point interconnection technology with low latency and high bandwidth that's at the core of today's servers. Specifically, it handles communication between CPUs, CPUs and memory, CPUs and chipsets, and so on. It's a technology that Intel developed after it got rid of the **Front Side Bus (FSB)** and chipset-integrated memory controllers. FSB was a shared bus that was shared between memory and I/O requests. That approach had much higher latency, didn't scale well, and had lower bandwidth and problems with situations in which there's a large amount of I/O happening on the memory and I/O side. After switching to an architecture where the memory controller was a part of the CPU (therefore, memory directly connects to it), it was essential for Intel to finally move to this kind of concept.

If you're more familiar with AMD CPUs, QPI is to Intel what HyperTransport bus on a CPU with built-in memory controller is to AMD CPUs.

As NVMe SSDs became faster, the PCI Express standard also needed to be updated, which is the reason why the latest version (PCIe 4.0 – the first products started shipping recently) was so eagerly anticipated. But now, the focus has switched to two other problems that need resolving for storage systems to work. Let's describe them briefly:

- Problem number one is simple. For a regular computer user, one or two NVMe SSDs will be enough in 99% of scenarios or more. Realistically, the only real reason why regular computer users need a faster PCIe bus is for a faster graphics cards. But for storage manufacturers, it's completely different. They want to produce enterprise storage devices that will have 20, 30, 50, 100, 500 NVMe SSDs in a single storage system—and they want that now, as SSDs are mature as a technology and are widely available.
- Problem number two is more complex. To add insult to injury, the latest generation of SSDs (for example, based on Intel Optane) can offer even lower latency and higher throughput. That's only going to get *worse* (even lower latencies, higher throughput) as technology evolves. For today's services—virtualization, cloud, and HPC—it's essential that the storage system is able to handle any load that we can throw at it. These technologies are a real game-changer in terms of how much faster storage devices can become, only if interconnects can handle it (QPI, FC, and many more). Two of these concepts derived from Intel Optane—**Storage Class Memory (SCM)** and **Persistent Memory (PM)** are the latest technologies that storage companies and customers want adopted into their storage systems, and fast.

- The third problem is how to transfer all of that bandwidth and I/O capability to the servers and infrastructures using them. This is why the concept of **NVMe over Fabrics (NVMe-OF)** was created, to try to work on the storage infrastructure stack to make NVMe much more efficient and faster for its consumers.

If you take a look at these advancements from a conceptual point of view, it was clear for decades that RAM-like memory is the fastest, lowest latency technology that we've had for the past couple of decades. It's also logical that we're moving workloads to RAM, as much as possible. Think of in-memory databases (such as Microsoft SQL, SAP Hana, and Oracle). They've been around the block for years.

These technologies fundamentally change the way we think about storage. Basically, no longer are we discussing storage tiering based on technology (SSD versus SAS versus SATA), or outright speed, as the speed is unquestionable. The latest storage technologies discuss storage tiering in terms of *latency*. The reason is very simple—let's say that you're a storage company and that you build a storage system that uses 50 SCM SSDs for capacity. For cache, the only reasonable technology would be RAM, hundreds of gigabytes of it. The only way you'd be able to work with storage tiering on a device like that is by basically *emulating* it in software, by creating additional technologies that will produce tiering-like services based on queueing, handling priority in cache (RAM), and similar concepts. Why? Because if you're using the same SCM SSDs for capacity, and they offer the same speed and I/O, you just don't have a way of tiering based on technology or capability.

Let's further describe this by using an available storage system to explain. The best device to make our point is Dell/EMC's PowerMax series of storage devices. If you load them with NVMe and SCM SSDs, the biggest model (8000) can scale to 15 million IOPS(!), 350 GB/s throughput at lower than 100 microseconds latency and up to 4 PB capacity. Think about those numbers for a second. Then add another number—on the frontend, it can have up to 256 FC/FICON/iSCSI ports. Just recently, Dell/EMC released new 32 Gbit/s FC modules for it. The smaller PowerMax model (2000) can do 7.5 million IOPS, sub-100 microsecond latency, and scale to 1 PB. It can also do all of the *usual EMC stuff*—replication, compression, deduplication, snapshots, NAS features, and so on. So, this is not just marketing talk; these devices are already out there, being used by enterprise customers:



Figure 3.30 – PowerMax 2000 – it seems small, but it packs a lot of punch

These are very important concepts for the future, as more and more manufacturers produce similar devices (and they are on the way). We fully expect the KVM-based world to embrace these concepts in large-scale environments, especially for infrastructures with OpenStack and OpenShift.

Summary

In this chapter, we introduced and configured various Open Source storage concepts for libvirt. We also discussed industry-standard approaches, such as iSCSI and NFS, as they are often used in infrastructures that are not based on KVM. For example, VMware vSphere-based environments can use FC, iSCSI, and NFS, while Microsoft-based environments can only use FC and iSCSI, from the list of subjects we covered in this chapter.

The next chapter will cover subjects related to virtual display devices and protocols. We'll provide an in-depth introduction to VNC and SPICE protocols. We will also provide a description of other protocols that are used for virtual machine connection. All that will help us to understand the complete stack of fundamentals that we need to work with our virtual machines, which we covered in the past three chapters.

Questions

1. What is a storage pool?
2. How does NFS storage work with libvirt?
3. How does iSCSI work with libvirt?
4. How do we achieve redundancy on storage connections?
5. What can we use for virtual machine storage except NFS and iSCSI?
6. Which storage backend can we use for object-based storage with libvirt?
7. How can we create a virtual disk image to use with a KVM virtual machine?
8. How does using NVMe SSDs and SCM devices change the way that we create storage tiers?
9. What are the fundamental problems of delivering tier-0 storage services for virtualization, cloud, and HPC environments?

Further reading

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

- What's new with RHEL8 file systems and storage: <https://www.redhat.com/en/blog/whats-new-rhel-8-file-systems-and-storage>
- oVirt storage: https://www.ovirt.org/documentation/administration_guide/#chap-Storage
- RHEL 7 storage administration guide: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/index
- RHEL 8 managing storage devices: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_storage_devices/index
- OpenFabrics CCIX, Gen-Z, OpenCAPI (overview and comparison): https://www.openfabrics.org/images/eventpresos/2017presentations/213_CCIXGen-Z_BBenton.pdf

6

Virtual Display Devices and Protocols

In this chapter, we will discuss the way in which we access our virtual machines by using virtual graphic cards and protocols. There are almost 10 available virtual display adapters that we can use in our virtual machines, and there are multiple available protocols and applications that we can use to access our virtual machines. If we forget about SSH for a second and any kind of console-based access in general, there are various protocols available on the market that we can use to access the console of our virtual machine, such as VNC, SPICE, and noVNC.

In Microsoft-based environments, we tend to use a **remote desktop protocol (RDP)**. If we are talking about **Virtual Desktop Infrastructure (VDI)**, then there are even more protocols available – **PC over IP (PCoIP)**, VMware Blast, and so on. Some of these technologies offer additional functionality, such as greater color depth, encryption, audio and filesystem redirection, printer redirection, bandwidth management, and USB and other port redirection. These are key technologies for your remote desktop experience in today's cloud-based world.

All of this means that we must put a bit more time and effort into getting to know various display devices and protocols, as well as how to configure and use them. We don't want to end up in situations in which we can't see the display of a virtual machine because we selected the wrong virtual display device, or in a situation where we try to open a console to see the content of a virtual machine and the console doesn't open.

In this chapter, we will cover the following topics:

- Using virtual machine display devices
- Discussing remote display protocols
- Using the VNC display protocol
- Using the SPICE display protocol
- Getting display portability with NoVNC
- Let's get started!

Using virtual machine display devices

To make the graphics work on virtual machines, QEMU needs to provide two components to its virtual machines: a virtual graphic adapter and a method or protocol to access the graphics from the client. Let's discuss these two concepts, starting with a virtual graphic adapter. The latest version of QEMU has eight different types of virtual/emulated graphics adapters. All of these have some similarities and differences, all of which can be in terms of features and/or resolutions supported or other, more technical details. So, let's describe them and see which use cases we are going to favor a specific virtual graphic card for:

- **tcx**: A SUN TCX virtual graphics card that can be used with old SUN OSes.
- **cirrus**: A virtual graphic card that's based on an old Cirrus Logic GD5446 VGA chip. It can be used with any guest OS after Windows 95.
- **std**: A standard VGA card that can be used with high-resolution modes for guest OSes after Windows XP.
- **vmware**: VMware's SVGA graphics adapter, which requires additional drivers in Linux guest OSes and VMware Tools installation for Windows OSes.
- **QXL**: The de facto standard paravirtual graphics card that we need to use when we use SPICE remote display protocol, which we will cover in detail a bit later in this chapter. There's an older version of this virtual graphics card called QXL VGA, which lacks some more advanced features, while offering lower overhead (it uses less memory).

- **Virtio:** A paravirtual 3D virtual graphics card that is based on the virgl project, which provides 3D acceleration for QEMU guest OSes. It has two different types (VGA and gpu). virtio-vga is commonly used for situations where we need multi-monitor support and OpenGL hardware acceleration. The virtio-gpu version doesn't have a built-in standard VGA compatibility mode.
- **cg3:** A virtual graphics card that we can use with older SPARC-based guest OSes.
- **none:** Disables the graphics card in the guest OS.

When configuring your virtual machine, you can select these options at startup or virtual machine creation. In CentOS 8, the default virtual graphics card that gets assigned to a newly created virtual machine is **QXL**, as shown in the following screenshot of the configuration for a new virtual machine:

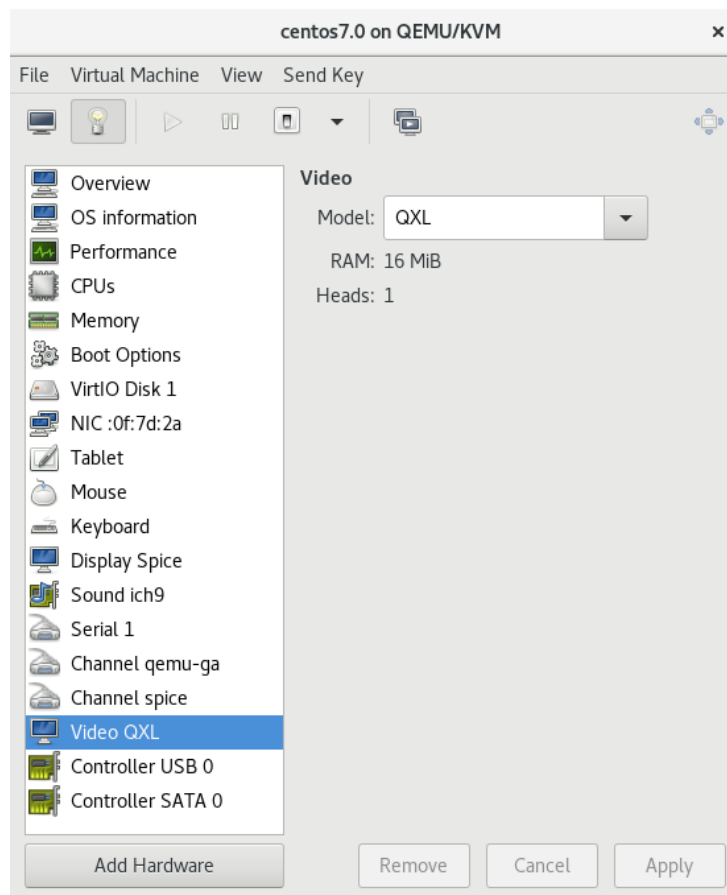


Figure 6.1 – Default virtual graphics card for a guest OS – QXL

Also, by default, we can select three of these types of virtual graphics cards for any given virtual machine, as these are usually pre-installed for us on any Linux server that's configured for virtualization:

- QXL
- VGA
- Virtio

Some of the new OSes running in KVM virtualization shouldn't use older graphics card adapters for a variety of reasons. For example, ever since Red Hat Enterprise Linux/CentOS 7, there's an advisory not to use the cirrus virtual graphics card for Windows 10 and Windows Server 2016. The reason for this is related to the instability of the virtual machine, as well as the fact that – for example – you can't use a full HD resolution display with the cirrus virtual graphics card. Just in case you start installing these guest OSes, make sure that you're using a QXL video graphics card as it offers the best performance and compatibility with the SPICE remote display protocol.

Theoretically, you could still use cirrus virtual graphics card for some of the *really* old guest OSes (older Windows NTs such as 4.0 and older client guest OSes such as Windows XP), but that's about it. For everything else, it's much better to either use a std or QXL driver as they offer the best performance and acceleration support. Furthermore, these virtual graphics cards also offer higher display resolutions.

There are some other virtual graphics cards available for QEMU, such as embedded drivers for various **System on a Chip (SoC)** devices, ati vga, bochs, and so on. Some of these are often used, such as SoCs – just remember all of the world's Raspberry Pis, and BBC Micro:bits. These new virtual graphics options are further expanded by **Internet of Things (IoT)**. So, there are loads of good reasons why we should pay close attention to what's happening in this market space.

Let's show this via an example. Let's say that we want to create a new virtual machine that is going to have a set of custom parameters assigned to it in terms of how we access its virtual display. If you remember in *Chapter 3, Installing KVM Hypervisor, libvirt, and ovirt*, we discussed various libvirt management commands (`virsh`, `virt-install`) and we also created some virtual machines by using `virt-install` and some custom parameters. Let's add to those and use a similar example:

```
virt-install --virt-type=kvm --name MasteringKVM01 --vcpus 2
--ram 4096 --os-variant=rhel8.0 --/iso/CentOS-8-x86_64-
1905-dvd1.iso --network=default --video=vga --graphics
vnc,password=Packt123 --disk size=16
```

Here's what's going to happen:

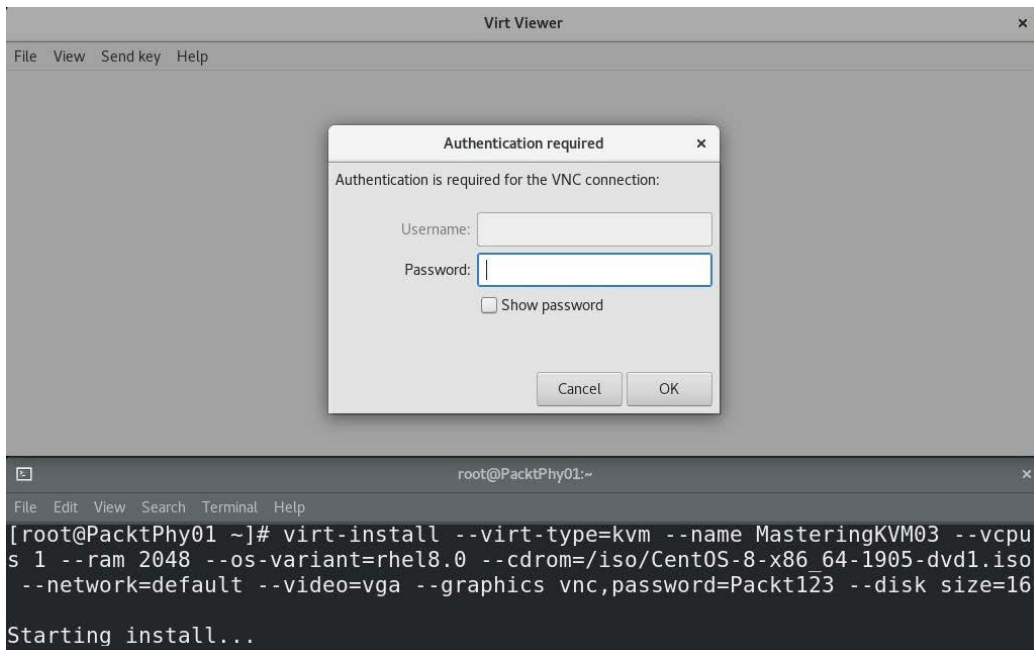


Figure 6.2 – KVM virtual machine with a VGA virtual graphics card is created.
Here, VNC is asking for a password to be specified

After we type in the password (PacKt123, as specified in the virt-install configuration option), we're faced with this screen:

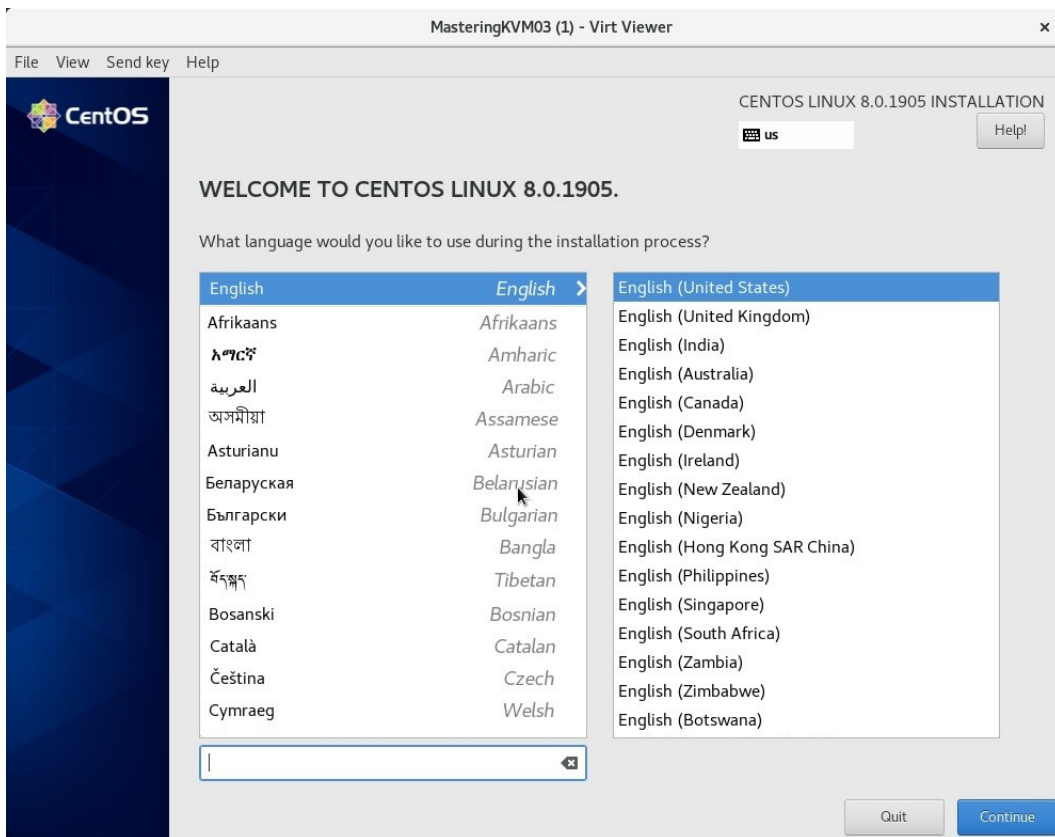


Figure 6.3 – VGA display adapter and its low default (640x480) initial resolution - a familiar resolution for all of us who grew up in the 80s

That being said, we just used this as an example of how to add an advanced option to the virt-install command – specifically, how to install a virtual machine with a specific virtual graphics card.

There are other, more advanced concepts of using real graphics cards that we installed in our computers or servers to forward their *capabilities* directly to virtual machines. This is *very* important for concepts such as VDI, as we mentioned earlier. Let's discuss these concepts for a second and use some real-world examples and comparisons to understand the complexity of VDI solutions on a larger scale.

Physical and virtual graphics cards in VDI scenarios

As we discussed in *Chapter 1, Understanding Linux Virtualization*, VDI is a concept that uses virtualization paradigm for client OSES. This means that end users connect *directly* to their virtual machines by running a client OS (for example, Windows 8.1, Windows 10, or Linux Mint) that is either *reserved* for them or *pooled*, which means that multiple users can access the same virtual machines and get access to their *data* via additional VDI capabilities.

Now, if we're talking about most business users, they just need something that we jokingly call a *typewriter*. This usage model relates to a scenario in which the user uses a client OS for reading and writing documents, email, and browsing the internet. And for these use cases, if we were to use any vendor-based solution out there (VMware's Horizon, Citrix Xen Desktop, or Microsoft's Remote Desktop Services-based VDI solutions), we could do so with any one of them.

However, there's a big *but*. What happens if the scenario includes hundreds of users who need access to 2D and/or 3D video acceleration? What happens if we are designing a VDI solution for a company creating designs – architecture, plumbing, oil and gas, and video production? Running VDI solutions based on CPU and software-based virtual graphics cards will get us nowhere, especially at scale. This is where Xen Desktop and Horizon will be much more feature-packed if we're talking about the technology level. And – to be quite honest – KVM-based methods aren't all that far behind in terms of display options, it's just that they lag in some other enterprise-class features, which we will discuss in later chapters, such as *Chapter 12, Scaling Out KVM with OpenStack*.

Basically, there are three concepts we can use to obtain graphics card performance for a virtual machine:

- We can use a software renderer that's CPU-based.
- We can reserve a GPU for a specific virtual machine (PCI passthrough).
- We can *partition* a GPU so that we can use it in multiple virtual machines.

Just to use the VMware Horizon solution as a metaphor, these solutions would be called CPU rendering, **Virtual Direct Graphics Acceleration (vDGA)**, and **Virtual Shared Graphics Acceleration (vSGA)**. Or, in Citrix, we'd be talking about HDX 3D Pro. In CentOS 8, we are talking about *mediated devices* in the shared graphics card scenario.

If we're talking about PCI passthrough, it definitely delivers the best performance as you can use a PCI-Express graphics card, forward it directly to a virtual machine, install a native driver inside the guest OS, and have the complete graphics card all for yourself. But that creates four problems:

- You can only have that PCI-Express graphics card forwarded to *one* virtual machine.
- As servers can be limited in terms of upgradeability, for example, you can't run 50 virtual machines like that on one physical server as you can't fit 50 graphics cards on a single server – physically or in terms of PCI-Express slots, where you usually have up to six in a typical 2U rack server.
- If you're using Blade servers (for example, HP c7000), it's going to be even worse as you're going to use half of the server density per blade chassis if you're going to use additional graphics cards as these cards can only be fitted to double-height blades.
- You're going to spend an awful lot of money scaling any kind of solution like this to hundreds of virtual desktops, or – even worse – thousands of virtual desktops.

If we're talking about a shared approach in which you partition a physical graphics card so that you can use it in multiple virtual machines, that's going to create another set of problems:

- You're much more limited in terms of which graphics card to use as there are maybe 20 graphics cards that support this usage model (some include NVIDIA GRID, Quadro, Tesla cards, and a couple of AMD and Intel cards).
- If you share the same graphics card with four, eight, 16, or 32 virtual machines, you have to be aware of the fact that you'll get less performance, as you're sharing the same GPU with multiple virtual machines.
- Compatibility with DirectX, OpenGL, CUDA, and video encoding offload won't be as good as you might expect, and you might be forced to use older versions of these standards.
- There might be additional licensing involved, depending on the vendor and solution.

The next topic on our list is how to use a GPU in a more advanced way – by using the GPU partitioning concept to provide parts of a GPU to multiple virtual machines. Let's explain how that works and gets configured by using an NVIDIA GPU as an example.

GPU partitioning using an NVIDIA vGPU as an example

Let's use an example to see how we can use the scenario in which we partition our GPU (NVIDIA vGPU) with our KVM-based virtual machine. This procedure is very similar to the SR-IOV procedure we discussed in *Chapter 4, Libvirt Networking*, where we used a supported Intel network card to present virtual functions to our CentOS host, which we then presented to our virtual machines by using them as uplinks for the KVM virtual bridge.

First, we need to check which kind of graphic cards we have, and it must be a supported one (in our case, we're using a Tesla P4). Let's use the `lshw` command to check our display devices, which should look similar to this:

```
# yum -y install lshw
# lshw -C display
*-display
    description: 3D controller
    product: GP104GL [Tesla P4]
    vendor: NVIDIA Corporation
    physical id: 0
    bus info: pci@0000:01:00.0
    version: a0
    width: 64 bits
    clock: 33MHz
    capabilities: pm msi pciexpress cap_list
    configuration: driver=vfio-pci latency=0
    resources: irq:15 memory:f6000000-f6ffffff
    memory:e0000000-efffffff memory:f0000000-f1ffffff
```

The output of this command tells us that we have a 3D-capable GPU – specifically, a NVIDIA GP104GL-based product. It tells us that this device is already using the `vfio-pci` driver. This driver is the native SR-IOV driver for **Virtualized Functions (VF)**. These functions are the core of SR-IOV functionality. We will describe this by using this SR-IOV-capable GPU.

The first thing that we need to do – which all of us NVIDIA GPU users have been doing for years – is to blacklist the nouveau driver, which gets in the way. And if we are going to use GPU partitioning on a permanent basis, we need to do this permanently so that it doesn't get loaded when our server starts. But be warned – this can lead to unexpected behavior at times, such as the server booting and not showing any output without any real reason. So, we need to create a configuration file for modprobe that will blacklist the nouveau driver. Let's create a file called `nouveauoff.conf` in the `/etc/modprobe.d` directory with the following content:

```
blacklist nouveau
options nouveau modeset 0
```

Then, we need to force our server to recreate the `initrd` image that gets loaded as our server starts and reboot the server to make that change is active. We are going to do that with the `dracut` command, followed by a regular `reboot` command:

```
# dracut --regenerate-all -force
# systemctl reboot
```

After the reboot, let's check if our `vfio` driver for the NVIDIA graphics card has loaded and, if it has, check the vGPU manager service:

```
# lsmod | grep nvidia | grep vfio
nvidia_vgpu_vfio 45011 0
nvidia 14248203 10 nvidia_vgpu_vfio
mdev 22078 2 vfio_mdev,nvidia_vgpu_vfio
vfio 34373 3 vfio_mdev,nvidia_vgpu_vfio,vfio_iommu_type1
# systemctl status nvidia-vgpu-mgr
nvidia-vgpu-mgr.service - NVIDIA vGPU Manager Daemon
   Loaded: loaded (/usr/lib/systemd/system/nvidia-vgpu-mgr.
   service; enabled; vendor preset: disabled)
   Active: active (running) since Thu 2019-12-12 20:17:36 CET;
   0h 3min ago
   Main PID: 1327 (nvidia-vgpu-mgr)
```

We need to create a UUID that we will use to present our virtual function to a KVM virtual machine. We will use the `uuidgen` command for that:

```
uuidgen
c7802054-3b97-4e18-86a7-3d68dff2594d
```

Now, let's use this UUID for the virtual machines that will share our GPU. For that, we need to create an XML template file that we will add to the existing XML files for our virtual machines in a copy-paste fashion. Let's call this `vsga.xml`:

```
<hostdev mode='subsystem' type='mdev' managed='no' model='vfio-
pci'>
  <source>
    <address uuid='c7802054-3b97-4e18-86a7-3d68dff2594d' />
  </source>
</hostdev>
```

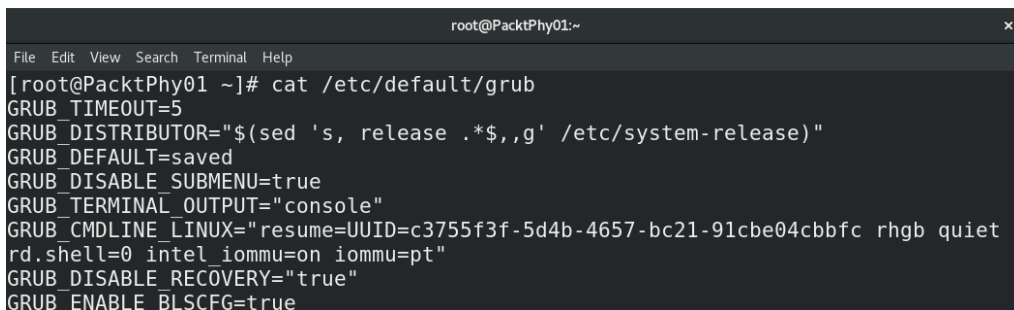
Use these settings as a template and just copy-paste the complete content to any virtual machine's XML file where you want to have access to our shared GPU.

The next concept that we need to discuss is the complete opposite of SR-IOV, where we're slicing a device into multiple pieces to present these pieces to virtual machines. In GPU passthrough, we're taking the *whole* device and presenting it directly to *one* object, meaning one virtual machine. Let's learn how to configure that.

GPU PCI passthrough

As with every advanced feature, enabling GPU PCI passthrough requires multiple steps to be done in sequence. By doing these steps in the correct order, we're directly presenting this hardware device to a virtual machine. Let's explain these configuration steps and do them:

1. To enable GPU PCI passthrough, we need to configure and enable IOMMU – first in our server's BIOS, then in our Linux distribution. We're using Intel-based servers, so we need to add `iommu` options to our `/etc/default/grub` file, as shown in the following screenshot:



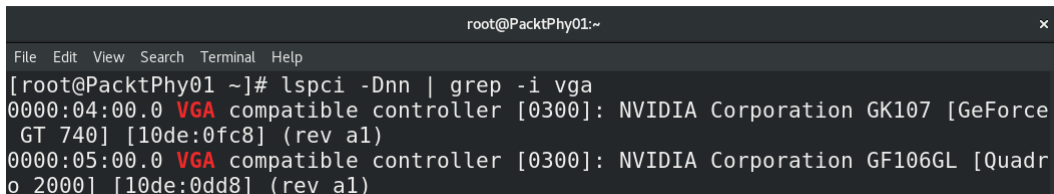
```
root@PacktPhy01:~
File Edit View Search Terminal Help
[root@PacktPhy01 ~]# cat /etc/default/grub
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$, ,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="resume=UUID=c3755f3f-5d4b-4657-bc21-91cbe04cbbfc rhgb quiet
rd.shell=0 intel_iommu=on iommu=pt"
GRUB_DISABLE_RECOVERY="true"
GRUB_ENABLE_BLS_CFG=true
```

Figure 6.4 – Adding `intel_iommu iommu=pt` options to a GRUB file

- The next step is to reconfigure the GRUB configuration and reboot it, which can be achieved by typing in the following commands:

```
# grub2-mkconfig -o /etc/grub2.cfg
# systemctl reboot
```

- After rebooting the host, we need to acquire some information – specifically, ID information about the GPU device that we want to forward to our virtual machine. Let's do that:



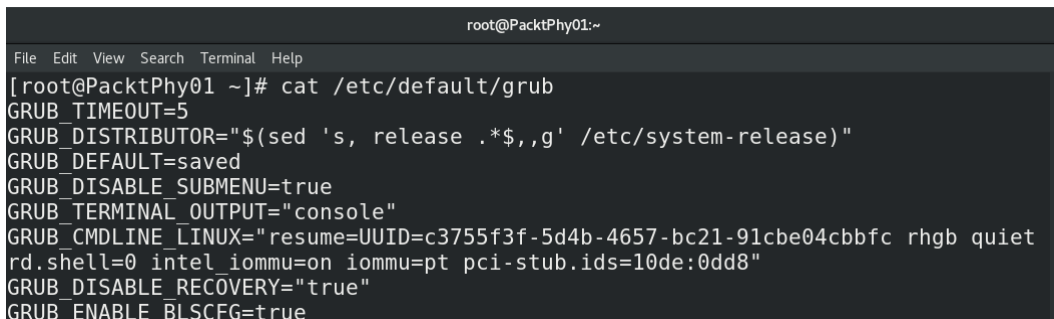
```
root@PacktPhy01:~
File Edit View Search Terminal Help
[root@PacktPhy01 ~]# lspci -Dnn | grep -i vga
0000:04:00.0 VGA compatible controller [0300]: NVIDIA Corporation GK107 [GeForce
GT 740] [10de:0fc8] (rev a1)
0000:05:00.0 VGA compatible controller [0300]: NVIDIA Corporation GF106GL [Quadro
2000] [10de:0dd8] (rev a1)
```

Figure 6.5 – Using `lspci` to display relevant configuration information

In our use case, we want to forward the Quadro 2000 card to our virtual machine as we're using the GT740 to hook up our monitors and the Quadro card is currently free of any workloads or connections. So, we need to take note of two numbers; that is, `0000:05:00.0` and `10de:0dd8`.

We will need both IDs going forward, with each one for defining which device we want to use and where.

- The next step is to explain to our host OS that it will not be using this PCI express device (Quadro card) for itself. In order to do that, we need to change the GRUB configuration again and add another parameter to the same file (`/etc/defaults/grub`):



```
root@PacktPhy01:~
File Edit View Search Terminal Help
[root@PacktPhy01 ~]# cat /etc/default/grub
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="resume=UUID=c3755f3f-5d4b-4657-bc21-91cbe04cbbfc rhgb quiet
rd.shell=0 intel iommu=on iommu=pt pci-stub.ids=10de:0dd8"
GRUB_DISABLE_RECOVERY="true"
GRUB_ENABLE_BLSCFG=true
```

Figure 6.6 – Adding the `pci-stub.ids` option to GRUB so that it ignores this device when booting the OS

Again, we need to reconfigure GRUB and reboot the server after this, so type in the following commands:

```
# grub2-mkconfig -o /etc/grub2.cfg
# systemctl reboot
```

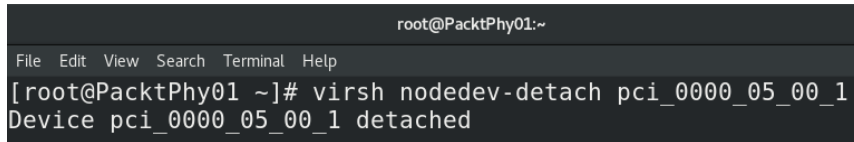
This step marks the end of the *physical* server configuration. Now, we can move on to the next stage of the process, which is how to use the now fully configured PCI passthrough device in our virtual machine.

- Let's check if everything was done correctly by using the `virsh nodedev-dumpxml` command on the PCI device ID:

```
root@PacktPhy01:~
File Edit View Search Terminal Help
[root@PacktPhy01 ~]# virsh nodedev-dumpxml pci_0000_05_00_0
<device>
  <name>pci_0000_05_00_0</name>
  <path>/sys/devices/pci0000:00/0000:00:02.0/0000:05:00.0</path>
  <parent>pci_0000_00_02_0</parent>
  <driver>
    <name>pci-stub</name>
  </driver>
  <capability type='pci'>
    <domain>0</domain>
    <bus>5</bus>
    <slot>0</slot>
    <function>0</function>
    <product id='0x0dd8'>GF106GL [Quadro 2000]</product>
    <vendor id='0x10de'>NVIDIA Corporation</vendor>
    <iommuGroup number='18'>
      <address domain='0x0000' bus='0x05' slot='0x00' function='0x1' />
      <address domain='0x0000' bus='0x05' slot='0x00' function='0x0' />
    </iommuGroup>
    <numa node='0' />
    <pci-express>
      <link validity='cap' port='0' speed='2.5' width='16' />
      <link validity='sta' speed='2.5' width='16' />
    </pci-express>
  </capability>
</device>
```

Figure 6.7 – Checking if the KVM stack can see our PCIe device

Here, we can see that QEMU sees two functions: 0x1 and 0x0. The 0x1 function is actually the GPU device's *audio* chip, which we won't be using for our procedure. We just need the 0x0 function, which is the GPU itself. This means that we need to mask it. We can do that by using the following command:



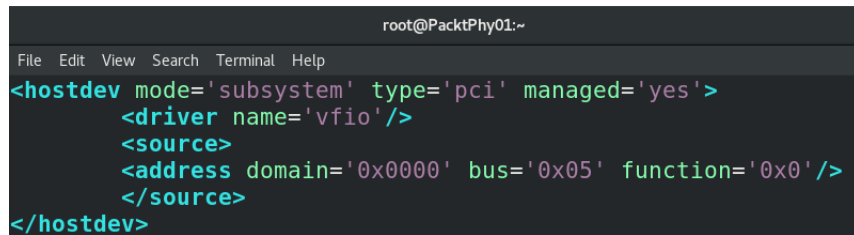
```

root@PacktPhy01:~
File Edit View Search Terminal Help
[root@PacktPhy01 ~]# virsh nodedev-detach pci_0000_05_00_1
Device pci_0000_05_00_1 detached

```

Figure 6.8 – Detaching the 0x1 device so that it can't be used for passthrough

- Now, let's add the GPU via PCI passthrough to our virtual machine. For this purpose, we're using a freshly installed virtual machine called `MasteringKVM03`, but you can use any virtual machine you want. We need to create an XML file that QEMU will use to know which device to add to a virtual machine. After that, we need to shut down the machine and import that XML file into our virtual machine. In our case, the XML file will look like this:



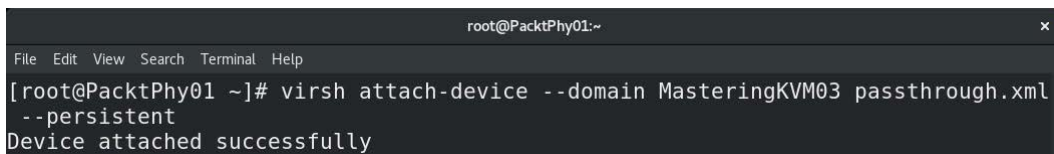
```

root@PacktPhy01:~
File Edit View Search Terminal Help
<hostdev mode='subsystem' type='pci' managed='yes'>
  <driver name='vfio' />
  <source>
    <address domain='0x0000' bus='0x05' function='0x0' />
  </source>
</hostdev>

```

Figure 6.9 – The XML file with our GPU PCI passthrough definition for KVM

- The next step is to attach this XML file to the `MasteringKVM03` virtual machine. We can do this by using the `virsh attach-device` command:



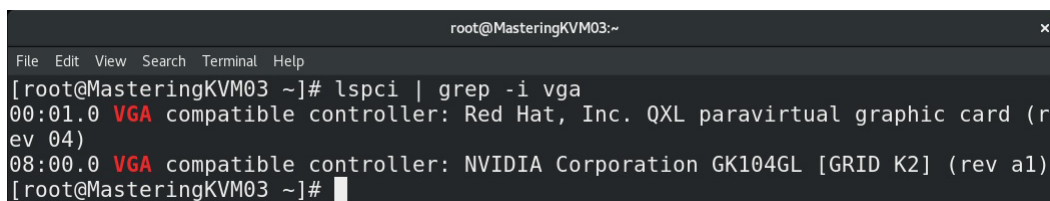
```

root@PacktPhy01:~
File Edit View Search Terminal Help
[root@PacktPhy01 ~]# virsh attach-device --domain MasteringKVM03 passthrough.xml
--persistent
Device attached successfully

```

Figure 6.10 – Importing the XML file into a domain/virtual machine

8. After the previous step, we can start our virtual machine, log in, and check if the virtual machine sees our GPU:

A terminal window titled 'root@MasteringKVM03:~' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the command 'lspci | grep -i vga' and its output: '00:01.0 VGA compatible controller: Red Hat, Inc. QXL paravirtual graphic card (rev 04)' and '08:00.0 VGA compatible controller: NVIDIA Corporation GK104GL [GRID K2] (rev a1)'.

```
root@MasteringKVM03:~  
File Edit View Search Terminal Help  
[root@MasteringKVM03 ~]# lspci | grep -i vga  
00:01.0 VGA compatible controller: Red Hat, Inc. QXL paravirtual graphic card (rev 04)  
08:00.0 VGA compatible controller: NVIDIA Corporation GK104GL [GRID K2] (rev a1)  
[root@MasteringKVM03 ~]#
```

Figure 6.11 – Checking GPU visibility in our virtual machine

The next logical step would be to install the NVIDIA driver for this card for Linux so that we can freely use it as our discrete GPU.

Now, let's move on to another important subject that is related to remote display protocols. We kind of danced around this subject in the previous part of this chapter as well, but now we are going to tackle it head-on.

Discussing remote display protocols

As we mentioned previously, there are different virtualization solutions, so it's only normal that there are different methods to *access* virtual machines. If you take a look at the history of virtual machines, we had a number of different display protocols taking care of this particular problem. So, let's discuss this history a bit.

Remote display protocols history

There will be people disputing this premise, but remote protocols started as text-only protocols. Whichever way you look at it, serial, text-mode terminals were here before we had X Windows or anything remotely resembling a GUI in the Microsoft, Apple, and UNIX-based worlds. Also, you can't dispute the fact that the telnet and rlogin protocols are also used to access remote display. It just so happens that the remote display that we're accessing by using telnet and rlogin is a text-based display. By extension, the same thing applies to SSH. And serial terminals, text consoles, and text-based protocols such as telnet and rlogin were some of the most commonly used starting points that go way back to the 1970s.

The end of the 1970s was an important time in computer history as there were numerous attempts to start mass-producing a personal computer for large amounts of people (for example, Apple II from 1977). In the 1980s, people started using personal computers more, as any Amiga, Commodore, Atari, Spectrum, or Amstrad fan will tell you. Keep in mind that the first real, publicly available GUI-based OSes didn't start appearing until Xerox Star (1981) and Apple Lisa (1983). The first widely available Apple-based GUI OS was Mac OS System 1.0 in 1984. Most of the other previously mentioned computers were all using a text-based OS. Even games from that era (and for many years to come) looked like they were drawn by hand while you were playing them. Amiga's Workbench 1.0 was released in 1985 and with its GUI and color usage model, it was miles ahead of its time. However, 1985 is probably going to be remembered for something else – this is the year that the first Microsoft Windows OS (v1.0) was released. Later, that became Windows 2.0 (1987), Windows 3.0 (1990), Windows 3.1 (1992), by which time Microsoft was already taking the OS world by storm. Yes, there were other OSes by other manufacturers too:

- Apple: Mac OS System 7 (1991)
- IBM: OS/2 v1 (1988), v1.2 (1989), v2.0 (1992), Warp 4 (1996)

All of these were just a tiny dot on the horizon compared to the big storm that happened in 1995, when Microsoft introduced Windows 95. It was the first Microsoft client OS that was able to boot to GUI by default since the previous versions were started from a command line. Then came Windows 98 and XP, which meant even more market share for Microsoft. The rest of that story is probably very familiar, with Vista, Windows 7, Windows 8, and Windows 10.

The point of this story is not to teach you about OS history per se. It's about noticing the trend, which is simple enough. We started with text interfaces in the command line (for example, IBM and MS DOS, early versions of Windows, Linux, UNIX, Amiga, Atari, and so on). Then, we slowly moved toward more visual interfaces (GUI). With advancements in networking, GPU, CPU, and monitoring technologies, we've reached a phase in which we want a shiny, 4K-resolution monitor with 4-megapixel resolutions, low latency, huge CPU power, fantastic colors, and a specific user experience. That user experience needs to be immediate, and it shouldn't really matter that we're using a local OS or a remote one (VDI, the cloud, or whatever the background technology is).

This means that along with all the hardware components that we just mentioned, other (software) components needed to be developed as well. Specifically, what needed to be developed were high-quality remote display protocols, which nowadays must be able to be extended to a browser-based usage model, as well. People don't want to be forced to install additional applications (clients) to access their remote resources.

Types of remote display protocols

Let's just mention some protocols that are very active on the market *now*:

- Microsoft Remote Desktop Protocol/Remote FX: Used by Remote Desktop Connection, this multi-channel protocol allows us to connect to Microsoft-based virtual machines.
- VNC: Short for Virtual Network Computing, this is a remote desktop sharing system that transmits mouse and keyboard events to access remote machines.
- SPICE: Short for Simple Protocol for Independent Computing Environments, this is another remote display protocol that can be used to access remote machines. It was developed by Qumranet, which was bought by Red Hat.

If we further expand our list to protocols that are being used for VDI, then the list increases further:

- Teradici PCoIP (PC over IP): A UDP-based VDI protocol that we can use to access virtual machines on VMware, Citrix and Microsoft-based VDI solutions
- VMware Blast Extreme: VMware's answer to PcoIP for VMware Horizon-based VDI solution
- Citrix HDX: Citrix's protocol for virtual desktops.

Of course, there are others that are available but not used as much and are way less important, such as the following:

- Colorado CodeCraft
- OpenText Exceed TurboX
- NoMachine
- FreeNX
- Apache Guacamole
- Chrome Remote Desktop
- Miranex

The major differences between regular remote protocols and fully featured VDI protocols are related to additional functionalities. For example, on PCoIP, Blast Extreme, and HDX, you can fine-tune bandwidth settings, control USB and printer redirection (manually or centrally via policies), use multimedia redirection (to offload media decoding), Flash redirection (to offload Flash), client drive redirection, serial port redirection, and dozens of other features. You can't do some of these things on VNC or Remote Desktop, for example.

Having said that, let's discuss two of the most common ones in the open source world: VNC and SPICE.

Using the VNC display protocol

When the VNC graphics server is enabled through libvirt, QEMU will redirect the graphics output to its inbuilt VNC server implementation. The VNC server will listen to a network port where the VNC clients can connect.

The following screenshot shows how to add a VNC graphics server. Just go to **Virtual Machine Manager**, open the settings of your virtual machine, and go to the **Display Spice** tab on the left-hand side:

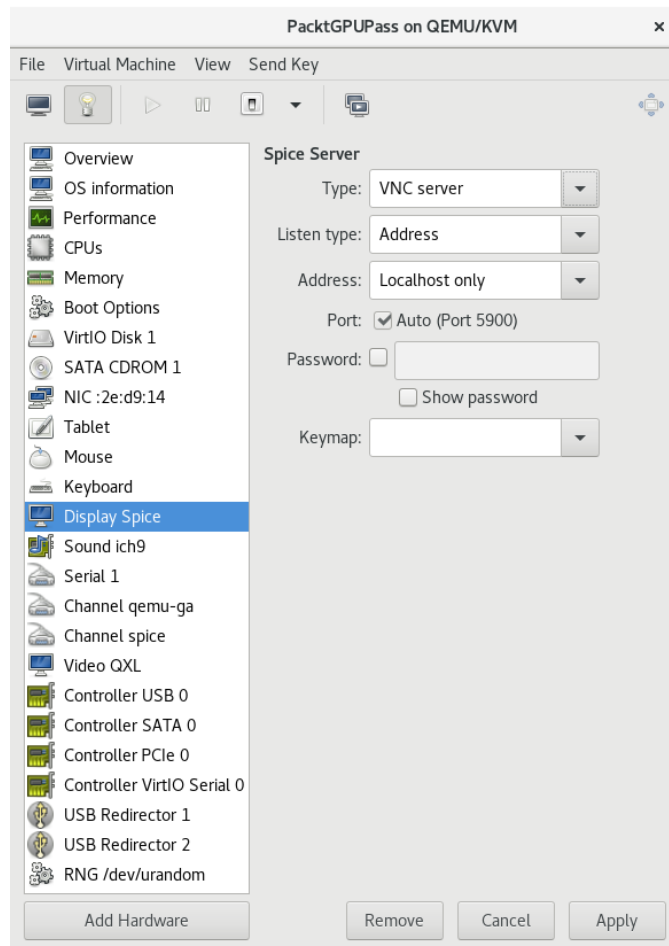


Figure 6.12 – VNC configuration for a KVM virtual machine

When adding VNC graphics, you will be presented with the options shown in the preceding screenshot:

- **Type:** The type of the graphics server. Here, it is **VNC server**.
- **Address:** VNC server listening address. It can be all, localhost, or an IP address. By default, it is **Localhost only**.
- **Port:** VNC server listening port. You can either choose auto, where libvirt defines the port based on the availability, or you can define one yourself. Make sure it does not create a conflict.
- **Password:** The password protecting VNC access.
- **Keymap:** If you want to use a specific keyboard layout instead of an auto detected one, you can do the same using the `virt-xml` command-line tool.

For example, let's add VNC graphics to a virtual machine called `PacktGPUPass` and then modify its VNC listening IP to `192.168.122.1`:

```
# virt-xml MasteringKVM03 --add-device --graphics type=vnc
# virt-xml MasteringKVM03 --edit --graphics
listen=192.168.122.1
```

This is how it looks in the `PacktVM01` XML configuration file:

```
<graphics type='vnc' port='-1' autoport='yes'
listen='192.168.122.1'>
  <listen type='address' address='192.168.122.1' />
</graphics>
```

You can also use `virsh` to edit `PacktGPUPass` and change the parameters individually.

Why VNC?

You can use VNC when you access virtual machines on LAN or to access the VMs directly from the console. It is not a good idea to expose virtual machines over a public network using VNC as the connection is not encrypted. VNC is a good option if the virtual machines are servers with no GUI installed. Another point that is in favor of VNC is the availability of clients. You can access a virtual machine from any operating system platform as there will be a VNC viewer available for that platform.

Using the SPICE display protocol

Like KVM, a **Simple Protocol for Independent Computing Environments (SPICE)** is one of the best innovations that came into open source virtualization technologies. It propelled the open source virtualization to a large **Virtual Desktop Infrastructure (VDI)** implementation.

Important Note

Qumranet originally developed SPICE as a closed source code base in 2007. Red Hat, Inc. acquired Qumranet in 2008, and in December 2009, they decided to release the code under an open source license and treat the protocol as an open standard.

SPICE is the only open source solution available on Linux that gives two-way audio. It has high-quality 2D rendering capabilities that can make use of a client system's video card. SPICE also supports multiple HD monitors, encryption, smart card authentication, compression, and USB passthrough over the network. For a complete list of features, you can visit <http://www.spice-space.org/features.html>. If you are a developer and want to know about the internals of SPICE, visit <http://www.spice-space.org/documentation.html>. If you are planning for VDI or installing virtual machines that need GUIs, SPICE is the best option for you.

SPICE may not be compatible with some older virtual machines as they do not have support for QXL. In those cases, you can use SPICE along with other video generic virtual video cards.

Now, let's learn how to add a SPICE graphics server to our virtual machine. This can be considered the best-performing virtual display protocol in the open source world.

Adding a SPICE graphics server

Libvirt now selects SPICE as the default graphics server for most virtual machine installations. You must follow the same procedures that we mentioned earlier for VNC to add the SPICE graphics server. Just change the VNC to SPICE in the dropdown. Here, you will get an additional option to select a **TLS port** since SPICE supports encryption:

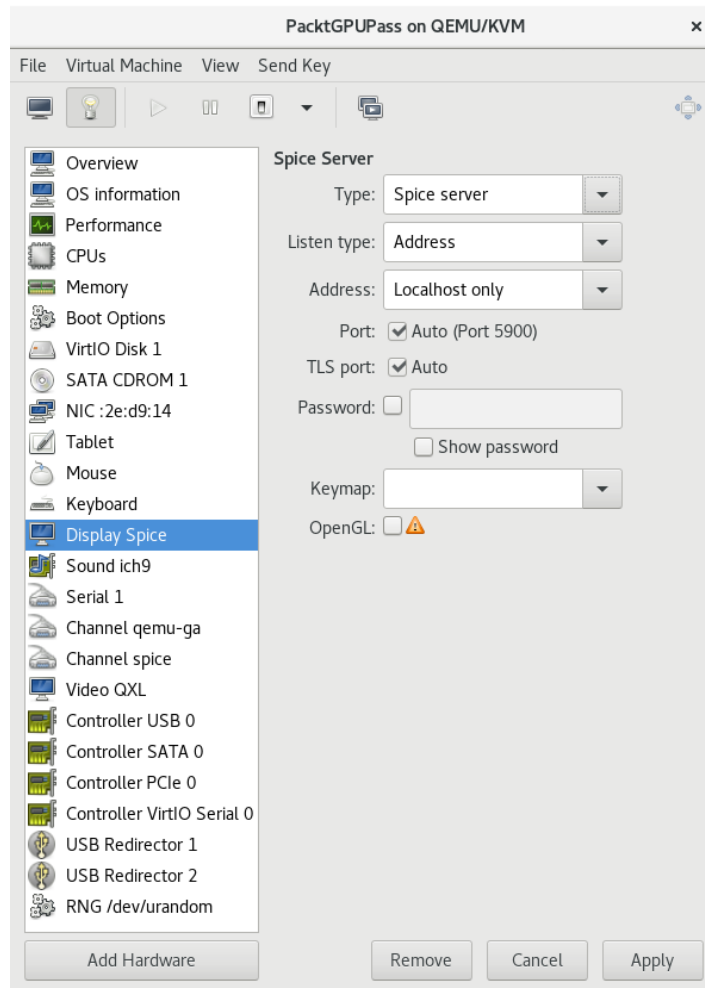


Figure 6.13 – SPICE configuration for a KVM virtual machine

To get to this configuration window, just edit the settings of your virtual machine. Go to the **Display Spice** options and select **Spice server** from the pull-down menu. All the other options are optional, so you don't necessarily have to do any additional configuration.

With this previous procedure completed, we've covered all the necessary topics regarding display protocols. Let's now discuss the various methods we can use to access the virtual machine console.

Methods to access a virtual machine console

There are multiple ways to connect to a virtual machine console. If your environment has full GUI access, then the easiest method is to use the virt-manager console itself. `virt-viewer` is another tool that can give you access to your virtual machine console. This tool is very helpful if you are trying to access a virtual machine console from a remote location. In the following example, we are going to make a connection to a remote hypervisor that has an IP of `192.168.122.1`. The connection is tunneled through an SSH session and is secure.

The first step is to set up an authentication system without a password between your client system and the hypervisor:

1. On the client machine, use the following code:

```
# ssh-keygen
# ssh-copy-id root@192.168.122.1
# virt-viewer -c qemu+ssh://root@192.168.122.1/system
```

You will be presented with a list of virtual machines available on the hypervisor. Select the one you have to access, as shown in the following screenshot:

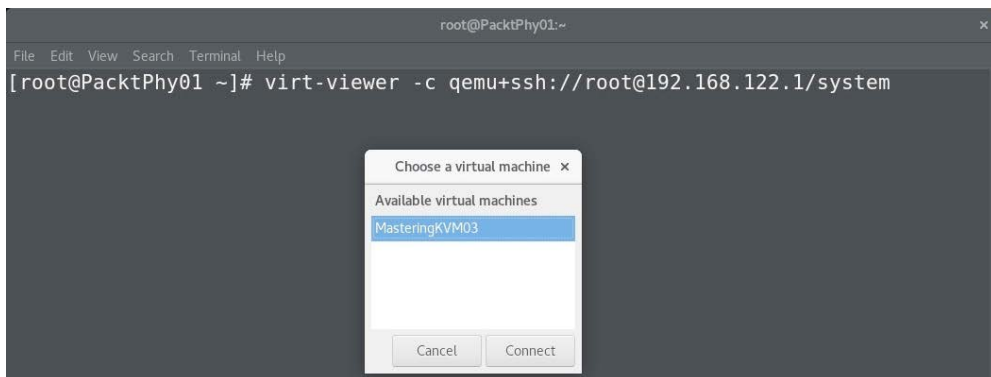


Figure 6.14 – virt-viewer selection menu for virtual machine access

2. To connect to a VM's console directly, use the following command:

```
# virt-viewer -c qemu+ssh://root@192.168.122.1/system
MasteringKVM03
```

If your environment is restricted to only a text console, then you must rely on your favorite `virsh` – to be more specific, `virsh console vm_name`. This needs some additional configuration inside the virtual machine OS, as described in the following steps.

3. If your Linux distro is using GRUB (not GRUB2), append the following line to your existing boot Kernel line in `/boot/grub/grub.conf` and shut down the virtual machine:

```
console=tty0 console=ttyS0,115200
```

If your Linux distro is using GRUB2, then the steps become a little complicated. Note that the following command has been tested on a Fedora 22 virtual machine. For other distros, the steps to configure GRUB2 might be different, though the changes that are required for GRUB configuration file should remain the same:

```
# cat /etc/default/grub (only relevant variables are shown)
```

```
GRUB_TERMINAL_OUTPUT="console"
```

```
GRUB_CMDLINE_LINUX="rd.lvm.lv=fedora/swap rd.lvm.lv=fedora/root rhgb quiet"
```

The changed configuration is as follows:

```
# cat /etc/default/grub (only relevant variables are shown)
```

```
GRUB_TERMINAL_OUTPUT="serial console"
```

```
GRUB_CMDLINE_LINUX="rd.lvm.lv=fedora/swap rd.lvm.lv=fedora/root console=tty0 console=ttyS0"
```

```
# grub2-mkconfig -o /boot/grub2/grub.cfg
```

4. Now, shut down the virtual machine. Then, start it again using `virsh`:

```
# virsh shutdown PacktGPUPass
```

```
# virsh start PacktGPUPass --console
```

5. Run the following command to connect to a virtual machine console that has already started:

```
# virsh console PacktGPUPass
```

You can also do this from a remote client, as follows:

```
# virsh -c qemu+ssh://root@192.168.122.1/system console PacktGPUPass
```

```
Connected to domain PacktGPUPass:
```

```
Escape character is ^]
```


In some cases, we have seen a console command stuck at `^]`. To work around this, press the *Enter* key multiple times to see the login prompt. Sometimes, configuring a text console is very useful when you want to capture the boot messages for troubleshooting purposes. Use `ctrl +]` to exit from the console.

Our next topic takes us to the world of noVNC, another VNC-based protocol that has a couple of major advantages over the *regular* VNC. Let's discuss these advantages and the implementation of noVNC now.

Getting display portability with noVNC

All these display protocols rely on having access to some type of client application and/or additional software support that will enable us to access the virtual machine console. But what happens when we just don't have access to all of these additional capabilities? What happens if we only have text mode access to our environment, but we still want to have GUI-based management of connections to our virtual machines?

Enter noVNC, a HTML5-based VNC client that you can use via a HTML5-compatible web browser, which is just fancy talk for *practically every web browser on the market*. It supports all the most popular browsers, including mobile ones, and loads of other features, such as the following:

- Clipboard copy-paste
- Supports resolution scaling and resizing
- It's free under the MPL 2.0 license
- It's rather easy to install and supports authentication and can easily be implemented securely via HTTPS

If you want to make noVNC work, you need two things:

- Virtual machine(s) that are configured to accept VNC connections, preferably with a bit of configuration done – a password and a correctly set up network interface to connect to the virtual machine, for instance. You can freely use `tigervnc-server`, configure it to accept connections on – for example – port 5901 for a specific user, and use that port and server's IP address for client connections.

- noVNC installation on a client computer, which you can either download from EPEL repositories or as a zip/tar.gz package and run directly from your web browser. To install it, we need to type in the following sequence of commands:

```
yum -y install novnc
```

```
cd /etc/pki/tls/certs
```

```
openssl req -x509 -nodes -newkey rsa:2048 -keyout /etc/  
pki/tls/certs/nv.pem -out /etc/pki/tls/certs/nv.pem -days  
365
```

```
websockify -D --web=/usr/share/novnc --cert=/etc/pki/tls/  
certs/nv.pem 6080 localhost:5901
```

The end result will look something like this:



Figure 6.15 – noVNC console configuration screen

Here, we can use our VNC server password for that specific console. After typing in the password, we get this:

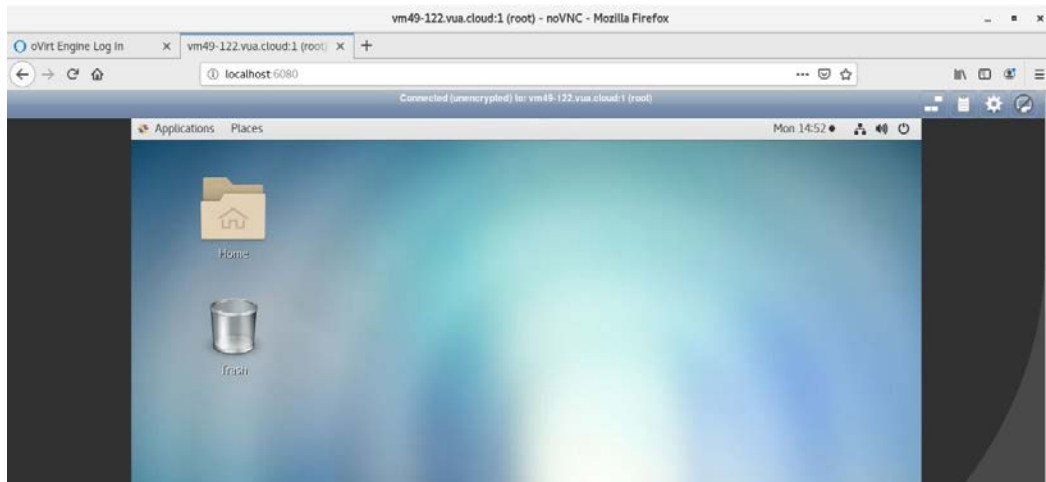


Figure 6.16 – noVNC console in action – we can see the virtual machine console and use it to work with our virtual machine

We can also use all these options in oVirt. During the installation of oVirt, we just need to select one additional option during the engine-setup phase:

```
--otopi-environment="OVESETUP_CONFIG/  
websocketProxyConfig=bool:True"
```

This option will enable oVirt to use noVNC as a remote display client, on top of the existing SPICE and VNC.

Let's take a look at an example of configuring a virtual machine in oVirt with pretty much all of the options that we've discussed in this chapter. Pay close attention to the **Monitors** configuration option:

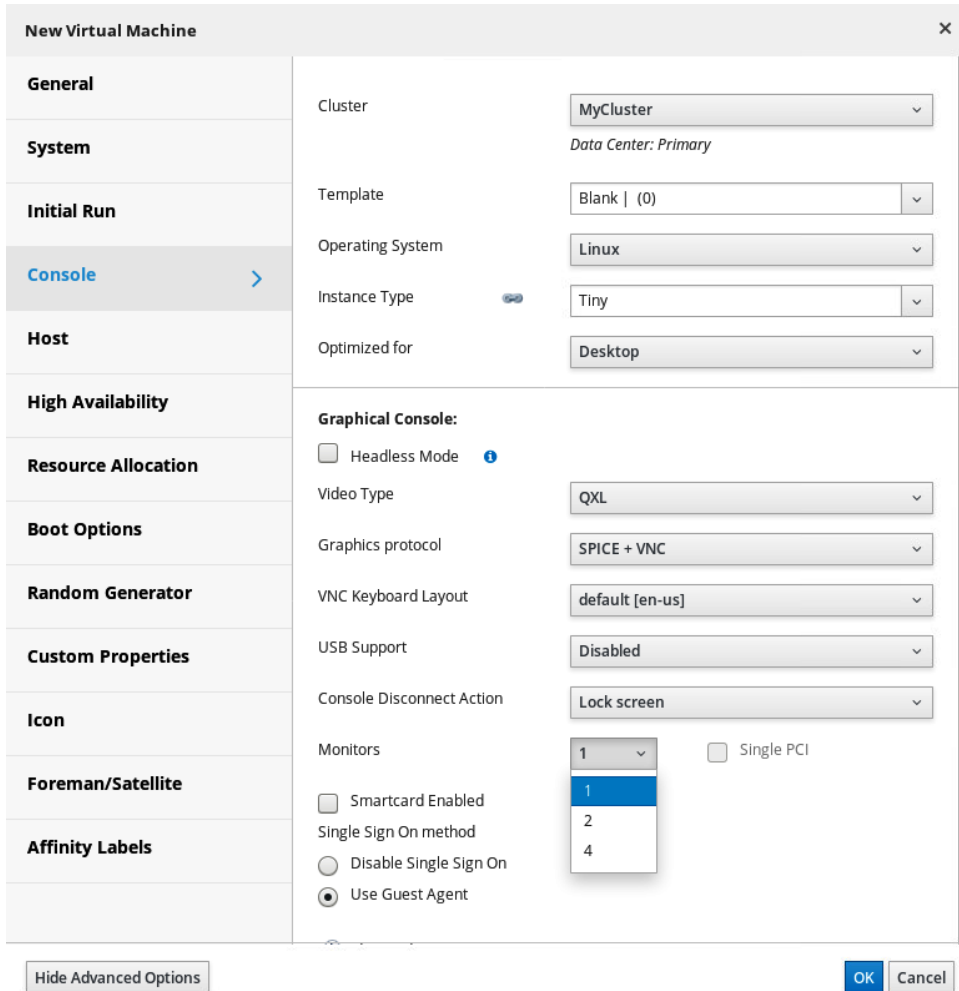


Figure 6.17 – oVirt also supports all the devices we discussed in this chapter

If we click on the **Graphics protocol** submenu, we will get the option to use SPICE, VNC, noVNC, and various combinations thereof. Also, at the bottom of the screen, we have available options for a number of monitors that we want to see in our remote display. This can be very useful if we want to have a high-performance multi-display remote console.

Seeing that noVNC has been integrated to noVNC as well, you can treat this as a sign of things to come. Think about it from this perspective – everything related to management applications in IT has steadily been moving to web-based applications for years now. It's only logical that the same things happen to virtual machine consoles. This has also been implemented in other vendors' solutions, so seeing noVNC being used here shouldn't be a big surprise.

Summary

In this chapter, we covered virtual display devices and protocols used to display virtual machine data. We also did some digging into the world of GPU sharing and GPU passthrough, which are important concepts for large-scale virtualized environments running VDI. We discussed some benefits and drawbacks to these scenarios as they tend to be rather complex to implement and require a lot of resources – financial resources included. Imagine having to do PCI passthrough for 2D/3D acceleration for 100 virtual machines. That would actually require buying 100 graphic cards, which is a big, big ask financially. Among the other topics we discussed, we went through various display protocols and options that can be used for console access to our virtual machines.

In the next chapter, we will take you through some regular virtual machine operations – installation, configuration, and life cycle management, including discussing snapshots and virtual machine migration.

Questions

1. Which types of virtual machine display devices can we use?
2. What are the main benefits of using a QXL virtual display device versus VGA?
3. What are the benefits and drawbacks of GPU sharing?
4. What are the benefits of GPU PCI passthrough?
5. What are the main advantages of SPICE versus VNC?
6. Why would you use noVNC?

Further reading

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

- Configuring and managing virtualization: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_virtualization/index
- QEMU documentation: <https://www.qemu.org/documentation/>
- NVIDIA virtual GPU software documentation: <https://docs.nvidia.com/grid/latest/grid-vgpu-release-notes-red-hat-el-kvm/index.html>
- Working with IOMMU groups: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/app-iommu

7

Virtual Machines: Installation, Configuration, and Life Cycle Management

In this chapter, we will discuss different ways of installing and configuring **virtual machines (VMs)**—from Command Prompt and/or a **graphical user interface (GUI)**. We will delve deeper into some tools and utilities that we have already used (`virt-manager`, `virt-install`, `oVirt`) and build upon our knowledge gained from previous chapters. Then, we will have a lengthy discussion about VM migration, one of the most fundamental aspects of virtualization, as it's pretty much unimaginable to use virtualization without migration options. To be able to configure our environment for VM migration, we will also use topics discussed in *Chapter 4, Libvirt Networking*, and *Chapter 5, Libvirt Storage*, as there are pre-requisites that need to be met for VM migration to work.

In this chapter, we will cover the following topics:

- Creating a new VM using `virt-manager`, using `virt` commands
- Creating a new VM using oVirt
- Configuring your VM
- Adding and removing virtual hardware from your VM
- Migrating VMs

Creating a new VM using `virt-manager`

`virt-manager` (a GUI tool for managing VMs) and `virt-install` (a command-line utility for managing VMs) are two of the most commonly used utilities in **Kernel-based VM (KVM)** virtualization. By using them, we can do practically everything to our VMs—create, start, stop, delete, and much more. We already had a chance to work with these two utilities in previous chapters, but we need to take a more structured approach to the subject as they offer loads of additional options that we haven't have a chance to discuss yet. We'll also add some other utilities that are a part of the `virt-*` command stack that are very, very useful.

Let's start with `virt-manager` and its familiar GUI.

Using `virt-manager`

`virt-manager` is the go-to GUI utility to manage KVM VMs. It's very intuitive and easy to use, albeit lacking in functionality a bit, as we will describe a bit later. This is the main `virt-manager` window:

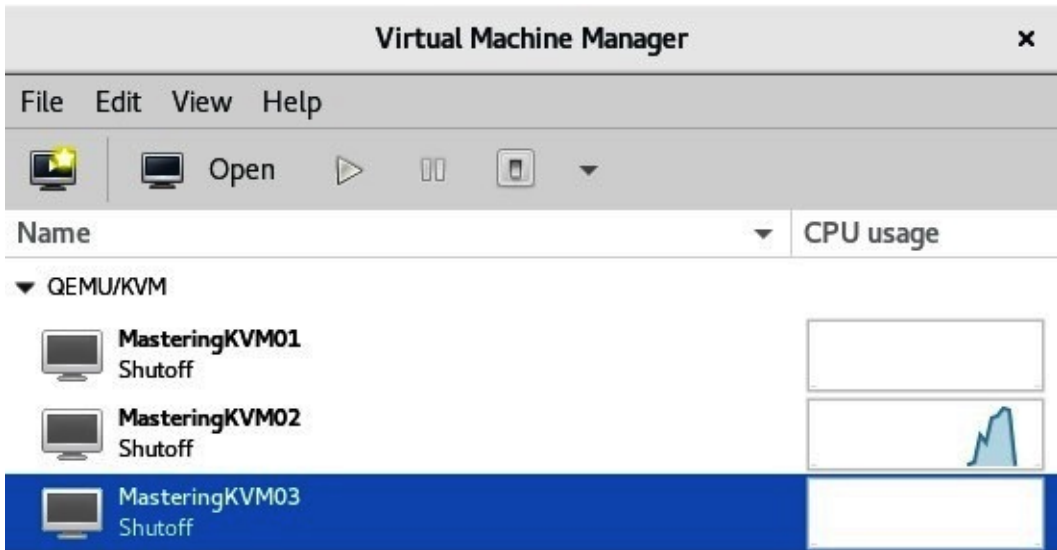


Figure 7.1 – Main virt-manager window

From this screenshot, we can already see that there are three VMs installed on this server. We can use the top-level menus (**F**ile, **E**dit, **V**iew, and **H**elp) to further configure our KVM server and/or VMs, as well as to connect to other KVM hosts on the network, as you can see in the following screenshot:

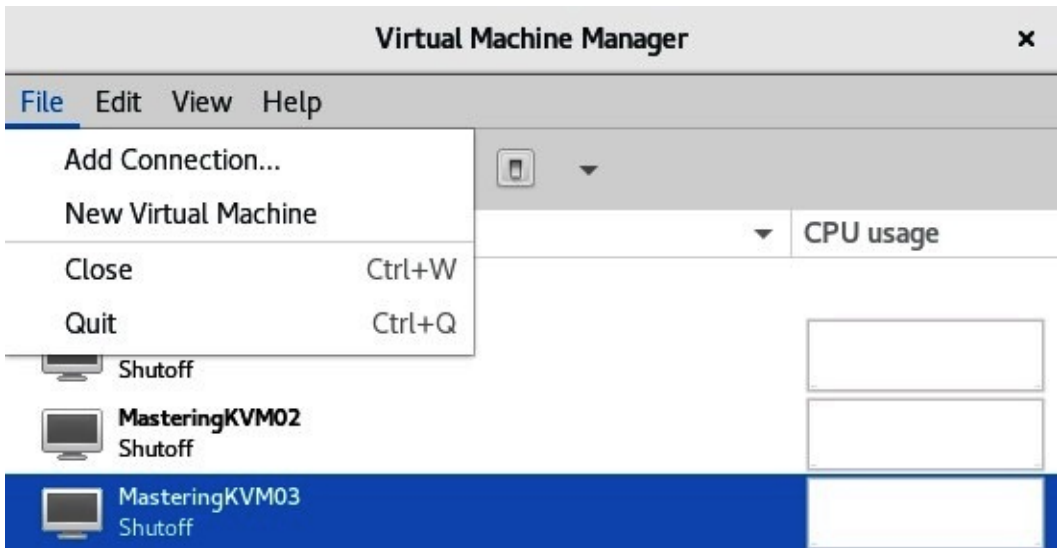


Figure 7.2 – Connecting to other KVM hosts by using the Add Connection... option

After we select the **Add Connection...** option, we will be greeted by a wizard to connect to the external host, and we just need to punch in some basic information—the username (it has to be a user that has administrative rights) and hostname or **Internet Protocol (IP)** address of the remote server. Before we do that, we also need to configure **Secure Shell (SSH)** keys on our local machine and copy our key to that remote machine, as this is the default authentication method for `virt-manager`. The process is shown in the following screenshot:

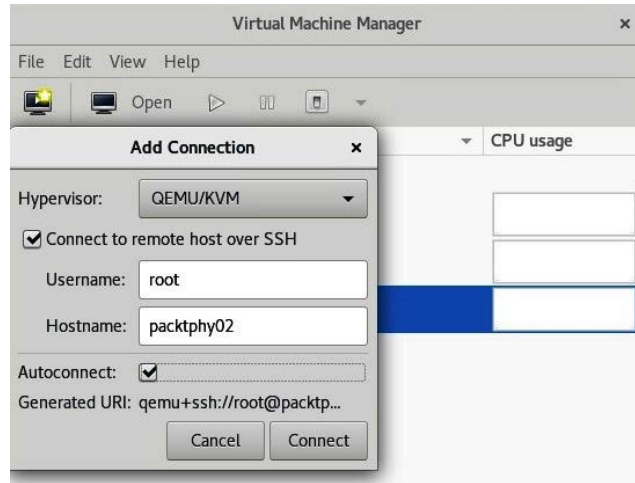


Figure 7.3 – Connecting to remote KVM host

At this point, you can start freely installing the VM on that remote KVM host, should you choose to do so, by right-clicking on the hostname and selecting **New**, as illustrated in the following screenshot:

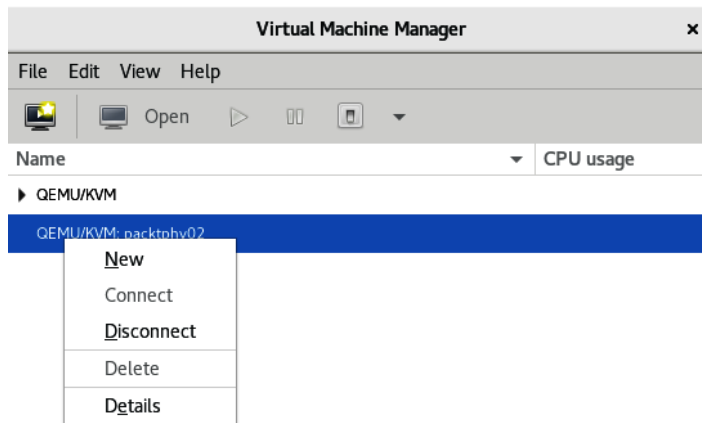


Figure 7.4 – Creating a new VM on a remote KVM host

As this wizard is the same as the wizard for installing VMs on your local server, we'll cover both of these scenarios in one go. The first step in the **New VM** wizard is selecting *where* you're installing your VM *from*. As you can see in the following screenshot, there are four available options:

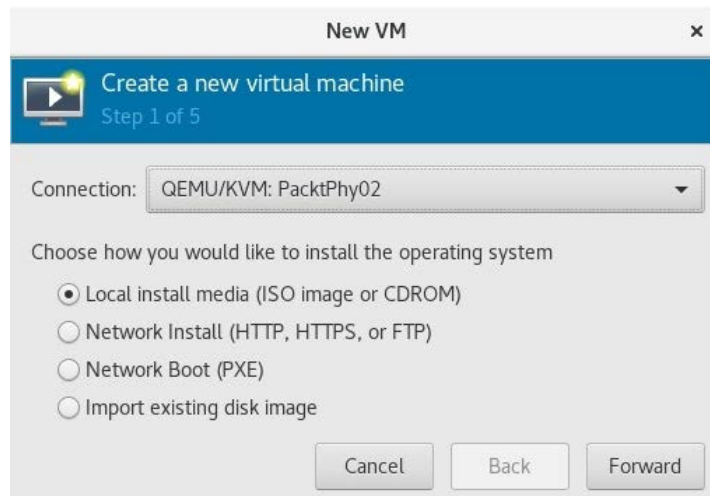


Figure 7.5 – Selecting boot media

The choices are as follows:

- If you already have an **International Organization for Standardization (ISO)** file available on your local machine (or as a physical device), select the first option.
- If you want to install from the network, select the second option.
- If you have a **Preboot eXecution Environment (PXE)** boot set up in your environment and you can boot your VM installation from the network, select the third option.
- If you have a VM disk and you just want to underlay that to a VM you're defining, select the fourth option.

Commonly, we're talking about network installations (second option) or PXE-booted network installations (third option), as these are the most popular use cases in production. The reason for this is very simple—there's absolutely no reason to waste local disk space on ISO files, which are quite big nowadays. For example, a CentOS 8 v1905 ISO file is roughly 8 **gigabytes (GB)** in size. If you need to be able to install multiple operating systems, or even multiple versions of these operating systems, you're better off with some sort of centralized storage space for ISO files only.

In VMware **ESX integrated (ESXi)**-based infrastructures, people often use ISO datastores or content libraries for this functionality. In Microsoft Hyper-V-based infrastructures, people usually have a **Server Message Block (SMB)** file share with ISO files needed for a VM installation. It would be quite pointless to have a copy of an operating system ISO per host, so some kind of a shared approach is much more convenient and is a good space-saving mechanism.

Let's say that we're installing a VM from a network (**HyperText Transfer Protocol (HTTP)**, **HyperText Transfer Protocol Secure (HTTPS)**, or **File Transfer Protocol (FTP)**). We're going to need a couple of things to proceed, as follows:

- A **Uniform Resource Locator (URL)** from which we can complete our installation—in our example, we are going to use `http://mirror.linux.duke.edu/pub/centos`. From this link choose the latest `8.x.x` directory, and then go to `BaseOS/x86_64/os`.
- Obviously, a functional internet connection—as fast as possible, as we are going to download all the necessary installation packages from the preceding URL.
- Optionally, we can open the **URL options** triangle and use additional options for the kernel line—most commonly, kickstart options with something such as the following:

```
ks=http://kickstart_file_url/file.ks
```

So, let's type that in, as follows:

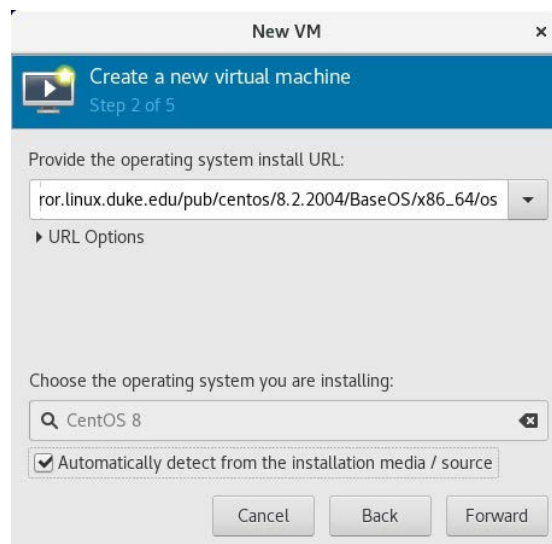


Figure 7.6 – URL and guest operating system selection

Note that we *manually* selected **Red Hat Enterprise Linux 8.0** as the target guest operating system as `virt-manager` doesn't currently recognize CentOS 8 (1905) as the guest operating system from the URL that we specified. If the operating system had been on the list of currently recognized operating systems, we could've just selected the **Automatically detect from installation media / source** checkbox, which you sometimes need to re-check and uncheck a couple of times before it works.

After clicking on the **Forward** button, we're faced with memory and **central processing unit (CPU)** settings for this VM. Again, you can go in two different directions here, as follows:

- Select the bare minimum of resources (for example, 1 **virtual CPU (vCPU)** and 1 GB of memory), and then change that afterward if you need more CPU horsepower and/or more memory.
- Select a decent amount of resources (for example, 2 vCPU and 4 GB of memory) with a specific usage in mind. For example, if the intended use case for this VM is a file server, you won't get an awful lot of performance if you add 16 vCPUs and 64 GB of memory to it, but there might be other use cases in which this will be appropriate.

The next step is configuring the VM storage. There are two available options, as we can see in the following screenshot:

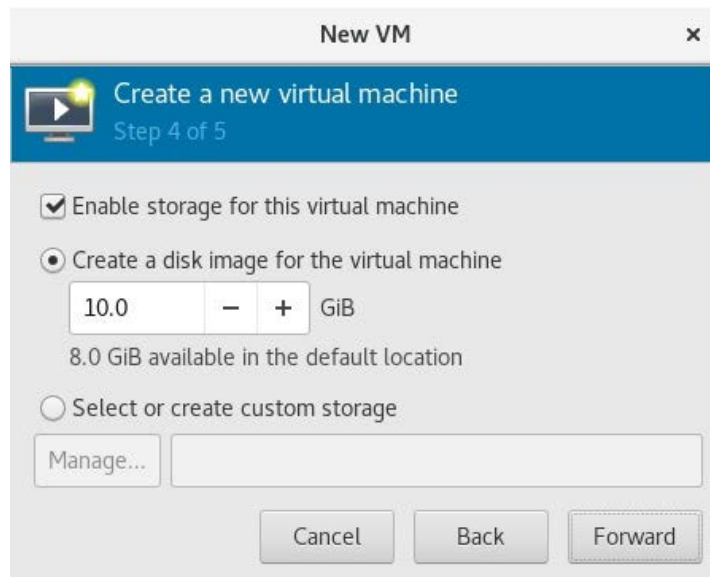


Figure 7.7 – Configuring VM storage

It's very important that you select a *proper* storage device for the VM, as you might have various problems in the future if you don't. For example, if you put your VM on the wrong storage device in a production environment, you'll have to migrate storage of that VM to another storage device, which is a tedious and time-consuming process that will have some nasty side effects if you have loads of VMs running on the source or destination storage device. For starters, it will seriously impact their performance. Then, if you have some dynamic workload management mechanism in your environment, it could trigger additional VM or VM storage movement in your infrastructure. Features such as VMware's **Distributed Resource Scheduler (DRS)**/Storage DRS, Hyper-V performance and resource optimization (with **System Center Operations Manager (SCOM)** integration), and oVirt/Red Hat Enterprise Virtualization cluster scheduling policies do things such as that. So, adopting the *think twice, do once* strategy might be the correct approach here.

If you select the first available option, **Create a disk image for the virtual machine**, `virt-manager` will create a VM hard disk in its default location—for **Red Hat Enterprise Linux (RHEL)** and CentOS, that's in the `/var/lib/libvirt/images` directory. Make sure that you have enough space for your VM hard disk. Let's say that we have 8 GB of space available in the `/var/lib/libvirt/images` directory and its underlying partition. If we leave everything as-is from the previous screenshot, we'd get an error message because we tried to create a 10 GB file on a local disk where only 8 GB is available.

After we click the **Forward** button again, we're at the final step of the VM creation process, where we can select the VM name (as it will appear in `virt-manager`), customize the configuration before the installation process, and select which virtual network the VM will use. We will cover the hardware customization of the VM a bit later in the chapter. After you click **Finish**, as shown in the following screenshot, your VM will be ready for deployment and—after we install the operating system—use:

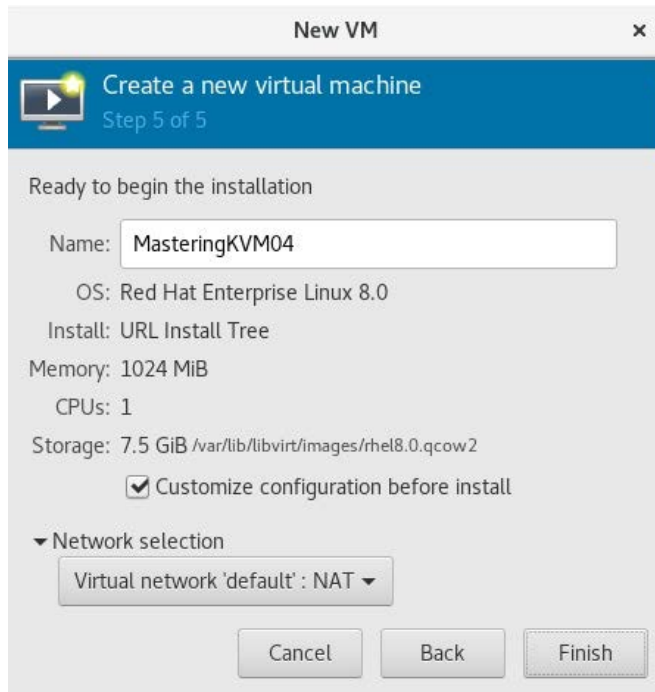


Figure 7.8 – Final virt-manager configuration step

Using `virt-manager` to create some VMs definitely wasn't a difficult task, but in real-life production environments, you won't necessarily find a GUI installed on a server. Therefore, our logical next task is to get to know command-line utilities to manage VMs—specifically, `virt-*` commands. Let's do that next.

Using `virt-*` commands

As previously mentioned, we need to learn some new commands to master the task of basic VM administration. For this specific purpose, we have stack of `virt-*` commands. Let's briefly go over some of the most important ones and learn how to use them.

virt-viewer

As we've already used the `virt-install` command heavily before (check out *Chapter 3, Installing a Kernel-based Virtual Machine (KVM) Hypervisor, libvirt, and ovirt*, where we installed quite a few VMs by using this command), we're going to cover the remaining commands.

Let's start with `virt-viewer`, as we've used this application before. Every time we double-click on a VM in `virt-viewer`, we open a VM console, and that happens to be `virt-viewer` in the background of this procedure. But if we wanted to use `virt-viewer` from a shell—as people often do—we need some more information about it. So, let's use a couple of examples.

First, let's connect to a local KVM called `MasteringKVM01`, which resides on the host that we're currently connected to as `root`, by running the following command:

```
# virt-viewer --connect qemu:///system MasteringKVM01
```

We could also connect to the VM in `kiosk` mode, which means that `virt-viewer` will close when we shut down the VM that we connect to. To do this, we would run the following command:

```
# virt-viewer --connect qemu:///system MasteringKVM01 --kiosk  
--kiosk-quit on-disconnect
```

If we need to connect to a *remote* host, we can also use `virt-viewer`, but we need a couple of additional options. The most common way to authenticate to a remote system is through SSH, so we can do the following:

```
# virt-viewer --connect qemu+ssh://username@remote-host/system  
VirtualMachineName
```

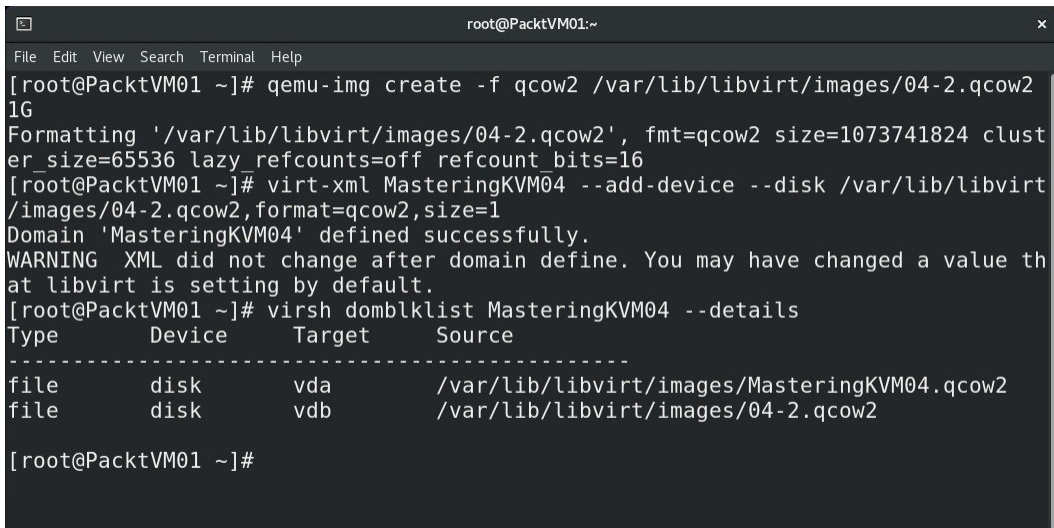
If we configured SSH keys and copied them to `username@remote-host`, this previous command wouldn't ask us for a password. But if we didn't, it is going to ask us for a password twice—to establish connection to the hypervisor and then to establish connection to the VM **Virtual Network Computing (VNC)** session.

virt-xml

The next command-line utility on our list is `virt-xml`. We can use it with `virt-install` command-line options to change the VM configuration. Let's start with a basic example—let's just enable the boot menu for the VM, as follows:

```
# virt-xml MasgteringKVM04 --edit --boot bootmenu=on
```

Then, let's add a thin-provisioned disk to the VM, in three steps— first, create the disk itself, and then attach it to the VM and check that everything worked properly. The output can be seen in the following screenshot:



```

root@PacktVM01:~
File Edit View Search Terminal Help
[root@PacktVM01 ~]# qemu-img create -f qcow2 /var/lib/libvirt/images/04-2.qcow2
1G
Formatting '/var/lib/libvirt/images/04-2.qcow2', fmt=qcow2 size=1073741824 clust
er_size=65536 lazy_refcounts=off refcount_bits=16
[root@PacktVM01 ~]# virt-xml MasteringKVM04 --add-device --disk /var/lib/libvirt
/images/04-2.qcow2,format=qcow2,size=1
Domain 'MasteringKVM04' defined successfully.
WARNING XML did not change after domain define. You may have changed a value th
at libvirt is setting by default.
[root@PacktVM01 ~]# virsh domblklist MasteringKVM04 --details
Type      Device      Target      Source
-----
file      disk        vda         /var/lib/libvirt/images/MasteringKVM04.qcow2
file      disk        vdb         /var/lib/libvirt/images/04-2.qcow2

[root@PacktVM01 ~]#

```

Figure 7.9 – Adding a thin-provision QEMU copy-on-write (qcow2) format virtual disk to a VM

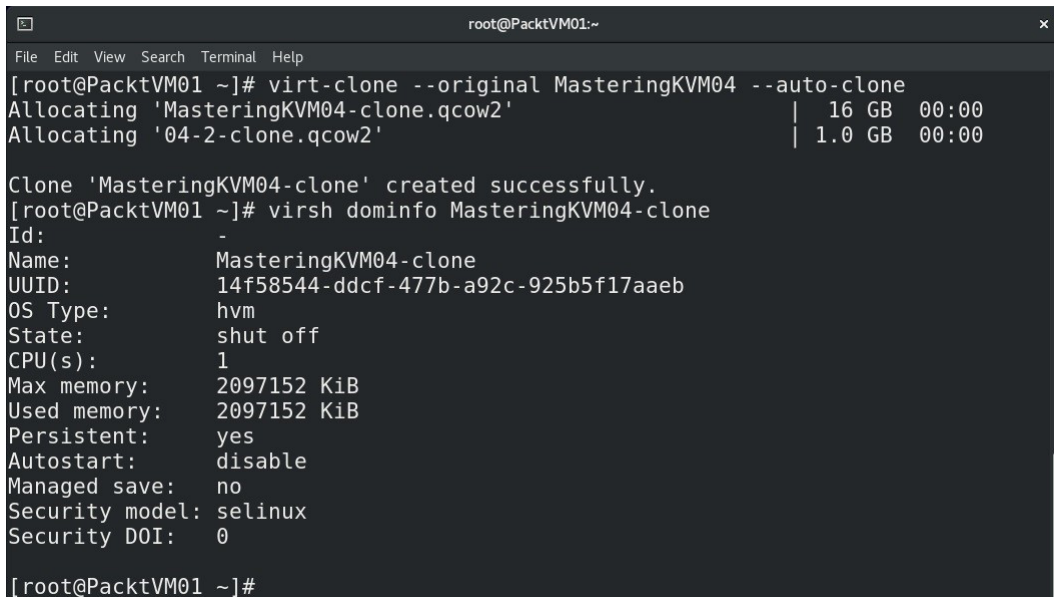
As we can see, `virt-xml` is quite useful. By using it, we added another virtual disk to our VM, and that's one of the simplest things that it can do. We can use it to deploy any additional piece of VM hardware to an existing VM. We can also use it to edit a VM configuration, which is really handy in larger environments, especially when you have to script and automate such procedures.

virt-clone

Let's now check `virt-clone` by using a couple of examples. Let's say we just want a quick and easy way to clone an existing VM without any additional hassle. We can do the following:

```
# virt-clone --original VirtualMachineName --auto-clone
```

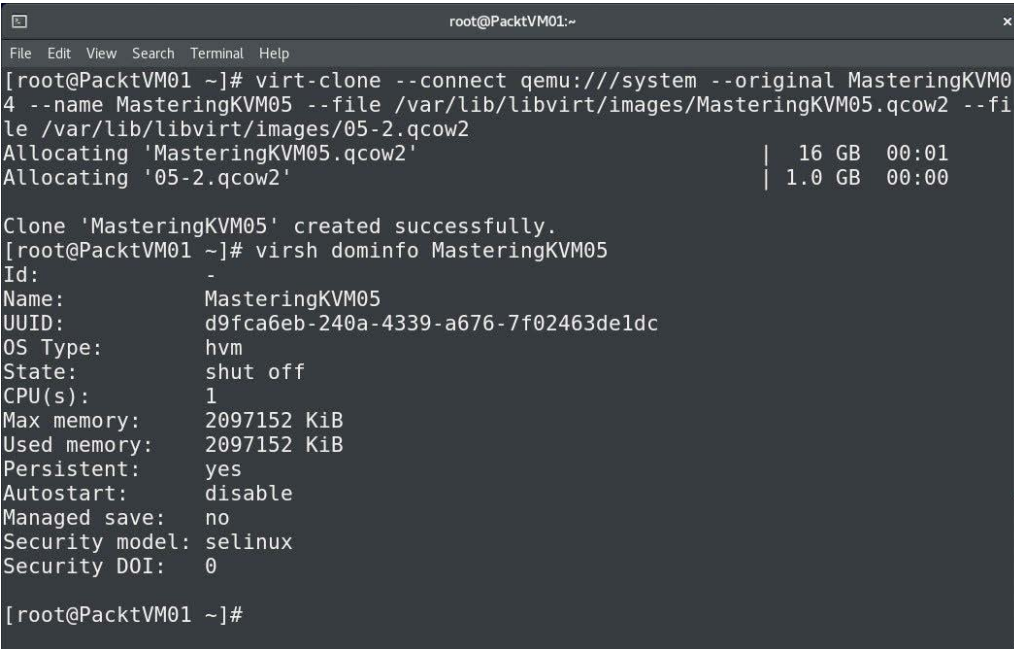
As a result, this will produce a VM named `VirtualMachineName-clone` that we can start using right away. Let's see this in action, as follows:



```
root@PacktVM01:~  
File Edit View Search Terminal Help  
[root@PacktVM01 ~]# virt-clone --original MasteringKVM04 --auto-clone  
Allocating 'MasteringKVM04-clone.qcow2' | 16 GB 00:00  
Allocating '04-2-clone.qcow2' | 1.0 GB 00:00  
  
Clone 'MasteringKVM04-clone' created successfully.  
[root@PacktVM01 ~]# virsh dominfo MasteringKVM04-clone  
Id: -  
Name: MasteringKVM04-clone  
UUID: 14f58544-ddcf-477b-a92c-925b5f17aueb  
OS Type: hvm  
State: shut off  
CPU(s): 1  
Max memory: 2097152 KiB  
Used memory: 2097152 KiB  
Persistent: yes  
Autostart: disable  
Managed save: no  
Security model: selinux  
Security DOI: 0  
  
[root@PacktVM01 ~]#
```

Figure 7.10 – Creating a VM clone with `virt-clone`

Let's see how this could be a bit more *customized*. By using `virt-clone`, we are going to create a VM named `MasteringKVM05`, by cloning a VM named `MasteringKVM04`, and we are going to customize virtual disk names as well, as illustrated in the following screenshot:

A terminal window titled 'root@PacktVM01:~' showing the execution of the 'virt-clone' command. The command clones 'MasteringKVM04' to 'MasteringKVM05' with custom disk names. It shows progress for allocating 'MasteringKVM05.qcow2' (16 GB) and '05-2.qcow2' (1.0 GB). After successful creation, the 'virsh dominfo MasteringKVM05' command is run, displaying VM details such as Name, UUID, OS Type, State, CPU(s), Max memory, Used memory, Persistent, Autostart, Managed save, Security model, and Security DOI.

```
root@PacktVM01:~# virt-clone --connect qemu:///system --original MasteringKVM04 --name MasteringKVM05 --file /var/lib/libvirt/images/MasteringKVM05.qcow2 --file /var/lib/libvirt/images/05-2.qcow2
Allocating 'MasteringKVM05.qcow2' | 16 GB 00:01
Allocating '05-2.qcow2' | 1.0 GB 00:00

Clone 'MasteringKVM05' created successfully.
[root@PacktVM01 ~]# virsh dominfo MasteringKVM05
Id:
Name: MasteringKVM05
UUID: d9fca6eb-240a-4339-a676-7f02463de1dc
OS Type: hvm
State: shut off
CPU(s): 1
Max memory: 2097152 KiB
Used memory: 2097152 KiB
Persistent: yes
Autostart: disable
Managed save: no
Security model: selinux
Security DOI: 0

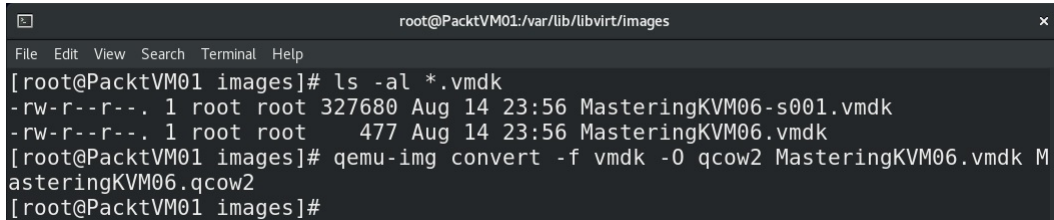
[root@PacktVM01 ~]#
```

Figure 7.11 – Customized VM creation: customizing VM names and virtual hard disk filenames

There are situations in real life that require you to convert VMs from one virtualization technology to another. The bulk of that work is actually converting the VM disk format from one format to another. That's what `virt-convert` is all about. Let's learn how it does its job.

qemu-img

Let's now check how we will convert a virtual disk to another format, and how we will convert a VM *configuration file* from one virtualization method to another. We will use an empty VMware VM as a source and convert its vmdk virtual disk and .vmx file to a new format, as illustrated in the following screenshot:

A terminal window titled 'root@PacktVM01:/var/lib/libvirt/images' showing the execution of the 'qemu-img convert' command. The terminal output shows the current directory contents and the successful conversion of 'MasteringKVM06.vmdk' to 'MasteringKVM06.qcow2'.

```
root@PacktVM01:/var/lib/libvirt/images
File Edit View Search Terminal Help
[root@PacktVM01 images]# ls -al *.vmdk
-rw-r--r--. 1 root root 327680 Aug 14 23:56 MasteringKVM06-s001.vmdk
-rw-r--r--. 1 root root   477 Aug 14 23:56 MasteringKVM06.vmdk
[root@PacktVM01 images]# qemu-img convert -f vmdk -O qcow2 MasteringKVM06.vmdk M
asteringKVM06.qcow2
[root@PacktVM01 images]#
```

Figure 7.12 – Converting VMware virtual disk to qcow2 format for KVM

If we are faced with projects that involve moving or converting VMs between these platforms, we need to make sure that we use these utilities as they are easy to use and understand and only require one thing—a bit of time. For example, if we have a 1 **terabyte (TB)** VMware virtual disk (**VM Disk (VMDK)** and flat VMDK file), it might take hours for that file to be converted to qcow2 format, so we have to be patient. Also, we need to be prepared to edit vmx configuration files from time to time as the conversion process from vmx to kvm format isn't 100% smooth, as we might expect it to be. During the course of this process, a new configuration file is created. The default directory for KVM VM configuration files is `/etc/libvirt/qemu`, and we can easily see **Extensible Markup Language (XML)** files in that directory—these are our KVM VM configuration files. Filenames represent VM names from the `virsh` list output.

There are also some new utilities in CentOS 8 that will make it easier for us to manage not only the local server but also VMs. The Cockpit web interface is one of those—it has the capability to do basic VM management on a KVM host. All we need to do is connect to it via a web browser, and we mentioned this web application in *Chapter 3, Installing a Kernel-based VM (KVM) Hypervisor, libvirt, and ovirt*, when discussing the deployment of oVirt appliances. So, let's familiarize ourselves with VM management by using Cockpit.

Creating a new VM using Cockpit

To use Cockpit for the management of our server and its VMs, we need to install and start Cockpit and its additional packages. Let's start with that, as follows:

```
yum -y install cockpit*
systemctl enable --now cockpit.socket
```

After this, we can start Firefox and point it to `https://kvm-host:9090/`, as this is the default port where Cockpit can be reached, and log in as `root` with the root password, which will give us the following **user interface (UI)**:

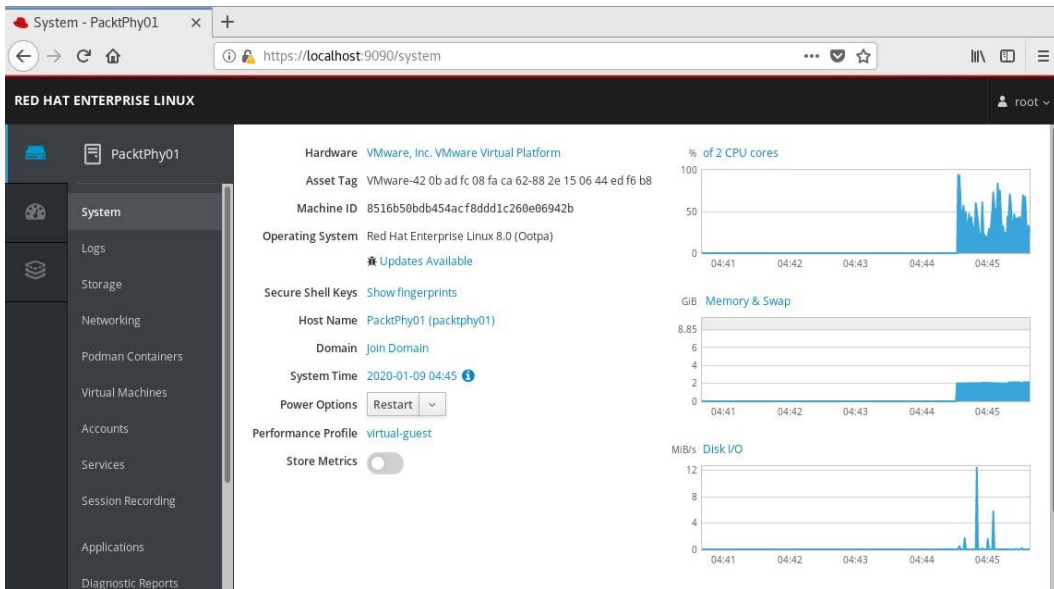


Figure 7.14 – Cockpit web console, which we can use to deploy VMs

In the previous step, when we installed `cockpit*`, we also installed `cockpit-machines`, which is a plugin for the Cockpit web console that enables us to manage `libvirt` VMs in the Cockpit web console. So, after we click on **VMs**, we can easily see all of our previously installed VMs, open their configuration, and install new VMs via a simple wizard, as illustrated in the following screenshot:

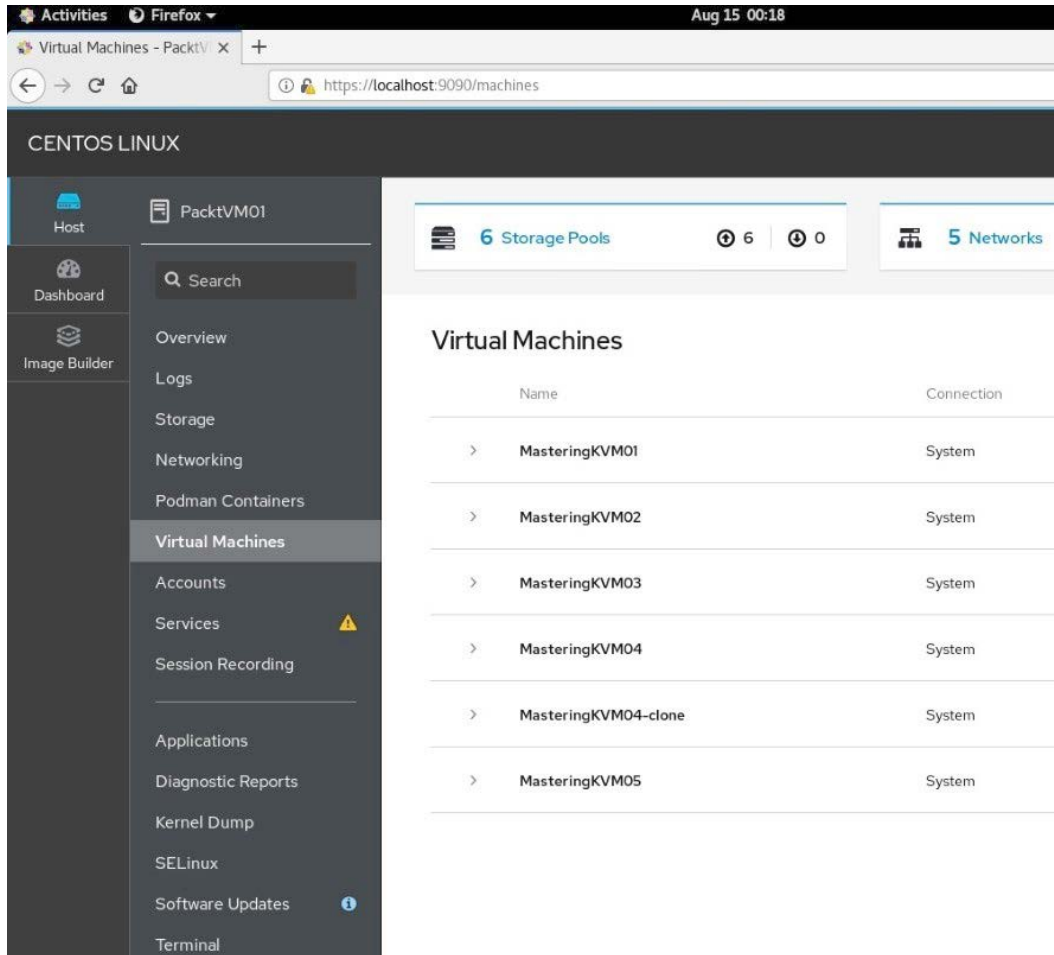


Figure 7.15 – Cockpit VM management

The wizard for VM installation is really simple—we just need to configure basic settings for our new VM and we can start installing, as follows:

Create New Virtual Machine ✕

Name

Installation Type

Installation Source

Operating System

Storage

Size 10

Memory 2

Immediately Start VM

Figure 7.16 – Installing KVM VM from Cockpit web console

Now that we covered how we can install VMs *locally*—meaning without some sort of centralized management application—let's go back and check how we can install VMs via oVirt.

Creating a new VM using oVirt

If we added a host to oVirt, when we log in to it, we can go to **Compute-VMs** and start deploying VMs by using a simple wizard. So, after clicking on the **New** button in that menu, we can do just that, and we will be taken to the following screen:

Figure 7.17 – New VM wizard in oVirt

Having in mind that oVirt is a centralized management solution for KVM hosts, we have *loads* of additional options when compared to local VM installation on a KVM host—we can select a cluster that will host this VM; we can use a template, configure the optimization and instance type, configure **high availability (HA)**, resource allocation, boot options... basically, it's what we jokingly refer to as *option paralysis*, although it's for our own benefit, as centralized solutions will always be a bit different than any kind of local solution.

At a minimum, we will have to configure general VM properties—name, operating system, and VM network interface. Then, we will move to the **System** tab, where we will configure memory size and virtual CPU count, as illustrated in the following screenshot:

The screenshot shows the 'New Virtual Machine' dialog box with the 'System' tab selected. The left sidebar contains the following tabs: General, System (selected), Initial Run, Console, Host, High Availability, Resource Allocation, Boot Options, Random Generator, Custom Properties, Icon, Foreman/Satellite, and Affinity Labels. The main area displays the following configuration options:

Property	Value
Cluster	Datacenter
Template	Blank (0)
Operating System	Linux
Instance Type	Small
Optimized for	Server
Memory Size	2048 MB
Maximum memory	8192 MB
Physical Memory Guaranteed	2048 MB
Total Virtual CPUs	1
Advanced Parameters	General
Hardware Clock Time Offset	default: (GMT+00:00) GMT Standard Time
Provide custom serial number policy	<input type="checkbox"/>

At the bottom of the dialog, there is a 'Hide Advanced Options' button on the left and 'OK' and 'Cancel' buttons on the right.

Figure 7.18 – Selecting VM configuration: virtual CPUs and memory

We will definitely want to configure boot options—attach a CD/ISO, add a virtual hard disk, and configure the boot order, as illustrated in the following screenshot:

The screenshot shows the 'New Virtual Machine' configuration window with the 'Boot Options' tab selected. The configuration is as follows:

- General:** Cluster: Datacenter
- System:** Data Center: Primar
- Initial Run:** Template: Blank | (0)
- Console:** Operating System: Linux
- Host:** Instance Type: Custom; Optimized for: Server
- High Availability:** (Collapsed)
- Resource Allocation:** (Collapsed)
- Boot Options:**
 - Boot Sequence:
 - First Device: Hard Disk
 - Second Device: [None]
 - Attach CD: CentOS-7-x86_64-NetInstall-1908.iso
 - Enable menu to select boot device
 - Linux Boot Options:
 - kernel path: [Empty field]
 - initrd path: [Empty field]
 - kernel parameters: [Empty field]
- Random Generator:** (Collapsed)
- Custom Properties:** (Collapsed)
- Icon:** (Collapsed)
- Foreman/Satellite:** (Collapsed)
- Affinity Labels:** (Collapsed)

Buttons at the bottom: Hide Advanced Options, OK, Cancel.

Figure 7.19 – Configuring VM boot options in oVirt

We can customize our VM post-installation by using `sysprep` or `cloud-init`, which we will discuss in *Chapter 9, Customizing a VM with cloud-init*.

Here's what the basic configuration in oVirt looks like:

Figure 7.20 – Installing KVM VM from oVirt: make sure that you select correct boot options

Realistically, if you're managing an environment that has more than two to three KVM hosts, you'll want to use some kind of centralized utility to manage them. oVirt is really good for that, so don't skip it.

Now that we have done the whole deployment procedure in a variety of different ways, it's time to think about the VM configuration. Keeping in mind that a VM is an object that has many important attributes—such as the number of virtual CPUs, amount of memory, virtual network cards, and so on—it's very important that we learn how to customize the VM settings. So, let's make that our next topic.

Configuring your VM

When we were using `virt-manager`, if you go all the way to the last step, there's an interesting option that you could've selected, which is the **Customize configuration before install** option. The same configuration window can be accessed if you check the VM configuration post-install. So, whichever way we go, we'll be faced with the full scale of configuration options for every VM hardware device that was assigned to the VM we just created, as can be seen in the following screenshot:

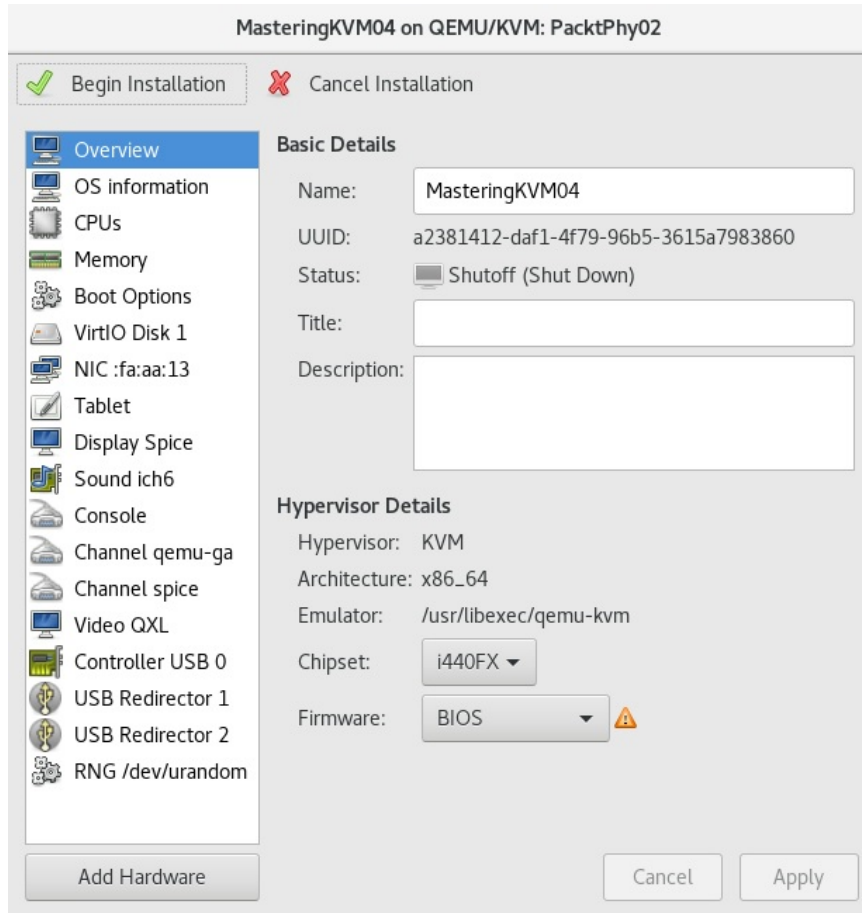


Figure 7.21 – VM configuration options

For example, if we click on the **CPUs** option on the left-hand side, you will see the number of available CPUs (current and maximum allocation), and we'll also see some pretty advanced options such as **CPU topology (Sockets/Cores/Threads)**, which enables us to configure specific **non-uniform memory access (NUMA)** configuration options. Here's what that configuration window looks like:

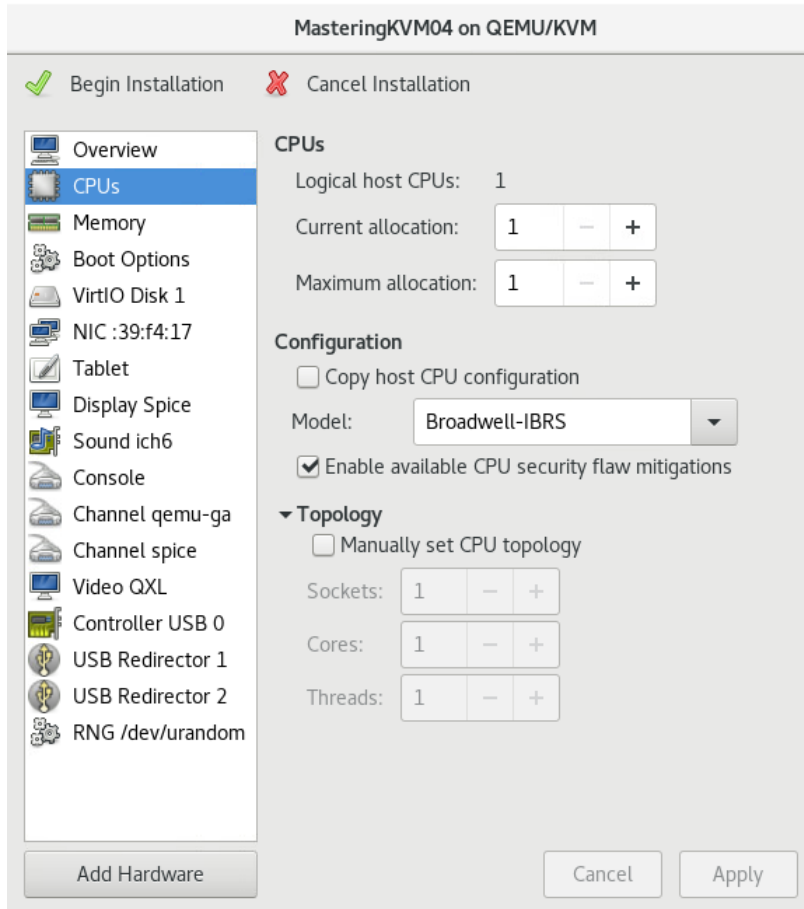


Figure 7.22 – VM CPU configuration

This is a *very* important part of VM configuration, especially if you're designing an environment that hosts loads of virtualized servers. Furthermore, it becomes even more important if virtualized servers host **input/output (I/O)**-intensive applications such as databases. If you want to learn more about this, you can check a link at the end of this chapter, in the *Further reading* section, as it will give you loads of additional information about VM design.

Then, if we open the **Memory** option, we can change memory allocation—again, in floating terms (current and maximum allocation). We'll discuss these options a bit later when we start working with `virt-*` commands. This is what a `virt-manager` **Memory** configuration option looks like:

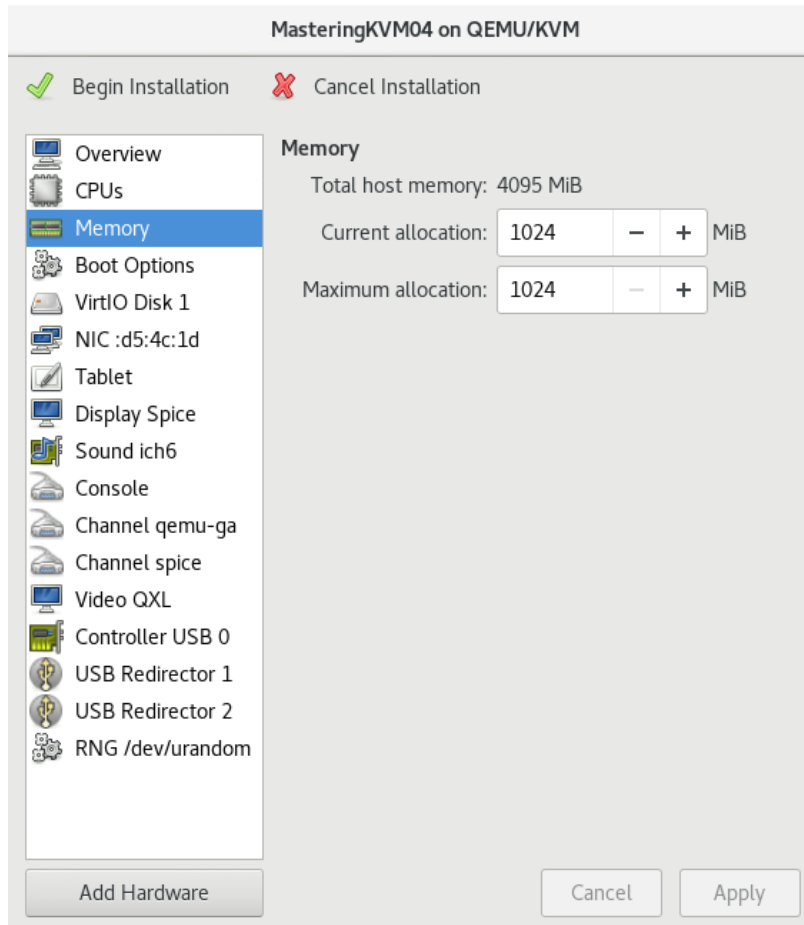


Figure 7.23 – VM memory configuration

One of the most important configuration option sets available in `virt-manager` is located in the **Boot Options** sub-menu, which is shown in the following screenshot:

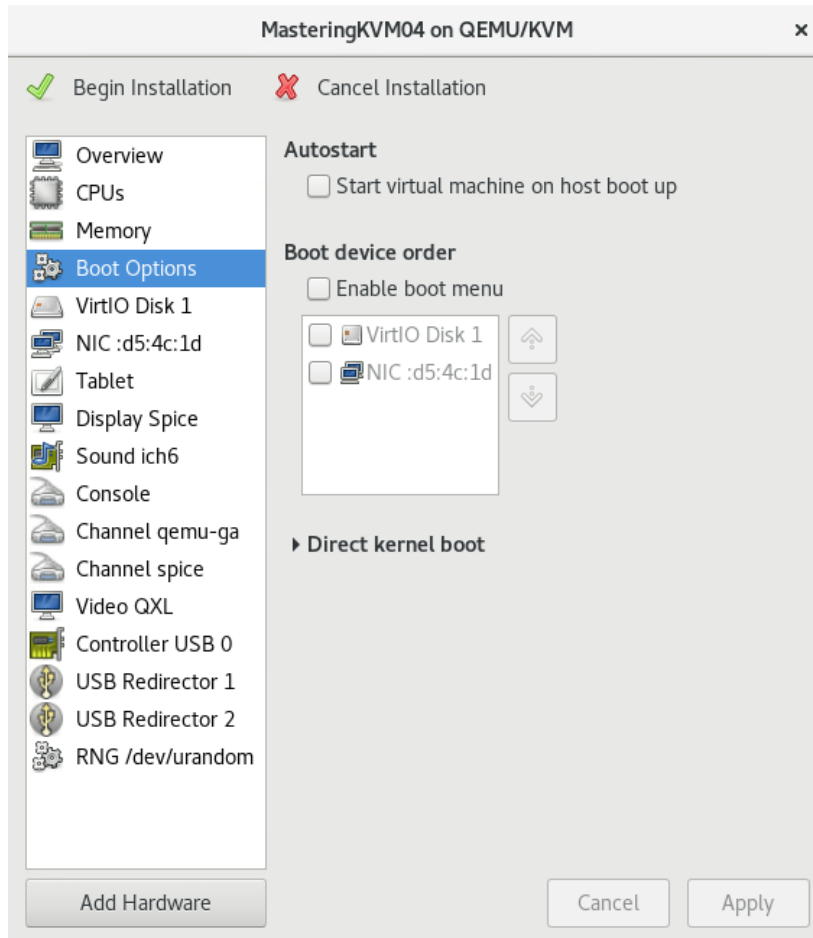


Figure 7.24 – VM boot configuration options

There, you can do two very important things, as follows:

- Select this VM to be auto-started with the host
- Enable the boot menu and select a boot device and boot device priorities

In terms of configuration options, by far the most feature-rich configuration menu for `virt-manager` is the virtual storage menu—in our case, **VirtIO Disk 1**. If we click on that, we're going to get the following selection of configuration options:

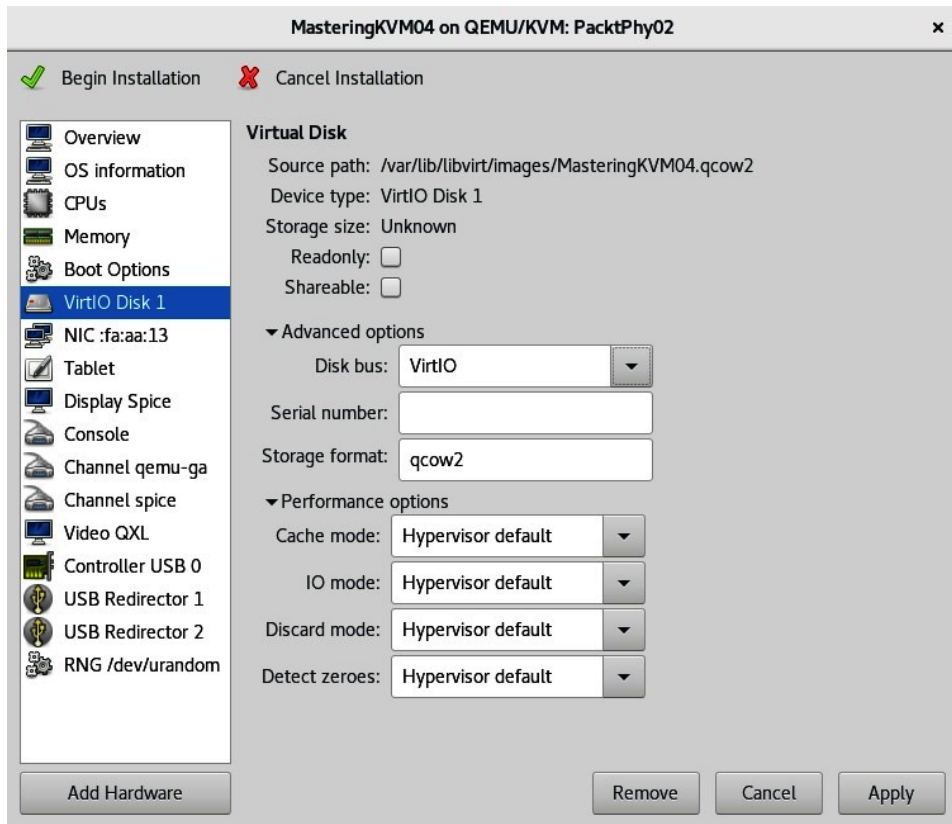


Figure 7.25 – Configuring VM hard disk and storage controller options

Let's see what the significance of some of these configuration options is, as follows:

- **Disk bus**—There are usually five options here, **VirtIO** being the default (and the best) one. Just as with VMware, ESXi, and Hyper-V, KVM has different virtual storage controllers available. For example, VMware has BusLogic, LSI Logic, Paravirtual, and other types of virtual storage controllers, while Hyper-V has the **integrated drive electronics (IDE)** and **small computer system interface (SCSI)** controllers. This option defines the storage controller that the VM is going to see inside its guest operating system.

- **Storage format**—There are two formats: `qcow2` and `raw` (`dd` type format). The most common option is `qcow2` as it offers the most flexibility for VM management—for example, it supports thin provisioning and snapshots.
- **Cache mode**—There are six types: `writethrough`, `writeback`, `directsync`, `unsafe`, `none`, and `default`. These modes explain how data gets written from an I/O that originated from the VM to the storage underlay below the VM. For example, if we're using `writethrough`, the I/O gets cached on the KVM host and is written through to the VM disk as well. On the other hand, if we're using `none`, there's no caching on the host (except for the disk `writeback` cache), and data gets written to the VM disk directly. Different modes have different pros and cons, but generally, `none` is the best option for VM management. You can read more about them in the *Further reading* section.
- **IO mode**—There are two modes: `native` and `threads`. Depending on this setting, the VM I/O will be either written via kernel asynchronous I/O or via pool of threads in the user space (which is the default value, as well). When working with `qcow2` format, it's generally accepted that `threads` mode is better as `qcow2` format first allocates sectors and then writes to them, which will hog vCPUs allocated to the VM and have direct influence on I/O performance.
- **Discard mode**—There are two available modes here, called `ignore` and `unmap`. If you select `unmap`, when you delete files from your VM (which translates to free space in your `qcow2` VM disk file), the `qcow2` VM disk file will shrink to reflect the newly freed capacity. Depending on which Linux distribution, kernel, and kernel patches you have applied and the **Quick Emulator (QEMU)** version, this function *might* only be available on a SCSI disk bus. It's supported for QEMU version 4.0+.
- **Detect zeroes**—There are three modes available: `off`, `on`, and `unmap`. If you select `unmap`, zero write will be translated as an unmapping operation (as explained in discard mode). If you set it to `on`, zero writes by the operating system will be translated to specific zero write commands.

During the lifespan of any given VM, there's a significant chance that we will reconfigure it. Whether that means adding or removing virtual hardware (of course, usually, it's adding), it's an important aspect of a VM's life cycle. So, let's learn how to manage that.

Adding and removing virtual hardware from your VM

By using the VM configuration screen, we can easily add additional hardware, or remove hardware as well. For example, if we click on the **Add Hardware** button in the bottom-left corner, we can easily add a device—let's say, a virtual network card. The following screenshot illustrates this process:

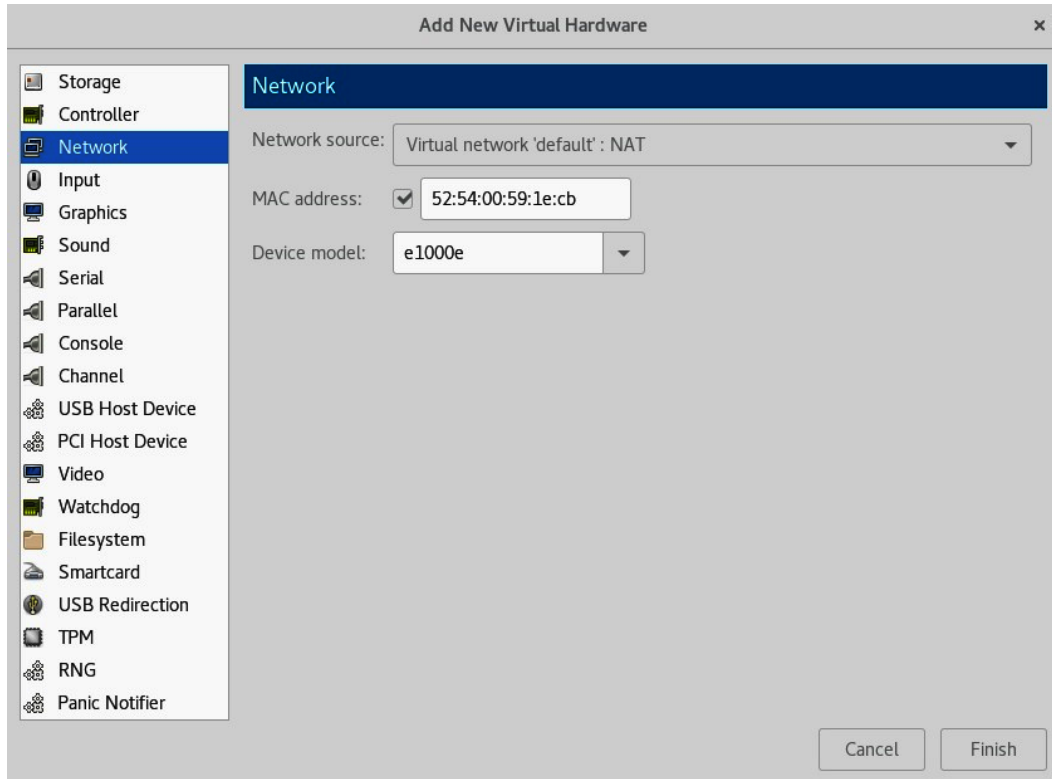


Figure 7.26 – After clicking on Add Hardware, we can select which virtual hardware device we want to add to our VM

On the other hand, if we select a virtual hardware device (for example, **Sound ich6**) and press the **Remove** button that will then appear, we can also remove this virtual hardware device, after confirming that we want to do so, as illustrated in the following screenshot:

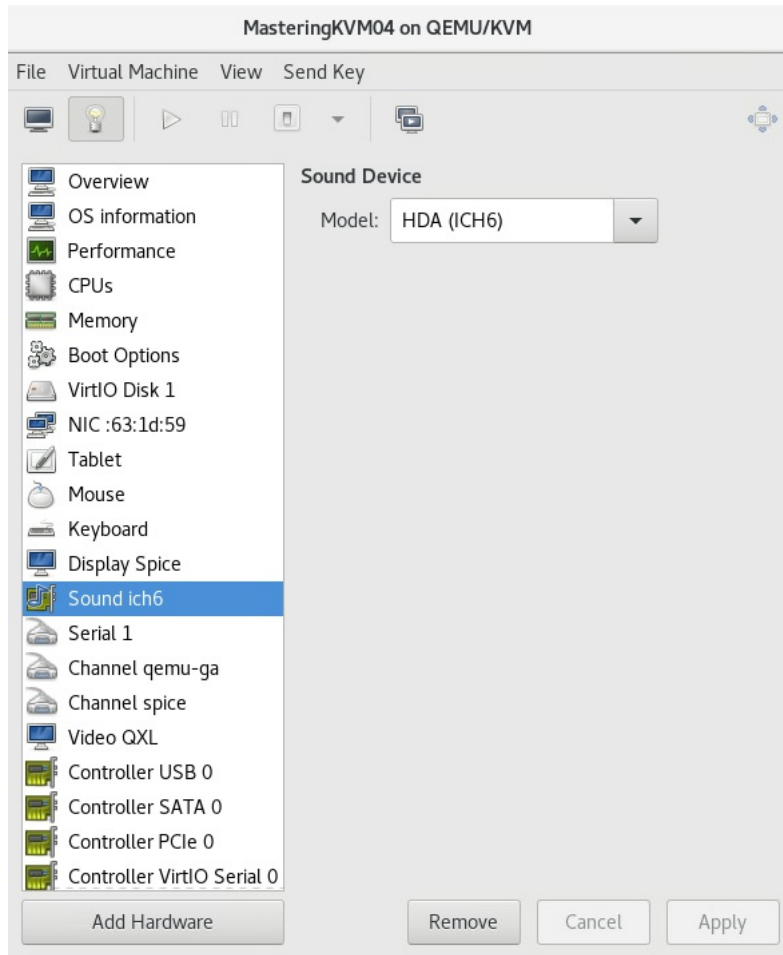


Figure 7.27 – Process for removing a VM hardware device: select it on the left-hand side and click Remove

As you can see, adding and removing VM hardware is as easy as one-two-three. We actually touched on the subject before, when we were working with virtual networking and storage (*Chapter 4, Libvirt Networking*), but there, we used shell commands and XML file definitions. Check out those examples if you want to learn more about that.

Virtualization is all about flexibility, and being able to place VMs on any given host in our environment is a huge part of that. Having that in mind, VM migration is one of the features in virtualization that can be used as a marketing poster for virtualization and its many advantages. What is VM migration all about? That's what we're going to learn next.

Migrating VMs

In simple terms, migration enables you to move your VM from one physical machine to another physical machine, with a very minimal downtime or no downtime. We can also move VM storage, which is a resource-hog type of operation that needs to be carefully planned and—if possible —executed after hours so that it doesn't affect other VMs' performance as much as it could.

There are various different types of migration, as follows:

- Offline (cold)
- Online (live)
- Suspended migration

There are also various different types of online migrations, depending on what you're moving, as follows:

- The compute part of the VM (moving the VM from one KVM host to another KVM host)
- The storage part of the VM (moving VM files from one storage pool to another storage pool)
- Both (moving the VM from host to host and storage pool to storage pool at the same time)

There are some differences in terms of which migration scenarios are supported if you're using just a plain KVM host versus oVirt or Red Hat Enterprise Virtualization. If you want to do a live storage migration, you can't do it on a KVM host directly, but you can easily do it if the VM is shut down. If you need a live storage migration, you will have to use oVirt or Red Hat Enterprise Virtualization.

We discussed **single-root input-output virtualization (SR-IOV)**, **Peripheral Component Interconnect (PCI)** device passthrough, **virtual graphics processing units (vGPUs)**, and similar concepts as well (in *Chapter 2, KVM as a Virtualization Solution*, and *Chapter 4, Libvirt Networking*). In CentOS 8, you can't live-migrate a VM that has either one of these options assigned to a running VM.

Whatever the use case is, we need to be aware of the fact that migration needs to be performed either as the `root` user or as a user that belongs to the `libvirt` user group (what Red Hat refers to as `system` versus `user libvirt` session).

There are different reasons why VM migration is a valuable tool to have in your arsenal. Some of these reasons are obvious; others, less so. Let's try to explain different use cases for VM migration and its benefits.

Benefits of VM migration

The most important benefits of VM live migration are listed as follows:

- **Increased uptime and reduced downtime**—A carefully designed virtualized environment will give you the maximum uptime for your application.
- **Saving energy and going green**—You can easily consolidate your VMs based on their load and usage to a smaller number of hypervisors during off hours. Once the VMs are migrated, you can power off the unused hypervisors.
- **Easy hardware/software upgrade process by moving your VM between different hypervisors**—Once you have the capability to move your VMs freely between different physical servers, the benefits are countless.

VM migration needs proper planning to be put in place. There are some basic requirements the migration looks for. Let's see them one by one.

The migration requirements for production environments are the following:

- The VM should be using a storage pool that is created on a shared storage.
- The name of the storage pool and the virtual disk's path should remain the same on both hypervisors (source and destination hypervisors).

Check out *Chapter 4, Libvirt Networking*, and *Chapter 5, Libvirt Storage*, to remind yourself how to create a storage pool using shared storage.

There are, as always, some rules that apply here. These are rather simple, so we need to learn them before starting migration processes. They are as follows:

- It is possible to do a live storage migration using a storage pool that is created on non-shared storage. You only need to maintain the same storage pool name and file location, but shared storage is still recommended in a production environment.
- If there is an unmanaged virtual disk attached to a VM that uses a **Fiber Channel (FC)**, an **Internet Small Computer Systems Interface (iSCSI)**, **Logical Volume Manager (LVM)**, and so on, the same storage should be available on both hypervisors.
- The virtual networks used by the VMs should be available on both hypervisors.

- A bridge that is configured for a networking communication should be available on both the hypervisors.
- Migration may fail if the major versions of `libvirt` and `qemu-kvm` on the hypervisors are different, but you should be able to migrate the VMs running on a hypervisor that has a lower version of `libvirt` or `qemu-kvm` to a hypervisor that has higher versions of those packages, without any issues.
- The time on both the source and destination hypervisors should be synced. It is highly recommended that you sync the hypervisors using the same **Network Time Protocol (NTP)** or **Precision Time Protocol (PTP)** servers.
- It is important that the systems use a **Domain Name System (DNS)** server for name resolution. Adding the host details on `/etc/hosts` will not work. You should be able to resolve the hostnames using the `host` command.

There are some pre-requisites that we need to have in mind when planning our environment for VM migration. For the most part, these pre-requisites are mostly the same for all virtualization solutions. Let's discuss these pre-requisites and, in general, how to set up our environment for VM migration next.

Setting up the environment

Let's build the environment to do VM migration—both offline and live migrations. The following diagram depicts two standard KVM virtualization hosts running VMs with a shared storage:

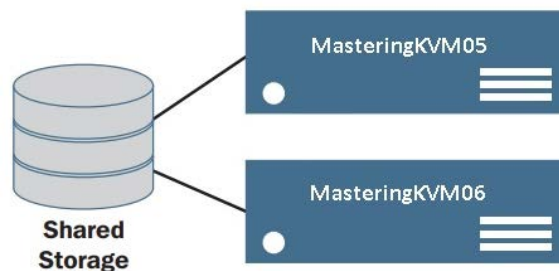


Figure 7.28 – VMs on shared storage

We start this by setting up a shared storage. In this example, we are using **Network File System (NFS)** for the shared storage. We are going to use NFS because it is simple to set up, thus helping you to follow the migration examples easily. In actual production, it is recommended to use iSCSI-based or FC-based storage pools. NFS is not a good choice when the files are large and the VM performs heavy I/O operations. Gluster is a good alternative to NFS, and we would recommend that you try it. Gluster is well integrated in `libvirt`.

We're going to create a NFS share on CentOS 8 server. It's going to be hosted in `/testvms` directory, which we're going to export via NFS. The name of the server is `nfs-01`. (in our case, IP address of `nfs-01` is `192.168.159.134`)

1. The first step is creating and exporting the `/testvms` directory from `nfs-01` and turning off SELinux (check *Chapter 5, Libvirt Storage, Ceph* section to see how):

```
# mkdir /testvms
# echo '/testvms *(rw,sync,no_root_squash)' >> /etc/exports
```

2. Then, allow the NFS service in the firewall by executing the following code:

```
# firewall-cmd --get-active-zones
public
interfaces: ens33
# firewall-cmd --zone=public --add-service=nfs
# firewall-cmd --zone=public --list-all
```

3. Start the NFS service, as follows:

```
# systemctl start rpcbind nfs-server
# systemctl enable rpcbind nfs-server
# showmount -e
```

4. Confirm that the share is accessible from your KVM hypervisors. In our case, it is `PacktPhy01` and `PacktPhy02`. Run the following code:

```
# mount 192.168.159.134:/testvms /mnt
```

5. If mounting fails, reconfigure the firewall on the NFS server and recheck the mount. This can be done by using the following commands:

```
firewall-cmd --permanent --zone=public --add-service=nfs
firewall-cmd --permanent --zone=public
--add-service=mountd
firewall-cmd --permanent --zone=public --add-service=rpc-
bind
firewall-cmd -- reload
```

6. Unmount the volume once you have verified the NFS mount point from both hypervisors, as follows:

```
# umount /mnt
```


7. On PacktPhy01 and PacktPhy02, create a storage pool named `testvms`, as follows:

```
# mkdir -p /var/lib/libvirt/images/testvms/  
# virsh pool-define-as --name testvms --type netfs  
--source-host 192.168.159.134 --source-path /testvms  
--target /var/lib/libvirt/images/testvms/  
# virsh pool-start testvms  
# virsh pool-autostart testvms
```

The `testvms` storage pool is now created and started on two hypervisors.

In this next example, we are going to isolate the migration and VM traffic. It is highly recommended that you do this isolation in your production environment, especially if you do a lot of migrations, as it will offload that demanding process to a separate network interface, thus freeing other congested network interfaces. So, there are two main reasons for this, as follows:

- **Network performance:** The migration of a VM uses the full bandwidth of the network. If you use the same network for the VM traffic network and the migration network, the migration will choke that network, thus affecting the servicing capability of the VM. You can control the migration bandwidth, but it will increase the migration time.

Here is how we create the isolation:

```
PacktPhy01 -- ens36 (192.168.0.5) <---switch-----> ens36  
(192.168.0.6) -- PacktPhy02  
ens37 -> br1 <-----switch-----> ens37 -> br1
```

`ens192` interfaces on `PacktPhy01` and `PacktPhy02` are used for migration as well as administrative tasks. They have an IP assigned and connected to a network switch. A `br1` bridge is created using `ens224` on both `PacktPhy01` and `PacktPhy02`. `br1` does not have an IP address assigned and is used exclusively for VM traffic (uplink for the switch that the VMs are connected to). It is also connected to a (physical) network switch.

- **Security reasons:** It is always recommended that you keep your management network and virtual network isolated for security reasons, as well. You don't want your users to mess with your management network, where you access your hypervisors and do the administration.

We will discuss three of the most important scenarios— offline migration, non-live migration (suspended), and live migration (online). Then, we will discuss storage migration as a separate scenario that requires additional planning and forethought.

Offline migration

As the name suggests, during offline migration, the state of the VM will be either shut down or suspended. The VM will be then resumed or started at the destination host. In this migration model, `libvirt` will just copy the VM's XML configuration file from the source to the destination KVM host. It also assumes that you have the same shared storage pool created and ready to use at the destination. As the first step in the migration process, you need to set up two-way passwordless SSH authentication on the participating KVM hypervisors. In our example, they are called `PacktPhy01` and `PacktPhy02`.

For the following exercises, disable **Security-Enhanced Linux (SELinux)** temporarily.

In `/etc/sysconfig/selinux`, use your favorite editor to modify the following line of code:

```
SELINUX=enforcing
```

This needs to be modified as follows:

```
SELINUX=permissive
```

Also, in the command line, as `root`, we need to temporarily set SELinux mode to `permissive`, as follows:

```
# setenforce 0
```

On `PacktPhy01`, as `root`, run the following command:

```
# ssh-keygen
```

```
# ssh-copy-id root@PacktPhy02
```

On `PacktPhy02`, as `root`, run the following commands:

```
# ssh-keygen
```

```
# ssh-copy-id root@PacktPhy01
```

You should now be able to log in to both of these hypervisors as `root` without typing a password.

Let's do an offline migration of `MasteringKVM01`, which is already installed, from `PacktPhy01` to `PacktPhy02`. The general format of the migration command looks similar to the following:

```
# virsh migrate migration-type options name-of-the-vm-destination-uri
```

On PacktPhy01, run the following code:

```
[PacktPhy01] # virsh migrate --offline --verbose --persistent
MasteringKVM01 qemu+ssh://PacktPhy02/system
Migration: [100 %]
```

On PacktPhy02, run the following code:

```
[PacktPhy02] # virsh list --all
# virsh list --all
Id Name State
-----
- MasteringKVM01 shut off
[PacktPhy02] # virsh start MasteringKVM01
Domain MasteringKVM01 started
```

When a VM is on shared storage and you have some kind of issue with one of the hosts, you could also manually register a VM on another host. That means that you might end up in a situation where the same VM is registered on two hypervisors, after you repair the issue on your host that had an initial problem. It's something that happens when you're manually managing KVM hosts without a centralized management platform such as oVirt, which wouldn't allow such a scenario. So, what happens if you're in that kind of situation? Let's discuss this scenario.

What if I start the VM accidentally on both the hypervisors?

Accidentally starting the VM on both the hypervisors can be a sysadmin's nightmare. It can lead to VM filesystem corruption, especially when the filesystem inside the VM is not cluster-aware. Developers of `libvirt` thought about this and came up with a locking mechanism. In fact, they came up with two locking mechanisms. When enabled, these will prevent the VMs from starting at the same time on two hypervisors.

The two locking mechanisms are as follows:

- `lockd`: `lockd` makes use of the POSIX `fcntl()` advisory locking capability. It was started by the `virtlockd` daemon. It requires a shared filesystem (preferably NFS), accessible to all the hosts that share the same storage pool.
- `sanlock`: This is used by oVirt projects. It uses a disk `paxos` algorithm for maintaining continuously renewed leases.

For `libvirt`-only implementations, we prefer `lockd` over `sanlock`. It is best to use `sanlock` for oVirt.

Enabling lockd

For image-based storage pools that are POSIX-compliant, you can enable `lockd` easily by uncommenting the following command in `/etc/libvirt/qemu.conf` or on both hypervisors:

```
lock_manager = "lockd"
```

Now, enable and start the `virtlockd` service on both the hypervisors. Also, restart `libvirtd` on both the hypervisors, as follows:

```
# systemctl enable virtlockd; systemctl start virtlockd
# systemctl restart libvirtd
# systemctl status virtlockd
```

Start `MasteringKVM01` on `PacktPhy02`, as follows:

```
[root@PacktPhy02] # virsh start MasteringKVM01
Domain MasteringKVM01 started
```

Start the same `MasteringKVM01` VM on `PacktPhy01`, as follows:

```
[root@PacktPhy01] # virsh start MasteringKVM01
error: Failed to start domain MasteringKVM01
error: resource busy: Lockspace resource '/var/lib/libvirt/
images/ testvms/MasteringKVM01.qcow2' is locked
```

Another method to enable `lockd` is to use a hash of the disk's file path. Locks are saved in a shared directory that is exported through NFS, or similar sharing, to the hypervisors. This is very useful when you have virtual disks that are created and attached using a multipath **logical unit number (LUN)**. `fcntl()` cannot be used in such cases. We recommend that you use the methods detailed next to enable the locking.

On the NFS server, run the following code (make sure that you're not running any virtual machines from this NFS server first!):

```
mkdir /flockd
# echo "/flockd *(rw,no_root_squash)" >> /etc/exports
# systemctl restart nfs-server
# showmount -e
Export list for :
/flockd *
/testvms *
```

Add the following code to both the hypervisors in `/etc/fstab` and type in the rest of these commands:

```
# echo "192.168.159.134:/flockd /var/lib/libvirt/lockd/flockd
nfs rsize=8192,wsiz=8192,timeo=14,intr,sync" >> /etc/fstab
# mkdir -p /var/lib/libvirt/lockd/flockd
# mount -a
# echo 'file_lockspace_dir = "/var/lib/libvirt/lockd/flockd"'
>> /etc/libvirt/qemu-lockd.conf
```

Reboot both hypervisors, and, once rebooted, verify that the `libvirtd` and `virtlockd` daemons started correctly on both the hypervisors, as follows:

```
[root@PacktPhy01 ~]# virsh start MasteringKVM01
Domain MasteringKVM01 started
[root@PacktPhy02 flockd]# ls
36b8377a5b0cc272a5b4e50929623191c027543c4facb1c6f3c35bacaa745
5ef
51e3ed692fdf92ad54c6f234f742bb00d4787912a8a674fb5550b1b826343
dd6
```

MasteringKVM01 has two virtual disks, one created from an NFS storage pool and the other created directly from a LUN. If we try to power it on the PacktPhy02 hypervisor host, MasteringKVM01 fails to start, as can be seen in the following code snippet:

```
[root@PacktPhy02 ~]# virsh start MasteringKVM01
error: Failed to start domain MasteringKVM01
error: resource busy: Lockspace resource
'51e3ed692fdf92ad54c6f234f742bb00d4787912a8a674fb5550b1b82634
3dd6' is locked
```

When using LVM volumes that can be visible across multiple host systems, it is desirable to do the locking based on the **universally unique identifier (UUID)** associated with each volume, instead of their paths. Setting the following path causes `libvirt` to do UUID-based locking for LVM:

```
lvm_lockspace_dir = "/var/lib/libvirt/lockd/lvmvolumes"
```

When using SCSI volumes that can be visible across multiple host systems, it is desirable to do locking based on the UUID associated with each volume, instead of their paths. Setting the following path causes `libvirt` to do UUID-based locking for SCSI:

```
scsi_lockspace_dir = "/var/lib/libvirt/lockd/scsivolumes"
```

As with `file_lockspace_dir`, the preceding directories should also be shared with the hypervisors.

Important note

If you are not able to start VMs due to locking errors, just make sure that they are not running anywhere and then delete the lock files. Start the VM again. We deviated a little from migration for the `lockd` topic. Let's get back to migration.

Live or online migration

In this type of migration, the VM is migrated to the destination host while it's running on the source host. The process is invisible to the users who are using the VMs. They won't even know that the VM they are using has been transferred to another host while they are working on it. Live migration is one of the main features that have made virtualization so popular.

Migration implementation in KVM does not need any support from the VM. It means that you can live-migrate any VMs, irrespective of the operating system they are using. A unique feature of KVM live migration is that it is almost completely hardware-independent. You should ideally be able to live-migrate a VM running on a hypervisor that has an **Advanced Micro Devices (AMD)** processor to an Intel-based hypervisor.

We are not saying that this will work in 100% of the cases or that we in any way recommend having this type of mixed environment, but in most of the cases, it should be possible.

Before we start the process, let's go a little deeper to understand what happens under the hood. When we do a live migration, we are moving a live VM while users are accessing it. This means that users shouldn't feel any disruption in VM availability when you do a live migration.

Live migration is a five-stage, complex process, even though none of these processes are exposed to the sysadmins. `libvirt` will do the necessary work once the VM migration action is issued. The stages through which a VM migration goes are explained in the following list:

1. **Preparing the destination:** When you initiate a live migration, the source `libvirt` (`Slibvirt`) will contact the destination `libvirt` (`Dlibvirt`) with the details of the VM that is going to be transferred live. `Dlibvirt` will pass this information to the underlying QEMU, with relevant options to enable live migration. QEMU will start the actual live migration process by starting the VM in pause mode and will start listening on a **Transmission Control Protocol (TCP)** port for VM data. Once the destination is ready, `Dlibvirt` will inform `Slibvirt`, with the details of QEMU. By this time, QEMU, at the source, is ready to transfer the VM and connects to the destination TCP port.
2. **Transferring the VM:** When we say transferring the VM, we are not transferring the whole VM; only the parts that are missing at the destination are transferred—for example, the memory and the state of the virtual devices (VM state). Other than the memory and the VM state, all other virtual hardware (virtual network, virtual disks, and virtual devices) is available at the destination itself. Here is how QEMU moves the memory to the destination:
 - a) The VM will continue running at the source, and the same VM is started in pause mode at the destination.
 - b) In one go, it will transfer all the memory used by the VM to the destination. The speed of transfer depends upon the network bandwidth. Suppose the VM is using 10 **gibibytes (GiB)**; it will take the same time to transfer 10 GiB of data using the **Secure Copy Protocol (SCP)** to the destination. In default mode, it will make use of the full bandwidth. That is the reason we are separating the administration network from the VM traffic network.
 - c) Once the whole memory is at the destination, QEMU starts transferring the dirty pages (pages that are not yet written to the disk). If it is a busy VM, the number of dirty pages will be high and it will take time to move them. Remember, dirty pages will always be there and there is no state of zero dirty pages on a running VM. Hence, QEMU will stop transferring the dirty pages when it reaches a low threshold (50 or fewer pages).QEMU will also consider other factors, such as iterations, the number of dirty pages generated, and so on. This can also be determined by `migrate-setmaxdowntime`, which is in milliseconds.

3. **Stopping the VM on the source host:** Once the number of dirty pages reaches the said threshold, QEMU will stop the VM on the source host. It will also sync the virtual disks.
4. **Transferring the VM state:** At this stage, QEMU will transfer the state of the VM's virtual devices and remaining dirty pages to the destination as quickly as possible. We cannot limit the bandwidth at this stage.
5. **Continuing the VM:** At the destination, the VM will be resumed from the paused state. Virtual **network interface controllers (NICs)** become active, and the bridge will send out gratuitous **Address Resolution Protocols (ARPs)** to announce the change. After receiving the announcement from the bridge, the network switches will update their respective ARP cache and start forwarding the data for the VM to the new hypervisors.

Note that *Steps 3, 4, and 5* will be completed in milliseconds. If some errors happen, QEMU will abort the migration and the VM will continue running on the source hypervisor. All through the migration process, `libvirt` services from both participating hypervisors will be monitoring the migration process.

Our VM called `MasteringKVM01` is now running on `PacktPhy01` safely, with `lockd` enabled. We are going to live-migrate `MasteringKVM01` to `PacktPhy02`.

We need to open the necessary TCP ports used for migration. You only need to do that at the destination server, but it's a good practice to do this in your whole environment so that you don't have to micro-manage these configuration changes as you need them in the future, one by one. Basically, you have to open the ports on all the participating hypervisors by using the following `firewall-cmd` command for the default zone (in our case, the `public` zone):

```
# firewall-cmd --zone=public --add-port=49152-49216/tcp
--permanent
```

Check the name resolution on both the servers, as follows:

```
[root@PacktPhy01 ~] # host PacktPhy01
PacktPhy01 has address 192.168.159.136
[root@PacktPhy01 ~] # host PacktPhy02
PacktPhy02 has address 192.168.159.135
[root@PacktPhy02 ~] # host PacktPhy01
PacktPhy01 has address 192.168.159.136
[root@PacktPhy02 ~] # host PacktPhy02
PacktPhy02 has address 192.168.159.135
```


Check and verify all the virtual disks attached are available at the destination, on the same path, with the same storage pool name. This is applicable to attached unmanaged (iSCSI and FC LUNs, and so on) virtual disks also.

Check and verify all the network bridges and virtual networks used by the VM available at the destination. After that, we can start the migration process by running the following code:

```
# virsh migrate --live MasteringKVM01 qemu+ssh://PacktPhy02/system --verbose --persistent
Migration: [100 %]
```

Our VM is using only 4,096 **megabytes (MB)** of memory, so all five stages completed in a couple of seconds. The `--persistent` option is optional, but we recommend adding this.

This is the output of `ping` during the migration process (10.10.48.24 is the IP address of MasteringKVM01):

```
# ping 10.10.48.24
PING 10.10.48.24 (10.10.48.24) 56(84) bytes of data.
64 bytes from 10.10.48.24: icmp_seq=12 ttl=64 time=0.338 ms
64 bytes from 10.10.48.24: icmp_seq=13 ttl=64 time=3.10 ms
64 bytes from 10.10.48.24: icmp_seq=14 ttl=64 time=0.574 ms
64 bytes from 10.10.48.24: icmp_seq=15 ttl=64 time=2.73 ms
64 bytes from 10.10.48.24: icmp_seq=16 ttl=64 time=0.612 ms
--- 10.10.48.24 ping statistics ---
17 packets transmitted, 17 received, 0% packet loss, time
16003ms
rtt min/avg/max/mdev = 0.338/0.828/3.101/0.777 ms
```

If you get the following error message, change cache to none on the virtual disk attached:

```
# virsh migrate --live MasteringKVM01 qemu+ssh://PacktPhy02/system --verbose
error: Unsafe migration: Migration may lead to data corruption
if disks use cache != none
# virt-xml MasteringKVM01 --edit --disk target=vda,cache=none
```

target is the disk to change the cache. You can find the target name by running the following command:

```
virsh dumpxml MasteringKVM01
```

You can try a few more options while performing a live migration, as follows:

- `--undefine domain`: Option used to remove a KVM domain from a KVM host.
- `--suspend domain`: Suspends a KVM domain—that is, pauses a KVM domain until we unsuspend it.
- `--compressed`: When we do a VM migration, this option enables us to compress memory. That will mean a faster migration process, based on the `-comp-methods` parameter.
- `--abort-on-error`: If the migration process throws an error, it is automatically stopped. This is a safe default option as it will help in situations where any kind of corruption might happen during the migration process.
- `--unsafe`: Kind of like the polar opposite of the `-abort-on-error` option. This option forces migration at all costs, even in the case of error, data corruption, or any other unforeseen scenario. Be very careful with this option—don't use it often, or in any situation where you want to be 100% sure that VM data consistency is a key pre-requisite.

You can read more about these options in the RHEL 7—Virtualization Deployment and Administration guide (you can find the link in the *Further reading* section at the end of this chapter). Additionally, the `virsh` command also supports the following options:

- `virsh migrate-setmaxdowntime <domain>`: When migrating a VM, it's inevitable that, at times, a VM is going to be unavailable for a short period of time. This might happen—for example—because of the hand-off process, when we migrate a VM from one host to the other, and we're just coming to the point of state equilibrium (that is, when the source and destination host have the same VM content and are ready to remove the source VM from the source host inventory and make it run on the destination host). Basically, a small pause happens as the source VM gets paused and killed, and the destination host VM gets unpaused and continues. By using this command, the KVM stack is trying to estimate how long this stopped phase will last. It's a viable option, especially for VMs that are really busy and are therefore changing their memory content a lot while we're migrating them.

- `virsh migrate-setspeed <domain> bandwidth`: We can treat this as a quasi-**Quality of Service (QoS)** option. By using it, we can set the amount of bandwidth in MiB/s that we're giving to the migration process. This is a very good option to use if our network is busy (for example, if we have multiple **virtual local area networks (VLANs)** going across the same physical network and we have bandwidth limitations because of it. Lower numbers will slow the migration process.
- `virsh migrate-setspeed <domain>`: We can treat this as a *get information* option to the `migrate-setspeed` command, to check which settings we assigned to the `virsh migrate-setspeed` command.

As you can see, migration is a complex process from a technical standpoint, and has multiple different types and loads of additional configuration options that you can use for management purposes. That being said, it's still such an important capability of a virtualized environment that it's very difficult to imagine working without it.

Summary

In this chapter, we covered different ways of creating VMs and configuring VM hardware. We also covered VM migration in detail, and live and offline VM migration. In the next chapter, we will work with VM disks, VM templates, and snapshots. These concepts are very important to understand as they will make your life administering a virtualized environment a lot easier.

Questions

1. Which command-line tools can we use to deploy VMs in `libvirt`?
2. Which GUI tools can we use to deploy VMs in `libvirt`?
3. When configuring our VMs, which configuration aspects should we be careful with?
4. What's the difference between online and offline VM migration?
5. What's the difference between VM migration and VM storage migration?
6. How can we configure bandwidth for the migration process?

Further reading

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

- Managing VMs with virt-manager: <https://virt-manager.org/>
- oVirt—Installing Linux VMs: https://www.ovirt.org/documentation/vmm-guide/chap-Installing_Linux_Virtual_Machines.html
- Cloning VMs: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_virtualization/cloning-virtual-machines_configuring-and-managing-virtualization
- Migrating VMs: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_virtualization/migrating-virtual-machines_configuring-and-managing-virtualization
- Caching: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-blockio-caching
- Influence of NUMA and memory locality on Microsoft SQL Server 2019 performance: https://www.daaam.info/Downloads/Pdfs/proceedings/proceedings_2019/049.pdf
- Virtualization deployment and administration guide: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/index

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:

```
Administrator: Command Prompt
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.

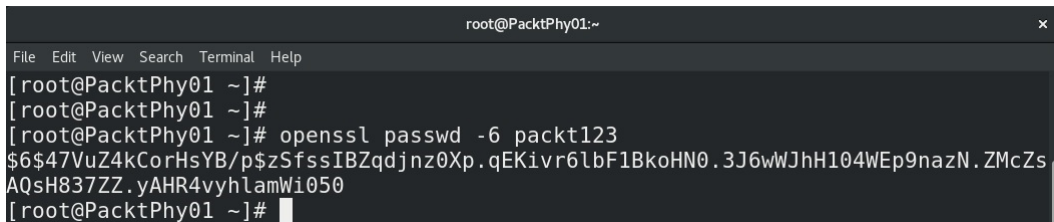
guestfish

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.

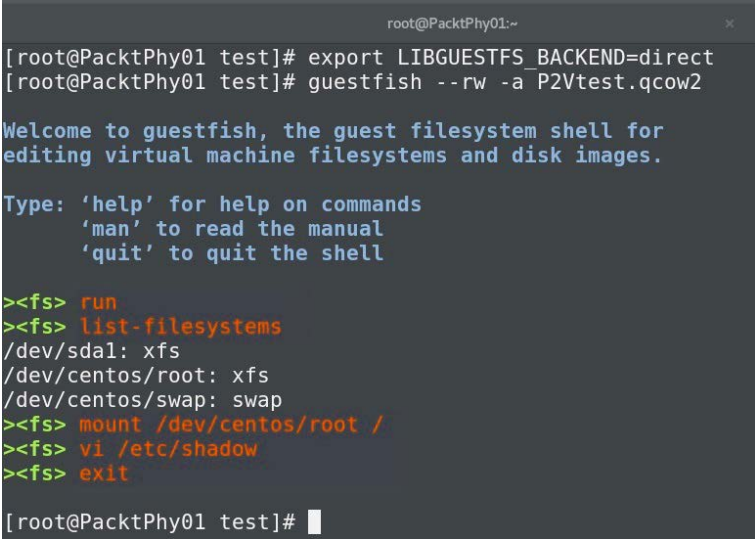
This is an ideal use case for `guestfish`. Let's say that our `qcow2` image is called `template.qcow2`. Let's change the root password to something else—for example, `packt123`. First, we need a hash for that password. The easiest way to do that would be to use `openssl` with the `-6` option (which equals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



```
root@PacktPhy01:~  
File Edit View Search Terminal Help  
[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:~
[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

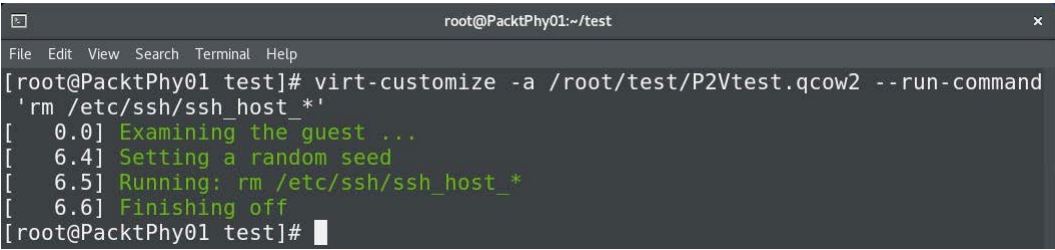
><fs> run
><fs> list-filesystems
/dev/sda1: xfs
/dev/centos/root: xfs
/dev/centos/swap: swap
><fs> mount /dev/centos/root /
><fs> vi /etc/shadow
><fs> exit
[root@PacktPhy01 test]#

```

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 (`guestfish 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:



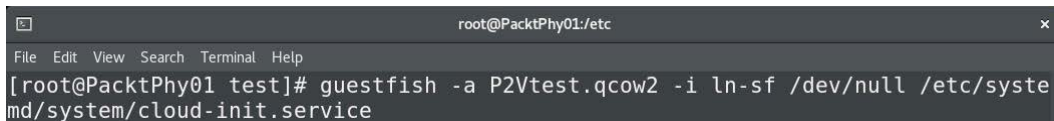
```

root@PacktPhy01:~/test
File Edit View Search Terminal Help
[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 reseat that VM if this was a VM template. So, let's use `guestfish` for the same purpose, as follows:



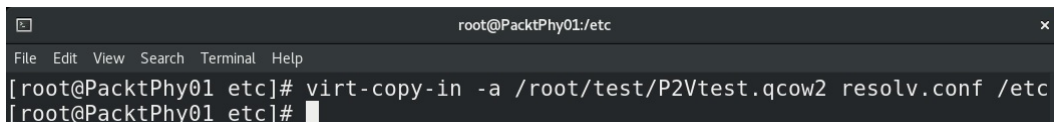
```
root@PacktPhy01:/etc
File Edit View Search Terminal Help
[root@PacktPhy01 test]# guestfish -a P2Vtest.qcow2 -i ln -sf /dev/null /etc/systemd/system/cloud-init.service
```

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
File Edit View Search Terminal Help
[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:

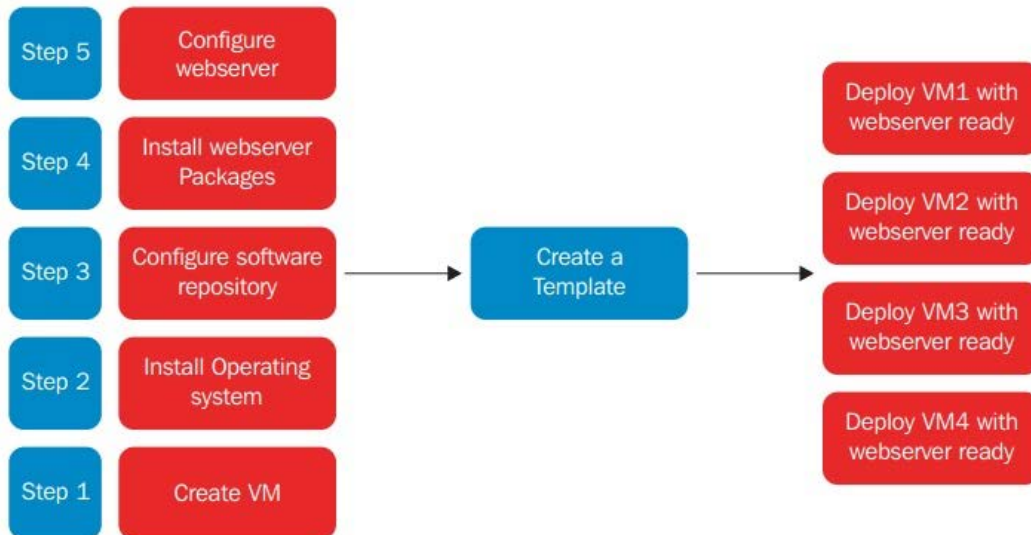


Figure 8.8 – Installing four Apache web servers by using VM templates

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
```

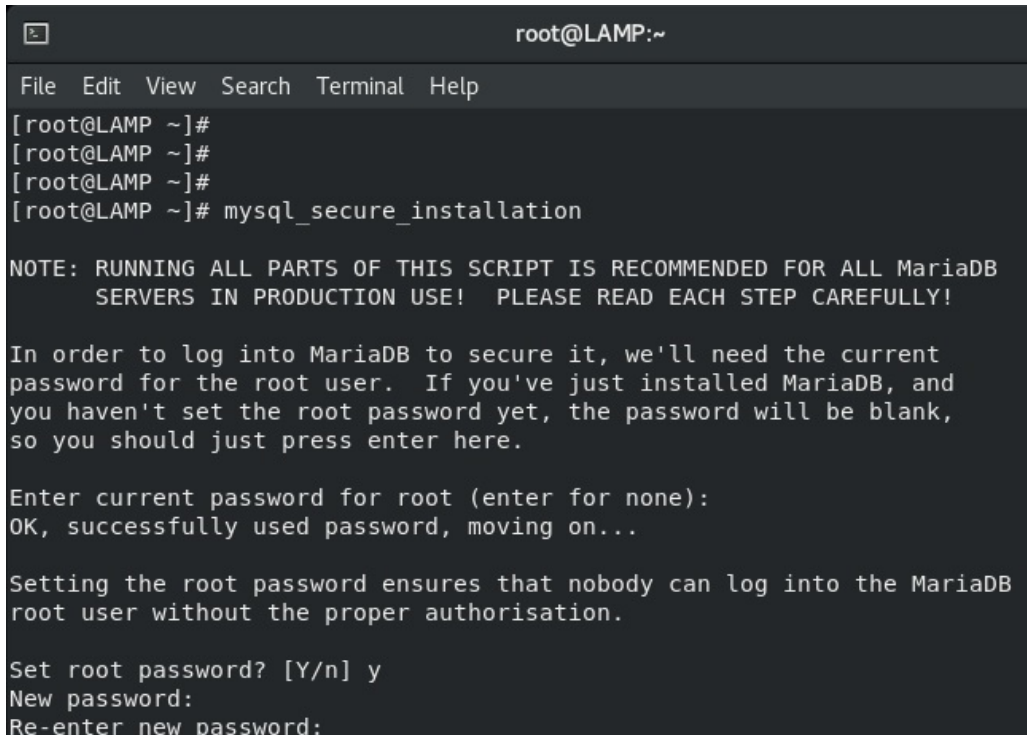
We also need to configure some security settings related to directory ownership—for example, **Security-Enhanced Linux (SELinux)** configuration for the Apache web server. Let's do that next, like this:

```
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:~
File Edit View Search Terminal Help
[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.

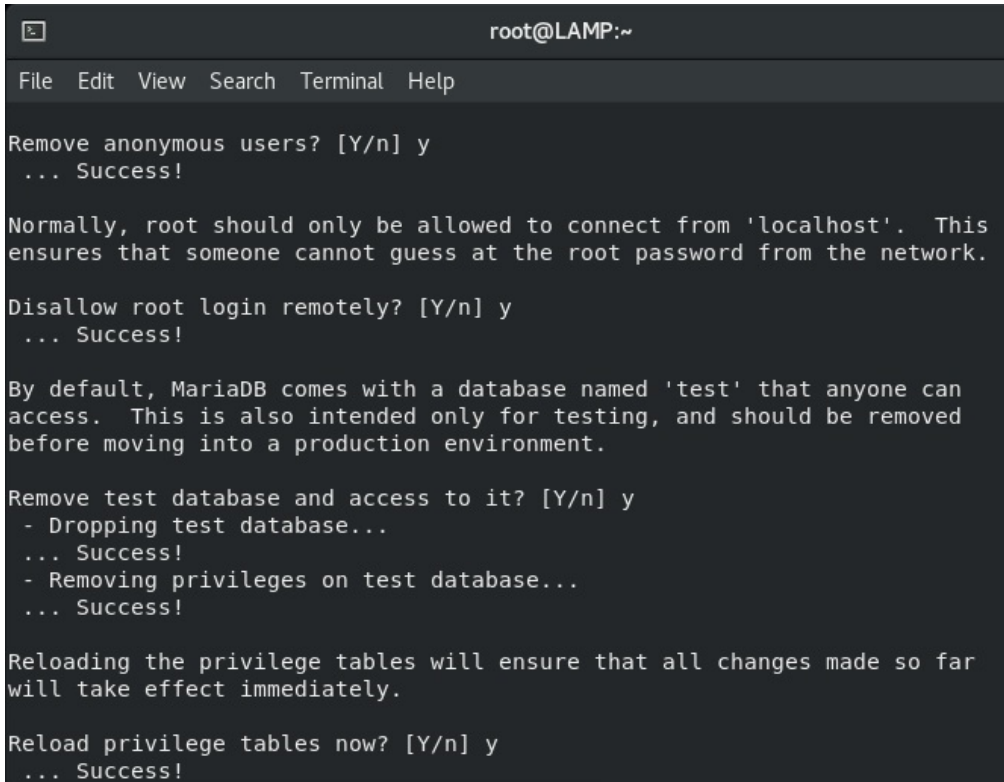
Enter current password for root (enter for none):
OK, successfully used password, moving on...

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:



```
root@LAMP:~
File Edit View Search Terminal Help
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.

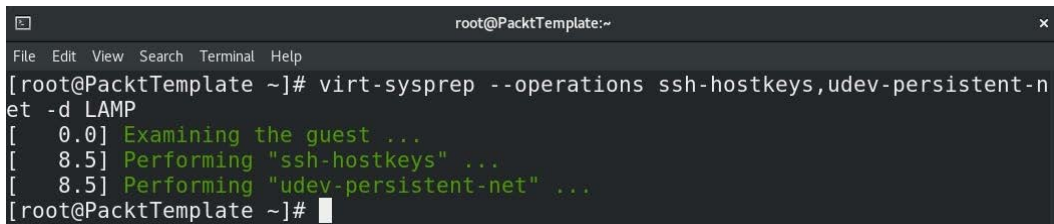
There are two ways to invoke `virt-sysprep` against a Linux VM: using the `-d` or `-a` option. The first option points to the intended guest using its name or **universally unique identifier (UUID)**, and the second one points to a particular disk image. This gives us the flexibility to use the `virt-sysprep` 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:~  
File Edit View Search Terminal Help  
[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:

A terminal window titled 'root@PacktTemplate:~' showing the execution of the 'virt-sysprep' command. The command is 'virt-sysprep --operations ssh-hostkeys,udev-persistent-net -d LAMP'. The output shows three lines of progress: '[0.0] Examining the guest ...', '[8.5] Performing "ssh-hostkeys" ...', and '[8.5] Performing "udev-persistent-net" ...'. The prompt returns to the root user at the shell.

```
root@PacktTemplate:~  
File Edit View Search Terminal Help  
[root@PacktTemplate ~]# virt-sysprep --operations ssh-hostkeys,udev-persistent-net -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.

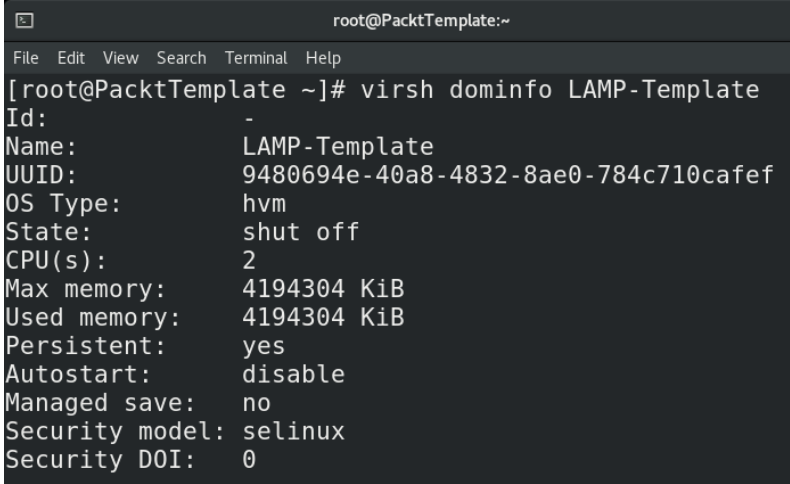
In order to rename a VM, use `virsh domrename` as root, like this:

```
# 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:~  
File Edit View Search Terminal Help  
[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:

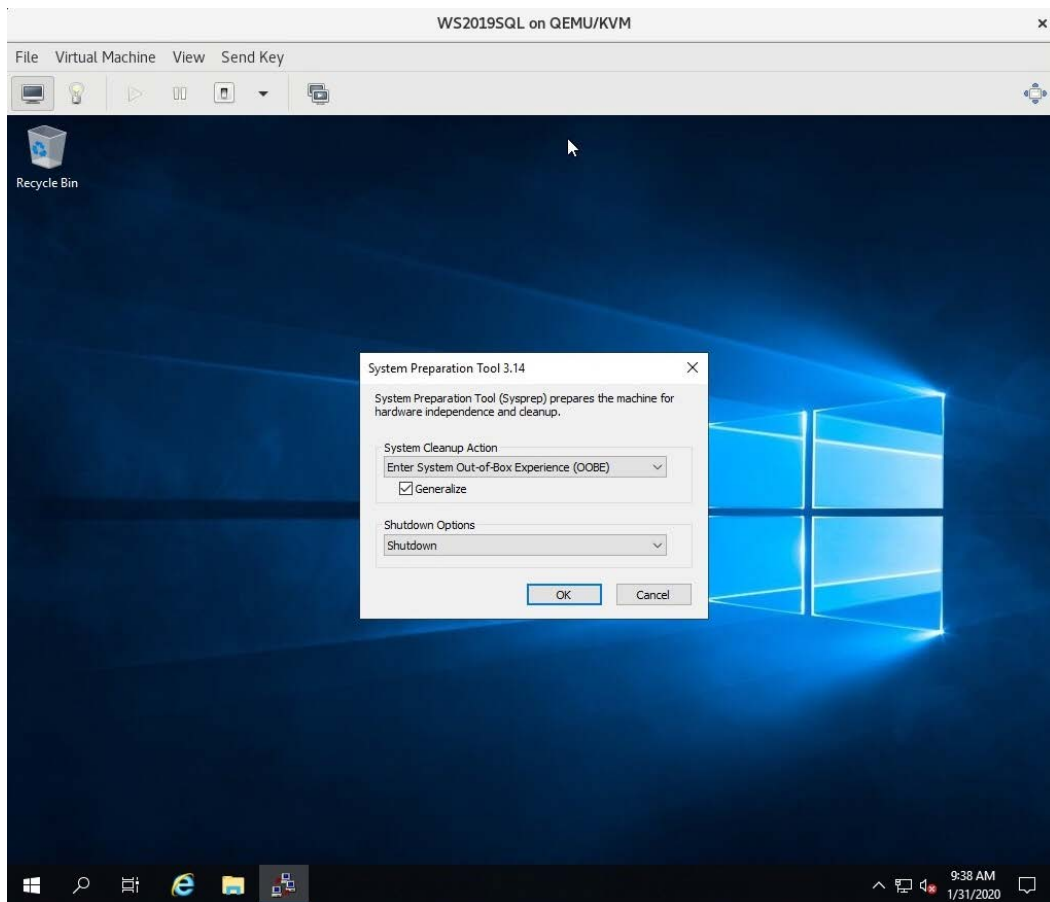


Figure 8.14 – Be careful with `sysprep` options; OOBE, generalize, and shutdown options are highly recommended

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:

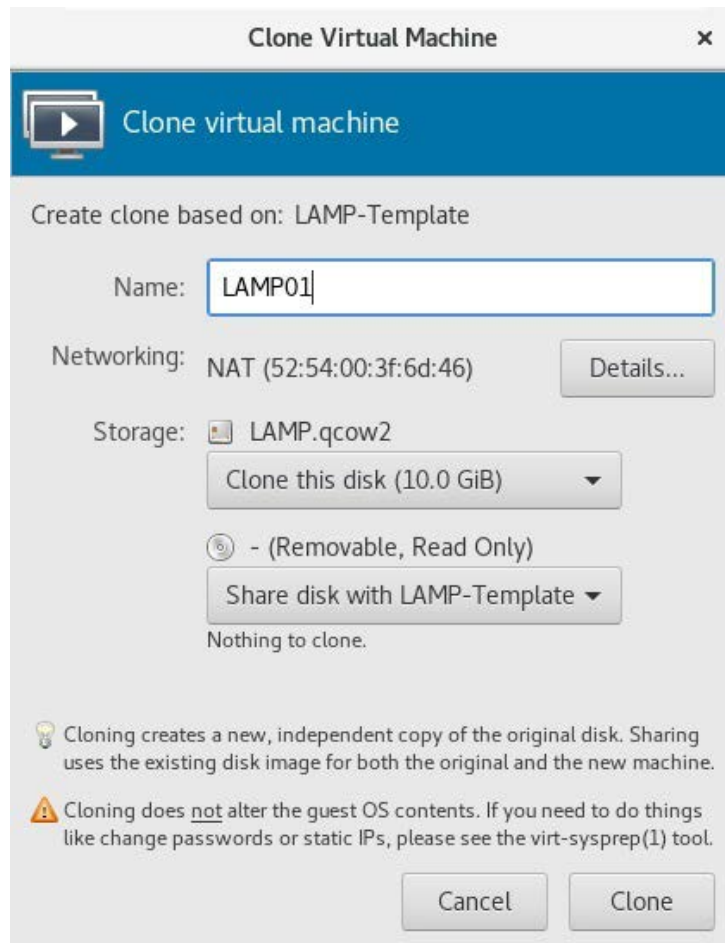
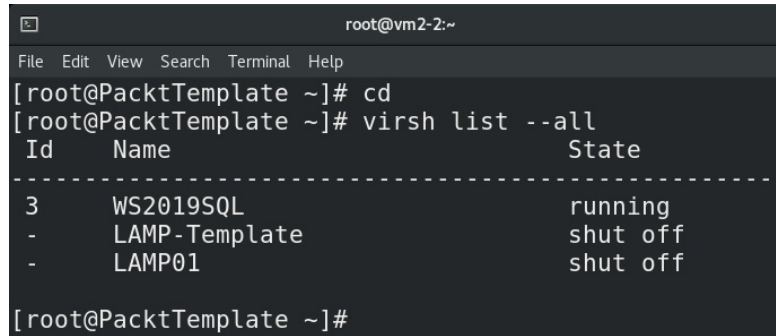


Figure 8.15 – Cloning a VM from VM Manager

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@vm2-2:~
File Edit View Search Terminal Help
[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:

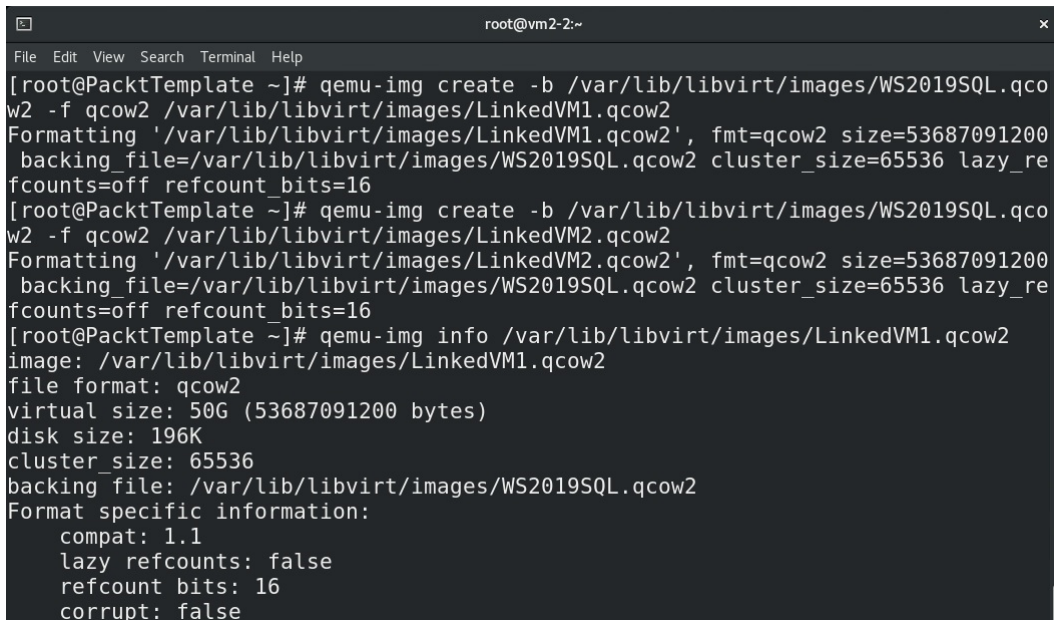
1. Create two new `qcow2` images using `/var/lib/libvirt/images/WS2019SQL.qcow2` as the backing file, like this:

```

# qemu-img create -b /var/lib/libvirt/images/WS2019SQL.qcow2 -f qcow2 /var/lib/libvirt/images/LinkedVM1.qcow2
# qemu-img create -b /var/lib/libvirt/images/WS2019SQL.qcow2 -f qcow2 /var/lib/libvirt/images/LinkedVM2.qcow2

```

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@vm2-2:~  
File Edit View Search Terminal Help  
[root@PacktTemplate ~]# qemu-img create -b /var/lib/libvirt/images/WS2019SQL.qcow2 -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_refcounts=off refcount_bits=16  
[root@PacktTemplate ~]# qemu-img create -b /var/lib/libvirt/images/WS2019SQL.qcow2 -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_refcounts=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
```

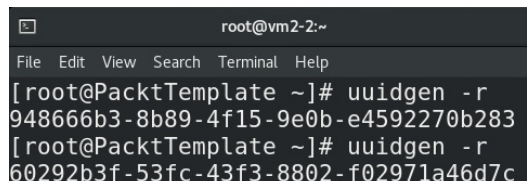
Figure 8.17 – Creating a linked clone image

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
```

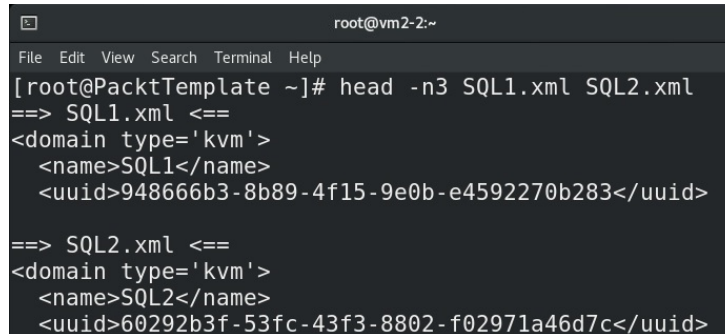
4. By using the `uuidgen -r` command, generate two random UUIDs. We will need them for our VMs. The process can be seen in the following screenshot:



```
root@vm2-2:~  
File Edit View Search Terminal Help  
[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:



```

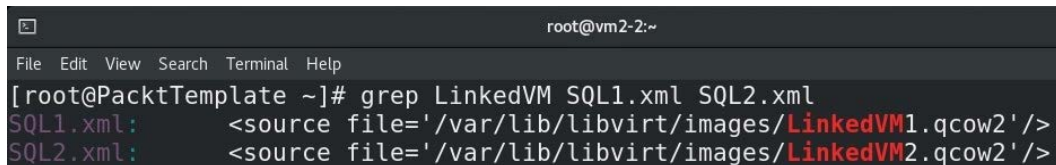
root@vm2-2:~
File Edit View Search Terminal Help
[root@PacktTemplate ~]# head -n3 SQL1.xml SQL2.xml
==> SQL1.xml <==
<domain type='kvm'>
  <name>SQL1</name>
  <uuid>948666b3-8b89-4f15-9e0b-e4592270b283</uuid>

==> 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:



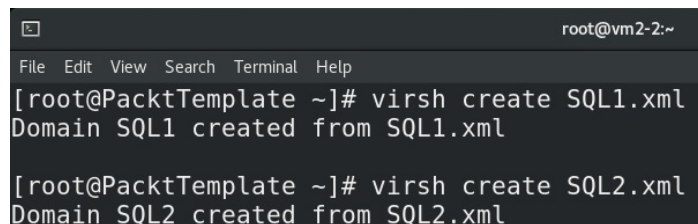
```

root@vm2-2:~
File Edit View Search Terminal Help
[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:



```

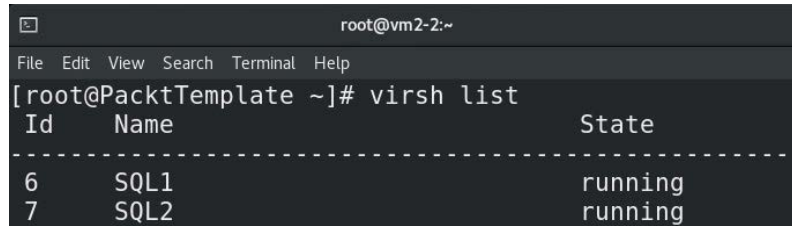
root@vm2-2:~
File Edit View Search Terminal Help
[root@PacktTemplate ~]# virsh create SQL1.xml
Domain SQL1 created from SQL1.xml

[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:



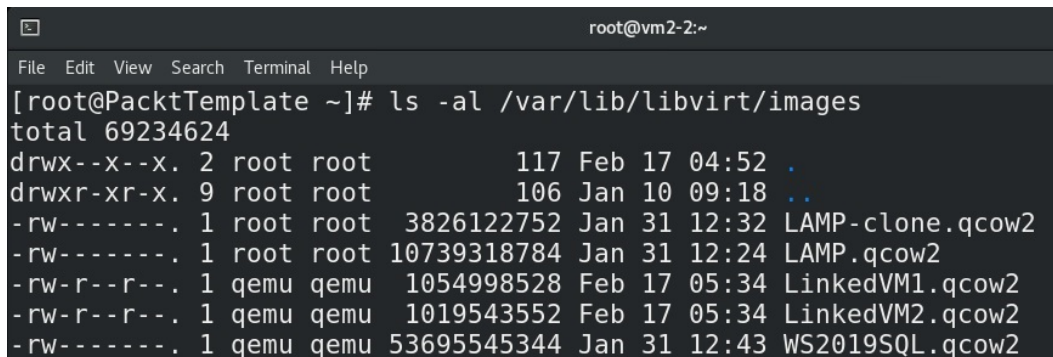
```

root@vm2-2:~
File Edit View Search Terminal Help
[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.qcow2` and `LinkedVM2.qcow2` files are roughly 50 times smaller than their base image:



```

root@vm2-2:~
File Edit View Search Terminal Help
[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

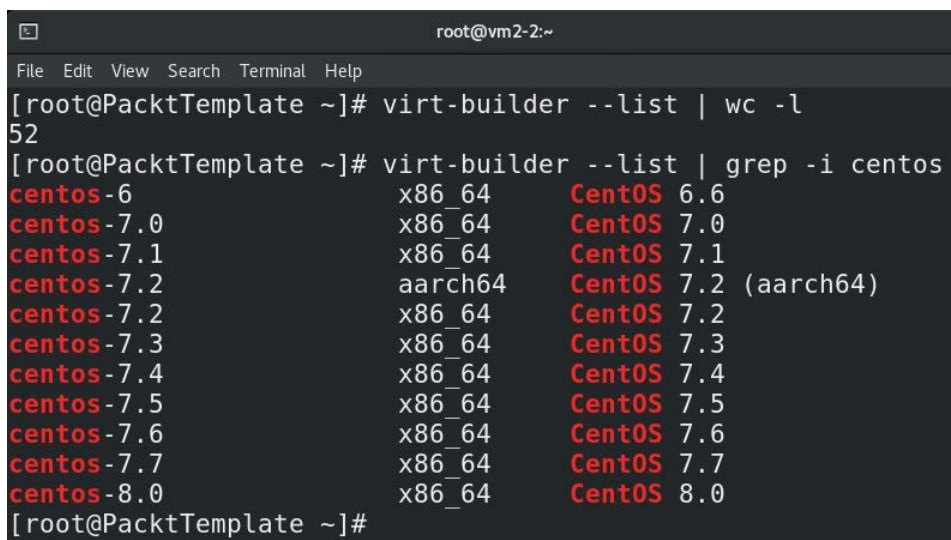
```

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:

A terminal window titled 'root@vm2-2:~' showing the output of the 'virt-builder --list' command. The first command is 'virt-builder --list | wc -l' which returns '52'. The second command is 'virt-builder --list | grep -i centos', which lists various CentOS distributions with their architectures and versions. The output is as follows:

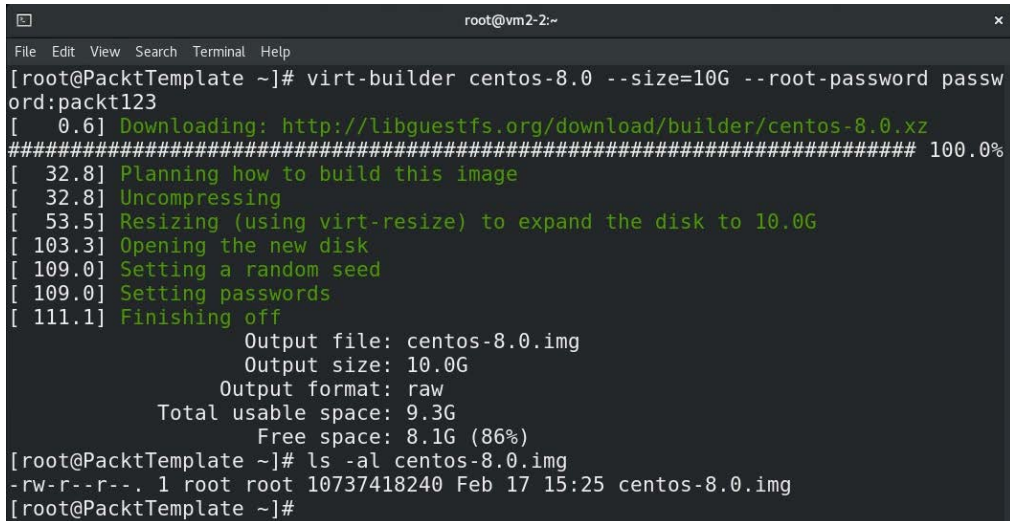
```
[root@PacktTemplate ~]# virt-builder --list | wc -l
52
[root@PacktTemplate ~]# virt-builder --list | grep -i centos
centos-6                x86_64      CentOS 6.6
centos-7.0              x86_64      CentOS 7.0
centos-7.1              x86_64      CentOS 7.1
centos-7.2              aarch64     CentOS 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:



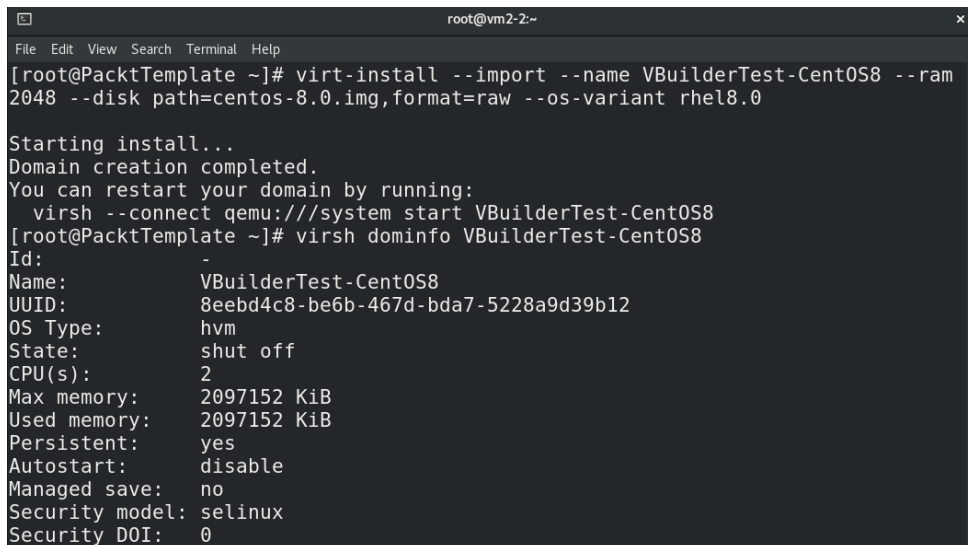
```

root@vm2-2:~
File Edit View Search Terminal Help
[root@PacktTemplate ~]# virt-builder centos-8.0 --size=10G --root-password password: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:



```

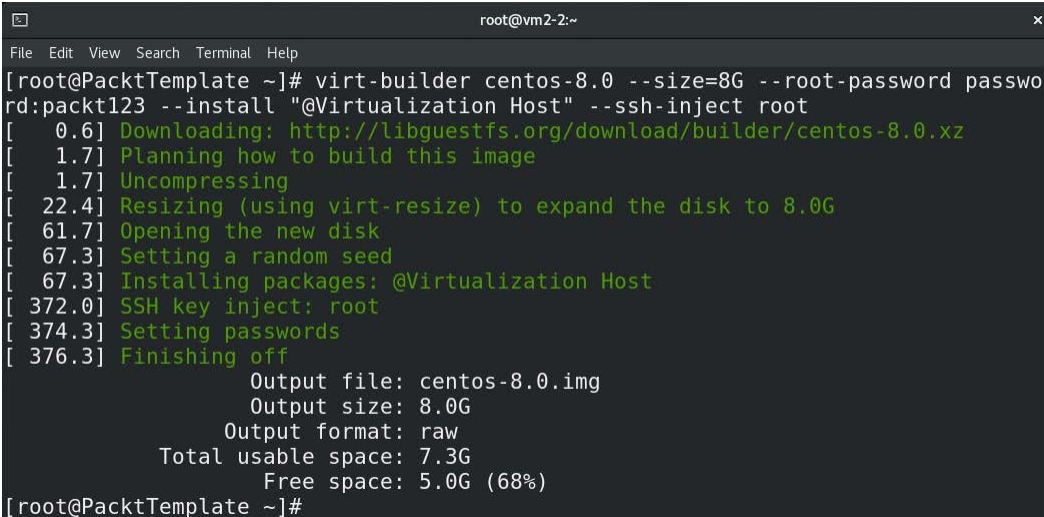
root@vm2-2:~
File Edit View Search Terminal Help
[root@PacktTemplate ~]# virt-install --import --name VBuilderTest-CentOS8 --ram 2048 --disk path=centos-8.0.img,format=raw --os-variant rhel8.0

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:

A terminal window titled 'root@vm2-2:~' showing the execution of the 'virt-builder' command. The command is: 'virt-builder centos-8.0 --size=8G --root-password password:packt123 --install "@Virtualization Host" --ssh-inject root'. The output shows a progress log with timestamps and descriptions of actions like downloading, planning, uncompressing, resizing, opening disk, setting seed, installing packages, SSH key injection, and setting passwords. It concludes with summary statistics: 'Output file: centos-8.0.img', 'Output size: 8.0G', 'Output format: raw', 'Total usable space: 7.3G', and 'Free space: 5.0G (68%)'.

```
root@vm2-2:~  
File Edit View Search Terminal Help  
[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

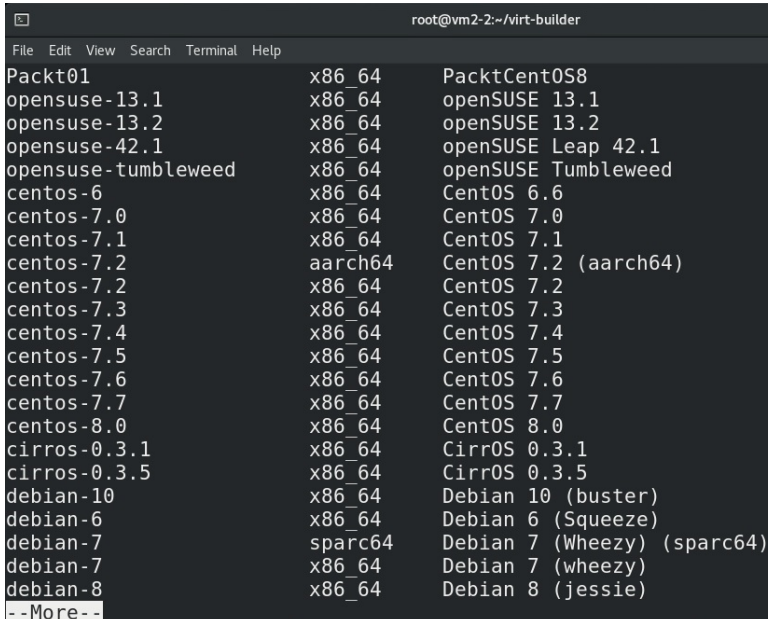
Obviously, there are some pre-defined `virt-builder` repositories (<http://libguestfs.org/> is one of them), but we can also create our own. If we go to the `/etc/virt-builder/repos.d` directory, we'll see a couple of files there (`libguestfs.conf` and its key, and so on). We can easily create our own 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=ccb4d840f5eb77d7d0ffbc4241fbf4d21fcc1acdd3679
c13174194810b17dc472566f6a29dba3a8992c1958b4698b6197e6a1689882
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:

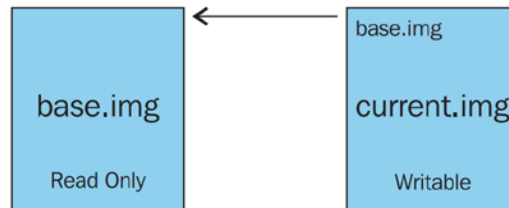


Figure 8.29 – Snapshot concept

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:

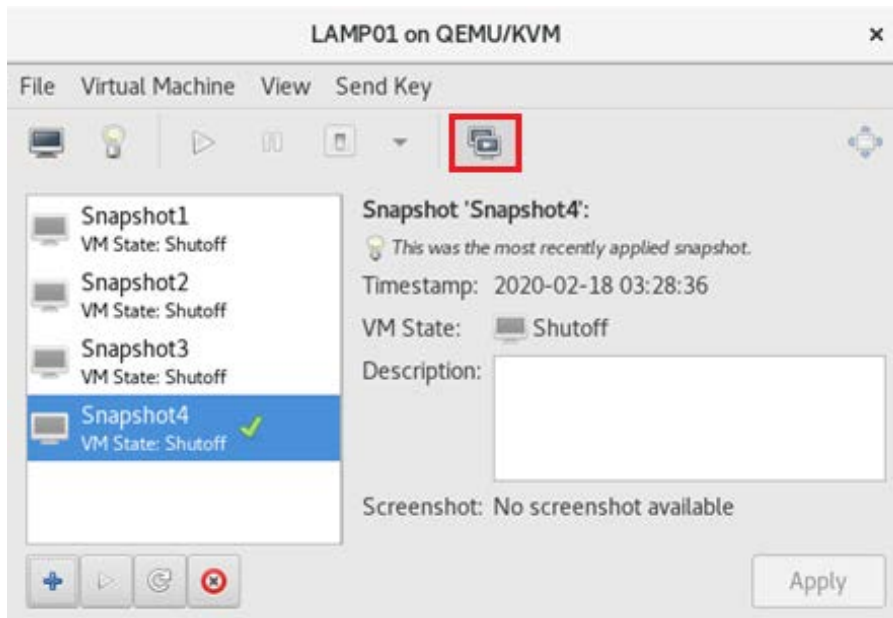


Figure 8.30 – Working with snapshots from virt-manager

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:

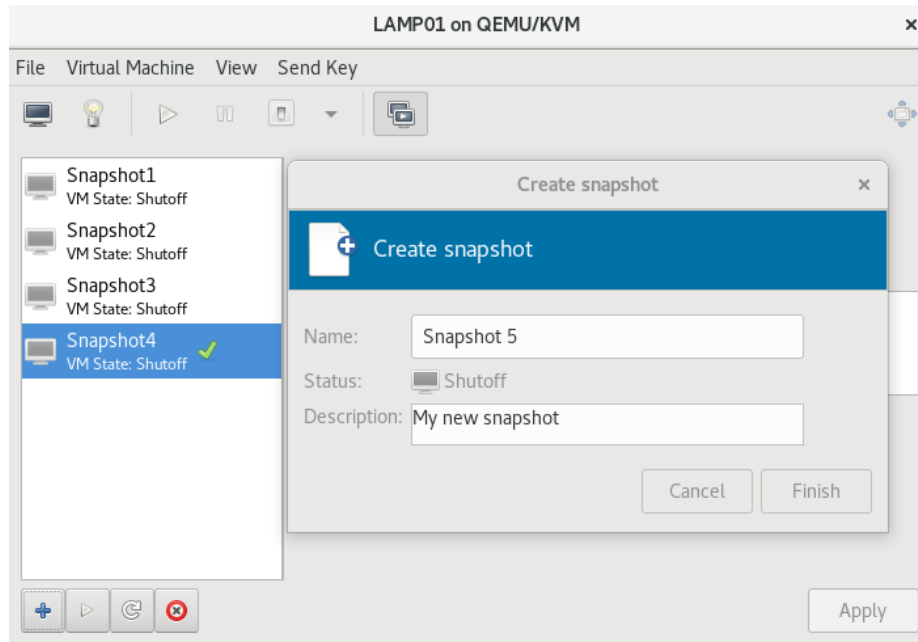


Figure 8.31 – Create snapshot wizard

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
```

- 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
```

- 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:

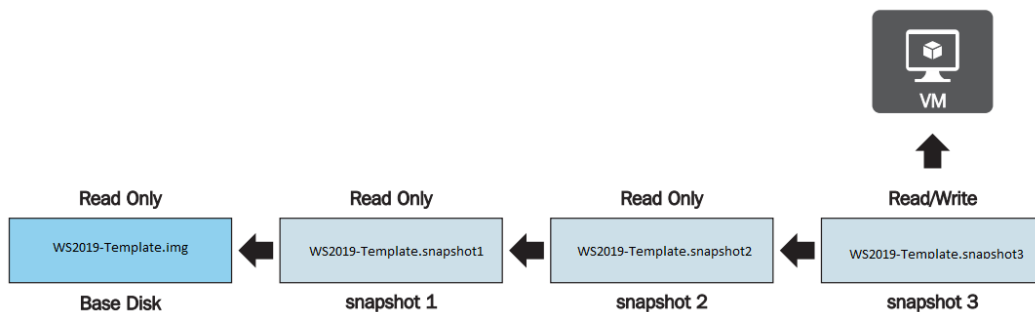


Figure 8.32 – Snapshot chain for our example VM

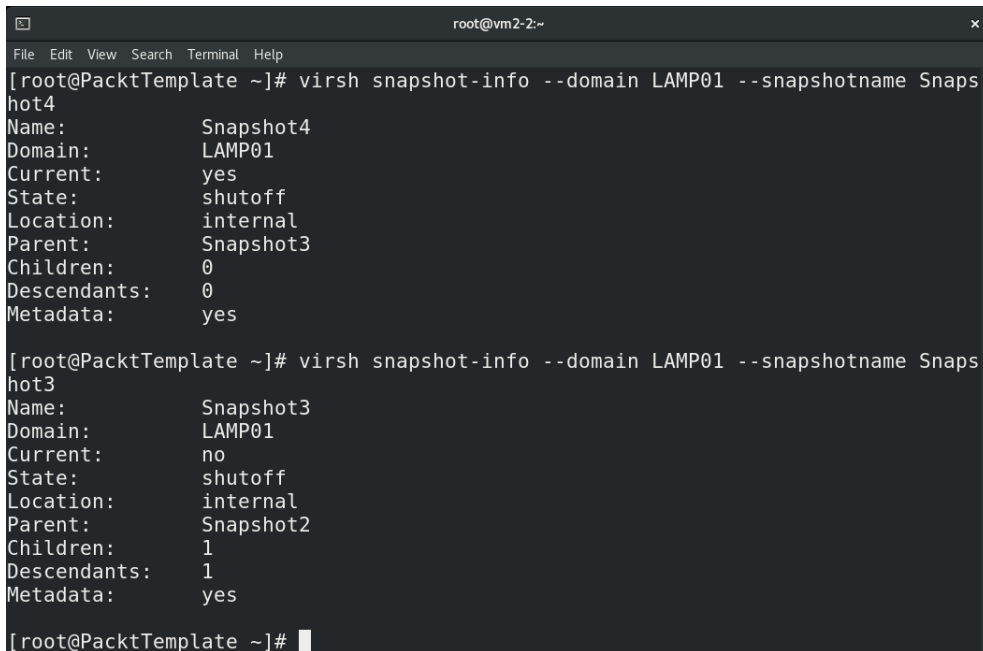
So, it has to be read as follows: `snapshot3` has `snapshot2` as its backing file; `snapshot2` has `snapshot1` as its backing file; and `snapshot1` has the base image as its backing file. Currently, `snapshot3` is the current active snapshot, where live guest writes happen.

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.

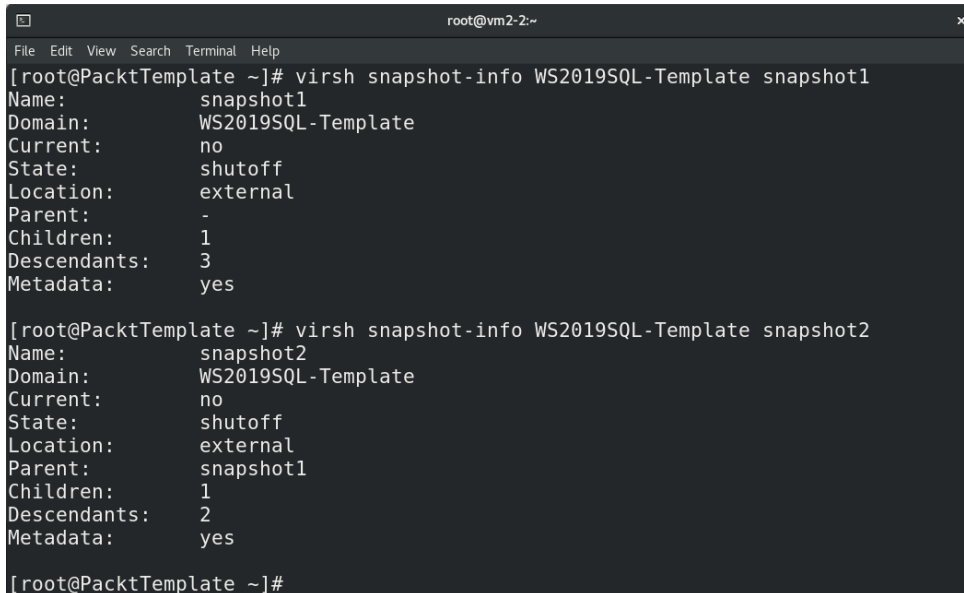
A caveat to this is the fact that `virt-manager` still doesn't support external snapshots, as evident by the following screenshot and the fact that when we created these snapshots just a couple of pages ago, we never got an option to select external snapshot as the snapshot type:



```
root@vm2-2:~  
File Edit View Search Terminal Help  
[root@PacktTemplate ~]# virsh snapshot-info --domain LAMP01 --snapshotname Snapshot4  
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 Snapshot3  
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@vm2-2-~
File Edit View Search Terminal Help
[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 snapshot, 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 following 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 images. 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/images/var/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:

- `libguestfs` 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>

Section 3: Automation, Customization, and Orchestration for KVM VMs

In this part of the book, you will get a complete understanding of how to customize KVM virtual machines by using `cloud-init` and `cloudbase-init`. This part also covers how to leverage the automation capabilities of Ansible to manage and orchestrate the KVM infrastructure.

This part of the book comprises the following chapters:

- *Chapter 9, Customizing a Virtual Machine with cloud-init*
- *Chapter 10, Automated Windows Guest Deployment and Customization*
- *Chapter 11, Ansible and Scripting for Orchestration and Automation*

9

Customizing a Virtual Machine with cloud-init

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

In this chapter, we will cover the following topics:

- What is the need for virtual machine customization?
- Understanding cloud-init
- cloud-init architecture

- How to install and configure cloud-init at boot time
- cloud-init images
- cloud-init data sources
- Passing metadata and user data to cloud-init
- Examples on how to use the cloud-config script with cloud-init

What is the need for virtual machine customization?

Once you really start using virtual machines and learn how to master them, you will notice that one thing seems to be happening a lot: virtual machine deployment. Since everything is so easy to configure and deploy, you will start to create new instances of virtual machines for almost anything, sometimes even to just check whether a particular application works on a particular version of the operating system. This makes your life as a developer and system administrator a lot easier, but creates its own set of problems. One of the most difficult ones is template management. Even if you have a small set of different servers and a relatively modest number of different configurations, things will start to add up, and if you decide to manage templates the normal way through the KVM, the sheer number of combinations will soon be too big.

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

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

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

Imagine this: you need to create a single application deployment to test your new application with different Linux distributions. All of your future virtual machines will need to have a unique identifier in the form of a hostname, a deployed SSH identity that will enable remote management through Ansible, and of course, your application. Your application has three dependencies – two in the form of packages that can be deployed through Ansible, but one depends on the Linux distribution being used and has to be tailored for that particular Linux distribution. To make things even more realistic, you expect that you will have to periodically repeat this test, and every time you will need to rebuild your dependencies.

There are a couple of ways you can create this environment. One is to simply manually install all the servers and create templates out of them. This means manually configuring everything and then creating a virtual machine template that will be deployed. If we intend to deploy to more than a couple of Linux distributions this is a lot of work. It becomes even more work once the distributions get upgraded since all the templates we are deploying from must be upgraded, often at different points in time. This means we can either manually update all the virtual machine templates, or perform a post-install upgrade on each of them. This is a lot of work and it is extremely slow. Add to that the fact that a test like this will probably involve running your test application on both new and old versions of virtual machine templates. In addition to all that, we need to solve the problem of customizing our network settings for each and every Linux distribution we are deploying. Of course, this also means that our virtual machine templates become far from generic. After a while, we are going to end up with tens of virtual machine templates for each test cycle.

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

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

Understanding cloud-init

We need to get a bit more technical in order to understand what cloud-init is and to understand what its limitations are. Since we are talking about a way to fully automatically reconfigure a system using simple configuration files, it means that some things need to be prepared in advance to make this complex process user friendly.

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

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

Why is this image important? Because it is prepared so that it can detect the cloud system it is running under, it determines whether cloud-init should be used or should be disabled, and after that, it reads and performs the configuration of the system itself.

Understanding cloud-init architecture

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

- The **generator** is the first one, and the simplest one: it will determine whether we are even trying to run cloud-init, and based on that, whether it should enable or disable the processing of data files. Cloud-init will not run if there are kernel command-line directives to disable it, or if a file called `/etc/cloud/cloud-init.disabled` exists. For more information on this and all the other things in this chapter, please read the documentation (start at <https://cloudinit.readthedocs.io/en/latest/topics/boot.html>) since it contains much more detail about switches and different options that cloud-init supports and that make it tick.
- The **local** phase tries to find the data that we included for the boot itself, and then it tries to create a running network configuration. This is a relatively simple task performed by a `systemd` service called `cloud-init-local.service`, which will run as soon as possible and will block the network until it's done. The concept of blocking services and targets is used a lot in cloud-init initialization; the reason is simple – to ensure system stability. Since cloud-init procedures modify a lot of core settings for a system, we cannot afford to let the usual startup scripts run and create a parallel configuration that could overrun the one created by cloud-init.
- The **network** phase is the next one, and it uses a separate service called `cloud-init.service`. This is the main service that will bring up the previously configured network and try to configure everything we scheduled in the data files. This will typically include grabbing all the files specified in our configuration, extracting them, and executing other preparation tasks. Disks will also be formatted and partitioned in this stage if such a configuration change is specified. Mount points will also get created, including those that are dynamic and specific to a particular cloud platform.

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

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

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

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

Why is all of this so important?

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

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

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

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

Important note

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

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

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

Installing and configuring cloud-init at boot time

The main thing that we are covering in this chapter is how to get cloud-init to run, and how to get all of its parts in the right place when the machine is being deployed, but this only scratches the surface of how cloud-init actually works. What you need to understand is that cloud-init runs as a service, configures the system, and follows what we told it to do in a certain way. After the system has booted, we can connect to it and see what was done, how, and analyze the logs. This could seem contrary to the idea of completely automatic deployment but it is there for a reason – whatever we do, there is always the possibility that we will need to debug the system or do some post-installation tasks that can also be automated.

Using cloud-init is not specifically confined to just debugging. After the system has booted, there is a large amount of data created by the system about how the boot was done, what actual cloud configuration the system is running on, and what was done in regard to customization. Any of your applications and scripts can then rely on this data and use it to run and detect certain configuration and deployment parameters. Check out this example, taken from a virtual machine in Microsoft Azure, running Ubuntu:

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

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

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

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

Cloud-init images

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

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

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

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

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

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























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

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

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

Cloud-init data sources

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

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

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

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

Passing metadata and user data to cloud-init

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

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

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

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

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

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

The main difference between these packages is the computer we are installing them on. `cloud-init` is to be installed on the computer we are configuring and is part of the deployment image. `cloud-utils` is the package intended to be used on the computer that will create the configuration.

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

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

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

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

Using cloud-init modules

When creating a configuration file, in cloud-init, pretty much like in any other software abstraction layer, we are dealing with modules that are going to translate our more-or-less universal configuration demands, such as *this package needs to be installed* into actual shell commands on a particular system. The way this is done is through **modules**. Modules are logical units that break down different functionalities into smaller groups and enable us to use different commands. You can check the list of all available modules at the following link: <https://cloudinit.readthedocs.io/en/latest/topics/modules.html>. It's quite a list, which will just further show you how well developed cloud-init is.

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

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

Important note

We are not going to deal with particular modules one by one since it would make this chapter too long and possibly turn it into a book on its own. If you plan on working with cloud-init, consult the module documentation since it will provide all the up-to-date information you need.

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

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

We are going to do the same with the Ubuntu image, after we get it from the internet. Ubuntu also publishes cloud versions of their distribution daily, for all supported versions. The main difference is that Ubuntu creates images that are designed to be 2.2 GB when full. We downloaded an image from <https://cloud.centos.org>; let's now get some information about it:

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

Figure 9.3 – Cloud-init image sizes

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

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

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

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

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

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

Figure 9.5 – Growing the Ubuntu and CentOS maximum image size to 10 GB via `qemu-img`
After growing our images, note that the size on the disk hasn't changed much:

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

Figure 9.6 – The real disk usage has changed only slightly

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

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

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

Figure 9.7 – Moving images to the KVM default system directory

We are going to create our first cloud-enabled deployment in the simplest way possible, by only repartitioning the disk and creating a single user with a single SSH key. The key belongs to the root of the host machine, so we can directly log in to the deployed machine after cloud-init is done.

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

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

Figure 9.8 – Creating an image disk for deployment

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

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

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

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

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

```
#cloud-config
users:
  - name: cloud
    ssh-authorized-keys:
      - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQACzh6
        6Gf1lNuMeenGywifUSW1T16uKW0IXnucNwoIynhymSm1fkTCqyxLk
        ImWbyd/tDFkbgTlei3qa245Xwt//5ny2fGitcSa7jWvkKvTLiPvxLP
        0CvcvGR4aiV/2TuxA1em3JweqpNppyuapH7u9q0SdxaG2gh3uViYl
        /+8uuzJLJJbxb/a8EK+szpdZq7bpLOvigOTgMan+LGN1sZc6lqE
        VD1j40tG3YNtk51xfKBLxwLpFq7JPfAv8DTMcdYqqqc5PhRnnKLak
        SUQ6OW0nv4fpa0MKuhalnrO72Zyur7FRf9XFvD+Uc7ABNpeyUTZVI
        j2dr5hjjFTPfZWUC96FEh root@localhost.localdomain
    sudo: ['ALL=(ALL) NOPASSWD:ALL']
    groups: users
    shell: /bin/bash
runcmd:
  - echo "AllowUsers cloud" >> /etc/ssh/sshd_config
  - restart ssh
```

Note that the file must follow the way YAML defines everything including the variables. Pay attention to the spaces and newlines, as the biggest problems with deployment come from misplaced newlines in the configuration.

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

```

[root@localhost ~]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:iNGTi2P1qCnKhl9dVZy5hDhdukBbax81VZN2Soc3Etg root@localhost
The key's randomart image is:
+---[RSA 2048]---+
|      .o.+=+++=|
|      .oooo==Eo==|
|      . = o.=.oo+o|
|      = * + o...|
|      = + S . . |
|      . = .      |
|      . . + .    |
|oo o            |
|oo.            |
+----[SHA256]-----+
[root@localhost ~]# ls -al .ssh
total 20
drwx-----. 2 root root  80 Jun 22 16:52 .
dr-xr-x---. 20 root root 4096 Jun 21 20:41 ..
-rw-----. 1 root root 1374 Apr 17 01:22 authorized_keys
-rw-----. 1 root root 1679 Jun 22 16:52 id_rsa
-rw-r--r--. 1 root root  396 Jun 22 16:52 id_rsa.pub
-rw-r--r--. 1 root root  366 Apr 27 11:23 known_hosts

```

Figure 9.10 – SSH keygen procedure done, SSH keys are present and accounted for. Keys are stored in the local `.ssh` directory, so we just need to copy them. When we are doing cloud deployments, we usually use this method of authentication, but cloud-init enables us to define any method of user authentication. It all depends on what we are trying to do and whether there are security policies in place that enforce one authentication method over another.

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

Next, we configured `sudo`. Our user needs to have root permissions since there are no other users defined for this machine. This means they need to be a member of the `sudo` group and have to have the right permissions defined in the `sudoers` file. Since this is a common setting, we only need to declare the variables, and cloud-init is going to put the settings in the right files. We will also define a user shell.

In this file, we can also define all the other users' settings available on Linux, a feature that is intended to help deploy user computers. If you need any of those features, check the documentation available here: <https://cloudinit.readthedocs.io/en/latest/topics/modules.html#users-and-groups>. All the extended user information fields are supported.

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

Now we are ready for our first deployment.

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

We need a way to connect the files we just created, the configuration, with the virtual machine. Usually, this is done in a couple of ways. The simplest way is usually to generate a `.iso` file that contains the files. Then we just mount the file as a virtual CD-ROM when we create the machine. On boot, cloud-init will look for the files automatically.

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

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

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

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

Figure 9.11 – Creating an ISO image

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

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

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

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

Now, let's start with our deployments.

The first deployment

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

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


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

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

Starting install...
Domain creation completed.
[root@localhost deploy-1]# virsh domifaddr deploy-1
Name          MAC address          Protocol  Address
-----
vnet0         52:54:00:55:93:9e    ipv4      192.168.122.2/24

[root@localhost deploy-1]# ssh cloud@192.168.122.120
^C
[root@localhost deploy-1]# ssh cloud@192.168.122.2
The authenticity of host '192.168.122.2 (192.168.122.2)' can't be established.
ECDSA key fingerprint is SHA256:WRucACTXNTbAlvwfuynPEgqo6FjJoLas6bLKymPJrEQ.
ECDSA key fingerprint is MD5:44:b5:04:e8:87:ad:24:19:01:a3:e9:8d:a7:0e:42:34.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.122.2' (ECDSA) to the list of known hosts.
[cloud@deploy-1 ~]$
```

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

The deployment will probably take a minute or two. As soon as the machine boots, it will get the IP address and we can SSH to it using our predefined key. The only thing that was not automated is accepting the fingerprint of the newly booted machine automatically.

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

```

2020-01-12 19:32:56,966 - util.py[DEBUG]: Cloud-init v. 19.3-41-gc4735dd3-0ubuntu1-18.04.1 running 'init-local' at Sun, 12 Jan 2020 19:32:56 +0000. Up 12.65 seconds.
2020-01-12 19:32:56,966 - main.py[DEBUG]: No kernel command line url found.
2020-01-12 19:32:56,966 - main.py[DEBUG]: Closing stdin.
2020-01-12 19:32:56,969 - util.py[DEBUG]: Writing to /var/log/cloud-init.log - ab: [644] 0 bytes
2020-01-12 19:32:56,969 - util.py[DEBUG]: Changing the ownership of /var/log/cloud-init.log to 102:4
2020-01-12 19:32:56,969 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/instance/boot-finished
2020-01-12 19:32:56,969 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/data/no-net
2020-01-12 19:32:56,969 - handlers.py[DEBUG]: start: init-local/check-cache: attempting to read from cache [check]
2020-01-12 19:32:56,969 - util.py[DEBUG]: Reading from /var/lib/cloud/instance/obj.pkl (quiet=False)
2020-01-12 19:32:56,970 - stages.py[DEBUG]: no cache found
2020-01-12 19:32:56,970 - handlers.py[DEBUG]: finish: init-local/check-cache: SUCCESS: no cache found
2020-01-12 19:32:56,970 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/instance
2020-01-12 19:32:56,976 - stages.py[DEBUG]: Using distro class <class 'cloudinit.distros.ubuntu.Distro'>
2020-01-12 19:32:56,977 - __init__.py[DEBUG]: Looking for data source in: ['NoCloud', 'None'], via packages ['', 'cloudinit.sources'] th
at matches dependencies ['FILESYSTEM']
2020-01-12 19:32:57,012 - __init__.py[DEBUG]: Searching for local data source in: ['DataSourceNoCloud']
2020-01-12 19:32:57,012 - handlers.py[DEBUG]: start: init-local/search-NoCloud: searching for local data from DataSourceNoCloud
2020-01-12 19:32:57,012 - __init__.py[DEBUG]: Seeing if we can get any data from <class 'cloudinit.sources.DataSourceNoCloud.DataSourceNoCloud'>
2020-01-12 19:32:57,012 - __init__.py[DEBUG]: Update datasource metadata and network config due to events: New instance first boot
2020-01-12 19:32:57,012 - util.py[DEBUG]: Running command ['systemd-detect-virt', '--quiet', '--container'] with allowed return codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,016 - util.py[DEBUG]: Running command ['running-in-container'] with allowed return codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,017 - util.py[DEBUG]: Running command ['lxc-is-container'] with allowed return codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /proc/1/environ (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 176 bytes from /proc/1/environ
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /proc/self/status (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 1290 bytes from /proc/self/status
2020-01-12 19:32:57,019 - util.py[DEBUG]: querying dmi data /sys/class/dmi/id/product_serial
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /sys/class/dmi/id/product_serial (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 1 bytes from /sys/class/dmi/id/product_serial
2020-01-12 19:32:57,019 - util.py[DEBUG]: dmi data /sys/class/dmi/id/product_serial returned
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/user-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/meta-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/vendor-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/network-config (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud-net/user-data (quiet=False)
2020-01-12 19:32:57,020 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud-net/meta-data (quiet=False)
2020-01-12 19:32:57,020 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud-net/vendor-data (quiet=False)
cloud-init.log

```

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

Another thing is how much actually happens below the surface completely automatically. Since this is CentOS, cloud-init has to deal with the SELinux security contexts in real time, so a lot of the information is simply that. There are also a lot of probes and tests going on. Cloud-init has to establish what the running environment is and what type of cloud it is running under. If something happens during the boot process and it in any way involves cloud-init, this is the first place to look.

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

The second deployment

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

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

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

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

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

Figure 9.16 – Growing our virtual machine image for deployment purposes

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

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

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

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

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

Now let's check the IP addresses:

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

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

Figure 9.18 – Check the virtual machine IP addresses

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

```
[root@localhost deploy-2]# ssh cloud@192.168.122.126
The authenticity of host '192.168.122.126 (192.168.122.126)' can't be established.
ECDSA key fingerprint is SHA256:4wD2aNIgUhKYIS8ByndSkH36hEe8xIhfb4Z/LLEyqTE.
ECDSA key fingerprint is MD5:a6:61:06:ac:f3:34:0d:0a:91:df:9c:8f:d7:54:d7:e3.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.122.126' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 18.04.3 LTS (GNU/Linux 4.15.0-74-generic x86_64)

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

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

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

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

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

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

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

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

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

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

```

2020-01-12 19:32:56,966 - util.py[DEBUG]: Cloud-init v. 19.3-41-gc4735dd3-0ubuntu1-18.04.1 running 'init-local' at Sun, 12 Jan 2
020 19:32:56 +0000. Up 12.65 seconds.
2020-01-12 19:32:56,966 - main.py[DEBUG]: No kernel command line url found.
2020-01-12 19:32:56,966 - main.py[DEBUG]: Closing stdin.
2020-01-12 19:32:56,969 - util.py[DEBUG]: Writing to /var/log/cloud-init.log - ab: [644] 0 bytes
2020-01-12 19:32:56,969 - util.py[DEBUG]: Changing the ownership of /var/log/cloud-init.log to 102:4
2020-01-12 19:32:56,969 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/instance/boot-finished
2020-01-12 19:32:56,969 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/data/no-net
2020-01-12 19:32:56,969 - handlers.py[DEBUG]: start: init-local/check-cache: attempting to read from cache [check]
2020-01-12 19:32:56,969 - util.py[DEBUG]: Reading from /var/lib/cloud/instance/obj.pkl (quiet=False)
2020-01-12 19:32:56,970 - stages.py[DEBUG]: no cache found
2020-01-12 19:32:56,970 - handlers.py[DEBUG]: finish: init-local/check-cache: SUCCESS: no cache found
2020-01-12 19:32:56,970 - util.py[DEBUG]: Attempting to remove /var/lib/cloud/instance
2020-01-12 19:32:56,976 - stages.py[DEBUG]: Using distro class <class 'cloudinit.distros.ubuntu.Distro'>
2020-01-12 19:32:56,977 - __init__.py[DEBUG]: Looking for data source in: ['NoCloud', 'None'], via packages ['', 'cloudinit.sources']
that matches dependencies ['FILESYSTEM']
2020-01-12 19:32:57,012 - __init__.py[DEBUG]: Searching for local data source in: ['DataSourceNoCloud']
2020-01-12 19:32:57,012 - handlers.py[DEBUG]: start: init-local/search-NoCloud: searching for local data from DataSourceNoCloud
2020-01-12 19:32:57,012 - __init__.py[DEBUG]: Seeing if we can get any data from <class 'cloudinit.sources.DataSourceNoCloud.DataSourceNoCloud'>
2020-01-12 19:32:57,012 - __init__.py[DEBUG]: Update datasource metadata and network config due to events: New instance first boot
2020-01-12 19:32:57,012 - util.py[DEBUG]: Running command ['systemd-detect-virt', '--quiet', '--container'] with allowed return
codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,016 - util.py[DEBUG]: Running command ['running-in-container'] with allowed return codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,017 - util.py[DEBUG]: Running command ['lxc-is-container'] with allowed return codes [0] (shell=False, capture=True)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /proc/1/environ (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 176 bytes from /proc/1/environ
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /proc/self/status (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 1290 bytes from /proc/self/status
2020-01-12 19:32:57,019 - util.py[DEBUG]: querying dmi data /sys/class/dmi/id/product_serial
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /sys/class/dmi/id/product_serial (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Read 1 bytes from /sys/class/dmi/id/product_serial
2020-01-12 19:32:57,019 - util.py[DEBUG]: dmi data /sys/class/dmi/id/product_serial returned
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/user-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/meta-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/vendor-data (quiet=False)
2020-01-12 19:32:57,019 - util.py[DEBUG]: Reading from /var/lib/cloud/seed/nocloud/network-config (quiet=False)
cloud-init.log

```

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

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

The third deployment

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

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

```
#cloud-config
users:
  - name: cloud
    ssh-authorized-keys:
      - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCh66Gf1NuMeenGywiFUSW1T16uKW0IXnucNwoIynhymSm1fkTCqyxLkImWbyd/tD
6itcSa7jWvkKvTL1PvxLP8CvcvGR4a1V/2TuxA1em3JweqpNppyuapH7u9q0SdxaG2gh3uV1Yl/+8uuzJLJJbxb/a8EK+szpdZq7bpL0vig0TgMa
tk5LxfkBLxwLpFq7JPFv8DTMcdYqqc5PhRnnKLakSU060W0nv4fpa0MKuha1nr072Zyur7FRf9XFvD+Uc7ABNpeyUTZVIj2drShjjFTPfZWUC9
ldomain
    sudo: ['ALL=(ALL) NOPASSWD:ALL']
    groups: sudo
    shell: /bin/bash

packages:
  - httpd

runcmd:
  - echo "AllowUsers cloud" >> /etc/ssh/sshd_config
  - restart ssh
```

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

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

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

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

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

We should wait for a while so that the VM gets deployed. After that, let's log in and check whether the image deployed correctly. We asked for `ht t p d` to be installed during the deployment. Was it?

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

Figure 9.23 – Checking whether `ht t p d` is installed but not started

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

After the installation

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

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

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

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

Everything that is created at runtime is written here, and available for users. Our demo machine was run under the local KVM hypervisor so cloud-init is not detecting a cloud, and consequently is unable to provide more data about the cloud, but we can see some interesting details. The first one is two files named `enabled` and `network-config-ready`. Both of them are empty but very important. The fact that they exist signifies that cloud-init is enabled, and that network has been configured and is working. If the files are not there, something went wrong and we need to go back and debug. More about debugging can be found at <https://cloudinit.readthedocs.io/en/latest/topics/debugging.html>.

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

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

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

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

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

Figure 9.25 – CentOS cloud-init version – quite old

Ubuntu comes with a much fresher version, 19.3:

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

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

Before you freak out, this is not as bad as it seems. Cloud-init decided to switch its versioning system a couple of years ago, so after 0.7.9 came 17.1. There were many changes and most of them are directly connected to the cloud-init command and configuration files. This means that the deployment will work, but a lot of things after we deploy will not. Probably the most visible difference is when we run `cloud-init --help`. For Ubuntu, this is what it looks like:

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

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

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

Figure 2.27 – Cloud-init features on Ubuntu

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

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

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

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

Figure 9.28 – Cloud-init features on CentOS

Since our example has a total of three running instances – one Ubuntu and two CentOS virtual machines – let's try to manually upgrade to the latest stable version of cloud-init available on CentOS. We can use our regular `yum update` command to achieve that, and the result will be as follows:

```
[cloud@deploy-3 ~]$ cloud-init -v
/usr/bin/cloud-init 18.5
[cloud@deploy-3 ~]$ cloud-init --help
usage: /usr/bin/cloud-init [-h] [--version] [--file FILES] [--debug] [--force]
                          {init,modules,single,query,dhclient-hook,features,analyze,devel,collect-logs,clean,status}
                          ...

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

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

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

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

We are not going to go into too much detail about the cloud-init CLI tool, since there is simply too much information available for a book like this, and as we can see, new features are being added quickly. You can freely check additional options by browsing at <https://cloudinit.readthedocs.io/en/latest/topics/cli.html>. In fact, they are being added so quickly that there is a `devel` option that holds new features while they are in active development. Once they are finished, they become commands of their own.

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

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

```

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

```

Figure 9.30 – Checking time consumption for cloud-init procedures

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

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

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

A tool like this makes it easier to optimize deployments. In our particular case, almost none of this makes sense, since we deployed simple machines with almost no changes to the default configuration, but being able to understand why the deployment is slow is a useful, if not essential, thing.

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

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

Figure 9.32 – Checking the boot time

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

```
cloud@deploy-2:/var/log$ cloud-init query --all
{
  "beta_keys": [
    "subplatform"
  ],
  "availability_zone": null,
  "base64_encoded_keys": [],
  "cloud_name": "unknown",
  "ds": {
    "doc": "EXPERIMENTAL: The structure and format of content scoped under the 'ds' key may change in subsequent releases of cloud-init.",
    "meta_data": {
      "dsmode": "net",
      "instance_id": "nocloud",
      "local_hostname": "deploy-2"
    }
  },
  "instance_id": "nocloud",
  "local_hostname": "deploy-2",
  "platform": "nocloud",
  "public_ssh_keys": [],
  "region": null,
  "sensitive_keys": [],
  "subplatform": "config-disk (/dev/sr0)",
  "userdata": "<redacted for non-root user> file:/var/lib/cloud/instance/user-data.txt",
  "v1": {
    "beta_keys": [
      "subplatform"
    ],
    "availability_zone": null,
    "cloud_name": "unknown",
    "instance_id": "nocloud",
    "local_hostname": "deploy-2",
    "platform": "nocloud",
    "public_ssh_keys": [],
    "region": null,
    "subplatform": "config-disk (/dev/sr0)"
  },
  "vendordata": "<redacted for non-root user> file:/var/lib/cloud/instance/vendor-data.txt"
}
```

Figure 9.33 – Querying cloud-init for information

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

Summary

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

In the next chapter, we're going to learn how to expand this usage model to Windows virtual machines by using cloudbase-init.

Questions

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

Further reading

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

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

10

Automated Windows Guest Deployment and Customization

Now that we have covered the different ways of deploying Linux-based **Virtual Machines (VMs)** in KVM, it's time to switch our focus to Microsoft Windows. Specifically, we'll work on Windows Server 2019 machines running on KVM, and cover prerequisites and different scenarios for the deployment and customization of Windows Server 2019 VMs. This book isn't based on the idea of **Virtual desktop infrastructure (VDI)** and desktop operating systems, which require a completely different scenario, approach, and technical implementation than virtualizing server operating systems.

In this chapter, we will cover the following topics:

- The prerequisites to creating Windows VMs on KVM
- Creating Windows VMs using the `virt-install` utility
- The customization of Windows VMs using `cloudbase-init`
- `cloudbase-init` customization examples
- Troubleshooting common `cloudbase-init` customization issues

The prerequisites to creating Windows VMs on KVM

When starting the installation of a guest operating system on KVM virtualization, we always have the same starting points. We need either of the following:

- An ISO file with operating system installation
- An image with a VM template
- An existing VM to clone and reconfigure

Let's start from scratch. We are going to create a Windows Server 2019 VM in this chapter. Version selection was made to keep in touch with the most recent release of Microsoft server operating systems on the market. Our goal will be to deploy a Windows Server 2019 VM template that we can use later for more deployments and `cloudbase-init`, and the tool of choice for this installation process is going to be `virt-install`. If you need to install an older version (2016 or 2012), you need to know two facts:

- They are supported on CentOS 8 out of the box.
- The installation procedure is the same as it will be with our Windows Server 2019 VM.

If you want to use Virtual Machine Manager to deploy Windows Server 2019, make sure that you configure the VM properly. That includes selecting the correct ISO file for the guest operating system installation, and connecting another virtual CD-ROM for `virtio-win` drivers so that you can install them during the installation process. Make sure that your VM has enough disk space on the local KVM host (60 GB+ is recommended), and that it has enough horsepower to run. Start with two virtual CPUs and 4 GB of memory, as this can easily be changed later.

The next step in our scenario is to create a Windows VM that we'll use throughout this chapter to customize via `cloudbase-init`. In a real production environment, we need to do as much configuration in it as possible – driver installation, Windows updates, commonly used applications, and so on. So, let's do that first.

Creating Windows VMs using the virt-install utility

The first thing that we need to do is to make sure we have the `virtio-win` drivers ready for installation – the VM will not work properly without them installed. So, let's first install that and the `libguestfs` packages, in case you don't have them already installed on your server:

```
yum -y install virtio-win libguestfs*
```

Then, it's time to start deploying our VM. Here are our settings:

- The Windows Server 2019 ISO is located at `/iso/windows-server-2019.iso`.
- The `virtio-win` ISO file is located in the default system folder, `/usr/share/virtio-win/virtio-win.iso`.
- We are going to create a 60 GB virtual disk, located at the default system folder, `/var/lib/libvirt/images`.

Now, let's start the installation process:

```
virt-install --name WS2019 --memory=4096 --vcpus 2 --cpu host --video qxl --features=hyperv_relaxed=on,hyperv_spinlocks=on,hyperv_vapic=on --clock hypervclock_present=yes --disk /var/lib/libvirt/images/WS2019.qcow2,format=qcow2,bus=virtio,cache=none,size=60 --cdrom /iso/windows-server-2019.iso --disk /usr/share/virtio-win/virtio-win.iso,device=cdrom --vnc --os-type=windows --os-variant=win2k19 --accelerate --noapic
```

When the installation process starts, we have to click **Next** a couple of times before we reach the configuration screen where we can select the disk where we want to install our guest operating system. On the bottom of that screen to the left, there's a button called **Load driver**, which we can now use, repeatedly, to install all of the necessary `virtio-win` drivers. Make sure that you untick the **Hide drivers that aren't compatible with this computer's hardware** checkbox. Then, add the following drivers one by one, from a specified directory, and select them with your mouse:

- `AMD64\2k19`: **Red Hat VirtIO SCSI controller.**
- `Balloon\2k19\amd64`: **VirtIO balloon driver.**
- `NetKVM\2k19\AMD64`: **Red Hat VirtIO Ethernet adapter.**
- `qemufwcfg\2k19\amd64`: **QEMU FWCfg device.**
- `qemupcserial\2k19\amd64`: **QEMU Serial PCI card.**
- `vioinput\2k19\amd64`: **VirtIO input driver and VirtIO input driver helper;** select both of them.
- `viornrg\2k19\amd64`: **VirtIO RNG device.**
- `vioscsi\2k19\amd64`: **Red Hat VirtIO SCSI pass-through controller.**
- `vioserial\2k19\amd64`: **VirtIO serial driver.**
- `viostor\2k19\amd64`: **Red Hat VirtIO SCSI controller.**

After that, click **Next** and wait for the installation process to finish.

You might be asking yourself: *why did we micro-manage this so early in the installation process, when we could've done this later?* The answer is two-fold – if we did it later, we'd have the following problems:

- There's a chance – at least for some operating systems – that we won't have all the necessary drivers loaded before the installation starts, which might mean that the installation will crash.
- We'd have loads of yellow exclamation marks in **Device Manager**, which is usually annoying to people.

Being as it is after deployment, our device manager is happy and the installation was a success:

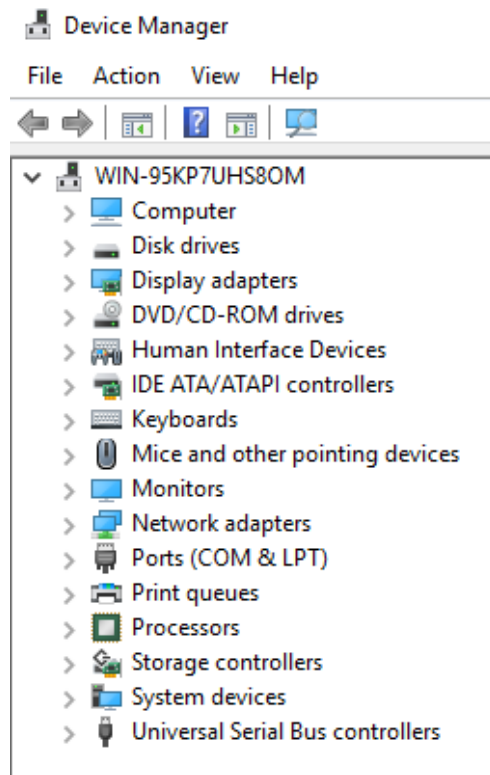


Figure 10.1 – The operating system and all drivers installed from the get-go

The only thing that's highly recommended post installation is that we install the guest agent from `virtio-win.iso` after we boot our VM. You will find an `.exe` file on the virtual CD-ROM, in `guest-agent` directory, and you just need to click the **Next** button until the installation is complete.

Now that our VM is ready, we need to start thinking about customization. Specifically, large-scale customization, which is a normal usage model for VM deployments in the cloud. This is why we need to use `cloudbase-init`, which is our next step.

Customizing Windows VMs using cloudbase-init

If you had the chance to go through *Chapter 9, Customizing a Virtual Machine with cloud-init*, we discussed a tool called `cloud-init`. What we used it for was guest operating system customization, specifically for Linux machines. `cloud-init` is heavily used in Linux-based environments, specifically in Linux-based clouds, to perform initialization and configuration of cloud VMs.

The idea behind `cloudbase-init` is the same, but it's aimed at Windows guest operating systems. Its base services start up when we boot a Windows guest operating system instance, as well as read through the configuration information and configure/initialize it. We are going to show a couple of examples of `cloudbase-init` operations a bit later in this chapter.

What can `cloudbase-init` do? The list of features is quite long, as `cloudbase-init` has a modular approach at its core, so it offers many plugins and interpreters, which can be used to further its reach:

- It can execute custom commands and scripts, most commonly coded in PowerShell, although regular CMD scripts are also supported.
- It can work with PowerShell remoting and the **Windows Remote Management (WinRM)** service.
- It can manage and configure disks, for example, to do a volume expansion.
- It can do basic administration, including the following:
 - a) Creating users and passwords
 - b) Setting up a hostname
 - c) Configuring static networking
 - d) Configuring MTU size
 - e) Assigning a license
 - f) Working with public keys
 - g) Synchronizing clocks

We mentioned earlier that our Windows Server 2019 VM is going to be used for cloudbase-init customization, so that's our next subject. Let's prepare our VM for cloudbase-init. We are going to achieve that by downloading the cloudbase-init installer and installing it. We can find the cloudbase-init installer by pointing our internet browser at <https://cloudbase-init.readthedocs.io/en/latest/intro.html#download>. The installation is simple enough, and it can work both in a regular, GUI fashion and silently. If you're used to using Windows Server Core or prefer silent installation, you can use the MSI installer for silent installation by using the following command:

```
msiexec /i CloudbaseInitSetup.msi /qn /l*v log.txt
```

Make sure that you check the cloudbase-init documentation for further configuration options as the installer supports additional runtime options. It's located at <https://cloudbase-init.readthedocs.io/en/latest/>.

Let's stick with the GUI installer as it's simpler to use, especially for a first-time user. First, the installer is going to ask about the license agreement and installation location – just the usual stuff. Then, we're going to get the following options screen:

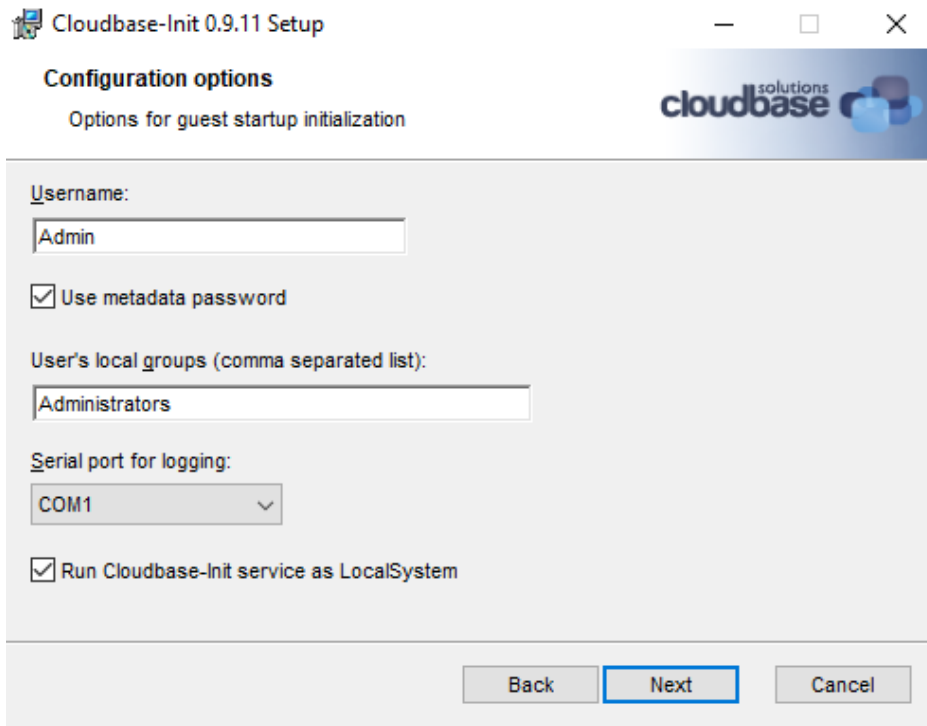


Figure 10.2 – Basic configuration screen

What it's asking us to do is to give permission to create the `cloudbase-init` configuration files (both `cloudbase-init-unattend.conf` and `cloudbase-init.conf`) with this specific future user in mind. This user will be a member of the local Administrators group and will be used for logging in when we start using the new image. This will be reflected in both of our configuration files, so if we select Admin here, that's the user that's going to get created. It's also asking us whether we want the `cloudbase-init` service to be run as a LocalSystem service, which we selected to make the whole process easier. The reason is really simple – this is the highest level of permission that we can give to our `cloudbase-init` services so that they can execute its operations. Translation: the `cloudbase-init` service will then be run as a LocalSystem service account, which has unlimited access to all local system resources.

The last configuration screen is going to ask us about running sysprep. Usually, we don't check the **Run sysprep to create a generalized image** box here, as we first need to create a `cloudbase-init` customization file and run sysprep after that. So, leave the following window open:

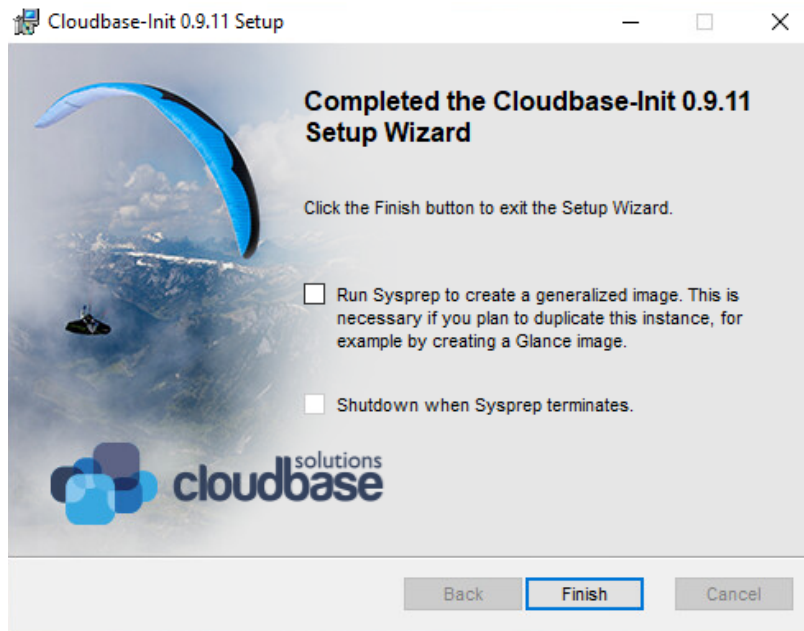


Figure 10.3 – The cloudbase-init installation wizard finishing up

Now that the `cloudbase-init` services are installed and configured, let's create a customization file that's going to configure this VM by using `cloudbase-init`. Again, make sure that this configuration screen is left open (with the completed setup wizard) so that we can easily start the whole process when we finish creating our `cloudbase-init` configuration.

cloudbase-init customization examples

After we finish the installation process, a directory with a set of files gets created in our installation location. For example, in our VM, a directory called `c:\Program Files\Cloudbase Solutions\Cloudbase-init\` was created, and it has the following set of subdirectories:

- `bin`: The location where some of the binary files are installed, such as `elevate`, `bsdtar`, `mcopy`, `mdir`, and so on.
- `conf`: The location of three main configuration files that we're going to work with, which is discussed a bit later.
- `LocalScripts`: The default location for PowerShell and similar scripts that we want to run post-boot.
- `Log`: The location where we'll store the `cloudbase-init` log files by default so that we can debug any issues.
- `Python`: The location where local installation of Python is deployed so that we can also use Python for scripting.

Let's focus on the `conf` directory, which contains our configuration files:

- `cloudbase-init.conf`
- `cloudbase-init-unattend.conf`
- `unattend.xml`

The way that `cloudbase-init` works is rather simple – it uses the `unattend.xml` file during the Windows sysprep phase to execute `cloudbase-init` with the `cloudbase-init-unattend.conf` configuration file. The default `cloudbase-init-unattend.conf` configuration file is easily readable, and we can use the example provided by the `cloudbase-init` project with the default configuration file explained step by step:

```
[DEFAULT]
# Name of the user that will get created, group for that user
username=Admin
groups=Administrators
firstlogonbehaviour=no
inject_user_password=true # Use password from the metadata
(not random).
```


The next part of the config file is about devices – specifically, which devices to inspect for a possible configuration drive (metadata):

```
config_drive_raw_hhd=true
config_drive_cdrom=true
# Path to tar implementation from Ubuntu.
bsdtar_path=C:\Program Files\Cloudbase Solutions\Cloudbase-
Init\bin\bsdtar.exe
mtools_path= C:\Program Files\Cloudbase Solutions\Cloudbase-
Init\bin\
```

We need to configure some settings for logging purposes as well:

```
# Logging level
verbose=true
debug=true
# Where to store logs
logdir=C:\Program Files (x86)\Cloudbase Solutions\Cloudbase-
Init\log\
logfile=cloudbase-init-unattend.log
default_log_levels=comtypes=INFO,suds=INFO,iso8601=WARN
logging_serial_port_settings=
```

The next part of the configuration file is about networking, so we'll use DHCP to get all the networking settings in our example:

```
# Use DHCP to get all network and NTP settings
mtu_use_dhcp_config=true
ntp_use_dhcp_config=true
```

We need to configure the location where the scripts are residing, the same scripts that we can use as a part of the `cloudbase-init` process:

```
# Location of scripts to be started during the process
local_scripts_path=C:\Program Files\Cloudbase Solutions\
Cloudbase-Init\LocalScripts\
```

The last part of the configuration file is about the services and plugins to be loaded, along with some global settings, such as whether to allow the `cloudbase-init` service to reboot the system or not and how we're going to approach the `cloudbase-init` shutdown process (`false=graceful service shutdown`):

```
# Services for loading
metadata_services=cloudbaseinit.metadata.services.configdrive.
ConfigDriveService, cloudbaseinit.metadata.services.
httpservice.HttpService,
cloudbaseinit.metadata.services.ec2service.EC2Service,
cloudbaseinit.metadata.services.maasservice.MaaSHttpService
# Plugins to load
plugins=cloudbaseinit.plugins.common.mtu.MTUPlugin,
        cloudbaseinit.plugins.common.sethostname.
SetHostNamePlugin
# Miscellaneous.
allow_reboot=false    # allow the service to reboot the system
stop_service_on_exit=false
```

Let's just get a couple of things out of the way from the get-go. Default configuration files already contain some settings that were deprecated, as you're going to find out soon enough. Specifically, settings such as `verbose`, `logdir` and `logfile` are already deprecated in this release, as you can see from the following screenshot, where `cloudbase-init` is complaining about those very options:

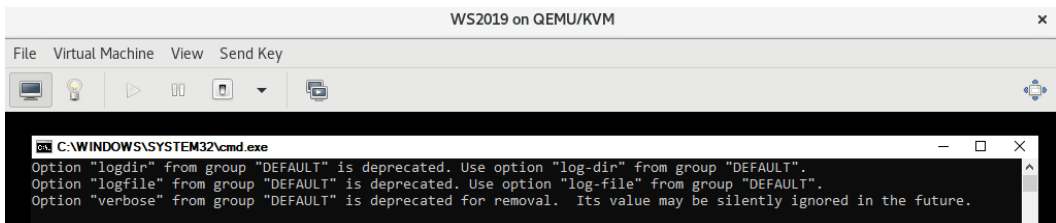


Figure 10.4 – `cloudbase-init` complaining about its own default configuration file options

If we want to start sysprepping with `cloudbase-init` by using the default configuration files, we are actually going to get a pretty nicely configured VM – it's going to be sysprepped, it's going to reset the administrator password and ask us to change it with the first login, and remove the existing administrator user and its directories. So, before we do this, we need to make sure that we save all of our administrator user settings and data (documents, installers, downloads, and so on) someplace safe. Also, the default configuration files will not reboot the VM by default, which might confuse you. We need to do a manual restart of the VM so that the whole process can start.

The easiest way to work with both `cloud-init` and `cloudbase-init` is by writing down a scenario of what needs to be done to the VM as it gets through the initialization process. So, we'll do just that – pick a load of settings that we want to be configured and create a customization file accordingly. Here are our settings:

- We want our VM to ask us to change the password post-sysprep and after the `cloudbase-init` process.
- We want our VM to take all of its network settings (the IP address, netmask, gateway, DNS servers, and NTP) from DHCP.
- We want to sysprep the VM so that it's unique to each scenario and policy.

So, let's create a `cloudbase-init-unattend.conf` config file that will do this for us. The first part of the configuration file was taken from the default config file:

```
[DEFAULT]
username=Admin
groups=Administrators
inject_user_password=true
config_drive_raw_hhd=true
config_drive_cdrom=true
config_drive_vfat=true
bsdtar_path=C:\Program Files\Cloudbase Solutions\Cloudbase-
Init\bin\bsdtar.exe
mtools_path= C:\Program Files\Cloudbase Solutions\Cloudbase-
Init\bin\
debug=true
default_log_levels=comtypes=INFO,suds=INFO,iso8601=WARN
logging_serial_port_settings=
mtu_use_dhcp_config=true
ntp_use_dhcp_config=true
```

As we decided to use PowerShell for all of the scripting, we created a separate directory for our PowerShell scripts:

```
local_scripts_path=C:\PS1
```

The rest of the file was also just copied from the default configuration file:

```
metadata_services=cloudbaseinit.metadata.services.base.  
EmptyMetadataService  
plugins=cloudbaseinit.plugins.common.mtu.MTUPlugin,  
        cloudbaseinit.plugins.common.sethostname.  
SetHostNamePlugin, cloudbaseinit.plugins.common.localscripts.  
LocalScriptsPlugin,cloudbaseinit.plugins.common.userdata.  
UserDataPlugin  
allow_reboot=false  
stop_service_on_exit=false
```

As for the `cloudbase-init.conf` file, the only change that we made was selecting the correct local script path (reasons to be mentioned shortly), as we will use this path in our next example:

```
[DEFAULT]  
username=Admin  
groups=Administrators  
inject_user_password=true  
config_drive_raw_hhd=true  
config_drive_cdrom=true  
config_drive_vfat=true
```

Also, part of our default config file contained paths for `tar`, `mttools`, and `debugging`:

```
bsdtar_path=C:\Program Files\Cloudbase Solutions\Cloudbase-  
Init\bin\bsdtar.exe  
mttools_path= C:\Program Files\Cloudbase Solutions\Cloudbase-  
Init\bin\  
debug=true
```

This part of the config file was also taken from the default config file, and we only changed `local_scripts_path` so that it's set to the directory that we're using to populate with PowerShell scripts:

```
first_logon_behaviour=no
default_log_levels=comtypes=INFO,suds=INFO,iso8601=WARN
logging_serial_port_settings=
mtu_use_dhcp_config=true
ntp_use_dhcp_config=true
local_scripts_path=C:\PS1
```

We can then go back to the `cloudbase-init` installation screen, check the `sysprep` option, and click **Finish**. After starting the `sysprep` process and going through with it, this is the end result:

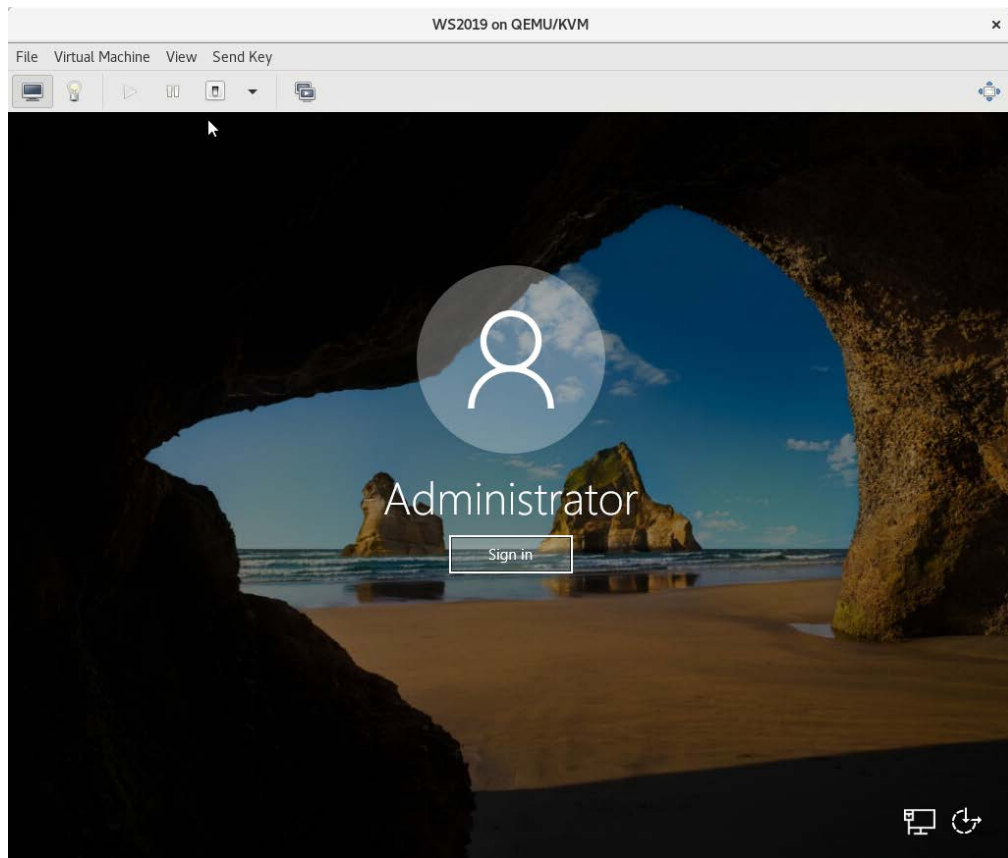


Figure 10.5 – When we press Sign in, we are going to be asked to change the administrator's password

Now, let's take this a step further and complicate things a bit. Let's say that you want to do the same process, but with additional PowerShell code that should do some additional configuration. Consider the following example:

- It should create another two local users called `packt1` and `packt2`, with a predefined password set to `Pa$w0rd`.
- It should create a new local group called `students`, and add `packt1` and `packt2` to this group as members.
- It should set the hostname to `Server1`.

The PowerShell code that enables us to do this should have the following content:

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Force
$password = "Pa$w0rd" | ConvertTo-SecureString -AsPlainText
-Force
New-LocalUser -name "packt1" -Password $password
New-LocalUser -name "packt2" -Password $password
New-LocalGroup -name "Students"
Add-LocalGroupMember -group "Students" -Member
"packt1", "packt2"
Rename-Computer -NewName "Server1" -Restart
```

Taking a look at the script itself, this is what it does:

- Sets the PowerShell execution policy to unrestricted so that our host doesn't stop our script execution, which it would do by default.
- Creates a password variable from a plaintext string (`Pa$w0rd`), which gets converted to a secure string that we can use with the `New-LocalUser` PowerShell cmdlet to create a local user.
- `New-LocalUser` is a PowerShell cmdlet that creates a local user. Mandatory parameters include a username and password, which is why we created a secure string.
- `New-LocalGroup` is a PowerShell cmdlet that creates a local group.
- `Add-LocalGroupMember` is a PowerShell cmdlet that allows us to create a new local group and add members to it.
- `Rename-Computer` is a PowerShell cmdlet that changes the hostname of a Windows computer.

We also need to call this code from `cloudbase-init` somehow, so we need to add this code as script. Most commonly, we'll use a directory called `LocalScripts` in the `cloudbase-init` installation folder for that. Let's call this script `userdata.ps1`, save the content mentioned previously to it in the folder, as defined in the `.conf` file (`c:\PS1`), and add a `cloudbase-init` parameter at the top of the file:

```
# ps1
$password = "Pa$$w0rd" | ConvertTo-SecureString -AsPlainText
-Force
New-LocalUser -name "packt1" -Password $password
New-LocalUser -name "packt2" -Password $password
New-LocalGroup -name "Students"
Add-LocalGroupMember -group "Students" -Member
"packt1", "packt2"
Rename-Computer -NewName "Server1" -Restart
```

After starting the `cloudbase-init` procedure again, which can be achieved by starting the `cloudbase-init` installation wizard and going through it as we did in the previous example, here's the end result in terms of users:

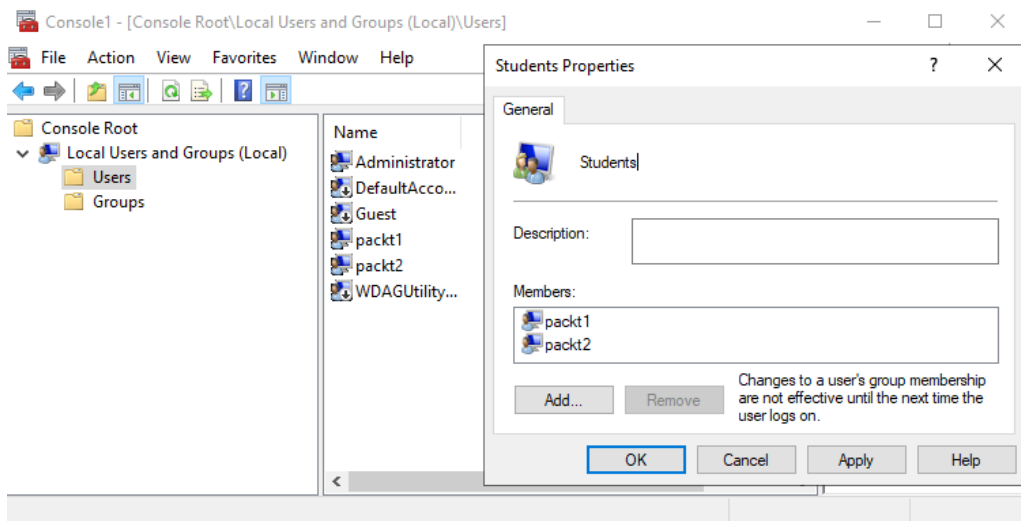


Figure 10.6 – The `packt1` and `packt2` users were created, and added to the group created by our PowerShell script

We can clearly see that the `packt1` and `packt2` users were created, along with a group called `Students`. We can then see that the `Students` group has two members – `packt1` and `packt2`. Also, in terms of setting the server name, we have the following:

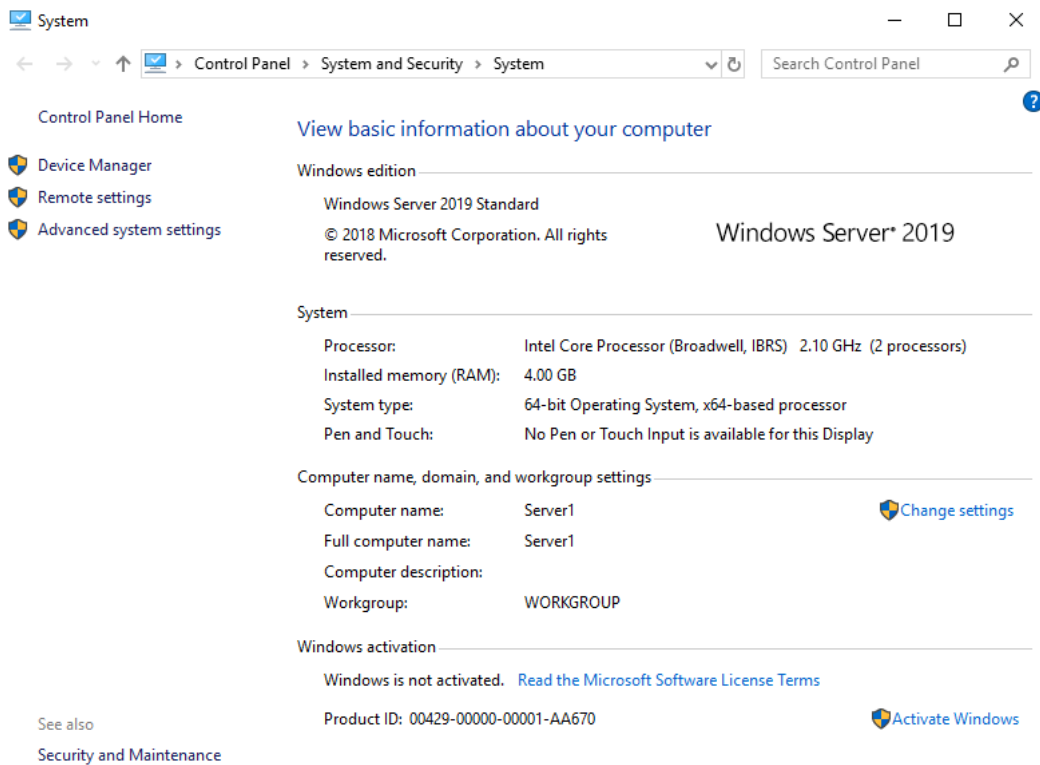


Figure 10.7 – Slika 1. Changing the server name via PowerShell script also works

Using `cloudbase-init` really isn't simple, and requires a bit of investment in terms of time and tinkering. But afterward, it will make our job much easier – not being forced to do pedestrian tasks such as these over and over again should be a reward enough, which is why we need to talk a little bit about troubleshooting. We're sure that you'll run into these issues as you ramp up your `cloudbase-init` usage.

Troubleshooting common cloudbase-init customization issues

In all honesty, you can freely say that the `cloudbase-init` documentation is not all that good. Finding examples of how to execute PowerShell or Python code is difficult at best, while the official page doesn't really offer any help in that respect. So, let's discuss some of the most common errors that happen while using `cloudbase-init`.

Although this seems counter-intuitive, we had much more success getting `cloudbase-init` to work with the latest development version instead of the latest stable one. We're not exactly sure what the problem is, but the latest development version (at the time of writing, this is version 0.9.12.dev125) worked for us right out of the gate. With version 0.9.11., we had massive issues with getting the PowerShell script to even start.

Apart from these issues, there are other issues that you will surely encounter as you get to know `cloudbase-init`. The first one is the reboot loop. This problem is really common, and it almost always happens because of two reasons:

- A mistake in the configuration file – a wrongly typed name of a module or an option, or something like that
- A mistake in some external file (location or syntax) that's being called as an external script to be executed in the `cloudbase-init` process

Making a mistake in configuration files is something that happens often, which throws `cloudbase-init` into a weird state that ends up like this:

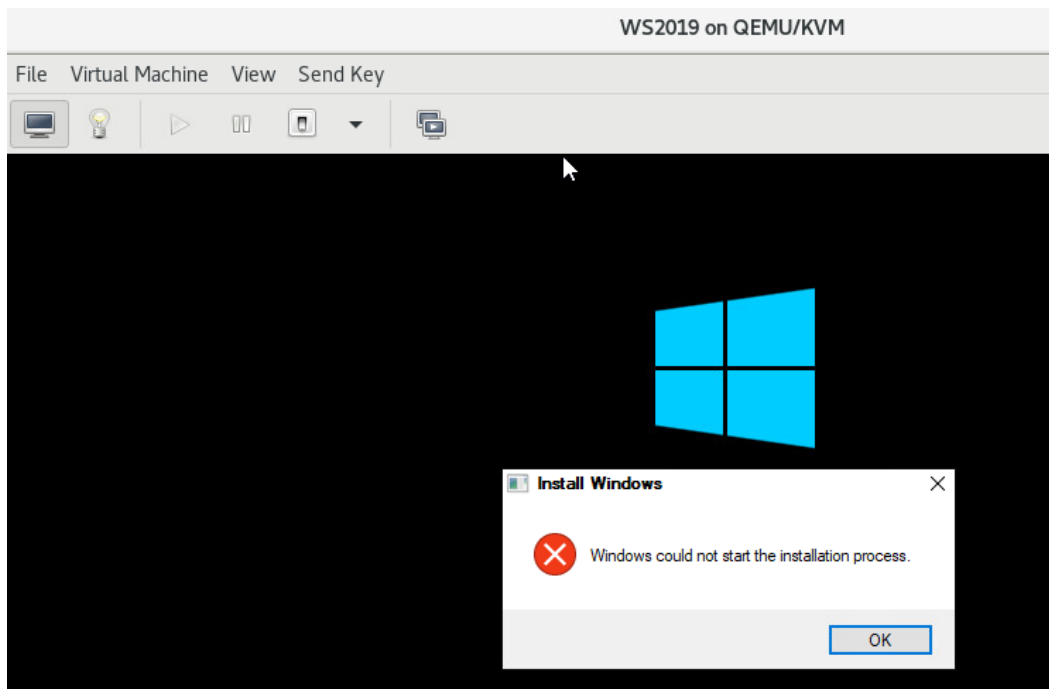


Figure 10.8 – Error in configuration

We've seen this situation multiple times. The real problem is the fact that sometimes it takes hours and hours of waiting, sometimes cycling through numerous reboots, but it's not just a regular reboot loop. It really seems that `cloudbase-init` is doing something – the CMD is started, you get no errors in it or on the screen, but it keeps doing something and then finishes like this.

Other issues that you might encounter are even more picky – for example, when `cloudbase-init` fails to reset the password during the `sysprep/cloudbase-init` process. This can happen if you manually change the account password that's being used by the `cloudbase-init` service (hence, why using `LocalSystem` is a better idea). That will lead to the failure of the whole `cloudbase-init` procedure, a part of which can be a failure to reset the password.

There's an even more obscure reason why this might happen – sometimes we manually manage system services by using the `services.msc` console and we deliberately disable services that we don't immediately recognize. If you set the `cloudbase-init` service to be disabled, it will fail in its process, as well. These services need to have automatic startup priority and shouldn't be manually reconfigured to be disabled.

A failure to reset the password can also happen because of some security policies – for example, if the password isn't complex enough. That's why we used a bit more of a complex password in our PowerShell script, as most of us system engineers learned that lesson a long time ago.

Also, sometimes companies have different security policies in place, which can lead to a situation in which some management application that takes care of – for example, software inventory – stops the `cloudbase-init` service or completely uninstalls it.

The most frustrating error that we can encounter is an error where the `cloudbase-init` process doesn't start scripts from its designated folder. After spending hours perfecting your Python, `bash`, `cmd`, or PowerShell script that needs to be added to the customization process, it's always maddening to see this happen. In order for us to be able to use these scripts, we need to use a specific plugin that is able to call the external script and execute it. That's why we usually use `UserDataPlugin` – both for executing and debugging reasons – as it can execute all of these script types and give us an error value, which we can then use for debugging purposes.

One last thing – make sure that you don't insert PowerShell code directly into the `cloudbase-init` configuration files in the `conf` folder. You'll only get a reboot loop as a reward, so be careful about that.

Summary

In this chapter, we worked with Windows VM customization, a topic that's equally as important as Linux VM customization. Maybe even more so, keeping in mind the market share numbers and the fact that a lot of people are using Windows in cloud environments, as well.

Now that we have covered all the bases in terms of working with VMs, templating, and customization, it's time to introduce a different approach to additional customization that's complementary to `cloud-init` and `cloudbase-init`. So, the next chapter is about that approach, which is based around Ansible.

Questions

1. Which drivers do we need to install onto Windows guest operating systems so that we can make a Windows template on the KVM hypervisor?
2. Which agent do we need to install onto Windows guest operating systems to have better visibility into the VM's performance data?
3. What is `sysprep`?
4. What is `cloudbase-init` used for?
5. What are the regular use cases for `cloudbase-init`?

Further reading

Please refer to the following links for more information:

- Microsoft LocalSystem account documentation: <https://docs.microsoft.com/en-us/windows/win32/ad/the-localsystem-account>
- `cloudbase-init` documentation: <https://cloudbase-init.readthedocs.io/en/latest/intro.html>
- The `cloudbase-init` plugin documentation: <https://cloudbase-init.readthedocs.io/en/latest/plugins.html>
- The `cloudbase-init` services documentation: <https://cloudbase-init.readthedocs.io/en/latest/services.html>

11

Ansible and Scripting for Orchestration and Automation

Ansible has become the de facto standard in today's open source community because it offers so much while asking so little of you and your infrastructure. Using Ansible with **Kernel-based Virtual Machine (KVM)** also makes a lot of sense, especially when you think about larger environments. It doesn't really matter if it's just a simple provisioning of KVM hosts that you want to do (install libvirt and related software), or if you want to uniformly configure KVM networking on hosts – Ansible can be invaluable for both. For example, in this chapter, we will use Ansible to deploy a virtual machine and multi-tier application that's hosted inside KVM virtual machines, which is a very common use case in larger environments. Then, we'll move to more pedantic subjects of combining Ansible and cloud-init since they differ in terms of timeline when they're applied and a way in which things get done. Cloud-init is an ideal automatic way for initial virtual machine configuration (hostname, network, and SSH keys). Then, we usually move to Ansible so that we can perform additional orchestration post-initial configuration – add software packages, make bigger changes to the system, and so on. Let's see how we can use Ansible and cloud-init with KVM.

In this chapter, we will cover the following topics:

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

Let's get started!

Understanding Ansible

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

Let's try to explain this with an example as this is the most convenient way of describing the problem and solution. Let's say that you're working for a company that needs to deploy 50 web servers to host a web application, with standard configuration. Standard configuration includes the software packages that you need to install, the services and network settings that need to be configured, the firewall rules that need to be configured, and the files that need to be copied from a network share to a local disk inside a virtual machine so that we can serve these files via a web server. How are you going to make that happen?

There are three basic approaches that come to mind:

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

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

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

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

Automation approaches

In general, all of this applies to administering systems and their parts, installing applications, and generally taking care of things inside the installed system. This can be considered an *old* approach to administration since it generally deals with services, not servers. At the same time, this kind of automation is decidedly focused on a single server or a small number of servers since it doesn't scale well. If we need to work on multiple servers, using regular scripts creates new problems. We need to take a lot of additional variables into account (different SSH keys, hostnames, and IP addresses) since scripts are more difficult to expand to work on multiple servers (which is easy in Ansible).

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

There's another kind of automation that is gathering traction in the DevOps community – Automation with a capital A. This is a way to automate systems operation across different machines – even across different operating systems. There are a couple of automation systems that enable this, and they can basically be divided into two groups: systems that use agents and agentless systems.

Systems that use agents

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

Almost all of them work in the same way. A small application – called an agent – is installed on the system that we need to monitor. After the application has been installed, it connects, or permits connections, from the central server, which handles everything regarding automation. Since you are reading this, you are probably familiar with systems like this. There are quite a few of them around, and chances are you've already run into one of them. To understand this principle, take a look at the following diagram:

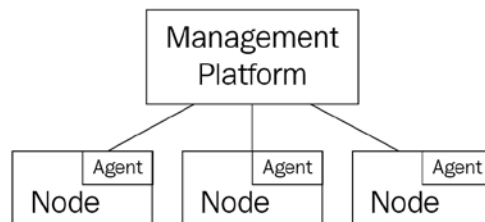


Figure 11.1 – The management platform needs an agent to connect to objects that need orchestration and automation

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

Agentless systems

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

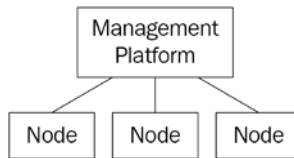


Figure 11.2 – The management platform doesn't need an agent to connect to objects that need orchestration and automation

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

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

Introduction to Ansible

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

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

This also means that Ansible has no way of detecting changes on the systems it controls; it is completely up to the configuration script we create to control what happens if something is not as we expect it to be.

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

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

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

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

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

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

The following architecture shows all of these parts working together:

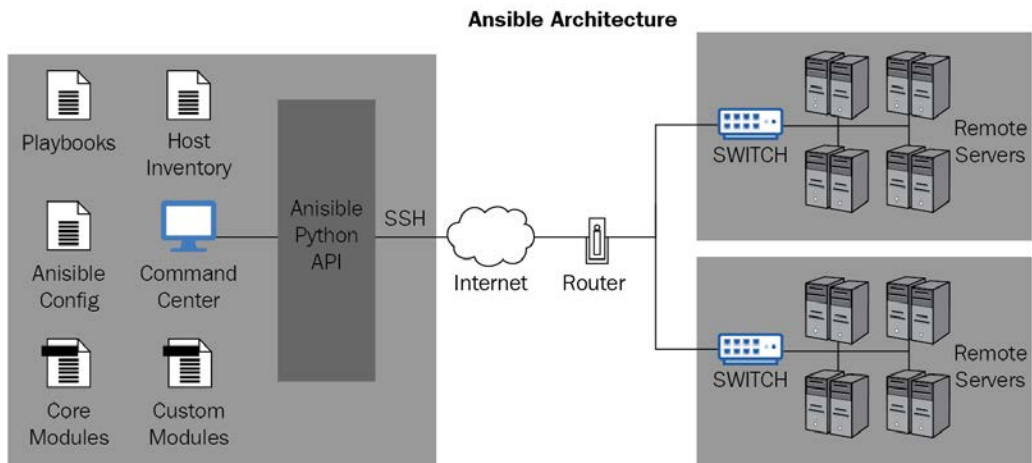


Figure 11.3 – Ansible architecture – Python API and SSH connections

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

Ansible Tower is a GUI-based utility that you can use to manage your Ansible-based environments. This started as a project called **AWX**, which is still very much alive today. But there are some key differences in the way in which AWX gets released versus how Ansible Tower gets released. The main one is the fact that Ansible Tower uses specific release versions while AWX takes a more *what OpenStack used to be* approach – a project that's moving forward rather quickly and has new releases very often.

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

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

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

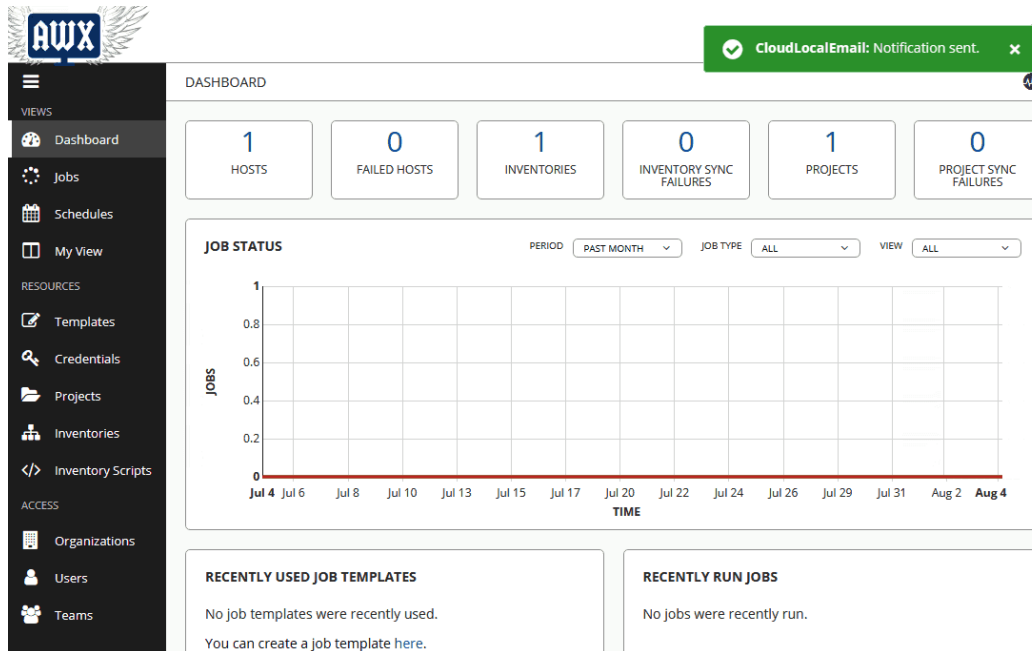


Figure 11.4 – Ansible AWX GUI for Ansible

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

Deploying and using AWX

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

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

The single-most important address we need to know about when we are talking about AWX is `https://github.com/ansible/awx`. This is the place where the project resides. The most up-to-date information is here, in `readme.md`, a file that is shown on the GitHub page. If you are unfamiliar with *cloning* from GitHub, do not worry – we are basically just copying from a special source that will enable you to copy only the things that have changed since you last got your version of the files. This means that in order to update to a new version, you only need to clone once more using the same exact command.

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

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

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

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

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

  git checkout -b <new-branch-name>

[root@awxdemo ~]#
```

Figure 11.5 – Git cloning the AWX files

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

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

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

Docker is the first one we will install. We are using CentOS 8 for this, so Docker is no longer part of the default set of packages. Therefore, we need to add the repository and then install the Docker engine. We are going to use the `-ce` package, which stands for Community Edition. We will also use the `--nobest` option to install Docker – without this option, CentOS will report that we are missing some dependencies:

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

Figure 11.6 – Deploying docker-ce package on CentOS 8

After that, we need to run the following command:

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

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

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

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

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

Figure 11.7 – Starting and enabling the Docker service

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

```
dnf install ansible
```

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

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

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

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

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

We also need to install the `make` package:

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

Figure 11.9 – Deploying GNU Make

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

```
curl -L https://github.com/docker/compose/releases/
download/1.25.0/docker-compose-`uname -s`-`uname -m` -o /usr/
local/bin/docker-compose
```

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

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

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

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

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

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

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

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

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

```
[root@awxdemo installer]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS            PORTS              NAMES
baldabec8a2a      ansible/awx:13.0.0 "tini -- /usr/bin/l... 4 minutes ago
Up 4 minutes      8052/tcp          awx_task
a5ba16f3529f      ansible/awx:13.0.0 "tini -- /bin/sh -c ... 4 minutes ago
Up 4 minutes      0.0.0.0:80->8052/tcp awx_web
8d5ae3c600f8      redis              "docker-entrypoint.s... 4 minutes ago
Up 4 minutes      6379/tcp          awx_redis
09ed8671d88a      postgres:10       "docker-entrypoint.s... 4 minutes ago
Up 4 minutes      5432/tcp          awx_postgres
```

Figure 11.10 – Checking the pulled and started docker images

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

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

Figure 11.11 – Checking the `awx_task` logs

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

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

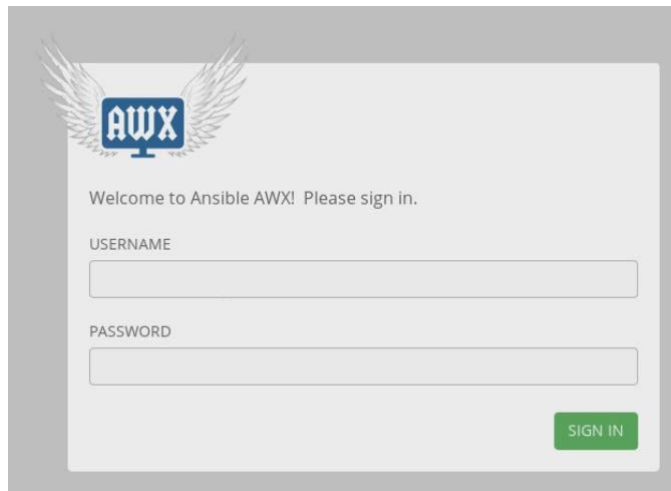


Figure 11.12 – AWX default login screen

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

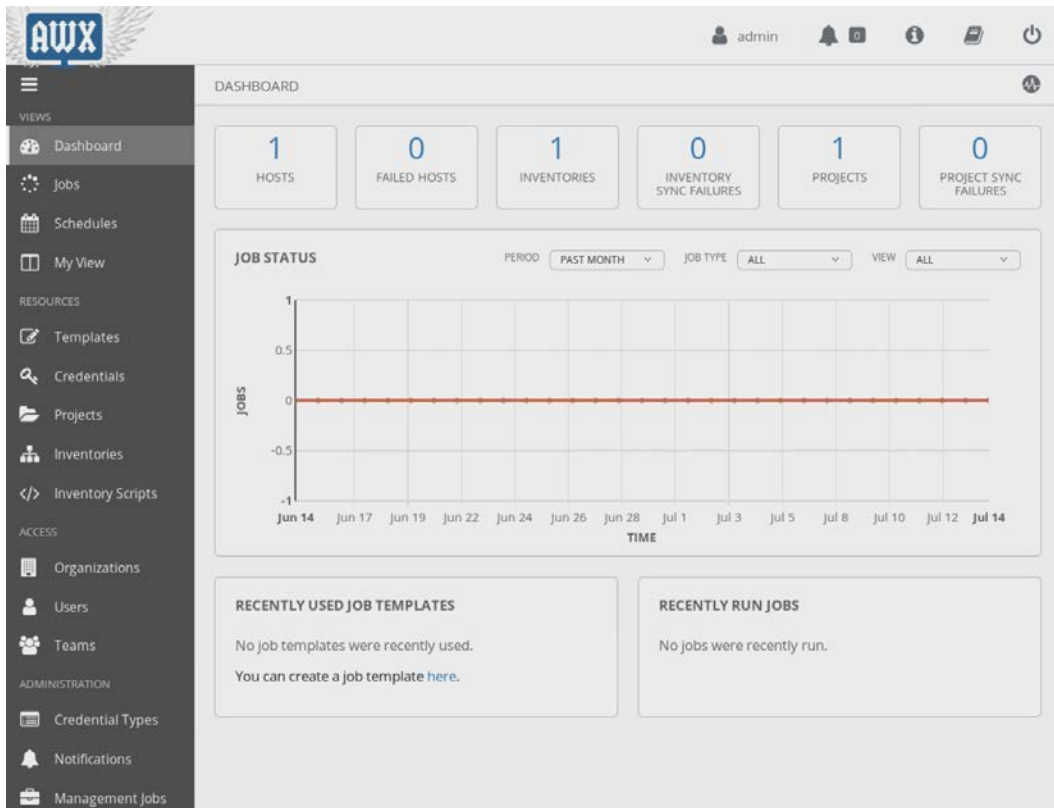


Figure 11.13 – Initial AWX dashboard after logging in

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

Figure 11.14 – Using a demo template in AWX

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

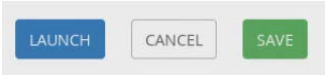
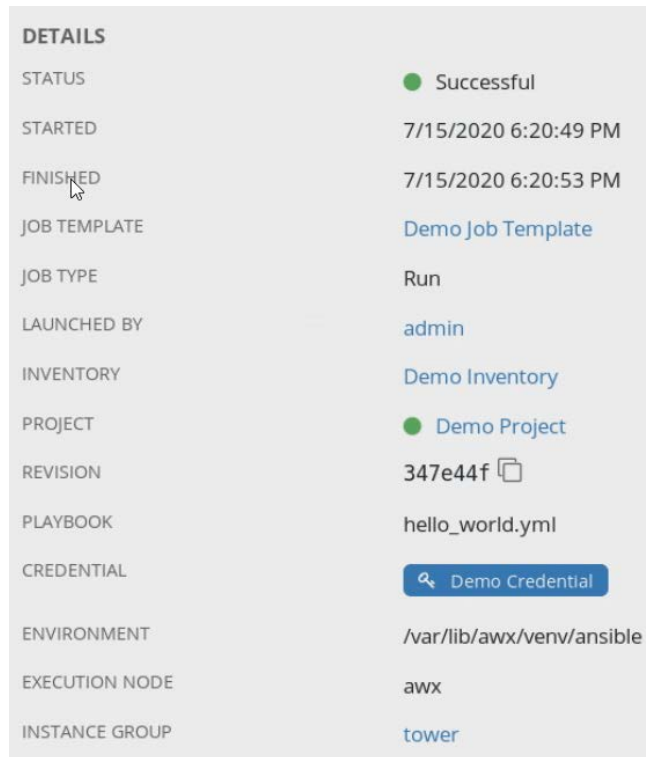


Figure 11.15 – By clicking on the Launch button, we can start our template job

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





DETAILS	
STATUS	● Successful
STARTED	7/15/2020 6:20:49 PM
FINISHED	7/15/2020 6:20:53 PM
JOB TEMPLATE	Demo Job Template
JOB TYPE	Run
LAUNCHED BY	admin
INVENTORY	Demo Inventory
PROJECT	● Demo Project
REVISION	347e44f 
PLAYBOOK	hello_world.yml
CREDENTIAL	 Demo Credential
ENVIRONMENT	/var/lib/awx/venv/ansible
EXECUTION NODE	awx
INSTANCE GROUP	tower

Figure 11.16 – Template job details

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

```

1
2 PLAY [Hello World Sample] 18:20:51
*****
3
4 TASK [Gathering Facts] 18:20:51
*****
5 ok: [localhost]
6
7 TASK [Hello Message] 18:20:52
*****
8 ok: [localhost] => {
9   "msg": "Hello World!"
10 }
11
12 PLAY RECAP 18:20:
*****
13 localhost : ok=2   changed=0   unreachable=0   f
   ailed=0   skipped=0   rescued=0   ignored=0
14

```

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

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

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

Deploying Ansible

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

Other than that, there are no databases (Ansible uses text files), no daemons (Ansible runs on demand), and no management of Ansible itself to speak of. Since nothing is running in the background, Ansible is easily upgraded – the only thing that can change is the way playbooks are structured, and that can easily be fixed. Ansible is based on the Python programming language, but its structure is simpler than that of a standard Python program. Configuration files and playbooks are either simple text files or YAML formatted text files, with YAML being a file format used to define data structures. Learning YAML is outside the scope of this chapter, so we will just presume that you understand simple data structures. The YAML files we'll be using as examples are simple enough to warrant almost no explanation, but if one is needed, it will be provided.

The installation can be as simple as running the following:

```
yum install ansible
```

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

```
apt install ansible
```

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

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

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

After running the preceding command, use the following code:

```
dnf install ansible
```

This is all it takes to install Ansible.

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

The machine that Ansible is installed in is also called the *control node*. It must be installed on a Linux host as Windows is not supported in this role. Ansible control nodes can be run inside virtual machines.

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

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

Provisioning a virtual machine using the `kvm_libvirt` module

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

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

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

- The first is a key fingerprint, as seen from the Ansible control node (server). When establishing a connection for the first time, SSH requires the user to verify and accept keys that the remote system presents. This is designed to prevent MITM attacks and is a good tactic in everyday use. But if we are in the position of having to configure freshly installed systems, *all* of them will require for us to accept their keys. This is time-consuming and complicated to do once we start using playbooks, so the first playbook you will start is probably going to disable key checks and logging into machines. Of course, this should only be used in a controlled environment since this lowers the security of the whole Ansible system.
- The second thing you need to know is that Ansible runs as a normal user. Having said that, maybe we do not want to connect to the remote systems as the current user. Ansible solves that by having a variable that can be set on individual computers or groups that indicates what username the system is going to use to connect to this particular computer. After connecting, Ansible allows us to execute commands on the remote system as a different user entirely. This is something that is commonly used since it enables us to reconfigure the machine completely and change users as if we were at the console.
- The third thing that we need to remember are the keys – SSH can log in by using interactive authentication, meaning via password or by using pre-shared keys that are exchanged once and then reused to establish the SSH session. There is also `ssh-agent`, which can be used to authenticate sessions.

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

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

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

Now, let's dig deeper.

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

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

```
[servers]
srv1.local
srv2.local
srv3.local

[workstations]
wrk1.local
wrk2.local
wrk3.local
```

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

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

```
All:
  Servers:
    Hosts:
      Srv1.local:
      Srv2.local:
      Srv3.local:

  Workstations:
    Hosts:
      Wrk1.local:
      Wrk2.local:
      Wrk3.local:

  Production:
    Hosts:
      Srv1.local:

    Workstations:
```


Important Note

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

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

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

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

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

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

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

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

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

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

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

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

1. First, we define a group:

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

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

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

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

Important Note

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

Now that we've created our first practical Ansible task, it's time to talk about how to make Ansible do many things at once while using a more *objective* approach. It's important to be able to create a single task or a couple of tasks that we can combine through a concept called a *playbook*, which can include multiple tasks/plays.

Working with playbooks

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

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

Ansible is installed on the system with the IP address 10.0.0.1, but as we already said, this is completely arbitrary. Ansible has a minimal footprint on the system that is used as a control node and can be installed on any system as long as it has connectivity to the rest of the network we are going to manage. We simply chose the first computer in our small network. One more thing to note is that the control node can be controlled by itself through Ansible. This is useful, but at the same time not a good thing to do. Depending on your setup, you will want to test not only playbooks, but individual commands before they are deployed to other machines – doing that on your control server is not a wise thing to do.

Now that Ansible is installed, we can try and do something with it. There are two distinct ways that Ansible can be run. One is by running a playbook, a file that contains tasks that are to be performed. The other way is by using a single task, sometimes called **ad hoc** execution. There are reasons to use Ansible either way – playbooks are our main tool, and you will probably use them most of the time. But ad hoc execution also has its advantages, especially if we are interested in doing something that we need done once, but across multiple servers. A typical example is using a simple command to check the version of an installed application or application state. If we need it to check something, we are not going to write a playbook.

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

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

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

```
ansible all -m ping
```

The following is the result of running the preceding command:

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

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

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

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

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

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

When Ansible runs, it tries to detect whether it ran correctly and whether it has changed the system state. In this particular task, the command that ran is just informative and does not change anything on the system, so the system state was unchanged.

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

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

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

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

The following is the output:

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

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

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

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

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

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

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

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

To prove this, we are going to see if the service is *not* installed or *absent* in Ansible terms. Remember that if the service was installed, this will uninstall it. Type in the following command:

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

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

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

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

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

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

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

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

Now, let's run the command again:

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

This should be the result:

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

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

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

These were all just starting examples so that we could get to know Ansible a little bit. Now, let's expand on this and create an Ansible playbook that's going to install KVM on our predefined set of hosts.

Installing KVM

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

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

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

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

    roles:
      - checkVirtualization
      - installKVM
```

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

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

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

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

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

2. When we downloaded the script from the GitHub, it created a few folders. In the one named `roles`, there are two subfolders that each contain a file; one is called `checkVirtualization` and the other is called `installKVM`.

You can probably already see where this is heading. First, let's see what `checkVirtualization` contains:

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

Figure 11.24 – Checking for CPU virtualization via the `lscpu` command

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

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

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

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

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

Figure 11.25 – Ansible task for installing the necessary `libvirt` packages

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

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

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

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

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

```
PLAY [Install KVM] *****
TASK [Gathering Facts] *****
ok: [10.0.0.3]
ok: [10.0.0.2]
ok: [10.0.0.4]
ok: [10.0.0.1]

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

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

Figure 11.26 – Interactive Ansible playbook monitoring

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

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

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

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

Figure 11.27 – Ansible playbook report

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

- We are going to start `virsh` and list the active VMs on all the parts of the cluster:

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

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

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

We are going to do two things: first, we are going to download a minimal ISO image for CentOS from the internet. Then, we are going to call `virsh`. This book will show you different ways to accomplish this task; downloading from the internet is one of the slowest:

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

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

Figure 11.29 – Downloading files in Ansible playbooks

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

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

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

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

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

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

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

```
ansible all -m shell -a "virt-install --name=COS7Core
--ram=2048 --vcpus=4 --cdrom=/var/lib/libvirt/boot/
CentOS-7-x86_64-Minimal-1810.iso --os-type=linux"
```

```
--os-variant=rhel7 --disk path=/var/lib/libvirt/images/
cos7vm.dsk,size=6"
```

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

Now, we can check if everything worked:

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

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

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

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

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

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

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

Using Ansible and cloud-init for automation and orchestration

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

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

Cloud-init works using two separate parts: one is the distribution file for the operating system we are deploying. This is not the usual OS installation file, but a specially configured machine template intended to be used as a cloud-init image.

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

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

The way this works is simple:

1. First, we distribute a machine template that is completely identical for all the machines that we are going to create. This means having one master copy and creating all the instances out of it.
2. Then, we pair the template with a specially crafted file that is created using cloud-init. Our template, regardless of the OS it uses, is capable of understanding different directives that we can set in the cloud-init file and will be reconfigured. This can be repeated as needed.

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

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

```
#cloud-config
package_upgrade: true
users:
  - name: ansible
    groups: wheel
    lock_passwd: false
    passwd: F1AppspmE+Lz8lMLW2PK5ohcuogevH
    shell: /bin/bash
    sudo: ['ALL=(ALL) NOPASSWD:ALL']
    ssh-authorized-keys:
      - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDf7E8+1x0AW3Wxez0mx1t0rLx
```

Figure 11.32 – Using cloud-init for additional configuration

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 11.33 – Checking the content of a directory

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

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

```
---
- name: download Centos core image
  hosts: cloudhosts
  tasks:
  - name: Copy to cloud instance
    copy:
      src: /clouddeploy/CentOS-7-x86_64-GenericCloud-1907.qcow2
      dest: /var/lib/libvirt/images/cloudsrv1/
      owner: qemu
      group: qemu
      mode: u=rw,g=rw,o=r
  - name: Copy cloud-init configuration
    copy:
      src: /clouddeploy/config.iso
      dest: /var/lib/libvirt/images/cloudsrv1/
      owner: qemu
      group: qemu
      mode: u=rw,g=rw,o=r
  - name: Create machine
    command: >
      virt-install --name=COS7Cloud --ram=1024 --vcpus=1 --os-type=linux --os-variant=rhel7
      --disk path=/var/lib/libvirt/images/cloudsrv1/CentOS-7-x86_64-GenericCloud-1907.qcow2,device=disk
      --disk /var/lib/libvirt/images/cloudsrv1/config.iso,device=cdrom --graphics none --import --noautoconsole
  - name: Start VM
    virt:
      name: COS7Cloud
      state: running
```

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

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

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

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

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

Back to the important part – the local installation. We are creating a machine with 1 GB of RAM and one CPU using the disk image we just copied. We're also attaching our `config.iso` file as a virtual CD/DVD. We are then importing this image and using no graphic terminal.

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

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

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

```
[root@vm0-101 ~]# ansible-playbook installvms.yaml
PLAY [download Centos core image] *****
TASK [Gathering Facts] *****
ok: [10.0.0.4]
ok: [10.0.0.2]
ok: [10.0.0.3]
TASK [Copy to cloud instance] *****
ok: [10.0.0.2]
ok: [10.0.0.4]
ok: [10.0.0.3]
TASK [Copy cloud-init configuration] *****
changed: [10.0.0.3]
changed: [10.0.0.2]
changed: [10.0.0.4]
TASK [Create machine] *****
changed: [10.0.0.4]
changed: [10.0.0.3]
changed: [10.0.0.2]
TASK [Start VM] *****
ok: [10.0.0.3]
ok: [10.0.0.4]
ok: [10.0.0.2]
PLAY RECAP *****
10.0.0.2      : ok=5  changed=2  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
10.0.0.3      : ok=5  changed=2  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
10.0.0.4      : ok=5  changed=2  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

Figure 11.35 – Checking our installation process

We can also check this using the command line:

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

The output of this command should look something like this:

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

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

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

Figure 11.36 – Checking our VMs

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

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

We should get something like this:

```
[root@vm0-101 ~]# ansible cloudhosts -m shell -a "virsh net-dhcp-leases --network default"
10.0.0.2 | CHANGED | rc=0 >>
Expiry Time      MAC address      Protocol  IP address      Hostname      Client ID or OUID
-----
2019-09-09 19:33:19 52:54:00:9a:e0:20 ipv4      192.168.122.38/24  -              -
10.0.0.4 | CHANGED | rc=0 >>
Expiry Time      MAC address      Protocol  IP address      Hostname      Client ID or OUID
-----
2019-09-09 19:33:19 52:54:00:a4:b8:21 ipv4      192.168.122.119/24  -              -
10.0.0.3 | CHANGED | rc=0 >>
Expiry Time      MAC address      Protocol  IP address      Hostname      Client ID or OUID
-----
2019-09-09 19:33:19 52:54:00:31:fc:7c ipv4      192.168.122.161/24  -              -
```

Figure 11.37 – Checking our VM network connectivity and network configuration

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

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

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

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

The following is the expected output:

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

Figure 11.38 – Using the virt module with Ansible

There is only one thing that we haven't covered yet – how to install multi-tiered applications. Pushing the definition to its smallest extreme, we are going to install a LAMP server using a simple playbook.

Orchestrating multi-tier application deployment on KVM VM

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

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

The difficult part is the package names: in our demonstration machine, we are using CentOS7 as the operating system and its package names are a little different. Apache is called `httpd` and `mysql` is replaced with `mariadb`, another engine that is compatible with MySQL. PHP is luckily the same as on other distributions. We also need another package named `python2-PyMySQL` (the name is case sensitive) in order to get our playbook to work.

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

One more thing: our script presumes that there is a file named `index.php` in the directory our playbook runs from, and that file will get copied to the remote system:

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

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

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

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

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

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

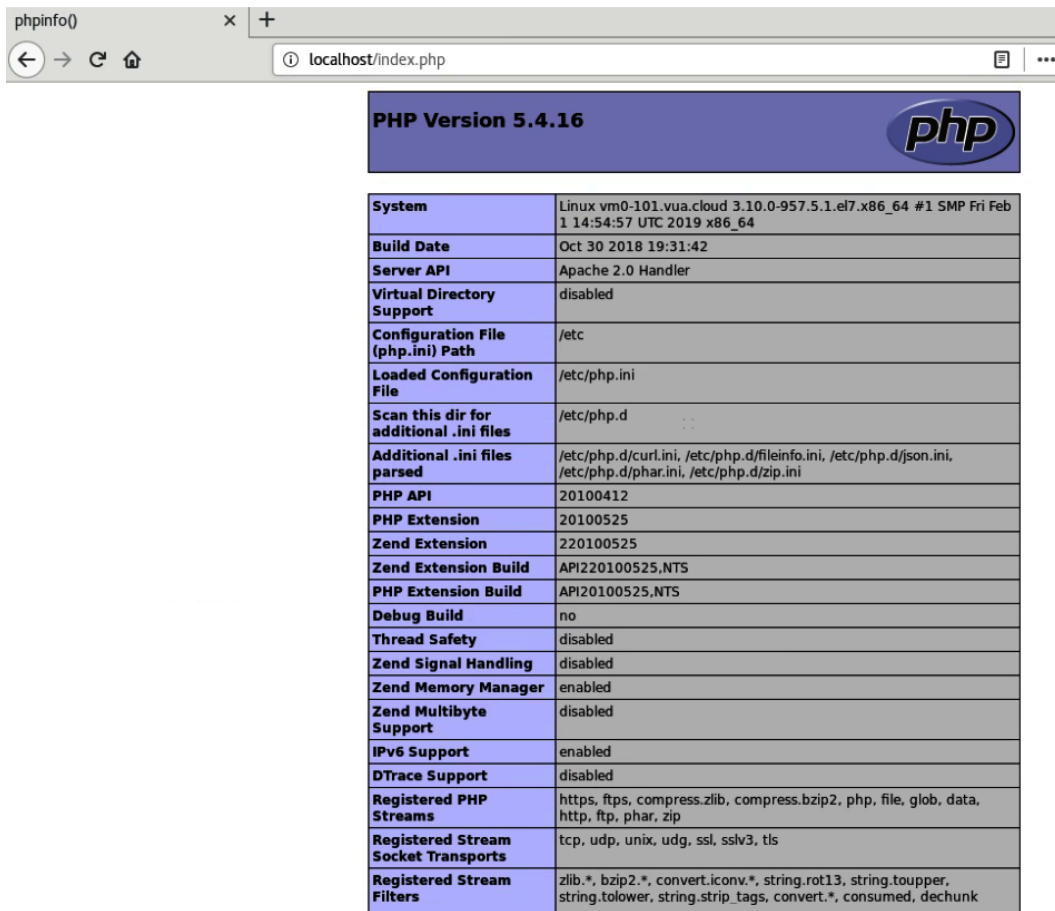
Figure 11.39 – Ansible LAMP playbook

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

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

Figure 11.40 – Testing if PHP works

Things can't get any simpler than that. In a normal deployment scenario, we would have something more complicated in the web server directory, such as a WordPress or Joomla installation, or even a custom application. The only thing that needs to change is the file that is copied (or a set of files) and the location of the database. Our file just prints information about the local `.php` installation:



PHP Version 5.4.16	
System	Linux vm0-101.vua.cloud 3.10.0-957.5.1.el7.x86_64 #1 SMP Fri Feb 1 14:54:57 UTC 2019 x86_64
Build Date	Oct 30 2018 19:31:42
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc
Loaded Configuration File	/etc/php.ini
Scan this dir for additional .ini files	/etc/php.d
Additional .ini files parsed	/etc/php.d/curl.ini, /etc/php.d/fileinfo.ini, /etc/php.d/json.ini, /etc/php.d/phar.ini, /etc/php.d/zip.ini
PHP API	20100412
PHP Extension	20100525
Zend Extension	220100525
Zend Extension Build	API220100525,NTS
PHP Extension Build	API20100525,NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	disabled
Zend Memory Manager	enabled
Zend Multibyte Support	disabled
IPv6 Support	enabled
DTrace Support	disabled
Registered PHP Streams	https, ftps, compress.zlib, compress.bzip2, php, file, glob, data, http, ftp, phar, zip
Registered Stream Socket Transports	tcp, udp, unix, udg, ssl, sslv3, tls
Registered Stream Filters	zlib.*, bzip2.*, convert.iconv.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed, dechunk

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

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

Learning by example – various examples of using Ansible with KVM

Now that we've covered simple and more complex Ansible tasks, let's think about how to use Ansible to further our configuration skills and overall compliance based on some kind of policy. The following are some things that we are going to leave as exercises for you:

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

- Task 4:

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

- a) Create a playbook that will run on a remote machine.
- b) Create a playbook that will install different roles on different servers.
- c) Create a playbook that will deploy a more complex application, such as WordPress.

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

Summary

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

The next chapter takes us to a world of cloud services – specifically OpenStack – where our Ansible knowledge is going to be very useful for large-scale virtual machine configuration as it's impossible to configure all of our cloud virtual machines by using any kind of manual utilities. Apart from that, we'll extend our knowledge of Ansible by integrating OpenStack and Ansible so that we can use both of these platforms to do what they do really well – manage cloud environments and configure their consumables.

Questions

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

Further reading

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

- What is Ansible?: <https://www.ansible.com/>
- Ansible documentation: <https://docs.ansible.com/>
- Ansible overview: <https://www.ansible.com/overview/it-automation>
- Ansible use cases: <https://www.ansible.com/use-cases>
- Ansible for continuous delivery: <https://www.ansible.com/use-cases/continuous-delivery>
- Integrating Ansible with Jenkins: <https://www.redhat.com/en/blog/integrating-ansible-jenkins-cicd-process>

Section 4: Scalability, Monitoring, Performance Tuning, and Troubleshooting

In this part of the book, you will learn about the scalability, monitoring, advanced performance tuning, and troubleshooting of KVM-based virtual machines and hypervisors.

This part of the book comprises the following chapters:

- *Chapter 12, Scaling out KVM with OpenStack*
- *Chapter 13, Scaling out KVM with AWS*
- *Chapter 14, Monitoring the KVM Virtualization Platform*
- *Chapter 15, Performance Tuning and Optimization for KVM VMs*
- *Chapter 16, Troubleshooting Guidelines for the KVM Platform*

12

Scaling Out KVM with OpenStack

Being able to virtualize a machine is a big thing, but sometimes, just virtualization is not enough. The problem is how to give individual users tools so that they can virtualize whatever they need, when they need it. If we combine that user-centric approach with virtualization, we are going to end up with a system that needs to be able to do two things: it should be able to connect to KVM as a virtualization mechanism (and not only KVM) and enable users to get their virtual machines running and automatically configured in a self-provisioning environment that's available through a web browser. OpenStack adds one more thing to this since it is completely free and based entirely on open source technologies. Provisioning such a system is a big problem due to its complexity, and in this chapter, we are going to show you – or to be more precise, point you – in the right direction regarding whether you need a system like this.

In this chapter, we will cover the following topics:

- Introduction to OpenStack
- Software-defined networking
- OpenStack components
- Additional OpenStack use cases

- Provisioning the OpenStack environment
- Integrating OpenStack with Ansible
- Let's get started!

Introduction to OpenStack

In its own words, **OpenStack** is a cloud operating system that is used to control a large number of different resources in order to provide all the essential services for **Infrastructure-as-a-Service (IaaS)** and **orchestration**.

But what does this mean? OpenStack is designed to completely control all the resources that are in the data center, and to provide both central management and direct control over anything that can be used to deploy both its own and third-party services. Basically, for every service that we mention in this book, there is a place in the whole OpenStack landscape where that service is or can be used.

OpenStack itself consists of several different interconnected services or service parts, each with its own set of functionalities, and each with its own API that enables full control of the service. In this part of this book, we will try to explain what different parts of OpenStack do, how they interconnect, what services they provide, and how to use those services to our advantage.

The reason OpenStack exists is because there was the need for an open source cloud computing platform that would enable creating public and private clouds that are independent of any commercial cloud platform. All parts of OpenStack are open source and were released under the Apache License 2.0. The software was created by a large, mixed group of individuals and large cloud providers. Interestingly, the first major release was the result of NASA (a US government agency) and Rackspace Technology (a large US hosting company) joining their internal storage and computing infrastructure solutions. These releases were later designated with the names Nova and Swift, and we will cover them in more detail later.

The first thing you will notice about OpenStack is its services since there is no single *OpenStack* service but an actual stack of services. The name *OpenStack* comes directly from this concept because it correctly identifies OpenStack as an open source component that acts as services that are, in turn, grouped into functional sets.

Once we understand that we are talking about autonomous services, we also need to understand that services in OpenStack are grouped by their function, and that some functions have more than one specialized service under them. We will try to cover as much as possible about different services in this chapter, but there are simply too many of them to even mention all of them here. All the documentation and all the whitepapers can be found at <http://openstack.org>, and we strongly suggest that you consult it for anything not mentioned here, and even for things that we mention but that could have changed by the time you read this.

The last thing we need to clarify is the naming – every service in OpenStack has its project name and is referred to by that name in the documentation. This might, at first glance, look confusing since some of the names are completely unrelated to the specific function a particular service has in the whole project, but using names instead of official designators for a function is far easier once you start using OpenStack. Take, for example, Swift. Swift's full name is *OpenStack Object Store*, but this is rarely mentioned in the documentation or its implementation. The same goes for other services or *projects* under OpenStack, such as Nova, Ironic, Neutron, Keystone, and over 20 other different services.

If you step away from OpenStack for a second, then you need to consider what cloud services are all about. The cloud is all about scaling – in terms of compute resources, storage, network, APIs – whatever. But, as always in life, as you scale things, you're going to run into problems. And these problems have their own *names* and *solutions*. So, let's discuss these problems for a minute.

The basic problems for cloud provider scalability can be divided into three groups of problems that need to be solved at scale:

- **Compute problems** (Compute = CPU + memory power): These problems are pretty straightforward to solve – if you need more CPU and memory power, you buy more servers, which, by design, means more CPU and memory. If you need a quality of service/**service-level agreement (SLA)** type of concept, we can introduce a concept such as compute resource pools so that we can slice the compute *pie* according to our needs and divide those resources between our clients. It doesn't matter whether our client is just a private person or a company buying into cloud services. In cloud technologies, we call our clients *tenants*.

- **Storage problems:** As you scale your cloud environments, things become really messy in terms of storage capacity, management, monitoring and – especially – performance. The performance side of that problem has a couple of most commonly used variables – read and write throughput and read and write IOPS. When you grow your environment from 100 hosts to 1,000 hosts or more, performance bottlenecks are going to become a major issue that will be difficult to tackle without proper concepts. So, the storage problem can be solved by adding additional storage devices and capacity, but it's much more involved than the compute problem as it needs much more configuration and money. Remember, every virtual machine has a statistical influence on other virtual machines' performance, and the more virtual machines you have, the greater this entropy is. This is the most difficult process to manage in storage infrastructure.
- **Network problems:** As the cloud infrastructure grows, you need thousands and thousands of isolated networks so that the network traffic of tenant A can't communicate with the network traffic of tenant B. At the same time, you still need to offer a capability where you can have multiple networks (usually implemented via VLANs in non-cloud infrastructures) per tenant and routing between these networks, if that's what the tenant needs.

This network problem is a scalability problem based on technology, as the technology behind VLAN was standardized years before the number of VLANs could become a scalability problem.

Let's continue our journey through OpenStack by explaining the most fundamental subject of cloud environments, which is scaling cloud networking via **software-defined networking (SDN)**. The reason for this is really simple – without SDN concepts, the cloud wouldn't really be scalable enough for customers to be happy, and that would be a complete showstopper. So, buckle up your seatbelts and let's do an SDN primer.

Software-defined networking

One of the straightforward stories about the cloud – at least on the face of it – should have been the story about cloud networking. In order to understand how simple this story should've been, we only need to look at one number, and that number is the **virtual LAN (VLAN ID)** number. As you might already be aware, by using VLANs, network administrators have a chance to divide a physical network into separate logical networks. Bearing in mind that the VLAN part of the Ethernet header can have up to 12 bits, the maximum number of these logically isolated networks is 4,096. Usually, the first and last VLANs are reserved (0 and 4095), as is VLAN 1.

So, basically, we're left with 4,093 separate logical networks in a real-life scenario, which is probably more than enough for the internal infrastructure of any given company. However, this is nowhere near enough for public cloud providers. The same problem applies to public cloud providers that use hybrid-cloud types of services to – for example – extend their compute power to the cloud.

So, let's focus on this network problem for a bit. Realistically, if we look at this problem from the cloud user perspective, data privacy is of utmost importance to us. If we look at this problem from the cloud provider perspective, then we want our network isolation problem to be a non-issue for our tenants. This is what cloud services are all about at a more basic level – no matter what the background complexity in terms of technology is, users have to be able to access all of the necessary services in as user-friendly a way as possible. Let's explain this by using an example.

What happens if we have 5,000 different clients (tenants) in our public cloud environment? What happens if every tenant needs to have five or more logical networks? We quickly realize that we have a big problem as cloud environments need to be separated, isolated, and fenced. They need to be separated from one another at a network level for security and privacy reasons. However, they also need to be routable, if a tenant needs that kind of service. On top of that, we need the ability to scale so that situations in which we need more than 5,000 or 50,000 isolated networks don't bother us. And, going back to our previous point – roughly 4,000 VLANs just isn't going to cut it.

There's a reason why we said that this should have been a straightforward story. The engineers among us see these situations in black and white – we focus on a problem and try to come to a solution. And the solution seems rather simple – we need to extend the 12-bit VLAN ID field so that we can have more available logical networks. How difficult can that be?

As it turns out, very difficult. If history teaches us anything, it's that various different interests, companies, and technologies compete for years for that *top dog* status in anything in terms of IT technology. Just think of the good old days of DVD+R, DVD-R, DVD+RW, DVD-RW, DVD-RAM, and so on. To simplify things a bit, the same thing happened here when the initial standards for cloud networking were introduced. We usually call these network technologies cloud overlay network technologies. These technologies are the basis for SDN, the principle that describes the way cloud networking works at a global, centralized management level. There are multiple standards on the market to solve this problem – VXLAN, GRE, STT, NVGRE, NVO3, and more.

Realistically, there's no need to break them all down one by one. We are going to take a simpler route – we're going to describe one of them that's the most valuable for us in the context of today (**VXLAN**) and then move on to something that's considered to be a *unified* standard of tomorrow (**GENEVE**).

First, let's define what an overlay network is. When we're talking about overlay networks, we're talking about networks that are built on top of another network in the same infrastructure. The idea behind an overlay network is simple – we need to disentangle the physical part of the network from the logical part of the network. If we want to do that in absolute terms (configure everything without spending massive amounts of time in the CLI to configure physical switches, routers, and so on), we can do that as well. If we don't want to do it that way and we still want to work directly with our physical network environment, we need to add a layer of programmability to the overall scheme. Then, if we want to, we can interact with our physical devices and push network configuration to them for a more top-to-bottom approach. If we do things this way, we'll need a bit more support from our hardware devices in terms of capability and compatibility.

Now that we've described what network overlay is, let's talk about VXLAN, one of the most prominent overlay network standards. It also serves as a basis for developing some other network overlay standards (such as GENEVE), so – as you might imagine – it's very important to understand how it works.

Understanding VXLAN

Let's start with the confusing part. VXLAN (IETF RFC 7348) is an extensible overlay network standard that enables us to aggregate and tunnel multiple Layer 2 networks across Layer 3 networks. How does it do that? By encapsulating a Layer 2 packet inside a Layer 3 packet. In terms of transport protocol, it uses UDP, by default on port 4789 (more about that in just a bit). In terms of special requests for VXLAN implementation – as long as your physical network supports MTU 1600, you can implement VXLAN as a cloud overlay solution easily. Almost all the switches you can buy (except for the cheap home switches, but we're talking about enterprises here) support jumbo frames, which means that we can use MTU 9000 and be done with it.

From the standpoint of encapsulation, let's see what it looks like:

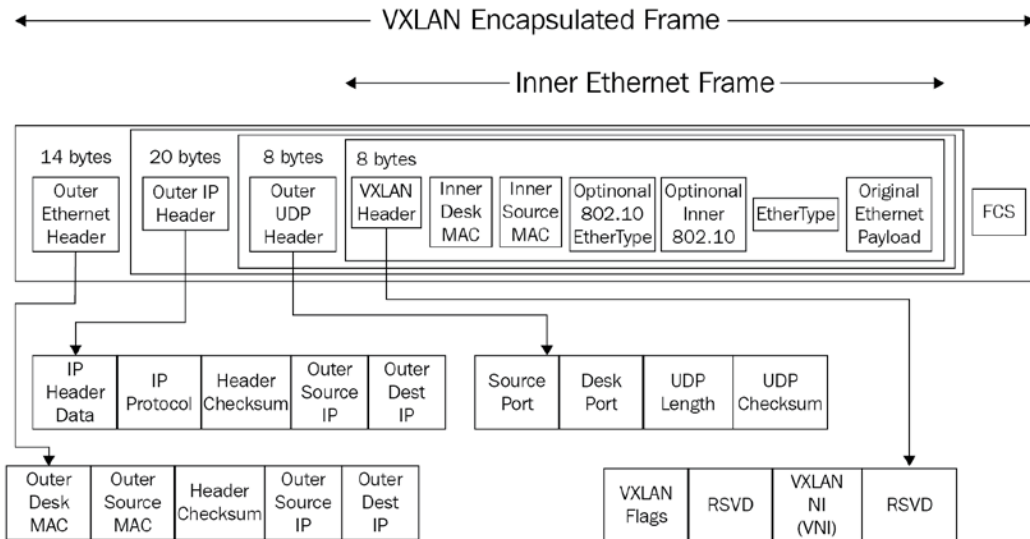


Figure 12.1 – VXLAN frame encapsulation

In more simplistic terms, VXLANs use tunneling between two VXLAN endpoints (called VTEPs; that is, VXLAN tunneling endpoints) that check **VXLAN network identifiers (VNIs)** so that they can decide which packets go where.

If this seems complicated, then don't worry – we can simplify this. From the perspective of VXLAN, a VNI is the same thing as a VLAN ID is to VLAN. It's a unique network identifier. The difference is just the size – the VNI field has 24 bits, compared to VLAN's 12. That means that we have 2^{24} VNIs compared to VLAN's 2^{12} . So, VXLANs – in terms of network isolation – are VLANs squared.

Why does VXLAN use UDP?

When designing overlay networks, what you usually want to do is reduce latency as much as possible. Also, you don't want to introduce any kind of overhead. When you consider these two basic design principles and couple that with the fact that VXLAN tunnels Layer 2 traffic inside Layer 3 (whatever the traffic is – unicast, multicast, broadcast), that literally means we should use UDP. There's no way around the fact that TCP's two methods – three-way handshakes and retransmissions – would get in the way of these basic design principles. In the simplest of terms, TCP would be too complicated for VXLAN as it would mean too much overhead and latency at scale.

In terms of VTEPs, just imagine them as two interfaces (implemented in software or hardware) that can encapsulate and decapsulate traffic based on VNIs. From a technology standpoint, VTEPs map various tenant's virtual machines and devices to VXLAN segments (VXLAN-backed isolated networks), perform package inspection, and encapsulate/decapsulate network traffic based on VNIs. Let's describe this communication with the help of the following diagram:

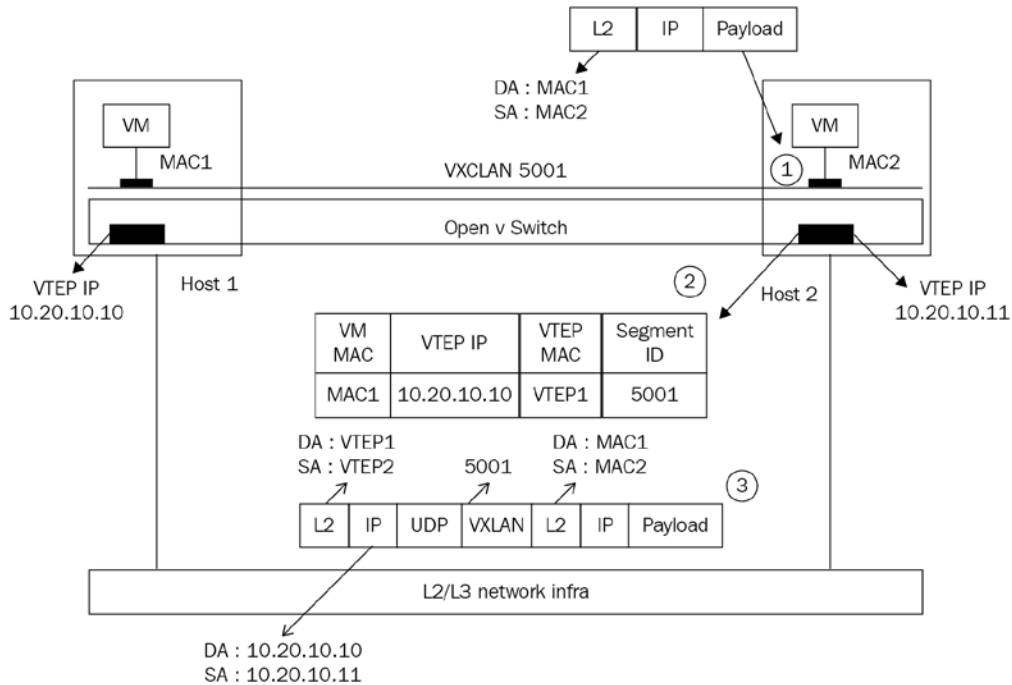


Figure 12.2 – VTEPs in unicast mode

In our open source-based cloud infrastructure, we're going to implement cloud overlay networks by using OpenStack Neutron or Open vSwitch, a free, open source distributed switch that supports almost all network protocols that you could possibly think of, including the already mentioned VXLAN, STT, GENEVE, and GRE overlay networks.

Also, there's a kind of gentleman's agreement in place in cloud networking regarding not using VXLANs from 1-4999 in most use cases. The reason for this is simple – because we still want to have our VLANs with their reserved range of 0-4095 in a way that is simple and not error-prone. In other words, by design, we leave network IDs 0-4095 for VLANs and start VXLANs with VNI 5000 so that it's really easy to differentiate between the two. Not using 5,000 VXLAN-backed networks out of 16.7 million VXLAN-backed networks isn't that much of a sacrifice for good engineering practices.

The simplicity, scalability, and extensibility of VXLAN also means more really useful usage models, such as the following:

- **Stretching Layer 2 across sites:** This is one of the most common problems regarding cloud networking, as we will describe shortly.
- **Layer 2 bridging:** Bridging a VLAN to a cloud overlay network (such as VXLAN) is *very* useful when onboarding our users to our cloud services as they can then just connect to our cloud network directly. Also, this usage model is heavily used when we want to physically insert a hardware device (for example, a physical database server or a physical appliance) into a VXLAN. If we didn't have Layer 2 bridging, imagine all the pain that we would have. All our customers running the Oracle Database Appliance would have no way to connect their physical servers to our cloud-based infrastructure.
- **Various offloading technologies:** These include load balancing, antivirus, vulnerability and antimalware scanning, firewall, IDS, IPS integration, and so on. All of these technologies enable us to have useful, secure environments with simple management concepts.

We mentioned that stretching Layer 2 across sites is a fundamental problem, so it's obvious that we need to discuss it. We'll do that next. Without a solution to this problem, you'd have very little chance of creating multiple data center cloud infrastructures efficiently.

Stretching Layer 2 across sites

One of the most common sets of problems that cloud providers face is how to stretch their environment across sites or continents. In the past, when we didn't have concepts such as VXLAN, we were forced to use some kind of Layer 2 VPN or MPLS-based technologies. These types of services are really expensive, and sometimes, our service providers aren't exactly happy with our *give me MPLS* or *give me Layer 2 access* requests. They would be even less happy if we mentioned the word *multicast* in the same sentence, and this was a set of technical criteria that was *often* used in the past. So, having the capability to deliver Layer 2 over Layer 3 fundamentally changes that conversation. Basically, if you have the capability to create a Layer 3-based VPN between sites (which you can almost always do), you don't have to be bothered with that discussion at all. Also, that significantly reduces the price of these types of infrastructure connections.

Consider the following multicast-based example:

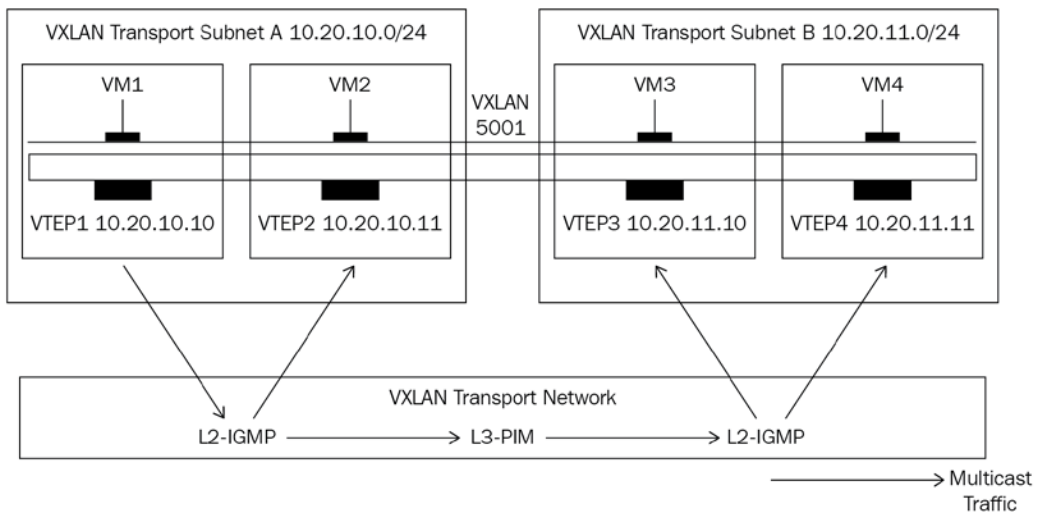


Figure 12.3 – Extending VXLAN segments across sites in multicast mode

Let's say that the left-hand side of this diagram is the first site and that the right-hand side of this diagram is the second site. From the perspective of VM1, it doesn't really matter that VM4 is in some other remote site as its segment (VXLAN 5001) *spans* across those sites. How? As long as the underlying hosts can communicate with each other over the VXLAN transport network (usually via the management network as well), the VTEPs from the first site can *talk* to the VTEPs from the second site. This means that virtual machines that are backed by VXLAN segments in one site can talk to the same VXLAN segments in the other site by using the aforementioned Layer 2-to-Layer 3 encapsulation. This is a really simple and elegant way to solve a complex and costly problem.

We mentioned that VXLAN, as a technology, served as a basis for developing some other standards, with the most important being GENEVE. As most manufacturers work toward GENEVE compatibility, VXLAN will slowly but surely disappear. Let's discuss what the purpose of the GENEVE protocol is and how it aims to become *the standard* for cloud overlay networking.

Understanding GENEVE

The basic problem that we touched upon earlier is the fact that history kind of repeated itself in cloud overlay networks, as it did many times before. Different standards, different firmwares, and different manufacturers supporting one standard over another, where all of the standards are incredibly similar but still not compatible with each other. That's why VMware, Microsoft, Red Hat, and Intel proposed GENEVE, a new cloud overlay standard that only defines the encapsulation data format, without interfering with the control planes of these technologies, which are fundamentally different. For example, VXLAN uses a 24-bit field width for VNI, while STT uses 64-bit. So, the GENEVE standard proposes no fixed field size as you can't possibly know what the future brings. Also, taking a look at the existing user base, we can still happily use our VXLANs as we don't believe that they will be influenced by future GENEVE deployments.

Let's see what the GENEVE header looks like:

GENEVE Header

V	Option Length	O	C	Reserved	Protocol Type
VNI					Reserved
Variable Length Options					

Figure 12.4 – GENEVE cloud overlay network header

The authors of GENEVE learned from some other standards (BGP, IS-IS, and LLDP) and decided that the key to doing things right is extensibility. This is why it was embraced by the Linux community in Open vSwitch and VMware in NSX-T. VXLAN is supported as the network overlay technology for **Hyper-V Network Virtualization (HNV)** since Windows Server 2016 as well. Overall, GENEVE and VXLAN seem to be two technologies that are surely here to stay – and both are supported nicely from the perspective of OpenStack.

Now that we've covered the most basic problem regarding the cloud – cloud networking – we can go back and discuss OpenStack. Specifically, our next subject is related to OpenStack components – from Nova through to Glance and then to Swift, and others. So, let's get started.

OpenStack components

When OpenStack was first formed as a project, it was designed from two different services:

- A computing service that was designed to manage and run virtual machines themselves
- A storage service that was designed for large-scale object storage

These services are now called OpenStack Compute or *Nova*, and OpenStack Object Store or *Swift*. These services were later joined by *Glance* or the OpenStack Image service, which was designed to simplify working with disk images. Also, after our SDN primer, we need to discuss OpenStack Neutron, the **Network-as-a-Service (NaaS)** component of OpenStack.

The following diagram shows the components of OpenStack:

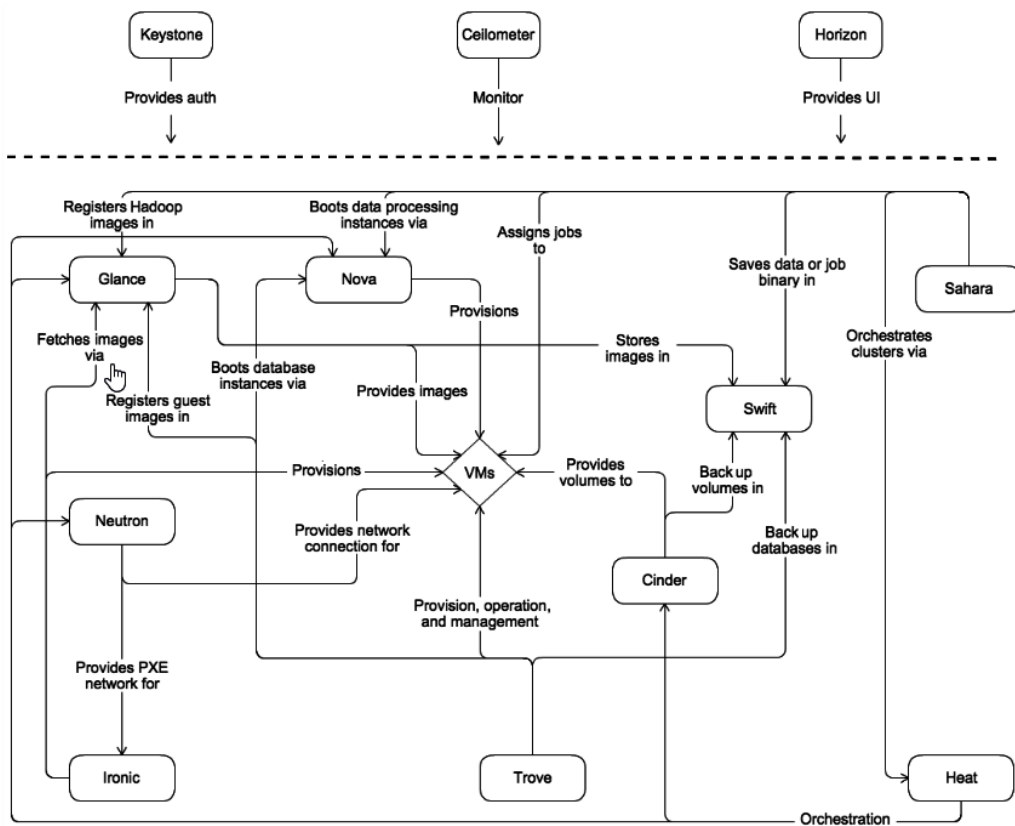


Figure 12.5 – Conceptual architecture of OpenStack (source: <https://docs.openstack.org/>)

We'll go through these in no particular order and will include additional services that are important. Let's start with **Swift**.

Swift

The first service we need to talk about is Swift. For that purpose, we are going to grab the project's own definition from the OpenStack official documentation and parse it to try and explain what services are fulfilled by this project, and what is it used for. The Swift website (<https://docs.openstack.org/swift/latest/>) states the following:

"Swift is a highly available, distributed, eventually consistent object/blob store. Organizations can use Swift to store lots of data efficiently, safely, and cheaply. It's built for scale and optimized for durability, availability, and concurrency across the entire dataset. Swift is ideal for storing unstructured data that can grow without bounds."

Having read that, we need to point out quite a few things that may be completely new to you. First and foremost, we are talking about storing data in a particular way that is not common in computing unless you have used unstructured data stores. Unstructured does not mean that this way of storing data is lacking structure; in this context, it means that we are the ones that are defining the structure of the data, but the service itself does not care about our structure, instead relying on the concept of objects to store our data. One result of this is something that may also sound unusual at first, and that is that the data we store in Swift is not directly accessible through any filesystem, or any other way we are used to manipulating files through our machines. Instead, we are manipulating data as objects and we must use the API that is provided as part of Swift to get the data objects. Our data is stored in *blobs*, or objects, that the system itself just labels and stores to take care of availability and access speed. We are supposed to know what the internal structure of our data is and how to parse it. On the other hand, because of this approach, Swift can be amazingly fast with any amount of data and scales horizontally in a way that is almost impossible to achieve using normal, classic databases.

Another thing worth mentioning is that this service offers highly available, distributed, and *eventually consistent* storage. This means that, first and foremost, the priority is for the data to be distributed and highly available, which are two things that are important in the cloud. Consistency comes after that but is eventually achieved. Once you come to use this service, you will understand what that means. In almost all usual scenarios where data is read and rarely written, it is nothing to even think about, but there are some cases where this can change the way we need to think about the way we go about delivering the service. The documentation states the following:

"Because each replica in Object Storage functions independently and clients generally require only a simple majority of nodes to respond to consider an operation successful, transient failures such as network partitions can quickly cause replicas to diverge. These differences are eventually reconciled by asynchronous, peer-to-peer replicator processes. The replicator processes traverse their local filesystems and concurrently perform operations in a manner that balances load across physical disks."

We can roughly translate this. Let's say that you have a three-node Swift cluster. In such a scenario, a Swift object will become available to clients after the PUT operation has been confirmed to have been completed on at least two nodes. So, if your goal is to create a low-latency, synchronous storage replication with Swift, there are other solutions available for that.

Having put aside all the abstract promises regarding what Swift offers, let's go into more details. High availability and distribution are the direct result of using a concept of *zones* and having multiple copies of the same data written onto multiple storage servers. Zones are nothing but a simple way of logically dividing the storage resources we have at our disposal and deciding on what kind of isolation we are ready to provide, as well as what kind of redundancy we need. We can group servers by the server itself, by the rack, by sets of servers across a Datacenter, in groups across different Datacenters, and in any combination of those. Everything really depends on the amount of available resources and the data redundancy and availability we need and want, as well as, of course, the cost that will accompany our configuration.

Based on the resources we have, we are supposed to configure our storage system in terms of how many copies it will hold and how many zones we are prepared to use. A copy of a particular data object in Swift is referred to as a *replica*, and currently, the best practices call for at least three replicas across no less than five zones.

A zone can be a server or a set of servers, and if we configure everything correctly, losing any one zone should have no impact on the availability or distribution of data. Since a zone can be as small as a server and as big as any number of data centers, the way we structure our zones has a huge impact on the way the system reacts to any failures and changes. The same goes for replicas. In the recommended scenario, configuration has a smaller number of replicas than the number of zones, so only some of the zones will hold some of these replicas. This means the system must balance the way data is written in order to evenly distribute both the data and the load, including both the writing and the reading load for the data. At the same time, the way we structure the zones will have an enormous impact on the cost – redundancy has a real cost in terms of server and storage hardware, and multiplying replicas and zones creates additional demands in regard to how much storage and computing power we need to allocate for our OpenStack installation. Being able to do this correctly is the biggest problem that a Datacenter architect has to solve.

Now, we need to go back to the concept of eventual consistency. Eventual consistency in this context means that data is going to be written to the Swift store and that objects are going to get updated, but the system will not be able to do a completely simultaneous write of all the data into all the copies (replicas) of the data across all zones. Swift will try to reconcile the differences as soon as possible and will be aware of these changes, so it serves new versions of the objects to whoever tries to read them. Scenarios where data is inconsistent due to a failure of some part of the system exist, but they are to be considered abnormal states of the system and need to be repaired rather than the system being designed to ignore them.

Swift daemons

Next, we need to talk about the way Swift is designed in regard to its architecture. Data is managed through three separate logical daemons:

- **Swift-account** is used to manage a SQL database that contains all the accounts defined with the object storage service. Its main task is to read and write the data that all the other services need, primarily in order to validate and find appropriate authentication and other data.

- **Swift-container** is another database process, but it is used strictly to map data into containers, a logical structure similar to AWS *buckets*. This can include any number of objects that are grouped together.
- **Swift-object** manages mapping to actual objects, and it keeps track of the location and availability of the objects themselves.

All these daemons are just in charge of data and make sure that everything is both mapped and replicated correctly. Data is used by another layer in the architecture: the presentation layer.

When a user wants to use any data object, it first needs to authenticate via a token that can be either externally provided or created by an authentication system inside Swift. After that, the main process that orchestrates data retrieval is Swift-proxy, which handles communication with three daemons that deal with the data. Provided that the user presented a valid token, it gets the data object delivered to the user request.

This is just the briefest of overviews regarding how Swift works. In order to understand this, you need to not only read the documentation but also use some kind of system that will perform low-level object retrieval and storage into and out of Swift.

Cloud services can't be scaled or used efficiently if we don't have orchestration services, which is why we need to discuss the next service on our list – **Nova**.

Nova

Another important service or project is Nova – an orchestration service that is used for providing both provisioning and management for computing instances at a large scale. What it basically does is allow us to use an API structure to directly allocate, create, reconfigure, and delete or *destroy* virtual servers. The following is a diagram of a logical Nova service structure:

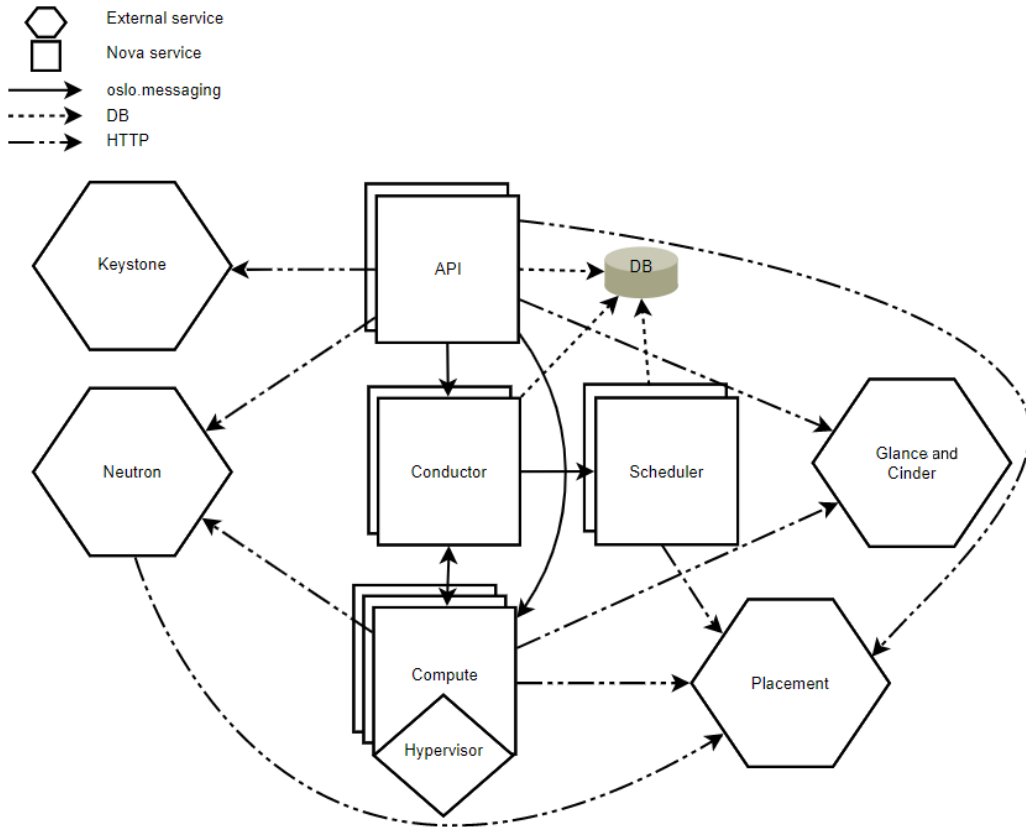


Figure 12.6 – Logical structure of the Nova service (openstack.org)

Most of Nova is a very complex distributed system written almost entirely in Python that consists of a number of working scripts that do the orchestration part and a gateway service that receives and carries through API calls. The API is also based on Python; it's a **Web Server Gateway Interface (WSGI)**-compatible application that handles calls. WSGI, in turn, is a standard that defines how a web application and a server should exchange data and commands. This means that, in theory, any system capable of using the WSGI standard can also establish communication with this service.

Aside from this multifaceted orchestration solution, there are two more services that are at the heart of Nova – the database and messaging queue. Neither of these is Python-based. We'll talk about messaging and databases first.

Almost all distributed systems must rely on queues to be able to perform their tasks. Messages need to be forwarded to a central place that will enable all daemons to do their tasks, and using the right messaging and queueing system is crucial for system speed and reliance. Nova currently uses RabbitMQ, a highly scalable and available system by itself. Using a production-ready system like this means that not only are there tools to debug the system itself, but there are a lot of reporting tools available for directly querying the messaging queue.

The main purpose of using a messaging queue is to completely decouple any clients from servers, and to provide asynchronous communication between different clients. There is a lot to be said on how the actual messaging works, but for this chapter, we will just refer you to the official documentation at <https://docs.openstack.org/nova/latest/>, since we are not talking about a couple of functions on a server but an entirely independent software stack.

The database is in charge of holding all the state data for the tasks currently being performed, as well as enabling the API to return information about the current state of different parts of Nova.

All in all, the system consists of the following:

- **nova-api**: The daemon that is directly facing the user and is responsible for accepting, parsing, and working through all the user API requests. Almost all the documentation that refers to `nova-api` is actually referring to this daemon, sometimes calling it just *API*, *controller*, or *cloud controller*. We need to explain a little bit more about Nova in order to understand that calling `nova-api` a controller is wrong, but since there exists a class inside a daemon named `CloudController`, a lot of users confuse this daemon for the whole distributed system.

`nova-api` is a powerful system since it can, by itself, process and sort out some API calls, getting the data from the database and working out what needs to be done. In a more common case, `nova-api` will just initiate a task and forward it in the form of messages to other daemons inside Nova.

- Another important daemon is the **scheduler**. Its main function is to go through the queue and determine when and where a particular request should run. This sounds simple enough, but given the possible complexity of the system, this *where and when* can lead to extreme gains or losses in performance. In order to solve this, we can choose how the scheduler makes decisions regarding choosing the right place to perform requests. Users can choose either to write their own request or to use one of the predetermined ones.

If we are choosing the ones provided by Nova, we have three choices:

- a) **Simple scheduler** determines where the request will be run based on the load on the hosts – it will monitor all the hosts and try to allocate the one that has the least load in a particular slice of time.
- b) **Chance** is the default way of scheduling. As its name suggests, it's the simplest algorithm – a host is randomly chosen from the list and given the request.
- c) **Zone scheduling** will also randomly choose a host but will do so from within a zone.

Now, we will look at *workers*, daemons that actually perform requests. There are three of these – network, volume, and compute:

- **nova-network** is in charge of the network. It will perform whatever is given to it from the queue that is related to anything on the network and will create interfaces and rules as needed. It is also in charge of IP address allocation; it will allocate both fixed and dynamically assigned addresses and take care of both the external and internal networks. Instances usually use one or more fixed IPs to enable management and connectivity, and these are usually local addresses. There are also floating addresses to enable connecting from the outside. This service has been obsolete since the OpenStack Newton release from 2016, although you can still use it in some legacy configurations.
- **nova-volume** handles storage volumes or, to be more precise, all the ways data storage can be connected to any instance. This includes standards such as iSCSI and AoE, which are targeted at encapsulating known common protocols, and providers such as Sheepdog, LeftHand, and RBD, which cover connections to open source and closed source storage systems such as CEPH or HP LeftHand.
- **nova-compute** is probably the easiest to describe – it is used to create and destroy new instances of virtual machines, as well as to update information about them in the database. Since this is a heavily distributed system, this also means that `nova-compute` must adapt itself to using different virtualization technologies and to completely different platforms. It also needs to be able to dynamically allocate and free resources. Primarily, it uses libvirt for its VM management, directly supporting KVM for creating and deleting new instances. This is the reason this chapter exists, since `nova-compute` using libvirt to start KVM machines is by far the most common way of configuring OpenStack, but support for different technologies extends a lot further. The libvirt interface also supports Xen, QEMU, LXC, and **user mode Linux (UML)**, and through different APIs, `nova-compute` can support Citrix, XCP, VMware ESX/ESXi vSphere, and Microsoft Hyper-V. This enables Nova to control all the currently used enterprise virtualization solutions from one central API.

As a side note, **nova-conductor** is there to process requests that require any conversion regarding objects, resizing, and database/proxy access.

The next service on our list is **Glance** – a service that is very important for virtual machine deployment as we want to do this from images. Let's discuss Glance now.

Glance

At first, having a separate service for cloud disk image management makes little sense, but when scaling any infrastructure, image management will become a problem that needs an API to be solved. Glance basically has this dual identity – it can be used to directly manipulate VM images and store them inside blobs of data, but at the same time it can be used to completely automatically orchestrate a lot of tasks when dealing with a huge number of images.

Glance is relatively simple in terms of its internal structure as it consists of an image information database, an image store that uses Swift (or a similar service), and an API that glues everything together. Database is sometimes called Registry, and it basically gives information about a given image. Images themselves can be stored on different types of stores, either from Swift (as blobs) on HTTP servers or on a filesystem (such as NFS).

Glance is completely nonspecific about the type of image store it uses, so NFS is perfectly okay and makes implementing OpenStack a little bit easier, but when scaling OpenStack, both Swift and Amazon S3 can be used.

When thinking about the place in the big OpenStack puzzle that Glance belongs to, we could describe it as being the service that Nova uses to find and instantiate images. Glance itself uses Swift (or any other storage) to store images. Since we are dealing with multiple architectures, we need a lot of different supported file formats for images, and Glance does not disappoint. Every disk format that is supported by different virtualization engines is supported by Glance. This includes both unstructured formats such as `raw` and structured formats such as VHD, VMDK, `qcow2`, VDI ISO, and AMI. OVF – as an example of an image container – is also supported.

Glance probably has the simplest API of them all, enabling it to be used even from the command line using `curl` to query the server and JSON as the format of the messages.

We'll finish this section with a small note directly from the Nova documentation: it explicitly states that everything in OpenStack is designed to be horizontally scalable but that, at any time, there should be significantly more computing nodes than any other type. This actually makes a lot of sense – computing nodes are the ones in charge of actually accepting and working on requests. The amount of storage nodes you'll need will depend on your usage scenario, and Glance's will inevitably depend on the capabilities and resources available to Swift.

The next service in line is **Horizon** – a *human-readable* GUI dashboard of OpenStack where we *consume* a lot of OpenStack visual information.

Horizon

Having explained the core services that enable OpenStack to do what it does the way it does in some detail, we need to address the user interaction. In almost every paragraph in this chapter, we refer to APIs and scripting interfaces as a way to communicate and orchestrate OpenStack. While this is completely true and is the usual way of managing large-scale deployments, OpenStack also has a pretty useful interface that is available as a web service in a browser. The name of this project is Horizon, and its sole purpose is to provide a user with a way of interacting with all the services from one place, called the dashboard. Users can also reconfigure most, if not all, the things in the OpenStack installation, including security, networking, access rights, users, containers, volumes, and everything else that exists in the OpenStack installation.

Horizon also supports plugins and *pluggable panels*. There is an active plugin marketplace for Horizon that aims at extending its functionality even further than it already has. If that's still not enough for your particular scenario, you can create your own plugins in Angular and get them to run in Horizon.

Pluggable panels are also a nice idea – without changing any defaults, a user or a group of users can change the way the dashboard looks and get more (or less) information presented to them. All of this requires a little bit of coding; changes are made in the config files, but the main thing is that the Horizon system itself supports such a customization model. You can find out more about the interface itself and the functions that are available to the user when we cover installing OpenStack and creating OpenStack instances in the *Provisioning the OpenStack environment* section.

As you are aware, networks don't really work all that well without name resolution, which is why OpenStack has a service called **Designate**. We'll briefly discuss Designate next.

Designate

Every system that uses any kind of network must have at least some kind of name resolution service in the form of a local or remote DNS or a similar mechanism.

Designate is a service that tries to integrate the *DNSaaS* concept in OpenStack in one place. When connected to Nova and Neutron, it will try to keep up-to-date records in regards to all the hosts and infrastructure details.

Another very important aspect of the cloud is how we manage identities. For that specific purpose, OpenStack has a service called **Keystone**. We'll discuss what it does next.

Keystone

Identity management is a big thing in cloud computing, simply because when deploying a large-scale infrastructure, not only do you need a way to scale your resources, but you also need a way to scale user management. A simple list of users that can access a resource is not an option anymore, mainly because we are not talking about simple users anymore. Instead, we are talking about domains containing thousands of users separated by groups and by roles – we are talking about multiple ways of logging in and providing authentication and authorization. Of course, this also can span multiple standards for authentication, as well as multiple specialized systems.

For these reasons, user management is a separate project/service in OpenStack named Keystone.

Keystone supports simple user management and the creation of users, groups, and roles, but it also supports LDAP, Oauth, OpenID Connect, SAML, and SQL database authentication and has its own API that can support every possible scenario for user management. Keystone is in a world by itself, and in this book, we will treat it as a simple user provider. However, it can be much more and can require a lot of configuration, depending on the case. The good thing is that, once installed, you will rarely need to think about this part of OpenStack.

The next service on our list is **Neutron**, the API/backend for (cloud) networking in OpenStack.

Neutron

OpenStack Neutron is an API-based service that aims to provide a simple and extensible cloud network concept as a development of what used to be called a *Quantum* service in older releases of OpenStack. Before this service, networking was managed by nova-network, which, as we mentioned, is a solution that's obsolete, with Neutron being the reason for this. Neutron integrates with some of the services that we've already discussed – Nova, Horizon, and Keystone. As a standalone concept, we can deploy Neutron to a separate server, which will then give us the ability to use the Neutron API. This is reminiscent of what VMware does in NSX with the NSX Controller concept.

When we deploy neutron-server, a web-based service that hosts the API connects to the Neutron plugin in the background so that we can introduce networking changes to our Neutron-managed cloud network. In terms of architecture, it has the following services:

- Database for persistent storage
- neutron-server
- External agents (plugins) and drivers

In terms of plugins, it has a *lot* of them, but here's a short list:

- Open vSwitch
- Cisco UCS/Nexus
- The Brocade Neutron plugin
- IBM SDN-VE
- VMware NSX
- Juniper OpenContrail
- Linux bridging
- ML2
- Many others

Most of these plugin names are logical, so you won't have any problems understanding what they do. But we'd like to mention one of these plugins specifically, which is the **Modular Layer 2 (ML2)** plugin.

By using the ML2 plugin, OpenStack Neutron can connect to various Layer 2 backends – VLAN, GRE, VXLAN, and so on. It also enables Neutron to go away from the Open vSwitch and Linux bridge plugins as its basic plugins (which are now obsolete). These plugins are considered to be too monolithic for Neutron's modular architecture, and ML2 has replaced them completely since the release of Havana (2013). ML2 today has many vendor-based plugins for integration. As shown by the preceding list, Arista, Cisco, Avaya, HP, IBM, Mellanox, and VMware all have ML2-based plugins for OpenStack.

In terms of network categories, Neutron supports two:

- **Provider networks:** Created by an OpenStack administrator, these are used for external connections on a physical level, which are usually backed by flat (untagged) or VLAN (802.1q tagged) concepts. These networks are shared since tenants use them to access their private infrastructure in hybrid cloud models or to access the internet. Also, these networks describe the way underlay and overlay networks interact, as well as their mappings.
- **Tenant networks, self-service networks, project networks:** These networks are created by users/tenants and their administrators so that they can connect their virtual resources and networks in whatever shape or form they need. These networks are isolated and usually backed by a network overlay such as GRE or VXLAN, as that's the whole purpose of tenant networks.

Tenant networks usually use some kind of SNAT mechanism to access external networks, and this service is usually implemented via virtual routers. The same concept is used in other cloud technologies such as VMware NSX-v and NSX-t, as well as Microsoft Hyper-V SDN technologies backed by Network Controller.

In terms of network types, Neutron supports multiple types:

- **Local:** Allows us to communicate within the same host.
- **Flat:** An untagged virtual network.
- **VLAN:** An 802.1Q VLAN tagged virtual network.
- **GRE, VXLAN, GENEVE:** Depending on the network overlay technologies, we select these network backends.

Now that we've covered OpenStack's usage models, ideas, and services, let's discuss additional ways in which OpenStack can be used. As you might imagine, OpenStack – being what it is – is highly capable of being used in many non-standard scenarios. We'll discuss these non-obvious scenarios next.

Additional OpenStack use cases

OpenStack has a lot of really detailed documentation available at <https://docs.openstack.org>. One of the more useful topics is the architecture and design examples, which both explain the usage scenarios and the ideas behind how a particular scenario can be solved using the OpenStack infrastructure. We are going to talk a lot about two different edge cases when we deploy our test OpenStack, but some things need to be said about configuring and running an OpenStack installation.

OpenStack is a complex system that encompasses not only computing and storage but also a lot of networking and supporting infrastructure. You will first notice that when you realize that even the documentation is neatly divided into an administration, architecture, operations, security, and virtual machine image guide. Each of these subjects is practically a topic for a single book, and a lot of things that guides cover are part experience, part best practice advice, and part assumptions based on best guesses.

There are a couple of things that are more or less common to all these use cases. First, when designing a cloud, you must try and get all the information about possible loads and your clients as soon as possible, even before a first server is booted. This will enable you to plan not only how many servers you need, but their location, the ratio of computing to storage nodes, the network topology, energy requirements, and all the other things that need to be thought through in order to create a working solution.

When deploying OpenStack, we are talking about a large-scale enterprise solution that is usually deployed for one of three reasons:

- *Testing and learning:* Maybe we need to learn how to configure a new installation, or we need to test a new computing node before we even go near production systems. For that reason, we need a small OpenStack environment, perhaps a single server that we can expand if there is a need for that. In practice, this system should be able to support probably a single user with a couple of instances. Those instances will usually not be the focus of your attention; they are going to be there just to enable you to explore all the other functionalities of the system. Deploying such a system is usually done the way we described in this chapter – using a readymade script that installs and configures everything so that we can focus on the part we are actually working on.
- *We have a need for a staging or pre-production environment:* Usually, this means that we need to either support the production team so they have a safe environment to work in, or we are trying to keep a separate test environment for storing and running instances before they are pushed into production.

Having such an environment is definitively recommended, even if you haven't had it yet, since it enables you and your team to experiment without fear of breaking the production environment. The downside is that this installation requires an environment that has to have some resources available for the users and their instances. This means we are not going to be able to get away with using a single server. Instead, we will have to create a cloud that will be, at least in some parts, as powerful as the production environment. Deploying such an installation is basically the same as production deployment since once it comes online, this environment will, from your perspective, be just another system in production. Even if we are calling it pre-production or test, if the system goes down, your users will inevitably call and complain. This is the same as what happens with the production environment; you will have to plan downtime, schedule upgrades, and try to keep it running as best as you can.

- *For production:* This one is demanding in another way – maintenance. When creating an actual production cloud environment, you will need to design it well, and then carefully monitor the system to be able to respond to problems. Clouds are a flexible thing from the user's perspective since they offer scaling and easy configuration, but being a cloud administrator means that you need to enable these configuration changes by having spare resources ready. At the same time, you need to pay attention to your equipment, servers, storage, networking, and everything else to be able to spot problems before the users see them. Has a switch failed over? Are the computing nodes all running correctly? Have the disks degraded in performance due to a failure? Each of these things, in a carefully configured system, will have minimal to no impact on the users, but if we are not proactive in our approach, compounding errors can quickly bring the system down.

Having distinguished between a single server and a full install in two different scenarios, we are going to go through both. The single server will be done manually using scripts, while the multi-server will be done using Ansible playbooks.

Now that we've covered OpenStack in quite a bit of detail, it's time to start using it. Let's start with some small things (a small environment to test) in order to provision a regular OpenStack environment for production, and then discuss integrating OpenStack with Ansible. We'll revisit OpenStack in the next chapter, when we start discussing scaling out KVM to Amazon AWS.

Creating a Packstack demo environment for OpenStack

If you just need a **Proof of Concept (POC)**, there's a very easy way to install OpenStack. We are going to use **Packstack** as it's the simplest way to do this. By using Packstack installation on CentOS 7, you'll be able to configure OpenStack in 15 minutes or so. It all starts with a simple sequence of commands:

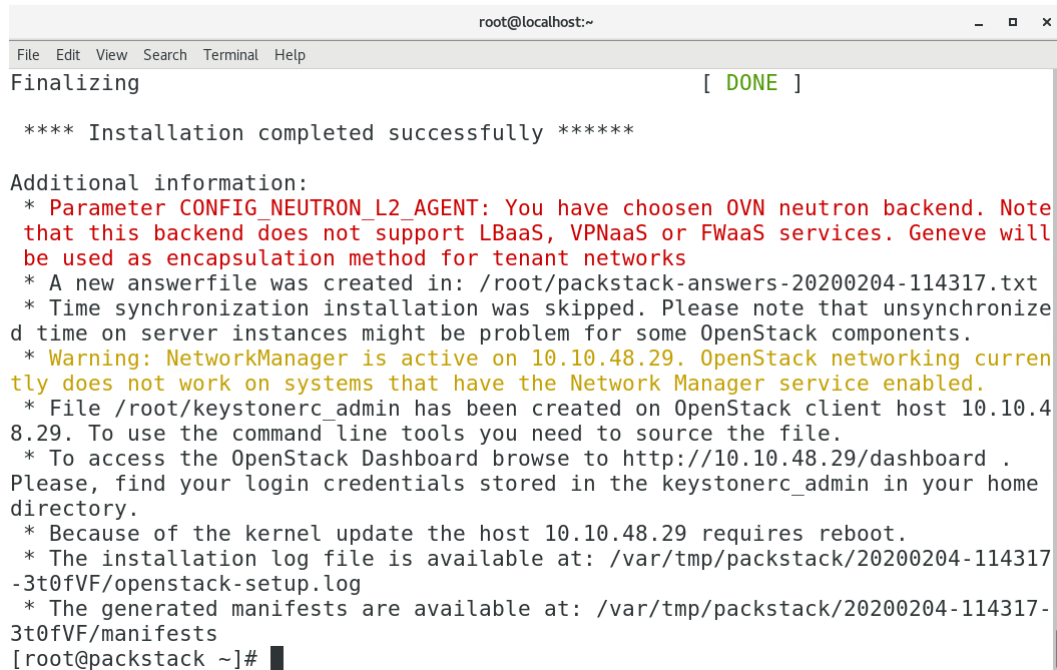
```
yum update -y
yum install -y centos-release-openstack-train
yum update -y
yum install -y openstack-packstack
packstack --allinone
```

As the process goes through its various phases, you'll see various messages, such as the following, which are quite nice as you get to see what's happening in real time with a decent verbosity level:

```
root@localhost:~
File Edit View Search Terminal Help
Preparing Nova Scheduler entries [ DONE ]
Preparing Nova VNC Proxy entries [ DONE ]
Preparing OpenStack Network-related Nova entries [ DONE ]
Preparing Nova Common entries [ DONE ]
Preparing Neutron LBaaS Agent entries [ DONE ]
Preparing Neutron API entries [ DONE ]
Preparing Neutron L3 entries [ DONE ]
Preparing Neutron L2 Agent entries [ DONE ]
Preparing Neutron DHCP Agent entries [ DONE ]
Preparing Neutron Metering Agent entries [ DONE ]
Checking if NetworkManager is enabled and running [ DONE ]
Preparing OpenStack Client entries [ DONE ]
Preparing Horizon entries [ DONE ]
Preparing Swift builder entries [ DONE ]
Preparing Swift proxy entries [ DONE ]
Preparing Swift storage entries [ DONE ]
Preparing Gnocchi entries [ DONE ]
Preparing Redis entries [ DONE ]
Preparing Ceilometer entries [ DONE ]
Preparing Aodh entries [ DONE ]
Preparing Puppet manifests [ DONE ]
Copying Puppet modules and manifests [ DONE ]
Applying 10.10.48.29_controller.pp
Testing if puppet apply is finished: 10.10.48.29 controller.pp [ \ ]
```

Figure 12.7 – Appreciating Packstack's installation verbosity

After the installation is finished, you will get a report screen that looks similar to this:



```
root@localhost:~
File Edit View Search Terminal Help
Finalizing [ DONE ]

**** Installation completed successfully ****

Additional information:
* Parameter CONFIG_NEUTRON_L2_AGENT: You have chosen OVN neutron backend. Note
that this backend does not support LBaaS, VPNaaS or FWaaS services. Geneve will
be used as encapsulation method for tenant networks
* A new answerfile was created in: /root/packstack-answers-20200204-114317.txt
* Time synchronization installation was skipped. Please note that unsynchroniz
ed time on server instances might be problem for some OpenStack components.
* Warning: NetworkManager is active on 10.10.48.29. OpenStack networking curren
tly does not work on systems that have the Network Manager service enabled.
* File /root/keystonerc_admin has been created on OpenStack client host 10.10.4
8.29. To use the command line tools you need to source the file.
* To access the OpenStack Dashboard browse to http://10.10.48.29/dashboard .
Please, find your login credentials stored in the keystonerc_admin in your home
directory.
* Because of the kernel update the host 10.10.48.29 requires reboot.
* The installation log file is available at: /var/tmp/packstack/20200204-114317
-3t0fVF/openstack-setup.log
* The generated manifests are available at: /var/tmp/packstack/20200204-114317-
3t0fVF/manifests
[root@packstack ~]#
```

Figure 12.8 – Successful Packstack installation

The installer has finished successfully, and it gives us a warning about `NetworkManager` and a kernel update, which means we need to restart our system. After the restart and checking the `/root/keystonerc_admin` file for our username and password, Packstack is alive and kicking and we can log in by using the URL mentioned in the previous screen's output (`http://IP_or_hostname_where_PackStack_is_deployed/dashboard`):

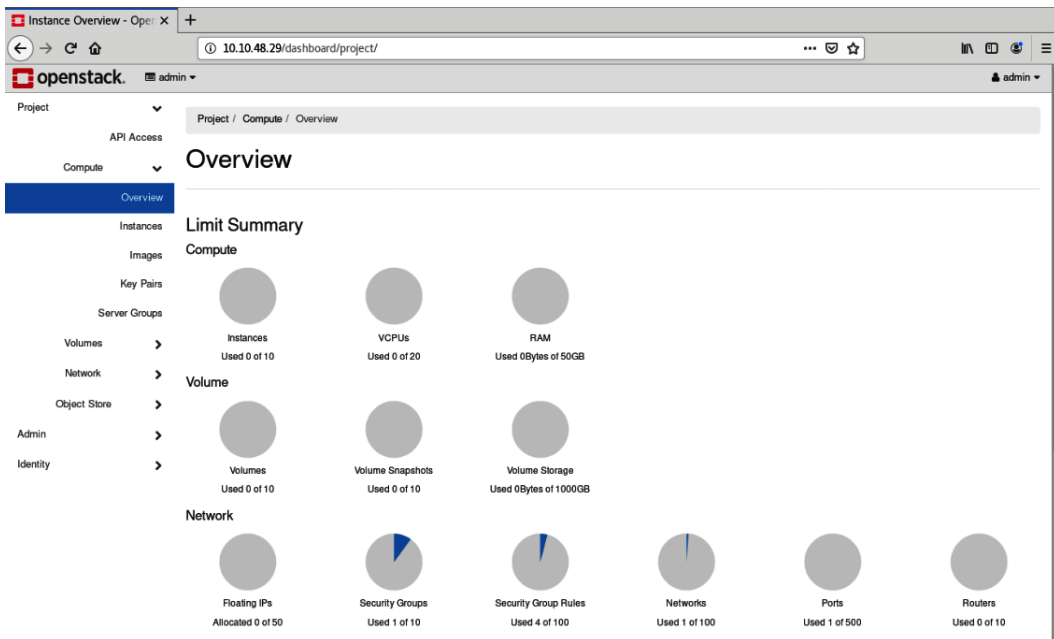


Figure 12.9 – Packstack UI

There's a bit of additional configuration that needs to be done, as noted in the Packstack documentation at <https://wiki.openstack.org/wiki/Packstack>. If you're going to use an external network, you need a static IP address without `NetworkManager`, and you probably want to either configure `firewalld` or stop it altogether. Other than that, you can start using this as your demo environment.

Provisioning the OpenStack environment

One of the tasks that is going to be the simplest and, at the same time, the hardest when you need to create your first OpenStack configuration is going to be provisioning. There are basically two ways you can go with this: one is to install services one at a time in a carefully prepared hardware configuration, while the other is to just use a *single server install* guide from the OpenStack site and create a single machine that will serve as your test bed. In this chapter, everything we do is created in such an instance, but before we learn how to install the system, we need to understand the differences.

OpenStack is a cloud operating system, and its main idea is to enable us to use multiple servers and other devices to create a coherent, easily configured cloud that can be managed from a central point, either through an API or through a web server. The size and type of the OpenStack deployment can be from one server running everything, to thousands of servers and storage units integrated across several Datacenters. OpenStack does not have a problem with large-scale deployment; the only real limiting factor is usually the cost and other requirements for the environment we are trying to create.

We mentioned scalability a few times, and this is where OpenStack shines in both ways. The amazing thing is that not only does it scale up easily but that it also scales down. An installation that will work perfectly fine for a single user can be done on a single machine – even on a single VM inside a single machine – so you will be able to have your own cloud within a virtual environment on your laptop. This is great for testing things but nothing else.

Having a bare-metal install that will follow the guidelines and recommended configuration requirements for particular roles and services is the only way to go forward when creating a working, scalable cloud, and obviously this is the way to go if you need to create a production environment. Having said that, between a single machine and a thousand server installs, there are a lot of ways that your infrastructure can be shaped and redesigned to support your particular use case scenario.

Let's first quickly go through an installation inside another VM, a task that can be accomplished in under 10 minutes on a faster host machine. For our platform, we decided on installing Ubuntu 18.04.3 LTS in order to be able to keep the host system to a minimum. The entire guide for Ubuntu regarding what we are trying to do is available at <https://docs.openstack.org/devstack/latest/guides/single-machine.html>.

One thing that we must point out is that the OpenStack site has a guide for a number of different install scenarios, both on virtual and bare-metal hardware, and they are all extremely easy to follow, simply because the documentation is straight to the point. There's also a simple install script that takes care of everything once a few steps are done manually by you.

Be careful with hardware requirements. There are some good sources available to cover this subject. Start here: <https://docs.openstack.org/newton/install-guide-rdo/overview.html#figure-hwreqs>.

Installing OpenStack step by step

The first thing we need to do is create a user that is going to install the entire system. This user needs to have `sudo` privileges since a lot of things require system-wide permissions.

Create a user either as root or through `sudo`:

```
useradd -s /bin/bash -d /opt/stack -m stack
chmod 755 /opt/stack
```

The next thing we need to do is allow this user to use `sudo`:

```
echo "stack ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers
```

We also need to install `git` and switch to our newly created user:

```
stack@openstack:~$ sudo apt install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  git-man libasn1-8-heimdal libcurl3-gnutls liberror-perl libgdbm-compat4 libgssapi3-heimdal
  libhcrypto4-heimdal libheimbase1-heimdal libheimntlm0-heimdal libhx509-5-heimdal
  libkrb5-26-heimdal libldap-2.4-2 libldap-common libnghttp2-14 libperl5.26 libroken18-heimdal
  librtmp1 libsasl2-2 libsasl2-modules libsasl2-modules-db libwind0-heimdal patch perl
  perl-modules-5.26
Suggested packages:
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs
  git-mediawiki git-svn libsasl2-modules-gssapi-mit | libsasl2-modules-gssapi-heimdal
  libsasl2-modules-ldap libsasl2-modules-otp libsasl2-modules-sql diffutils-doc perl-doc
  libterm-readline-gnu-perl | libterm-readline-perl-perl make
The following NEW packages will be installed:
  git git-man libasn1-8-heimdal libcurl3-gnutls liberror-perl libgdbm-compat4 libgssapi3-heimdal
  libhcrypto4-heimdal libheimbase1-heimdal libheimntlm0-heimdal libhx509-5-heimdal
  libkrb5-26-heimdal libldap-2.4-2 libldap-common libnghttp2-14 libperl5.26 libroken18-heimdal
  librtmp1 libsasl2-2 libsasl2-modules libsasl2-modules-db libwind0-heimdal patch perl
  perl-modules-5.26
0 upgraded, 25 newly installed, 0 to remove and 0 not upgraded.
Need to get 12.8 MB of archives.
After this operation, 80.6 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

Figure 12.10 – Installing `git`, the first step in deploying OpenStack

Now for the fun part. We are going to clone (copy the latest version of) `devstack`, the installation script that will provide everything we need to be able to run and use OpenStack on this machine:

```
stack@openstack:~$ git clone https://opendev.org/openstack/devstack
Cloning into 'devstack'...
remote: Enumerating objects: 44764, done.
remote: Counting objects: 100% (44764/44764), done.
remote: Compressing objects: 100% (20257/20257), done.
remote: Total 44764 (delta 31643), reused 36536 (delta 23819)
Receiving objects: 100% (44764/44764), 9.10 MiB | 900.00 KiB/s, done.
Resolving deltas: 100% (31643/31643), done.
stack@openstack:~$ cd devstack/
stack@openstack:~/devstack$ _
```

Figure 12.11 – Cloning devstack by using git

A little bit of configuration is now needed. Inside the `samples` directory, in the directory we just cloned, there is a file called `local.conf`. Use it to configure all the things the installer needs. Networking is one thing that has to be configured manually – not just the local network, which is the one that connects you to the rest of the internet, but also the internal network address space, which is going to get used for everything OpenStack needs to do between instances. Different passwords for different services also need to be set. All of this can be read in the sample file. Directions on how to exactly configure this are both on the web at the address we gave you earlier, and inside the file itself:

```
# Sample ``local.conf`` for user-configurable variables in ``stack.sh``
# NOTE: Copy this file to the root DevStack directory for it to work properly.
# ``local.conf`` is a user-maintained settings file that is sourced from ``stackrc``.
# This gives it the ability to override any variables set in ``stackrc``.
# Also, most of the settings in ``stack.sh`` are written to only be set if no
# value has already been set; this lets ``local.conf`` effectively override the
# default values.

# This is a collection of some of the settings we have found to be useful
# in our DevStack development environments. Additional settings are described
# in https://docs.openstack.org/devstack/latest/configuration.html#local-conf
# These should be considered as samples and are unsupported DevStack code.

# The ``localrc`` section replaces the old ``localrc`` configuration file.
# Note that if ``localrc`` is present it will be used in favor of this section.
[[local|localrc]]

# Minimal Contents
# -----

# While ``stack.sh`` is happy to run without ``localrc``, devlife is better when
# there are a few minimal variables set:

# If the ``*_PASSWORD`` variables are not set here you will be prompted to enter
# values for them by ``stack.sh`` and they will be added to ``local.conf``.
ADMIN_PASSWORD=nomoresecret
DATABASE_PASSWORD=stackdb
RABBIT_PASSWORD=stackqueue
SERVICE_PASSWORD=$ADMIN_PASSWORD

# ``HOST_IP`` and ``HOST_IPV6`` should be set manually for best results if
# the NIC configuration of the host is unusual, i.e. ``eth1`` has the default
# route but ``eth0`` is the public interface. They are auto-detected in
# ``stack.sh`` but often is indeterminate on later runs due to the IP moving
:
```

Figure 12.12 – Installer configuration

There will be some issues with this installation process, and as a result, installation might break twice because of the following reasons:

- Ownership of `/opt/stack/.cache` is `root:root`, instead of `stack:stack`. Please correct this ownership before running the installer;
- An installer problem (a known one), as it fails to install a component and then fails. Solution is rather simple - there's a line that needs to be changed in a file in `inc` directory, called `python`. At time of writing, line 192 of that file needs to be changed from `$cmd_pip $upgrade \` to `$cmd_pip $upgrade --ignore-installed \`

In the end, after we collected all the data and modified the file, we settled on this configuration:

```
[[local|localrc]]
FLOATING_RANGE=192.168.61.222/24
FIXED_RANGE=10.11.10.0/24
ADMIN_PASSWORD=secretpass
DATABASE_PASSWORD=dbpass
RABBIT_PASSWORD=rabbitpass
SERVICE_PASSWORD=$ADMIN_PASSWORD

LOGFILE=$DEST/logs/stack.sh.log

LOGDAYS=2

SWIFT_REPLICAS=1

SWIFT_DATA_DIR=$DEST/data
~
```

Figure 12.13 – Example configuration

Most of these parameters are understandable, but let's cover two of them first: `FLOATING_RANGE` and `FIXED_RANGE`. The `FLOATING_RANGE` parameter tells our OpenStack installation which network scope will be used for *private* networks. On the other hand, `FIXED_RANGE` is the network scope that will be used by OpenStack-provisioned virtual machines. Basically, virtual machines provisioned in OpenStack environments will be given internal addresses from `FIXED_RANGE`. If a virtual machine needs to be available from the outside world as well, we will assign a network address from `FLOATING_RANGE`. Be careful with `FIXED_RANGE` as it shouldn't match an existing network range in your environment.

One thing we changed from what is given in the guide is that we reduced the number of replicas in the Swift installation to one. This gives us no redundancy, but reduces the space used for storage and speeds things up a little. Do not do this in the production environment.

Depending on your configuration, you may also need to set the `HOST_IP` address variable in the file. Here, set it to your current IP address.

Then, run `./stack.sh`.

Once you've run the script, a really verbose installation should start and dump a lot of lines on your screen. Wait for it to finish – it is going to take a while and download a lot of files from the internet. At the end, it is going to give you an installation summary that looks something like this:

```
This is your host IP address: 192.168.61.129
This is your host IPv6 address: ::1
Horizon is now available at http://192.168.61.129/dashboard
Keystone is serving at http://192.168.61.129/identity/
The default users are: admin and demo
The password: secretpass

WARNING:
Using lib/neutron-legacy is deprecated, and it will be removed in the future

Services are running under systemd unit files.
For more information see:
https://docs.openstack.org/devstack/latest/systemd.html

DevStack Version: ussuri
Change: 455be66098353b08dabf38ec7256998de89ac755 Merge "Remove conflicting packages in Ubuntu" 2020-01-30 00:01:06 +0000
OS Version: Ubuntu 18.04 bionic
stack@openstack:~/devstack$ _
```

Figure 12.14 – Installation summary

Once this is done, if everything is okay, you should have a complete running version of OpenStack on your local machine. In order to verify that, connect to your machine using a web browser; a welcome screen should appear:



Figure 12.15 – OpenStack login screen

After logging in with the credentials that are written on your machine, after the installation (the default administrator name is `admin` and the password is the one you set in `local.conf` when installing the service), you are going to be welcomed by a screen showing you the stats for your cloud. The screen you are looking at is actually a Horizon dashboard and is the main screen that provides you with all you need to know about your cloud at a glance.

OpenStack administration

Looking at the top-left corner of Horizon, we can see that there are three distinct sections that are configured by default. The first one – **Project** – covers everything about our default instance and its performance. This is where you can create new instances, manage images, and work on server groups. Our cloud is just a core installation, so we only have one server and two defined zones, which means that we have no server groups installed:

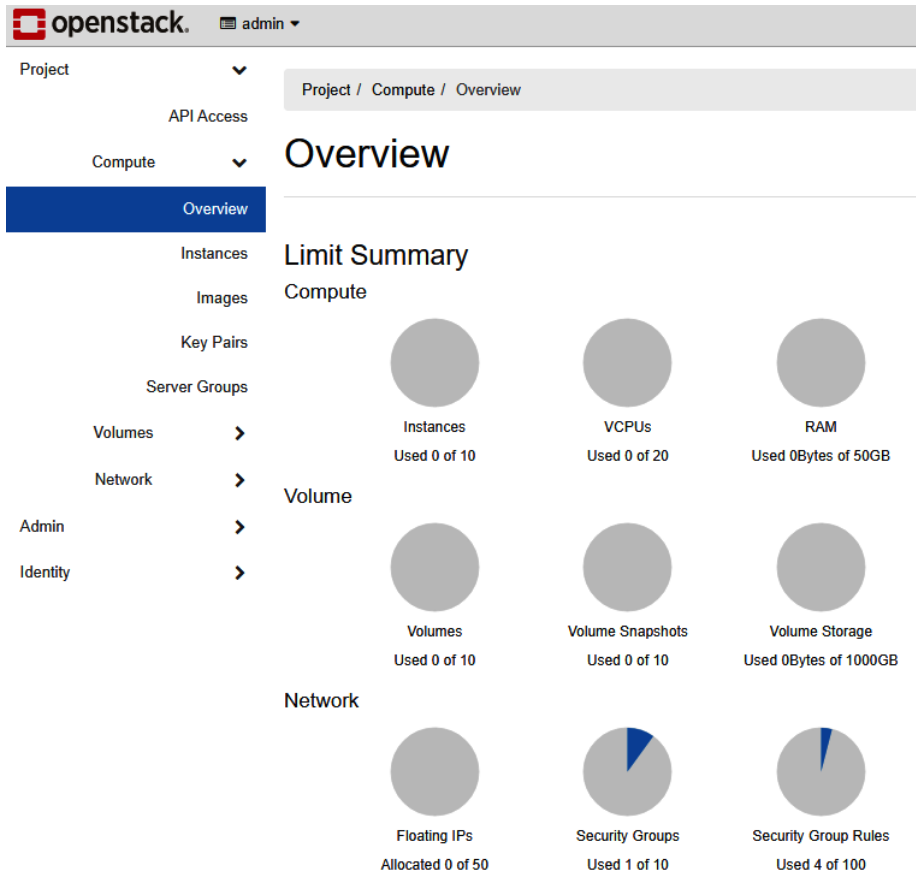


Figure 12.16 – Basic Horizon dashboard

First, let's create a quick instance to show how this is done:

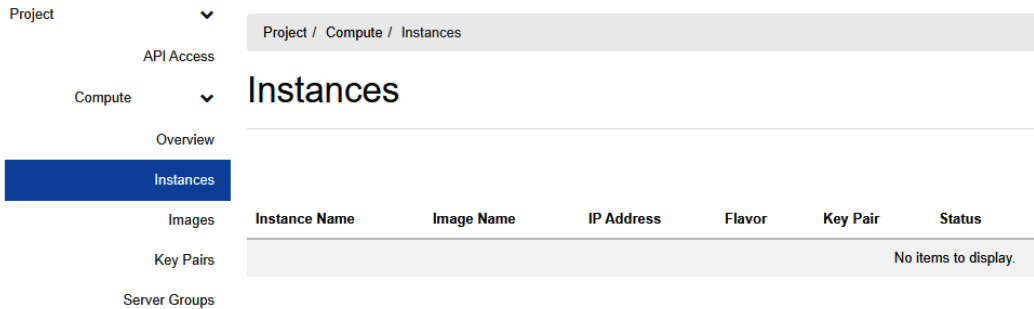


Figure 12.17 – Creating an instance

Follow these steps to create an instance:

1. Go to **Launch Instance** in the far-right part of the screen. A window will open that will enable you to give OpenStack all the information it needs to create a new VM instance:

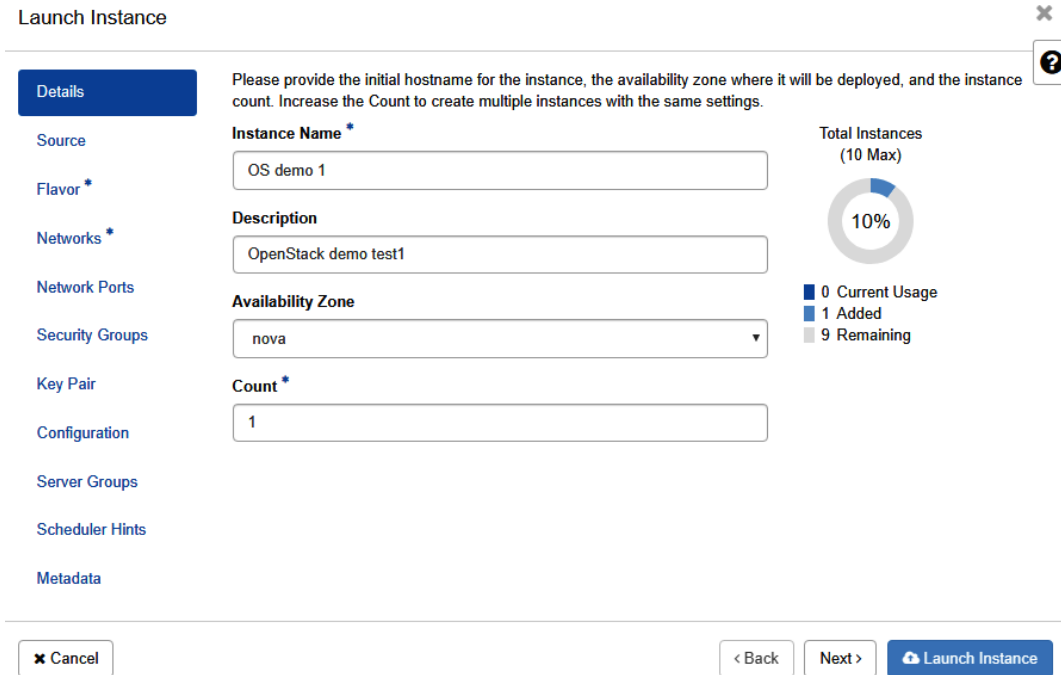


Figure 12.18 – Launch Instance wizard

- On the next screen, you need to supply the system with the image source. We already mentioned glances – these images are taken from the Glance store and can be either an image snapshot, a ready-made volume, or a volume snapshot. We can also create a persistent image if we want to. One thing that you'll notice is that there are two differences when comparing this process to almost any other deployment. The first is that we are using a ready-made image by default as one was provided for us. Another big thing is the ability to create a new persistent volume to store our data in, or to have it deleted when we are done with the image, or have it not be created at all.

Choose the one image you have allocated in the public repository; it should be called something similar to the one shown in the following screenshot. CirrOS is a test image provided with OpenStack. It's a minimal Linux distribution that is designed to be as small as possible and enable easy testing of the whole cloud infrastructure but to be as unobtrusive as possible. CirrOS is basically an OS placeholder. Of course, we need to click on the **Launch Instance** button to go to the next step:

Launch Instance
✕

Details

Source

Flavor *

Networks *

Network Ports

Security Groups

Key Pair

Configuration

Server Groups

Scheduler Hints

Metadata

Instance source is the template used to create an instance. You can use an image, a snapshot of an instance (image snapshot), a volume or a volume snapshot (if enabled). You can also choose to use persistent storage by creating a new volume.

Select Boot Source

Image
▼

Create New Volume

Yes
No

Volume Size (GB) *

1

Delete Volume on Instance Delete

Yes
No

Allocated

Name	Updated	Size	Type	Visibility
> cirros-0.4.0-x86_64-disk	1/30/20 2:19 PM	12.13 MB	qcow2	Public

Available 0 Select one

Q
Click here for filters or full text search.
✕

Name	Updated	Size	Type	Visibility
<i>No available items</i>				

✕ Cancel
< Back
Next >
Launch Instance

Figure 12.19 – Selecting an instance source

- The next important part of creating a new image is choosing a flavor. This is another one of those peculiarly named things in OpenStack. A flavor is a combination of certain resources that basically creates a computing, memory, and storage template for new instances. We can choose from instances that have as little as 64 MB of RAM and 1 vCPU and go as far as our infrastructure can provide:

Launch Instance ✕

Details

Source

Flavor ^{*}

Networks ^{*}

Network Ports

Security Groups

Key Pair

Configuration

Server Groups

Scheduler Hints

Metadata

Flavors manage the sizing for the compute, memory and storage capacity of the instance. ?

Allocated

Name	VCPUS	RAM	Total Disk	Root Disk	Ephemeral Disk	Public
Select an item from Available items below						

▼ Available 12 Select one

Name	VCPUS	RAM	Total Disk	Root Disk	Ephemeral Disk	Public
▶ m1.nano	1	64 MB	1 GB	1 GB	0 GB	Yes
▶ m1.micro	1	128 MB	1 GB	1 GB	0 GB	Yes
▶ cirros256	1	256 MB	1 GB	1 GB	0 GB	Yes
▶ m1.tiny	1	512 MB	1 GB	1 GB	0 GB	Yes
▶ ds512M	1	512 MB	5 GB	5 GB	0 GB	Yes
▶ ds1G	1	1 GB	10 GB	10 GB	0 GB	Yes
▶ m1.small	1	2 GB	20 GB	20 GB	0 GB	Yes
▶ ds2G	2	2 GB	10 GB	10 GB	0 GB	Yes
▶ m1.medium	2	4 GB	40 GB	40 GB	0 GB	Yes
▶ ds4G	4	4 GB	20 GB	20 GB	0 GB	Yes
▶ m1.large	4	8 GB	80 GB	80 GB	0 GB	Yes
▶ m1.xlarge	8	16 GB	160 GB	160 GB	0 GB	Yes

Figure 12.20 – Selecting an instance flavor

In this particular example, we are going to choose `cirros256`, a flavor that is basically designed to provide our test system with as few resources as is feasible.

4. The last thing we actually need to choose is the network connectivity. We need to set all the adapters our instance will be able to use while running. Since this is a simple test, we are going to use both adapters we have, both the internal and external one. They are called `public` and `shared`:

Launch Instance ✕

Details ?

Source

Flavor

Networks

Network Ports

Security Groups

Key Pair

Configuration

Server Groups

Scheduler Hints

Metadata

Networks provide the communication channels for instances in the cloud.

▼ Allocated 2 Select networks from those listed below.

	Network	Subnets Associated	Shared	Admin State	Status	
⇅ 1	➤ public	public-subnet ipv6-public-subnet	No	Up	Active	⌵
⇅ 2	➤ shared	shared-subnet	Yes	Up	Active	⌵

▼ Available 0 Select at least one network

🔍 Click here for filters or full text search. ✕

Network	Subnets Associated	Shared	Admin State	Status
<i>No available items</i>				

✕ Cancel < Back Next > 🔴 Launch Instance

Figure 12.21 – Instance network configuration

Now, we can launch our instance and it will be able to boot. Once you click on the **Launch Instance** button, it is going to take probably under a minute to create a new instance. The screen showing its current progress and instance status will auto update while the instance is being deployed.

Once this is done, our instance will be ready:

<input type="checkbox"/>	OS demo 1	cirros-0.4.0-x86_64-disk	shared 192.168.233.155 public 192.168.61.24, 2001:db8::1b9
--------------------------	------------------	--------------------------	---

Figure 12.22 – The instance is ready

We'll quickly create another instance, and then create a snapshot so that we can show you how image management works. If you click on the **Create snapshot** button on the right-hand side of the instance list, Horizon will create a snapshot and immediately put you in the interface meant for image administration:

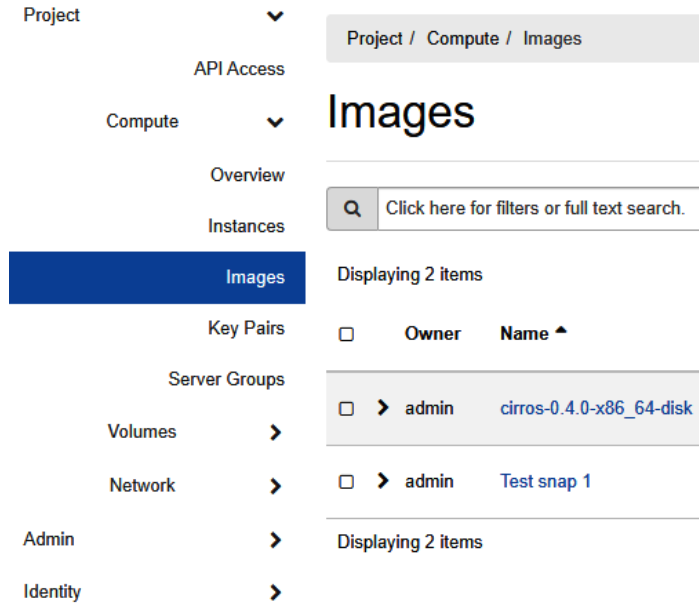


Figure 12.23 – Images

Now, we have two different snapshots: one that is the start image and another that is an actual snapshot of the image that is running. So far, everything has been simple. What about creating an instance out of a snapshot? It's just a click away! What you need to do is just click on the **Launch Instance** button on the right and go through the wizard for creating new instances.

The end result of our short example of instance creation should be something like this:

<input type="checkbox"/>	OS test 3	Test snap 1	public 192.168.61.221, 2001:db8::263 shared 192.168.233.191	cirros256
<input type="checkbox"/>	OS demo 2	cirros-0.4.0-x86_64-disk	shared 192.168.233.121 public 192.168.61.98, 2001:db8::1b6	cirros256
<input type="checkbox"/>	OS demo 1	cirros-0.4.0-x86_64-disk	shared 192.168.233.155 public 192.168.61.24, 2001:db8::1b9	cirros256

Figure 12.24 – New instance creation finished

What we can see is all the information we need on our instances, what their IP addresses are, their flavor (which translates into what amount of resources are allocated for a particular instance), the availability zone that the image is running in, and information on the current instance state. The next thing we are going to check out is the **Volumes** tab on the left. When we created our instances, we told OpenStack to create one permanent volume for the first instance. If we now click on **Volumes**, we should see the volume under a numeric name:

The screenshot shows the OpenStack dashboard interface. On the left, a sidebar contains navigation links: Project, API Access, Compute, Volumes (highlighted in blue), Snapshots, Groups, Group Snapshots, Network, Admin, and Identity. The main content area displays the breadcrumb 'Project / Volumes / Volumes' and the title 'Volumes'. Below the title, it indicates 'Displaying 1 item' and shows a table with the following data:

<input type="checkbox"/>	Name	Description	Size	Status
<input type="checkbox"/>	73238817-a806-4af7-82fe-511f5108c6d4	-	1GiB	In-use

Figure 12.25 – Volumes

From this screen, we can now snapshot the volume, reattach it to a different instance, and even upload it as an image to the repository.

The third tab on the left-hand side, named **Network**, contains even more information about our currently configured setup.

If we click on the **Network Topology** tab, we will get the whole network topology of our currently running network, shown in a simple graphical display. We can choose from **Topology** and **Graph**, both of which basically represent the same thing:

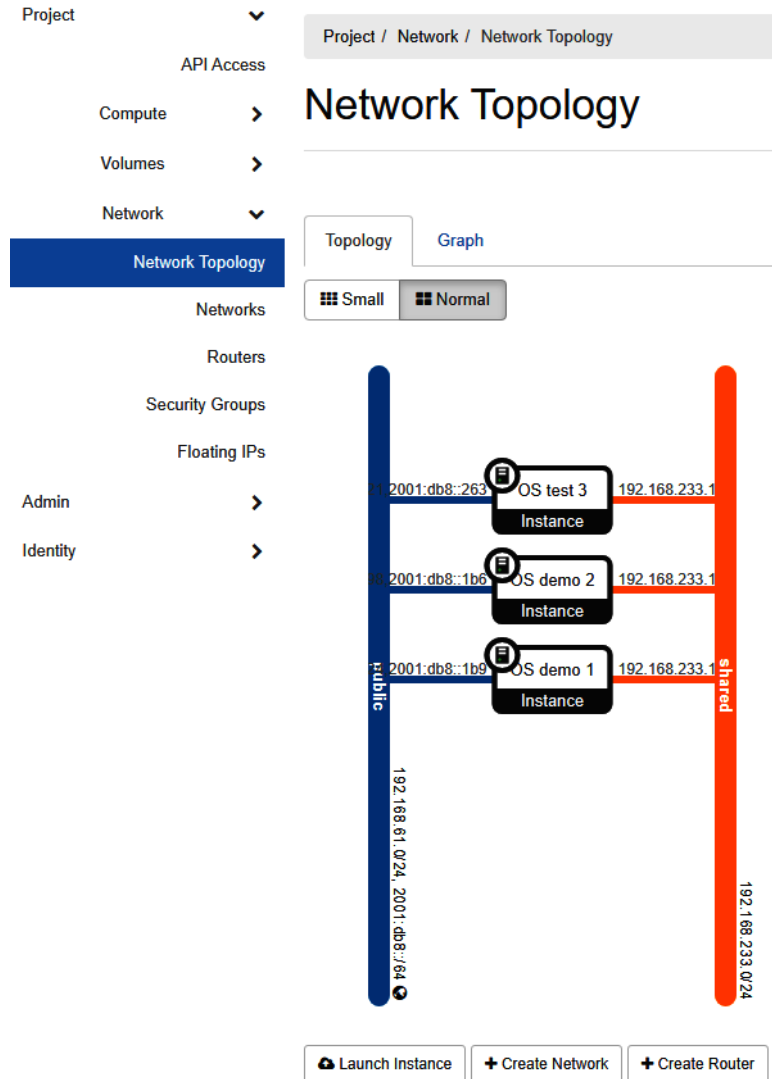


Figure 12.26 – Network topology

If we need to create another network or change anything in the network matrix, we can do so here. We consider this to be really administrator-friendly, on top of being documentation-friendly. Both of these points make our next topic – day-to-day administration – much easier.

Day-to-day administration

We are more or less finished with the most important options that are in any way connected to the administration of our day-to-day tasks in the **Project** Datacenter. If we click on the tab named **Admin**, we will notice that the menu structure we've opened looks a lot like the one under **Project**. This is because, now, we are looking at administration tasks that have something to do with the infrastructure of the cloud, not the infrastructure of our particular logical Datacenter, but the same building blocks exist in both of these. However, if we – for example – open **Compute**, a completely different set of options exist:

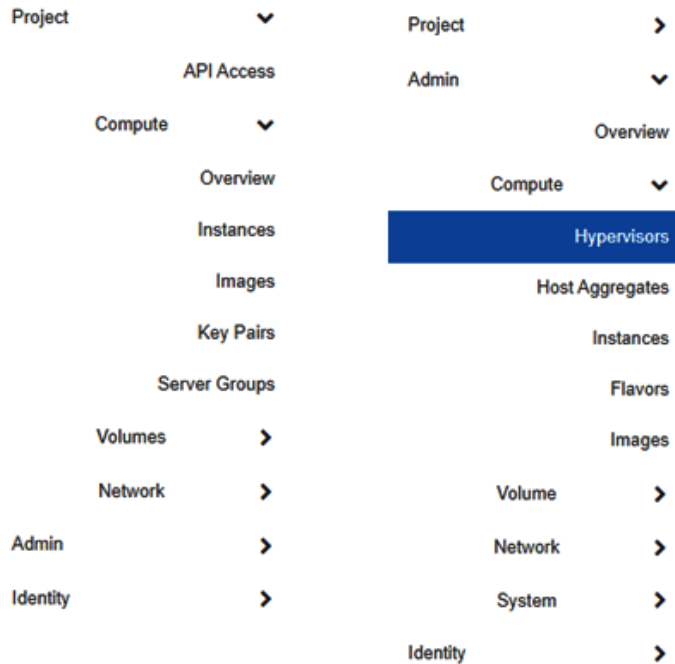


Figure 12.27 – Different available configuration options

This part of the interface is used to completely administer parts that form our infrastructure and those that define different things we can use while working in our *Datacenter*. When logged in as a user, we can add and remove virtual machines, configure networks, and use resources, but to put resources online, add new hypervisors, define flavors, and do these kinds of tasks that completely change the infrastructure, we need to be assigned the administrative role. Some of the functions overlap, such as both the administrative part of the interface and the user-specific part, which have control over instances. However, the administrative part has all these functions, and users can have their set of commands tweaked so that they are, for instance, unable to delete or create new instances.

The administrative view enables us to monitor our nodes on a more direct level, not only through the services they provide, but also through raw data about a particular host and the resources utilized on it:

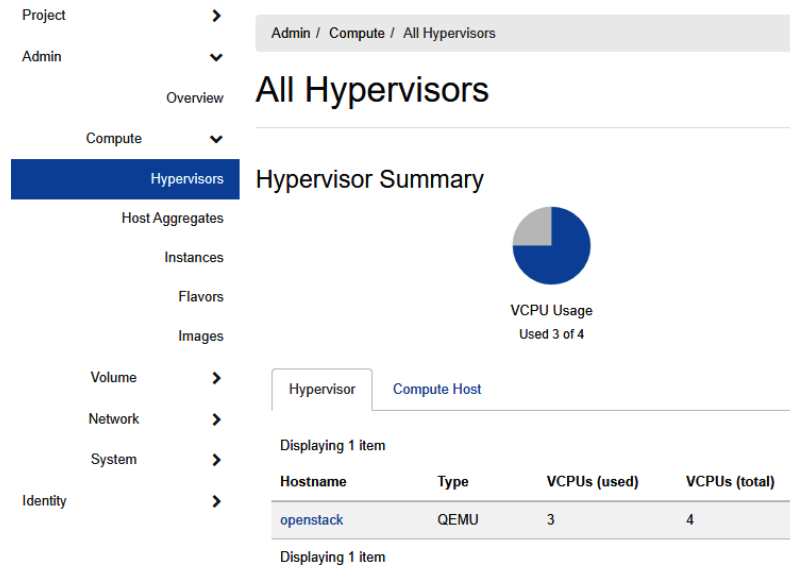


Figure 12.28 – Available hypervisors in our Datacenter

Our Datacenter has only one hypervisor, but we can see the amount of resources physically available on it, and the share of those resources the current setup is using at this particular moment.

Flavors are also one important part of the whole of OpenStack. We already mentioned them as predefined sets of resource presets that form a platform that the instance is going to run on. Our test setup has a few of them defined, but we can delete the ones that are shipped in this setup and create new ones tailored to our needs.

Since the point of the cloud is to optimize resource management, flavors play a big part in this concept. Creating flavors is not an easy task in terms of planning and design. First and foremost, it requires deep knowledge of what is possible on a given hardware platform, how much and what computing resources even exist, and how to utilize it to the full extent possible. So, it is essential that we plan and design things properly. The other thing is that we actually need to understand what kind of load we are preparing for. Is it memory-intensive? Do we have many small services that require a lot of nodes with a simple configuration? Are we going to need a lot of computing power and/or a lot of storage? The answers to those questions are something that will not only enable us to create what our clients want, but also create flavors that will have users utilizing our infrastructure in full.

The basic idea is to create flavors that will give individual users just enough resources to get their job done in a satisfactory way. This is not obvious in a deployment that has 10 instances, but once we run into thousands, a flavor that always leaves 10 percent of the storage unused is quickly going to eat into our resources and limit our ability to serve more users. Striking this balance between what we have and what we give users to use in a particular way is probably the hardest task in planning and designing our environments:

Create Flavor ✕

Flavor Information * Flavor Access

Name *

ID ⓘ

VCPUs *

RAM (MB) *

Root Disk (GB) *

Ephemeral Disk (GB)

Swap Disk (MB)

RX/TX Factor

Flavors define the sizes for RAM, disk, number of cores, and other resources and can be selected when users deploy instances.

Figure 12.29 – Create Flavor wizard

Creating a flavor is a simple task. We need to do the following:

1. Give it a name; an ID will be assigned automatically.
2. Set the number of vCPUs and RAM for our flavor.

3. Select the size of a base disk, and an ephemeral disk that doesn't get included in any of the snapshots and gets deleted when a virtual machine is terminated.
4. Select the amount of swap space.
5. Select the RX/TX factor so that we can create a bit of QoS on the network level. Some flavors will need to have more network traffic priority than others.

OpenStack allows a particular project to have more than one flavor, and for a single flavor to belong to different projects. Now that we've learned that, let's work with our user identities and assign them some objects.

Identity management

The last tab on the left-hand side is **Identity**, which is responsible for handling users, roles, and projects. This is where we are going to configure not only our usernames, but the user roles, groups, and projects a particular user can use:

<input type="checkbox"/>	User Name	Description	Email	User ID	Enabled	Domain Name	Actions
<input type="checkbox"/>	admin	-		1db1304b539a449c9d70440e618e1fb4	Yes	Default	Edit
<input type="checkbox"/>	demo	-	demo@example.com	6e850c68bb14e1496b37b82a91084c	Yes	Default	Edit
<input type="checkbox"/>	alt_demo	-	alt_demo@example.com	d929d27e7cc4480384c54561ce9121f	Yes	Default	Edit
<input type="checkbox"/>	nova	-		48b53227bae8448e83380940461d9db	Yes	Default	Edit
<input type="checkbox"/>	glance	-		132e38a908ab42c18d14cc8501a5b612	Yes	Default	Edit
<input type="checkbox"/>	cinder	-		d835c5b6a8ba49c2b999302684cc027	Yes	Default	Edit
<input type="checkbox"/>	neutron	-		546cc3048b442e297544955a6cc8905	Yes	Default	Edit
<input type="checkbox"/>	placement	-		4db784672b4e119768ab6d74c316dd	Yes	Default	Edit

Figure 12.30 – Users, Groups, and Roles management

We are not going to go too much into how users are managed and installed, but just cover the basics of user management. As always, the original documentation on the OpenStack site is the place to go to learn more. Make sure that you check out this link: <https://docs.openstack.org/keystone/pike/admin/cli-manage-projects-users-and-roles.html>.

In short, once you create a project, you need to define which users are going to be able to see and work on a particular project. In order to ease administration, users can also be part of groups, and you can then assign whole groups to a project.

The point of this structure is to enable the administrator to limit the users not only to what they can administer, but also to how many of the available resources are available for a particular project. Let's use an example for this. If we go to **Projects** and edit an existing project (or create a new one), we'll see a tab called **Quota** in the configuration menu, which looks like this:

Resource	Quota Value
Instances *	10
VCPUs *	20
RAM (MB) *	51200
Metadata Items *	128
Key Pairs *	100
Server Groups *	10
Server Group Members *	10
Injected Files *	5
Injected File Content (Bytes) *	10240
Length of Injected File Path *	255

Figure 12.31 – Quotas on the default project

Once you create a project, you can assign all the resources in the form of quotas. This assignment limits the maximum available resources for a particular group of instances. The user has no overview of the whole system; they can only *see* and utilize resources available through the project. If a user is part of multiple projects, they can create, delete, and manage instances based on their role in the project, and the resources available to them are specific to a project.

We'll discuss OpenStack/Ansible integration next, as well as some potential use cases for these two concepts to work together. Keep in mind that the larger the OpenStack environment is, the more use cases we will find for them.

Integrating OpenStack with Ansible

Dealing with any large-scale application is not easy, and not having the right tool can make it impossible. OpenStack provides a lot of ways for us to directly orchestrate and manage a huge horizontal deployment, but sometimes, this is not enough. Luckily, in our arsenal of tools, we have another one – **Ansible**. In *Chapter 11, Ansible for Orchestration and Automation*, we covered some other, smaller ways to use Ansible to deploy and configure individual machines, so we are not going to go back to that. Instead, we are going to focus on things that Ansible is good for in the OpenStack environment.

One thing that we must make clear, though, is that using Ansible in an OpenStack environment can be based on two very distinct scenarios. One is using Ansible to handle deployed instances, in a way that would pretty much look the same across all the other cloud or bare-metal deployments. You, as an administrator of a large number of instances, create a management node that is nothing more than a Python-enabled server with added Ansible packages and playbooks. After that, you sort out the inventory for your deployment and are ready to manage your instances. This scenario is not what this part of this chapter is about.

What we are talking about here is using Ansible to manage the cloud itself. This means we are not deploying instances inside the OpenStack cloud; we are deploying compute and storage nodes for OpenStack itself.

The environment we are talking about is sometimes referred to as **OpenStack-Ansible (OSA)** and is common enough to have its own deployment guide, located at the following URL: <https://docs.openstack.org/project-deploy-guide/openstack-ansible/latest/>.

The requirements for a minimal installation in OpenStack-Ansible are considerably greater than those in a single VM or on a single machine. The reason for this is not just that the system needs all the resources; it's the tools that need to be used and the philosophy behind it all.

Let's quickly go through what Ansible means in terms of OpenStack:

- Once configured, it enables the quick deployment of any kind of resource, be it storage or computing.
- It makes sure you are not forgetting to configure something in the process. When deploying a single server, you will have to make sure that everything works and that errors in configuration are easy to spot, but when deploying multiple nodes, errors can creep in and degrade the performance of part of the system without anyone noticing. The normal deployment practice to avoid this is an installation checklist, but Ansible is a much better solution than that.

- More streamlined configuration changes. Sometimes, we need to apply a configuration change across the whole system or some part of it. This can be frustrating if not scripted.

So, having said all that, let's quickly go through <https://docs.openstack.org/openstack-ansible/latest/> and see what the official documentation says about how to deploy and use Ansible and OpenStack.

What exactly does OpenStack offer to the administrator in regard to Ansible? The simplest answer is playbooks and roles.

To use Ansible to deploy OpenStack, you basically need to create a deployment host and then use Ansible to deploy the whole OpenStack system. The whole workflow goes something like this:

1. Prepare the deployment host
2. Prepare the target hosts
3. Configure Ansible for deployment
4. Run playbooks and let them install everything
5. Check whether OpenStack is correctly installed

When we are talking about deployment and target hosts, we need to make a clear distinction: the deployment host is a single entity holding Ansible, scripts, playbooks, roles, and all the supporting bits. The target hosts are the actual servers that are going to be part of the OpenStack cloud.

The requirements for installation are straightforward:

- The operating system should be a minimal installation of Debian, Ubuntu CentOS, or openSUSE (experimental) with the latest kernel and full updates applied.
- Systems should also run Python 2.7, have SSH access enabled with public key authentication, and have NTP time sync enabled. This covers the deployment host.
- There are also usual recommendations for different types of nodes. Computing nodes must support hardware-assisted virtualization, but that's an obvious requirement.
- There is a requirement that should go without saying, and that is to use multicore processors, with as many cores as possible, to enable some services to run much faster.

Disk requirements are really up to you. OpenStack suggests using fast disks if possible, recommending SSD drives in a RAID, and large pools of disks available for block storage.

- Infrastructure nodes have requirements that are different than the other types of nodes since they are running a few databases that grow over time and need at least 100 GB of space. The infrastructure also runs its services as containers, so it will consume resources in a particular way that will be different than the other compute nodes.

The deployment guide also suggests running a logging host since all the services create logs. The recommended disk space is at least 50 GB for logs, but in production, this will quickly grow in orders of magnitude.

OpenStack needs a fast, stable network to work with. Since everything in OpenStack will depend on the network, every possible solution that will speed up network access is recommended, including using 10G and bonded interfaces. Installing a deployment server is the first step in the overall process, which is why we'll do that next.

Installing an Ansible deployment server

Our deployment server needs to be up to date with all the upgrades and have Python, `git`, `ntp`, `sudo`, and `ssh` support installed. After you've installed the required packages, you need to configure the `ssh` keys to be able to log into the target hosts. This is an Ansible requirement and is also a best practice that leverages security and ease of access.

The network is simple – our deployment host must have connectivity to all the other hosts. The deployment host should also be installed on the L2 of the network, which is designed for container management.

Then, the repository should be cloned:

```
# git clone -b 20.0.0 https://opendev.org/openstack/openstack-ansible /opt/openstack-ansible
```

Next, an Ansible bootstrap script needs to be run:

```
# scripts/bootstrap-ansible.sh
```

This concludes preparing the Ansible deployment server. Now, we need to prepare the target computers we are going to use for OpenStack. Target computers are currently supported on Ubuntu Server (18.04) LTS, CentOS 7, and openSUSE 42.x (at the time of writing, there still isn't CentOS 8 support). You can use any of these systems. For each of them, there is a helpful guide that will get you up and running quickly: <https://docs.openstack.org/project-deploy-guide/openstack-ansible/latest/deploymenthost.html>. We'll just explain the general steps to ease you into installing it, but in all truth, just copy and paste the commands that have been published for your operating system from <https://www.openstack.org/>.

No matter which system you decide to run on, you have to be completely up to date with system updates. After that, install the `linux-image-extra` package (if it exists for your kernel) and install the `bridge-utils`, `debootstrap`, `ifenslave`, `lsuf`, `lvm2`, `chrony`, `openssh-server`, `sudo`, `tcpdump`, `vlan`, and Python packages. Also, enable bonding and VLAN interfacing. All these things may or may not be available for your system, so if something is already installed or configured, just skip over it.

Configure the NTP time sync in `chrony.conf` to synchronize time across the whole deployment. You can use any time source you like, but for the system to work, time must be in sync.

Now, configure the `ssh` keys. Ansible is going to deploy using `ssh` and key-based authentication. Just copy the public keys from the appropriate user on your deployment machine to `/root/.ssh/authorized_keys`. Test this setup by simply logging in from the deployment host to the target machine. If everything is okay, you should be able to log in without any password or any other prompt. Also, note that the root user on the deployment host is the default user for managing everything and that they have to have their `ssh` keys generated in advance since they are used not only on the target hosts but also in all the containers for different services running across the system. These keys must exist when you start to configure the system.

For storage nodes, please note that LVM volumes will be created on the local disks, thus overwriting any existing configuration. Network configuration is going to be done automatically; you just need to ensure that Ansible is able to connect to the target machines.

The next step is configuring our Ansible inventory so that we can use it. Let's do that now.

Configuring the Ansible inventory

Before we can run the Ansible playbooks, we need to finish configuring the Ansible inventory so that it points the system to the hosts it should install on. We are going to quote the verbatim, available at <https://docs.openstack.org/project-deploy-guide/openstack-ansible/queens/configure.html>:

1. *Copy the contents of the `/opt/openstack-ansible/etc/openstack_deploy` directory to the `/etc/openstack_deploy` directory.*
2. *Change to the `/etc/openstack_deploy` directory.*
3. *Copy the `openstack_user_config.yml.example` file to `/etc/openstack_deploy/openstack_user_config.yml`.*
4. *Review the `openstack_user_config.yml` file and make changes to the deployment of your OpenStack environment.*

Once inside the configuration file, review all the options. `openstack_user_config.yml` defines which hosts run which services and nodes. Before committing to installing, please review the documentation mentioned in the previous paragraph.

One thing that stands out on the web is `install_method`. You can choose either `source` or `distro`. Each has its pros and cons:

- `Source` is the simplest installation as it's done directly from the sources on the OpenStack official site and contains an environment that's compatible with all systems.
- The `distro` method is customized for the particular distribution you are installing on by using specific packages known to work and known as being stable. The major drawback of this is that updates are going to be much slower since not only OpenStack needs to be deployed but also information about all the packages on distributions, and that setup needs to be verified. As a result, expect long waits between when the upgrade reaches the *source* and gets to your *distro* installation. After installing, you must go with your primary choice; there is no mechanism for switching from one to the other.

The last thing you need to do is open the `user_secrets.yml` file and assign passwords for all the services. You can either create your own passwords or use a script provided just for this purpose.

Running Ansible playbooks

As we go through the deployment process, we will need to start a couple of Ansible playbooks. We need to use these three provided playbooks in this order:

- `setup-hosts.yml` : The initial Ansible playbook that we use to provision the necessary services on our OpenStack hosts.
- `setup-infrastructure.yml`: The Ansible playbook that deploys some more services, such as RabbitMQ, repository server, Memcached, and so on.
- `setup-openstack.yml`: The Ansible playbook that deploys the remaining services – Glance, Cinder, Nova, Keystone, Heat, Neutron, Horizon, and so on.

All of these Ansible playbooks need to be finished successfully so that we can integrate Ansible with Openstack. So, the only thing left is to run the Ansible playbooks. We need to start with the following command:

```
# openstack-ansible setup-hosts.yml
```

You can find the appropriate files in `/opt/openstack-ansible/playbooks`. Now, run the remaining setups:

```
# openstack-ansible setup-infrastructure.yml
```

```
# openstack-ansible setup-openstack.yml
```

All the playbooks should finish without unreachable or failed plays. And with that – congratulations! You have just installed OpenStack.

Summary

In this chapter, we spent a lot of time describing the architecture and inner workings of OpenStack. We discussed software-defined networking and its challenges, as well as different OpenStack services such as Nova, Swift, Glance, and so on. Then, we moved on to practical issues, such as deploying Packstack (let's just call that OpenStack for proof of concept), and full OpenStack. In the last part of this chapter, we discussed OpenStack-Ansible integration and what it might mean for us in larger environments.

Now that we've covered the *private* cloud aspect, it's time to grow our environment and expand it to a more *public* or *hybrid*-based approach. In KVM-based infrastructures, this usually means connecting to AWS to convert your workloads and transfer them there (public cloud). If we're discussing the hybrid type of cloud functionality, then we have to introduce an application called Eucalyptus. For the hows and whys, check out the next chapter.

Questions

1. What is the main problem with VLAN as a cloud overlay technology?
2. Which types of cloud overlay networks are being used on the cloud market today?
3. How does VXLAN work?
4. What are some of the most common problems with stretching Layer 2 networks across multiple sites?
5. What is OpenStack?
6. What are the architectural components of OpenStack?
7. What is OpenStack Nova?
8. What is OpenStack Swift?
9. What is OpenStack Glance?
10. What is OpenStack Horizon?
11. What are OpenStack flavors?
12. What is OpenStack Neutron?

Further reading

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

- OpenStack documentation: <https://docs.openstack.org>
- Arista VXLAN overview: https://www.arista.com/assets/data/pdf/Whitepapers/Arista_Networks_VXLAN_White_Paper.pdf
- Red Hat – What is GENEVE?: <https://www.redhat.com/en/blog/what-geneve>
- Cisco – Configuring Virtual Networks Using OpenStack: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus1000/kvm/config_guide/network/5x/b_Cisco_N1KV_KVM_Virtual_Network_Config_5x/configuring_virtual_networks_using_openstack.pdf
- Packstack: <http://rdoproject.org>

13

Scaling out KVM with AWS

Virtualization is a hard problem if you take a look at it up close – it is complicated to emulate complete computers in order to be able to run operating systems on them. For obvious reasons, getting those virtual machines into a cloud is even harder. After that, things really start to get messy. Conceptually, creating clusters of machines that can run on demand is even more complicated, based on the sheer number of machines that must run at the same time. Also, there's a need to create not only emulated computers but also all the networking and infrastructure to support larger deployments. Creating a global cloud – one that not only runs millions of machines but is also almost omnipresent in even the most remote parts of the globe – is a task that few companies have ever tried, and only a couple have succeeded. This chapter will cover those big cloud providers in general, and then Amazon, as the biggest of them all. Our main idea is to present what makes Amazon tick, how it relates to the rest of the topics covered in this book, and how to use the services Amazon enables in the real world on real machines.

Amazon Web Services (AWS) is a unique set of tools, services, and infrastructure that enable cloud services on a really massive scale, a scale that is so huge that it becomes hardly comprehensible. When we are talking about thousands of sites using millions of servers to run billions of applications, even enumerating these things becomes a big problem, and management is something that can easily span not only a single chapter, but probably multiple books. We are going to try to introduce you to the most important services and parts of the AWS cloud, and try to explain what they can be used for and when.

In this chapter, we will cover the following topics:

- Introduction to AWS
- Preparing and converting virtual machines for AWS
- Building hybrid KVM clouds with Eucalyptus

Introduction to AWS

While talking about cloud services, AWS is one that needs almost no introduction, although few people actually understand how big and complex a system the whole Amazon cloud is. What is completely certain is that, at this time, it is unquestionably the biggest and most used service on the planet.

Before we do anything else, we need to talk about how and why AWS is so important, not only in regard to its impact on the internet but also on any task that even remotely tries to provide some kind of scaling.

As we already have done a couple of times in this book, we will start with the basic premise of the AWS cloud – to provide a broadly scalable and distributed solution that will encompass all possible scenarios for performing any type of workload on the internet.

In almost every other place in this book where we mentioned the cloud, we talked about scaling up, but when we try to describe AWS as being able to scale, we are talking about probably one of the largest providers of capacity and scalability on the planet.

Right now, there are more or less four really big cloud providers on the internet: AWS, Microsoft Azure, Google Cloud Platform, and Alibaba. Since all the numbers are confidential for business reasons, the number of servers and sheer capacity they can provide is something analysts try to estimate, or more frequently guess, but it has to be in the millions.

Approaching the cloud

Although on the surface they are now competing for the same cloud market, all the players came from different backgrounds, and the way they use their infrastructure even now is vastly different. In this market, Amazon was first, and it got a head start that in IT seems almost unbelievable – roughly 6 *years*. Amazon introduced its Amazon Web Services in 2006 but it started the development of the service a couple of years earlier. There is even a blog post that mentions the service published back in 2004. The idea for AWS was basically conceived once Amazon realized it had a vast infrastructure that was unmatched in the market, and that by expanding it and offering it as a service, it could make a profit. At that point in time, the infrastructure they had was used to provide the Amazon.com service.

This idea was different from anything on the market. People were used to having collocated computers in data centers, and being able to rent a server in the *cloud*, but the concept of renting just the part of the stack they needed instead of the entire hardware infrastructure was something new. The first big service AWS offered was simple, and was even named like that – **Simple Storage Service (S3)**, along with **Elastic Compute Cloud (EC2)**. S3 was basically cloud-backed storage that offered almost unlimited storage resources for those who could pay for them in pretty much the same way it is available even today. EC2 offered computing resources.

Offerings expanded to a **Content Delivery Network (CDN)** and much more during the next 6 years, while the competition was still trying to get to grips with what the cloud actually meant.

We'll come back to the services AWS offers in a moment, but only after we mention the competition they eventually got in what has become a market worth hundreds of *billions* of dollars yearly.

Microsoft realized it would need to build up an infrastructure to support itself and its customers sometime in the late 2000s. Microsoft had its own business support infrastructure in place to run the company, but there were no public services offered to the general public at that time. That changed once **Microsoft Azure** was introduced in 2010. Initially, it was called **Windows Azure**, and it was mainly created to run services for both Microsoft and its partners, mainly on Windows. Very quickly, Microsoft realized that offering just a Microsoft operating system in the cloud was going to cost them a lot of customers, so Linux was also offered as an operating system that could be installed and used.

Azure now runs as a publicly available cloud, but a large portion of it is still used by Microsoft and its services, most notably **Office 365** and a myriad of Microsoft training and marketing solutions.

Google, on the other hand, came to the market in a different way. They also realized the need for a cloud offering but limited their first engagement with the cloud to offering a single service called **App Engine**, in 2008. It was a service targeted at the web developer community, and Google even gave 10,000 free licenses for the usage of the service in a limited way. At its core, this service and almost all the services that came after it came out with the premise that the web needs services that will enable developers to quickly deploy something that may or may not scale and that may or may not work. Therefore, giving it for free meant that a lot of developers were inclined to use the service just for simple testing.

Google now also has a vast number of services offered, but when you take a look from outside at the actual services and the pricing, it seems that Google has created its cloud as a way to lease out extra capacity it has available in its data centers.

Multi-cloud

Looking a few years back, both Azure and Google Cloud Platform had a viable cloud service, but compared to what AWS was offering, their services were simply not up to par. AWS was the biggest player, both in terms of market share, but also in people's minds. Azure was considered as being more Microsoft oriented, although more than half of the servers running on it are Linux-based, and Google just wasn't perceived as a competitor; their cloud looked more like a side business than a viable proposal to run a cloud.

Then came **multi-cloud**. The idea is simple – do not use a single cloud to deploy your services; use multiple cloud providers to provide both *data redundancy*, *availability*, and *failover*, and most important – *cost reduction and agility*. It may sound strange, but one of the biggest costs when using a cloud service is getting data out of it. Usually, getting data into the cloud, be it the user uploading data, or you deploying data on a server, is either free or has an extremely low cost, which makes sense, since you are more likely to use more services on this particular cloud if you have a lot of data online. Once you need to extract your data, it becomes expensive. This is intentional, and it keeps users locked into the cloud provider. Once you upload your data, it is much cheaper to just keep it on the servers and not try to work with it offline. But the data is not the only thing that has to be considered when talking about multi-cloud; services are also part of the equation.

Why multi-cloud?

Many companies (and we must stress that multi-cloud users are mostly big companies because of the costs involved) are scared of being locked into a particular platform or technology. One of the biggest questions is what happens if a platform changes so much that the company has to redo part of its infrastructure? Imagine that you are a multibillion-dollar company running an enterprise application for hundreds of thousands of your own users. You chose the cloud for the usual reasons – to keep capital expenditures down and to be able to scale your services. You decided to go with one of the big providers. Suddenly, your provider decides it is going to change technologies and will phase out some part of the infrastructure. When it comes to shifts like that it usually means that your current setup will slowly become much more expensive, or you are going to lose some part of the available functionalities. Thankfully, these things also typically stretch into years, as no sane cloud provider is going to go through a strategic change overnight.

But a change is a change, and you as a corporation have a choice – stay with the provider and face the consequences in the form of a much higher price for your systems – or redesign the systems, which will also cost money, and may take years to finish – sometimes decades.

So, a lot of companies decided on a very conservative strategy – to design a system that could run on any cloud, and that means using the lowest common denominator of all available technologies. This also means that the system can be migrated from cloud to cloud in a matter of days. Some of them then decided to even run the system on different clouds at the same time. This is an extremely conservative approach, but it works.

Another big reason to use a multi-cloud strategy is the complete opposite of the one that we just mentioned. Sometimes, the idea is to use a particular cloud service or services that are the best in a very specialized task. This means choosing different services from different providers to perform separate tasks but to do it as efficiently as possible. In the long run, it will also mean having to change providers and systems from time to time, but if the core system that the company uses is designed with that in mind, this approach can have its benefits.

Shadow IT

There is another way that a company can become a multi-cloud environment without even knowing it, this is usually called **Shadow IT**. If a company does not have a strict security policy and rules, some of the workers might start using services that are not part of the services that they are provided with by the company. It could be a cloud storage container, a videoconferencing system, or a mailing list provider. In bigger companies, it could even be that entire departments start using something from different cloud providers without even realizing it. All of a sudden, there is company data on a server that is outside of the scope that company's IT covers or is able to cover.

One of the better examples of this particular phenomenon was how the usage of video conferencing services changed during the COVID-19 virus worldwide pandemic. Almost all companies had an established communication system, usually a messaging system that covered the whole company. And then, literally overnight, the pandemic put all workers in their homes. Since communication is the crucial thing in running a company, everyone decided to switch to video and audio conferencing in the span of a week, globally. What happened next can and probably will become a bestselling book theme one day. Most companies tried to stick with their solution but almost universally that attempt failed on the first day, either because the service was too basic or too outdated to be used as both an audio and video conferencing solution, or because the service was not designed for the sheer volume of calls and requests and crashed.

People wanted to coordinate, so suddenly nothing was off the table. Every single video conferencing solution suddenly became a candidate. Companies, departments, and teams started experimenting with different conferencing systems, and cloud providers soon realized the opportunity – almost all the services instantly became free for even sizeable departments, allowing, in some cases, up to 150 people to participate in conferences.

A lot of services crashed due to demand, but big providers mostly were able to scale up to the volume required to keep everything running.

Since the pandemic was global, a lot of people decided they also needed a way to talk to their family. So individual users started using different services at home, and when they decided something worked, they used it at work. In a matter of days, companies became a multi-cloud environment with people using one provider for communication, another for email, a third for storage, and a fourth for backups. The change was so quick that sometimes IT was informed of the change a couple of days after the systems went online and the people were already using them.

This change was so enormous that at the time we are writing this book, we cannot even try to predict how many of these services are going to become a regular part of the company toolset, once users realize something works better than the company-provided software. These services further prove this point by being able to work continuously through a major disaster like this, so there is only so much a company-wide software usage policy can do to stop this chaotic multi-cloud approach.

Market share

One of the first things everyone mentions as soon as cloud computing companies and services are mentioned is the market share each one of them has. We also need to address this point, since we said that we are talking about the *biggest one* or the *second one*. Before multi-cloud became a thing, market share was divided basically between AWS, with the biggest market share; Azure as a distant second; followed by Google and a big group of *small* providers, such as Alibaba, Oracle, IBM, and such.

Once multi-cloud became a thing, the biggest problem became how to establish who had the biggest actual market share. All the big companies started using different cloud providers and just simply trying to add up the market share of the providers became difficult. From different polls, it is clear that Amazon is still the leading provider but that companies are slowly starting to use other providers together with Amazon services.

What this means is that, right now, AWS is still by far the cloud provider of choice but the choice itself is no longer about a single provider. People are using other providers as well.

Big infrastructure but no services

Sometimes, trying to divide the market share also has another point of view that must be considered. If we are talking about cloud providers, we usually think that we are talking about companies that have the biggest infrastructure created to support cloud services. Sometimes, in reality, we are actually comparing those companies that have the biggest portfolio of the services on the market. What do we mean by this?

There is a distinct company that has a big cloud presence but uses its own infrastructure almost exclusively to deliver its own content – Facebook. Although it's hard to compare infrastructure sizes in terms of the number of servers, data centers, or any other metric, since those numbers are a closely guarded secret, Facebook has an infrastructure that is in the same order of magnitude in size as AWS. The real difference is that this infrastructure is not going to serve services for third parties, and in reality, it was never meant to do so; everything that Facebook created was tailor-made to support itself, including choosing locations for the data centers, configuring and deploying hardware, and creating software. Facebook is not going to suddenly turn into another AWS; it's too big to do that. Available infrastructure does not always correlate with cloud market share.

Pricing

Another topic we have to cover, if just to mention it, is the one of pricing. Almost every mention of the cloud in this book is technical. We compared probably every possible metric that made any sense, from **IOPS**, through **GHz**, to **PPS** on the network side, but the cloud is not only a technical problem – when you have to put it in use, someone has to pay for it.

Pricing is a hot topic in the cloud world since the competition is fierce. All the cloud providers have their pricing strategies, and rebates and special deals are almost the norm, and all that turns understanding pricing into a nightmare, especially if you are new to all the different models. One thing is certain, all the providers will say that they are going to charge you only for what you use but defining what they actually mean by that can be a big problem.

When starting to plan the costs of deployment, you should first stop and try to define what you need, how much of it you need, and whether you are using the cloud in the way it is meant to be used. By far the most common mistake is to think that the cloud is in any form similar to using a normal server. The first thing people notice is the price of a particular instance, in a particular data center, running a particular configuration. The price will usually be either the monthly cost of the instance, and will usually be prorated, so you will pay only for the part that you use, or the price will be given for a different time unit – per day, per hour, or maybe even per second. This should be your first clue: you pay for using the instance, so in order to keep the costs down, do not keep your instances running all the time. This also means that your instances must be designed to be quickly brought up and down on demand so using the *standard* approach of installing a single or multiple servers and running them all the time is not necessarily a good option here.

When choosing instances, the options are literally too numerous to name here. All the cloud providers have their own idea of what people need, so you can not only choose simple things such as the number of processors or the amount of memory but also get an OS preinstalled, and get wildly varied types of storage and networking options. Storage is an especially complicated topic we are just going to quickly scratch the surface of here and only mention later. All the cloud providers offer two things – some sort of storage meant to be attached to instances, and some sort of storage that is meant to be used as a service. What a given provider offers can depend on the instance you are trying to request, the data center you are trying to request it in, and a number of other factors. Expect that you will have to balance three things: capacity, pricing, and speed. Of course, here, we are talking about instance storage. Storage as a service is even more complicated and with that, you have to think about pricing and capacity, but also about other factors like latency, bandwidth, and so on.

For example, AWS enables you to choose from a variety of services that go from database storage, file storage, and long-term backup storage, to different types of block and object storage. In order to use these services optimally, you need to first understand what is being offered, how it's being offered, what the different costs involved are, and how to use them to your advantage.

Another thing that you will notice quickly is that when the cloud providers said that everything is a service, they really meant it. It is completely possible to have a running application without having a single server instance. Tasks can be accomplished by stitching different services together, and this is by design. This creates an enormously flexible infrastructure, one that scales quickly and easily, but requires not only a different way of writing code but a completely different mindset when designing the solution you need. If you have no experience, find an expert, since this is the fundamental problem with your solution. It has to run on the cloud, not be running on your virtual machines that happen to be in the cloud.

Our advice to you is simple – *read a lot of documentation*. All the providers have excellent resources that will enable you to understand what their service provides, and how, but what the thousands of pages will not tell you is how it compares to the competition, and more importantly, what is the optimal way of connecting the services together. When paying for cloud services, expect that you will make a mistake once in a while and pay for it. This is why it's useful to use a pay-as-you-go option when getting services deployed – if you make a mistake, you will not run up a huge bill; your infrastructure will simply stop.

The other thing to mention when talking about pricing is that everything costs a little, but the composite price for a given configuration can be huge. Any additional resource will cost money. An *internal link between servers, external IP address, firewall, load balancer, virtual switch*, these are all the things that we usually don't even think about when designing infrastructure, but once we need them in the cloud, they can become expensive. Another thing to expect is that some of the services have different contexts – for example, network bandwidth can have a different price if you are transferring data between instances or to the outside world. The same goes for storage – as we mentioned earlier in this chapter, most providers will charge you different prices when storing and getting data out of the cloud.

Data centers

A couple of times in this chapter, we have mentioned **data centers**, and it is important that we talk a little bit about them. Data centers are at the core of the cloud infrastructure, and in more ways than you may think of. When we talked about a lot of servers, we mentioned that we usually group them into racks, and put the racks into data centers. As you are probably aware, a data center is in its essence a group of racks with all the infrastructure that servers need to function optimally, both in terms of power and data, but also when it comes to cooling, securing, and all the other things required to keep the servers running. They also need to be logically divided into **risk zones** that we usually call **fault domains**, so that we can avert various risks associated with the *we deployed everything on one rack* or *we deployed everything on one physical server* scenarios.

A data center is a complex infrastructure element in any scenario since it requires a combination of things to be efficient, secure, and redundant. Putting a group of servers in a rack is easy enough, but providing *cooling* and *power* is not a simple task. Add to that the fact that the cooling, power, and data all have to be *redundant* if you want your servers to work, and that all that needs to be secure, both from fires, floods, earthquakes and people, and the cost of running a real data center can be high. Of course, a data center running a couple of hundred servers is not as complex as the one running thousands or even tens of thousands, and the prices rise with the size of the facility. Add to that that having multiple data centers creates additional infrastructure challenges in connecting them so costs add up.

Now multiply that cost by a hundred since this is the number of data centers each of the cloud providers keep around the world. Some of the centers are small, some are huge but the name of the game is simple – *networking*. In order to be a truly global provider, all of them have to have a data center, or a couple of servers at least, as close to you as possible. If you are reading this in one of the bigger cities in almost any big country in the world, chances are there is an AWS, Microsoft, or Google-owned server in a radius of 100 miles from you. All the providers try to have at least one data center in every big city in every country since that can enable them to offer a range of services extremely quickly. This concept is called **Point of Contact (POC)** and means that when connecting to the provider's cloud, you just need to get to the nearest server, and after that, the cloud will make sure your services are as quick as possible.

But when we are talking about data centers that actually belong to Amazon or the others, we are still dealing with a large-scale operation. Here, numbers are in the hundreds, and their location is also a secret, mainly for security reasons. They all have a few things in common. They are a highly automated operation situated somewhere in the vicinity of a major power source, a major source of cooling, or a major data hub. Ideally, they would be placed in a spot that has all those three things, but that is usually not possible.

Placement is the key

Different companies have different strategies since choosing a good place to build a data center can mean a lot of cost savings. Some even go to extremes. Microsoft, for instance, has a data center completely submerged in the ocean to facilitate cooling.

When providing a service for a particular user, your main concern is usually speed and latency, and that in turn means that you want your server or your service to run in the data center that is closest to the user. For that purpose, all cloud providers divide their data centers geographically, which in turn enables administrators to deploy their services in the optimal part of the internet. But at the same time, this creates a typical problem with resources – there are places on the planet that have a small number of available data centers but are heavily populated, and there are places that are quite the opposite. This in turn has a direct influence on the price of resources. When we talked about pricing, we mentioned different criteria; now we can add another one – location. The location of a data center is usually given as a **region**. This means that AWS, or any other provider for that matter, is not going to give you the location of their data center, but instead will say *users in this region would be best served by this group of servers*. As a user, you have no idea where the particular servers are, but instead, you only care about the region as given to you by the provider. You can find the names of service regions and their codes here:

Name	Code
US East (Ohio)	us-east-2
US East (N. Virginia)	us-east-1
US West (N. California)	us-west-1
US West (Oregon)	us-west-2
Asia Pacific (Hong Kong)	ap-east-1
Asia Pacific (Mumbai)	ap-south-1
Asia Pacific (Osaka-Local)	ap-northeast-3
Asia Pacific (Seoul)	ap-northeast-2
Asia Pacific (Singapore)	ap-southeast-1
Asia Pacific (Sydney)	ap-southeast-2
Asia Pacific (Tokyo)	ap-northeast-1
Canada (Central)	ca-central-1
Europe (Frankfurt)	eu-central-1
Europe (Ireland)	eu-west-1
Europe (London)	eu-west-2
Europe (Paris)	eu-west-3
Europe (Stockholm)	eu-north-1
Middle East (Bahrain)	me-south-1
South America (São Paulo)	sa-east-1

Figure 13.1 – Service regions on AWS with the names used in the configuration

Choosing a service provided by a region that is in heavy demand can be expensive, and that same service can be much cheaper if you choose a server that is somewhere else. This is the beauty of the cloud – you can use the services that suit you and your budget.

Sometimes price and speed are not the most important things. For example, legal frameworks such as **GDPR**, a European regulation on personal data collection, processing, and movement, basically states that companies from Europe must use a data center in Europe since they are covered by the regulation. Using a US region in this case could mean that a company could be legally liable (unless the company running this cloud service is a part of some other framework that allows this – such as **Privacy Shield**).

AWS services

We need to talk a little bit about what AWS offers in terms of services since understanding services is one thing that will enable you to use the cloud appropriately. On AWS, all the available services are sorted into groups by their purpose. Since AWS has hundreds of services, the AWS management console, the first page you will see once you log in, will at first be a daunting place.

You will probably be using the AWS Free Tier for learning purposes, so the first step is to actually open an AWS Free account. Personally, I used my own personal account. For the Free account, we need to use the following URL: <https://aws.amazon.com/free/>, and follow along with the procedure. It just asks for a couple of pieces of information, such as email address, password, and AWS account name. It will ask you for credit card info as well, to make sure that you don't abuse the AWS account.

After signing up, we can log in and get to the AWS dashboard. Take a look at this screenshot:

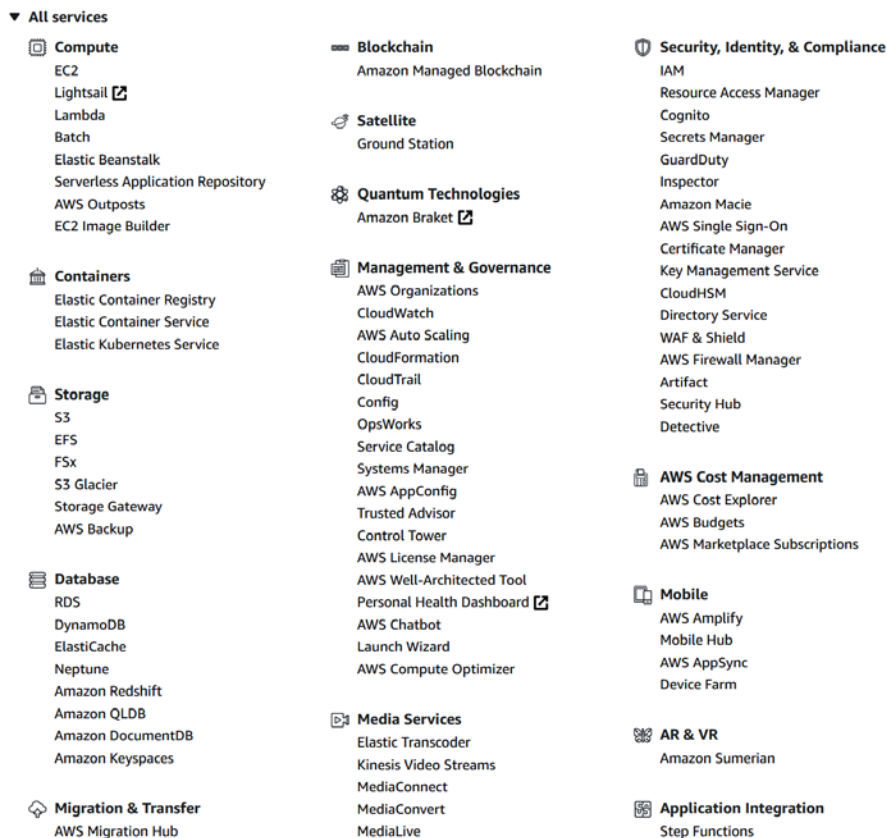


Figure 13.2 – Amazon services







Every single thing here is a link, and they all point to different services or pages with subservices. Moreover, this screenshot shows only about a third of all available services. There is no point in covering them all in this book; we are just going to use three of all these to show how AWS connects to our KVM infrastructure, but once you get the hang of it, you will slowly begin to understand how everything connects and what to use in a particular moment. What really helps is that AWS has great documentation, and that all the different services have provisioning wizards that help you find the thing you are looking for.

In this particular chapter, we are going to use these three services: **IAM**, **EC2**, and **S3**.

All of these are of course abbreviations, but other services just use project names, such as **CloudFront** or **Global Accelerator**. In any case, your first point of order should be to start using them, not just read about them; it is much easier to understand the structure once you put it to good use.

In this chapter, we used a Free account, and almost everything we did was free, so there is no reason for you not to try to use the AWS infrastructure yourself. AWS tries to be helpful there as much as it can, so if you scroll down on the console page, you will find these helpful icons:

Build a solution
Get started with simple wizards and automated workflows.

<p>Launch a virtual machine</p> <p>With EC2 2-3 minutes</p> 	<p>Build a web app</p> <p>With Elastic Beanstalk 6 minutes</p> 	<p>Build using virtual servers</p> <p>With Lightsail 1-2 minutes</p> 
<p>Register a domain</p> <p>With Route 53 3 minutes</p> 	<p>Connect an IoT device</p> <p>With AWS IoT 5 minutes</p> 	<p>Start migrating to AWS</p> <p>With CloudEndure Migration 1-2 minutes</p> 

▶ See more

Figure 13.3 – Some AWS wizards, documentation, and videos – all very helpful

All of these are simple scenarios that will get you up and running in a couple of minutes, for free. Amazon realizes that first-time users of the cloud are overwhelmed with all the choices, so they try to get your first machine running in a couple of minutes to show you how easy it is.

Let's get you acquainted with the services we are going to use, which we're going to do by using a scenario. We want to migrate a machine that was running in our local KVM installation into Amazon AWS. We are going to go through the whole process step by step, but we first need to understand what we need. The first thing, obviously, is the ability to run virtual machines in the cloud. In the AWS universe, this is EC2 or Amazon Elastic Compute Cloud in full.

EC2

EC2 is one of the few real core services that basically runs everything there is to run in the AWS cloud. It is a scalable computing capacity provider for the whole infrastructure. It enables running different instances or virtual computing environments, using various configurations of storage, memory, CPU, and networking, and it also provides everything else those instances need, including security, storage volumes, zones, IP addresses, and virtual networks. Some of these services are also available separately in case you need more complex scenarios, for example, a lot of different storage options exist, but the core functionality for the instances is provided by EC2.

S3

The full name of this service is actually Amazon Simple Storage Service, hence the name *Amazon S3*. The idea is to give you the ability to store and retrieve any amount of data, anytime you need it, using one or more of the methods offered. The most important concept we are going to use is an *S3 bucket*. A bucket is a logical storage element that enables you to group objects you store. Think of it as a name for a storage container you will later use to store things, whatever those things may be. You can name your buckets however you want, but there is a thing we must point out – the names of buckets have to be *globally unique*. This means that when you name a bucket, it must have a name that is not repeated anywhere else in any of the regions. This makes sure that your bucket will have a unique name, but it also means that trying to create a generic-sounding name such as `bucket1` or `storage` is probably not going to work.

Once you create a bucket, you can upload and download data from it using the web, a CLI, or an API. Since we are talking about a global system, we must also point out that data is stored in the region you specify when creating the bucket, and is kept there unless you specify you want some form of multi-region redundancy. Have that in mind when deploying buckets, since once you start using the data in the bucket, your users or your instances need to get the data, and latency can become a problem. Due to legal and privacy concerns, data never leaves your dedicated region unless you explicitly specify otherwise.

A bucket can store any number of objects, but there is a limit of 100 buckets per account. If that is not enough, you can request (and pay) to have that limit raised to 1,000 buckets.

Also, take a close look at other different options for storing and moving data – there are different types of storage that may or may not fit your needs and budget, such as, for example, S3 Glacier, which offers much cheaper options for storing large amounts of data, but is expensive if you need to get the data out.

IAM

AWS Identity and Access Management (IAM) is the service we need to use since it enables access management and permissions for all the objects and services. In our example, we are going to use it to create policies, users, and roles necessary to accomplish our task.

Other services

There is simply no way to mention all the services AWS offers in simple form. We mentioned only the ones that were necessary and tried to point you in the right direction. It is up to you to try and see what your usage scenario is, and how to configure whatever satisfies your particular needs.

So far, we have explained what AWS is and how complex it can become. We have also mentioned the most commonly used parts of the platform and started explaining what their functions are. We are going to expand on that as we actually migrate a machine from our local environment into AWS. This is going to be our next task.

Preparing and converting virtual machines for AWS

If you search for it on Google, migrating machines from KVM to AWS is easy, and all that is required is to follow the instructions at this link: https://docs.amazonaws.cn/en_us/vm-import/latest/userguide/vm-import-ug.pdf

If you actually try to do it, you will quickly understand that, given *basic* knowledge of the way AWS works, you will not be able to follow the instructions. This is why we choose to do this simple task as an example of using AWS to quickly create a working VM in the cloud.

What do we want to do?

Let's define what we are doing – we decided to migrate one of our machines into the AWS cloud. Right now, our machine is running on our local KVM server, and we want it running on AWS as soon as possible.

The first thing we must emphasize is that there is no live migration option for this. There is no simple tool that you can point to the KVM machine and move it to AWS. We need to do it step by step, and the machine needs to be off. After quickly consulting the documentation, we created a plan. Basically, what we need to do is the following:

1. Stop our virtual machine.
2. Convert the machine to a format that is compatible with the import tool used in AWS.
3. Install the required AWS tools.
4. Create an account that will be able to do the migration.
5. Check whether our tools are working.
6. Create an S3 bucket.
7. Upload the file containing our machine into the bucket.
8. Import the machine to EC2.
9. Wait for the conversion to finish.
10. Prepare the machine to start.
11. Start the machine in the cloud.

So, let's start working on that:

1. A good place to start is by taking a look at our machines on our workstation. We will be migrating the machine named `deploy-1` to test our AWS migration. It's a core installation of CentOS 7 and is running on a host using the same Linux distribution. For that, we obviously need to have privileges:

```
[cloud@workstation ~]$ virsh list
Id      Name                               State
-----
[cloud@workstation ~]$ sudo su
[sudo] password for cloud:
[root@workstation cloud]# virsh list
Id      Name                               State
-----
1       deploy-1                           running
```

Figure 13.4 – Selecting a VM for our migration process

The next thing to do is to stop the machine – we cannot migrate machines that are running since we need to convert the volume that the machine is using in order to make it compatible with the import tool on EC2.

2. The documentation available at <https://docs.aws.amazon.com/vm-import/latest/userguide/vmimport-image-import.html> states that:

"When importing a VM as an image, you can import disks in the following formats: Open Virtualization Archive (OVA), Virtual Machine Disk (VMDK), Virtual Hard Disk (VHD/VHDX), and raw. With some virtualization environments, you would export to Open Virtualization Format (OVF), which typically includes one or more VMDK, VHD, or VHDX files, and then package the files into an OVA file."

In our particular case, we are going to use the `.raw` format, since it is compatible with the import tool, and is fairly simple to convert from the `.qcow2` format KVM uses, into this format.

Once our machine has been stopped, we need to do the conversion. Find the image on disk and use `qemu-img` to do the conversion. The only parameter is the files; the converter understands what it needs to do by detecting the extensions:

```
[root@workstation deploy-1]# ls
centos1.qcow2  deploy-1-cidata.iso  meta-data  user-data
[root@workstation deploy-1]# qemu-img convert centos1.qcow2 deploy1.raw
[root@workstation deploy-1]# ls
centos1.qcow2  deploy-1-cidata.iso  deploy1.raw  meta-data  user-data
[root@workstation deploy-1]# █
```

Figure 13.5 – Converting a qcow2 image to raw image format

We only need to convert the image file, containing the disk image for the system; other data was left out of the installation of the VM. We need to have in mind that we are converting to a format that has no compression so your file size can significantly increase:

```
[root@workstation deploy-1]# ls -al
total 941644
drwxr-xr-x. 2 root root      107 Apr 11 01:43 .
drwx--x--x. 6 root root      158 Jan 13 16:52 ..
-rw-r--r--. 1 root root 43188224 Apr 11 01:27 centos1.qcow2
-rw-r--r--. 1 qemu qemu 374784 Jan 12 18:51 deploy-1-cidata.iso
-rw-r--r--. 1 root root 8589934592 Apr 11 01:44 deploy1.raw
-rw-r--r--. 1 root root      26 Jan 12 16:27 meta-data
-rw-r--r--. 1 root root      629 Jan 12 17:42 user-data
[root@workstation deploy-1]#
```

Figure 13.6 – The conversion process and the corresponding capacity change

We can see that our file increased from 42 MB to 8 GB just because we had to remove the advanced features `qcow2` offers for data storage. The free tier offers only 5 GB of storage, so please make sure to configure the raw image size correspondingly.

Our next obvious step is to upload this image to the cloud since the conversion is done there. Here, you can use different methods, either GUI or CLI (API is also a possibility but is way too complicated for this simple task).

3. AWS has a CLI tool that facilitates working with services. It's a simple command-line tool compatible with most, if not all, the operating systems you can think of:

```
[root@workstation ~]# curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/in % Total % Received % Xferd Average Speed Time Time Time Current
                Dload Upload Total Spent Left Speed
    0   0  0    0    0    0  ---:--:-- 0st100 31.1M 100 31.1M
[root@workstation ~]#
[root@workstation ~]# unzip awscliv2.zip
```

Figure 13.7 – Downloading and uncompressing AWS CLI

We're using `curl` to download a file, and its `-o` option to say what the name of the output file is going to be. Obviously, we need to unzip the ZIP file so that we can use it. The installation process of the tool is also referenced in the documentation. We are talking about a simple download, after which we have to extract the tool. Since there is no installer, the tool will not be in our path, so from now on, we need to reference it by the absolute path.

Before we can use the AWS CLI, we need to configure it. This tool has to know how it is going to connect to the cloud, which user it's going to use, and has to have all the permissions granted in order for the tool to be able to get the data uploaded to AWS, and then imported and converted into the EC2 image. Since we do not have that configured, let's switch to the GUI on AWS and configure the things we need.

Important note
 From now on, if something looks edited in the screenshots, it probably is. To enable things to work seamlessly, AWS has a lot of personal and account data on the screen.

4. We will go into **Identity and Access Management** or IAM, which looks like the following screenshot. Go to **Services | Security**, click on **Identity & Compliance**, and click on **IAM**. This is what you should see:

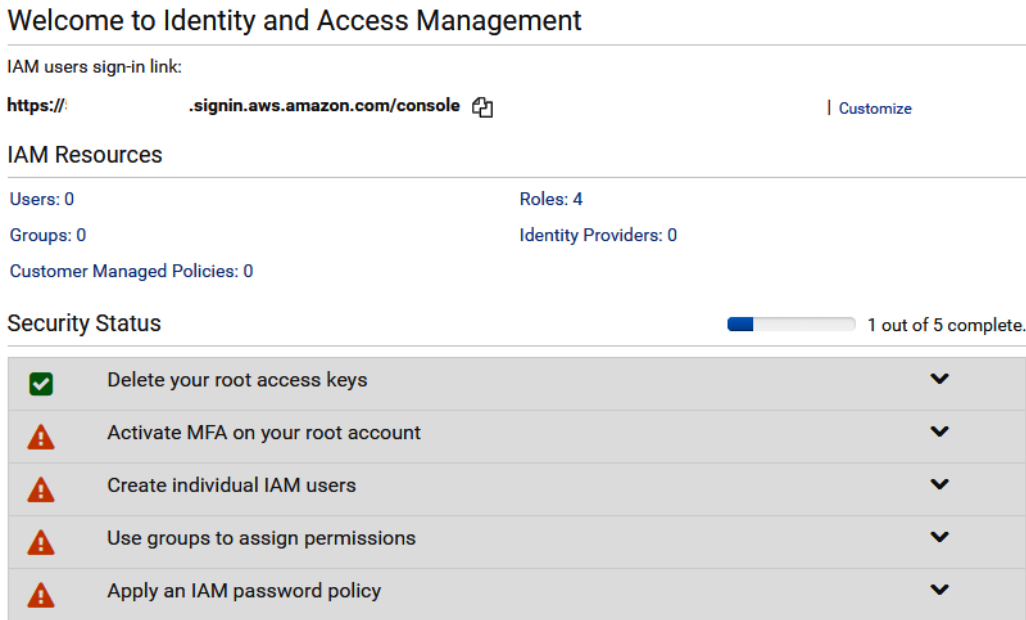


Figure 13.8 – IAM console

We need to choose users on the left side of the screen. This will give us access to the user console. We will create a user named `Administrator`, a group named `administrators`, and apply appropriate permissions to them. Then we are going to join the user to the group. On the first screen, you can choose both options, **Programmatic access** and **AWS Management Console access**. The first one enables you to use the AWS CLI, and the second one enables the user to log in to the management console if we need this account to configure something. We choose only the first one but will add the API key later:

Add user 1 2 3 4 5

Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name*

[+ Add another user](#)

Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)

Access type* **Programmatic access**
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

AWS Management Console access
Enables a **password** that allows users to sign-in to the AWS Management Console.

Figure 13.9 – Configuring user permissions

After clicking on the appropriate option, we can set the initial password for the user. This user will have to change it as soon as they log in:

Add user



Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name*

[+ Add another user](#)

Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)

- Access type*
- Programmatic access**
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, development tools.
 - AWS Management Console access**
Enables a **password** that allows users to sign-in to the AWS Management Console.

- Console password*
- Autogenerated password
 - Custom password

Show password

Figure 13.10 – Setting an initial password to be changed later

We will also create a group for this user. Do that by choosing the appropriate button in the upper part of the screen:

Add user



Set permissions

Add user to group

Copy permissions from existing user

Attach existing policies directly

Add user to an existing group or create a new one. Using groups is a best-practice way to manage user's permissions by job functions. [Learn more](#)

Add user to group

Create group
Refresh

Showing 1 result

Group	Attached policies

Figure 13.11 – Creating a group for our user

We can assign appropriate policies directly to the user, but having policies assigned to groups, and then assigning users to appropriate groups is a better option, saving a lot of time when we need to remove some permissions from users. Once you click the **Create group** button on the left, you will be able to name and create the group. Below the name box are a lot of predefined policies that we can use to configure a strict user policy. We can also create custom policies, but we are not going to do that:

Create group

Create a group and select the policies to be attached to the group. Using groups is a best-practice way to manage users' permissions

Group name

Filter policies

		Policy name ▼	Type
<input type="checkbox"/>	▶	AdministratorAccess	Job function
<input type="checkbox"/>	▶	AlexaForBusinessDeviceSetup	AWS managed
<input type="checkbox"/>	▶	AlexaForBusinessFullAccess	AWS managed
<input type="checkbox"/>	▶	AlexaForBusinessGatewayExecution	AWS managed
<input type="checkbox"/>	▶	AlexaForBusinessPolyDelegatedAccessPolicy	AWS managed

Figure 13.12 – Create a group wizard and policies

For this to work, we are going to create a group with permissions that are way over the top for this task. We are essentially giving the user all the permissions across the cloud. Filter the policies by **AWS managed – job function**:

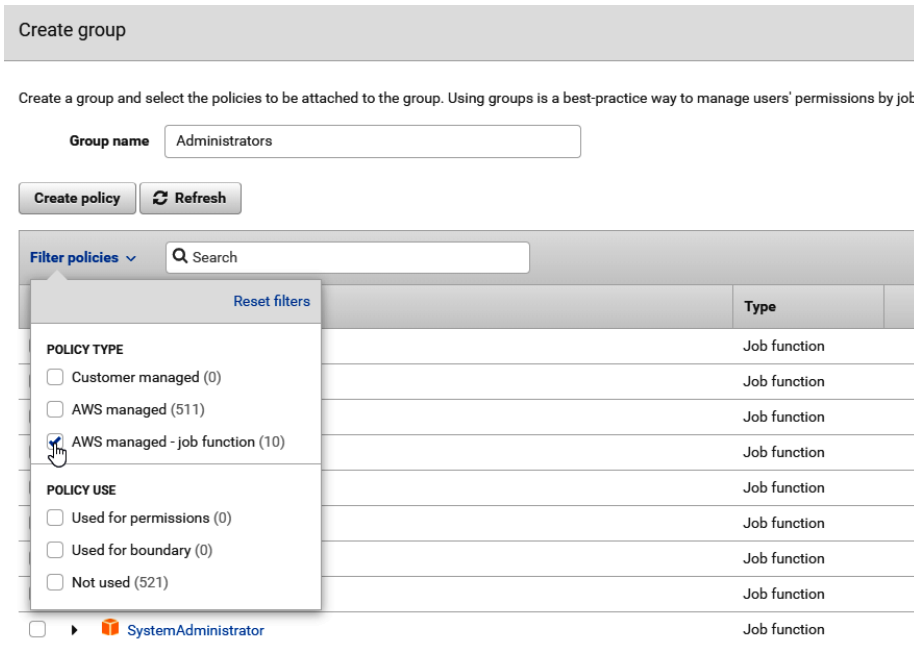


Figure 13.13 – Filtering policies

We are going to use the `AdministratorAccess` policy for this example. This policy is very important, as it allows us to give all available permissions to the `Administrators` group that we're creating. Now select `AdministratorAccess` and click **Create group** in the lower right of the screen:

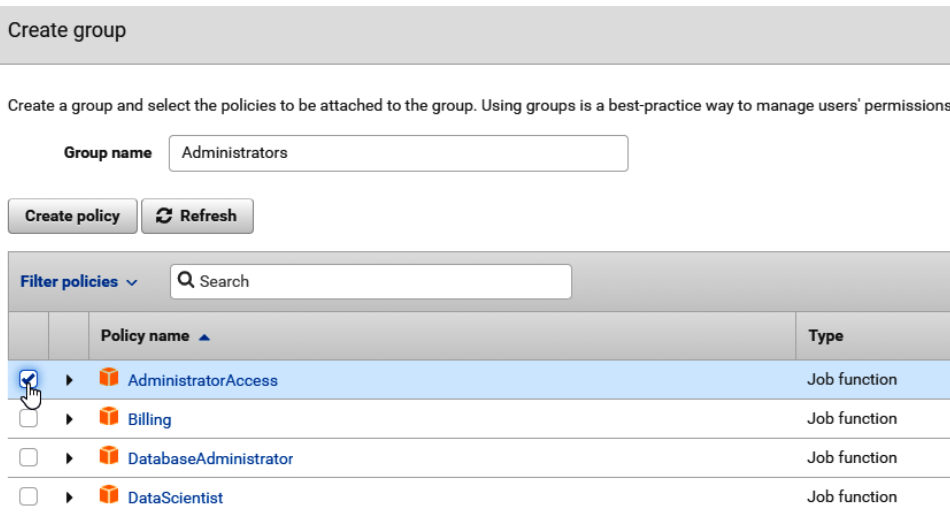


Figure 13.14 – Selecting a policy for our group

The next step is the tags: you can create different attributes or `tags` that can be used later for identity management.

5. Tagging can be done by using almost anything – name, email, job title, or whatever you need. We are going to leave this empty:

Add user



Add tags (optional)

IAM tags are key-value pairs you can add to your user. Tags can include user information, such as an email address, or can be descriptive, such as a job title. You can use the tags to organize, track, or control access for this user. [Learn more](#)

Key	Value (optional)	Remove
<input type="text" value="Add new key"/>	<input type="text"/>	

You can add 50 more tags.

Figure 13.15 – Adding tags

Let's review what we have configured so far. Our user is a member of the group we just created, and they have to reset the password as soon as they log in:

Add user



Review

Review your choices. After you create the user, you can view and download the autogenerated password and access key.

User details

User name	Administrator
AWS access type	AWS Management Console access - with a password
Console password type	Custom
Require password reset	Yes
Permissions boundary	Permissions boundary is not set

Permissions summary

The user shown above will be added to the following groups.

Type	Name
Group	Administrators
Managed policy	IAMUserChangePassword

Tags

No tags were added.

Figure 13.16 – Reviewing the user configuration with group, policy, and tag options

Accept these and add the user. You should be greeted with a reassuring green message box that will give you all the relevant details about what just happened. There is also a direct link to the console for management access, so you can share that with your new user:

Add user



Success

You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.

Users with AWS Management Console access can sign-in at: [https://\[redacted\].signin.aws.amazon.com/console](https://[redacted].signin.aws.amazon.com/console)

[Download .csv](#)

	User	Email login instructions
▼	✔ Administrator	Send email

- ✔ Created user Administrator
- ✔ Attached policy IAMUserChangePassword to user Administrator
- ✔ Added user Administrator to group Administrators
- ✔ Created login profile for user Administrator

Figure 13.17 – User creation was successful

Once the user has been created, we need to enable their `Access Key`. This is a normal concept in using different command-line utilities. It enables us to provide a way for an application to do something as a given user, and not give the application the username or the password. At the same time, we can give each application its own key, so when we want to revoke access, we can simply disable the key.

Click on **Create access key** in the middle of the screen:

Access keys

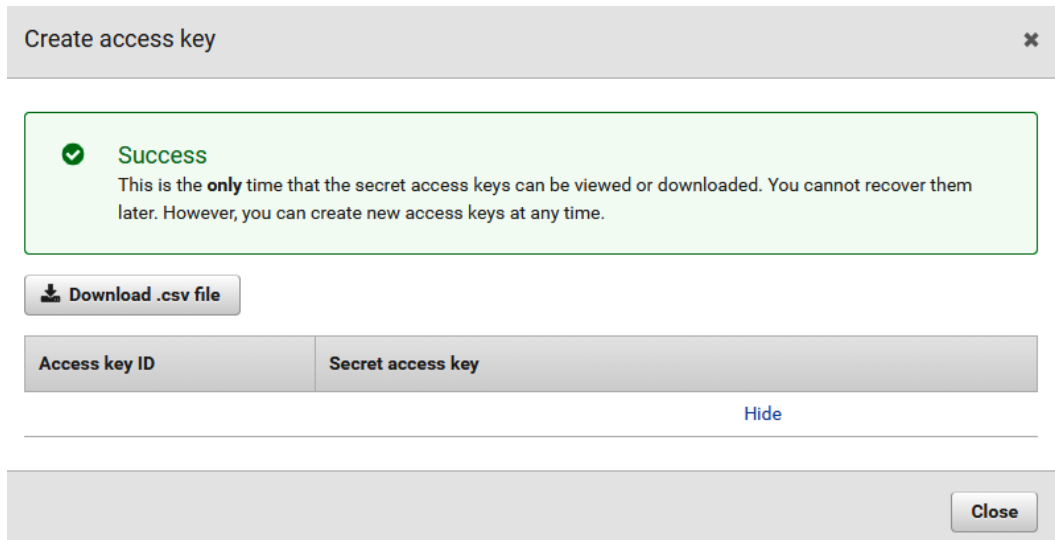
Use access keys to make secure REST or HTTP Query protocol requests to AWS service APIs.

[Create access key](#)

Access key ID	Created	Last used

Figure 13.18 – Creating an access key

A couple of things need to be said about this key. There are two fields – one is the key itself, which is `Access key ID`, the other is the secret part of the key, which is `Secret access key`. In regard to security, this is completely the same as having a username and password for a particular user. You are given only one opportunity to see and download the key, and after that, it is gone. This is because we are dealing with hashed information here, and AWS is *not* storing your keys, but hashes of them. This means there is no way to retrieve a key if you didn't save it. It also means if somebody grabs a key, let's say by reading it off a screenshot, they can identify themselves as the user that has the key assigned. The good thing is that you can create as many keys as you want and revoking them is only a question of deleting them here. So, save the key somewhere safe:



The screenshot shows the 'Create access key' dialog box in the AWS console. At the top, there is a title bar 'Create access key' with a close button (X). Below the title bar is a green success message box with a checkmark icon. The message reads: 'Success. This is the **only** time that the secret access keys can be viewed or downloaded. You cannot recover them later. However, you can create new access keys at any time.' Below the message is a button labeled 'Download .csv file'. Underneath is a table with two columns: 'Access key ID' and 'Secret access key'. The 'Secret access key' column is currently hidden, indicated by a 'Hide' link below the table. At the bottom right of the dialog box is a 'Close' button.

Access key ID	Secret access key

Figure 13.19 – Access key was created successfully

We are finished with the GUI for now. Let's go back and install the AWS CLI:

1. We just need to start the installation script and let it finish its job. This is done by starting the file called `install` in the `aws` directory:

```
inflating: aws/dist/botocore/data/appconfig/2019-10-09/paginators-1.json
creating: aws/dist/botocore/data/events/2015-10-07/
inflating: aws/dist/botocore/data/events/2015-10-07/service-2.json
inflating: aws/dist/botocore/data/events/2015-10-07/examples-1.json
inflating: aws/dist/botocore/data/events/2015-10-07/paginators-1.json
creating: aws/dist/botocore/data/comprehendmedical/2018-10-30/
inflating: aws/dist/botocore/data/comprehendmedical/2018-10-30/service-2.json
inflating: aws/dist/botocore/data/comprehendmedical/2018-10-30/paginators-1.json
root@workstation ~]#
root@workstation ~]# sudo ./aws/install
You can now run: /usr/local/bin/aws --version
root@workstation ~]# aws
ash: aws: command not found...
root@workstation ~]# /usr/local/bin/aws --version
aws-cli/2.0.7 Python/3.7.3 Linux/3.10.0-1062.el7.x86_64 botocore/2.0.0dev11
root@workstation ~]#
root@workstation ~]#
root@workstation ~]#
root@workstation ~]# /usr/local/bin/aws configure
WS Access Key ID [None]:
WS Secret Access Key [None]:
default region name [None]: us-west-2
default output format [None]: table
root@workstation ~]# █
```

Figure 13.20 – Installing AWS CLI

Remember what we said about absolute paths? The `aws` command is not in the user path; we need to call it directly. Use `configure` as a parameter. Then, use the two parts of the key we saved in the previous step. From now on, every command we give using the AWS CLI is interpreted as having been run as the user Administrator that we just created on the cloud.

The next step is to create a bucket on S3. This can be done in one of two ways. We can do it through our newly configured CLI, or we can use the GUI. We are going to take the "pretty" way and use the GUI in order to show how it looks and behaves.

2. Select S3 as the service in the console. There is a button at the top right labeled **Create bucket** – click it. The following screen will appear. Now create a bucket that is going to store your virtual machine in its raw format. In our case, we labeled the bucket `importkvm` but choose a different name. Make sure that you take note of the `region` pull-down menu – this is the AWS location where your resource will be created. Remember that the name has to be unique; if everybody who bought this book tried to use this name, only the first one would succeed. Fun fact: if by the time you read this, we haven't deleted this bucket, nobody will be able to create another with the same name, and only those of you reading this exact sentence will understand why. This wizard is quite big in terms of screen estate and might not fit on a single book page, so let's split it into two parts:

Amazon S3 > Create bucket

Create bucket

General configuration

Bucket name

Bucket name must be unique and must not contain spaces or uppercase letters. [See rules for bucket naming](#)

Region

Figure 13.21 – Wizard for creating an S3 bucket – selecting bucket name and region

The second part of this wizard is related to settings:

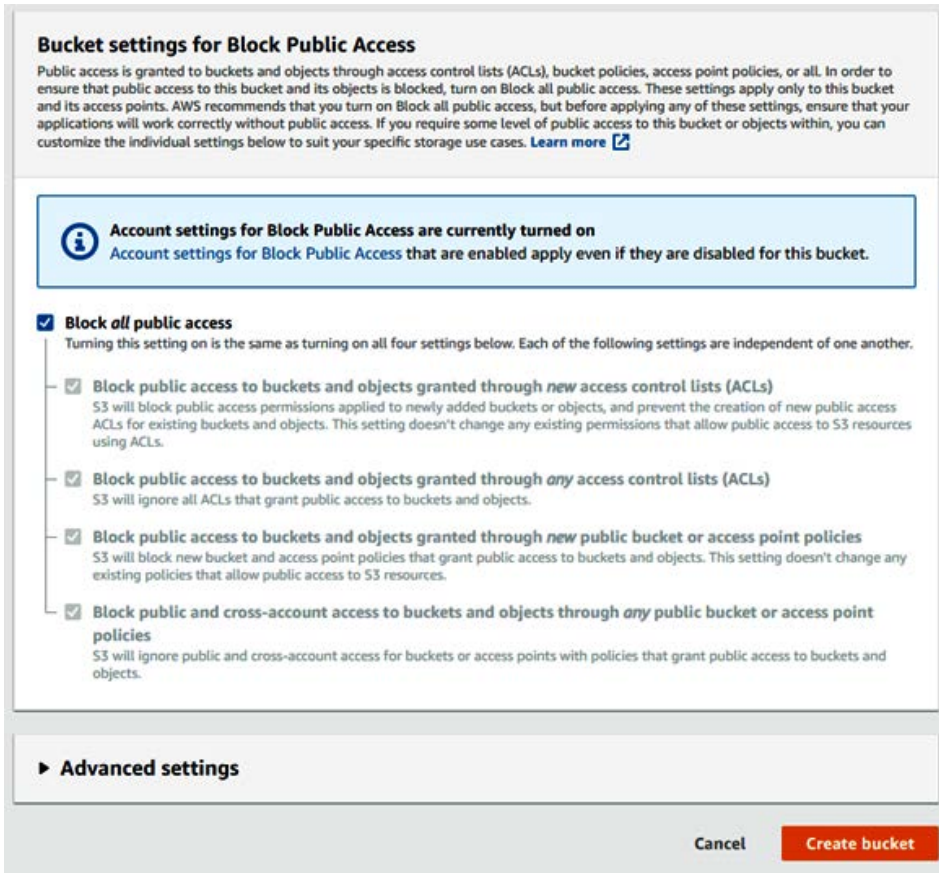


Figure 13.22 – Bucket settings

Do not change the access to public – there is really no need to; nobody but you is ever going to need to access this particular bucket and the file in it. By default, this option is pre-selected, and we should leave it as it is. This should be the end result:

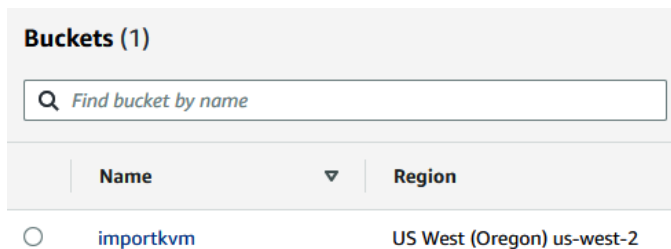


Figure 13.23 – S3 bucket created successfully

OK, having done that, it's time for some waiting on the next command to finish. In the next step, we are going to use our AWS CLI to copy the `.raw` file onto S3.

Important note

Depending on the type of account, from this point on, it is possible that we will have to pay for some of the services that we create since they may overdraw the *free tier* enabled on your account. If you do not enable anything expensive, you should be fine, but always take look at your **Cost management** dashboard, and check that you are still in the black.

Uploading an image to EC2

The next step is to upload an image to EC2 so that we can actually run that image as a virtual machine. Let's start the upload process – this is why we installed the AWS CLI utility in the first place:

1. Use the AWS CLI with the following parameters:

```
[root@workstation deploy-1]# /usr/local/bin/aws s3 cp deploy1.raw s3://importkvm
upload: ./deploy1.raw to s3://importkvm/deploy1.raw
[root@workstation deploy-1]#
```

Figure 13.24 – Using the AWS CLI to copy a virtual machine raw image to an S3 bucket

That's the end result. Since we are talking about 8 GB of data, you will have to wait for some time, depending on your upload speed. The syntax for the AWS CLI is pretty straightforward. You can use most Unix commands that you know, both `ls` and `cp` do their job. The only thing to remember is to give your bucket name in the following format as the destination: `s3://<bucketname>`.

2. After that, we do an `ls` – it will return the bucket names, but we can list their contents by using the bucket name. In this example, you can also see it took us something like 15 minutes to transfer the file from the moment we created the bucket:

```
[root@workstation deploy-1]# /usr/local/bin/aws s3 ls
2020-04-11 01:34:14 importkvm
[root@workstation deploy-1]# /usr/local/bin/aws s3 ls importkvm
2020-04-11 01:48:59 8589934592 deploy1.raw
[root@workstation deploy-1]# █
```

Figure 13.25 – Transferring the file

And now starts the fun part. We need to import the machine into EC2. To do that, we need to do a few things before we will be able to do the conversion. The problem is related to permissions – AWS services are unable to talk to each other by default. Therefore, you have to give explicit permission to each of them to do the importing. In essence, you have to let EC2 talk to S3 and get the file from the bucket.

3. For upload purposes, we will introduce another AWS concept – `.json` files. A lot of things in AWS are stored in `.json` format, including all the settings. Since the GUI is rarely used, this is the quickest way to communicate data and settings, so we must also use it. The first file we need is `trust-policy.json`, which we are using to create a role that will enable the data to be read from the S3 bucket:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "vmie.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "sts:ExternalId": "vmimport"
        }
      }
    }
  ]
}
```

Just create a file with the name `trust-policy.json`, and get the preceding code typed in. Do not change anything. The next one up is the file named `role-policy.json`. This one has some changes that you have to make. Take a closer look inside the file and find the lines where we mention our bucket name (`importkvm`). Delete our name and put the name of your bucket instead:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
```

```
    "Effect": "Allow",
    "Action": [
        "s3:GetBucketLocation",
        "s3:ListBucket",
        "s3:GetObject"
    ],
    "Resource": [
        "arn:aws:s3:::importkvm",
        "arn:aws:s3:::importkvm/*"
    ],
},
{
    "Effect": "Allow",
    "Action": [
        "s3:GetObject",
        "s3:GetBucketLocation",
        "s3:ListBucket",
        "s3:GetBucketAcl",
        "s3:PutObject"
    ],
    "Resource": [
        "arn:aws:s3:::importkvm",
        "arn:aws:s3:::importkvm/*"
    ],
},
{
    "Effect": "Allow",
    "Action": [
        "ec2:ModifySnapshotAttribute",
        "ec2:CopySnapshot",
        "ec2:RegisterImage",
        "ec2:Describe*"
    ],
    "Resource": "*"
}
]
```

Now it's time to put it all together and finally upload our virtual machine to AWS.

4. Execute these two commands, disregard whatever happens in the formatting – both of them are one-liners, and the filename is the last part of the command:

```
/usr/local/bin/aws iam create-role --role-name vmimport
--assume-role-policy-document file://trust-policy.json
/usr/local/bin/aws iam put-role-policy --role-name
vmimport --policy-name vmimport --policy-document file://
role-policy.json
```

You should get a result something like this:

CreateRole					
Role					
Arn	RoleName	CreateDate	Path	RoleId	RoleName
arn:aws:iam:::role/vmimport	vmimport	2020-04-11T10:05:08+00:00	/	A.....JPF4Z	vmimport
AssumeRolePolicyDocument					
Version	2012-10-17				
Statement					
Action	Effect				
sts:AssumeRole	Allow				
Condition					
StringEquals					
sts:Externalid	vmimport				
Principal					
Service	vmie.amazonaws.com				

Figure 13.26 – Result of createrole

This confirms that the role was given the permissions it needs. The second command should not return any output.

We're almost done. The last step is to create yet another .json file that will describe to EC2 what we are actually importing and what to do with it.

5. The file we're creating needs to look like this:

```
[
  {
    "Description": "Test deployment",
    "Format": "raw",
```

```

    "Userbucket": {
      "S3Bucket": "importkvm",
      "S3Key": "deploy1.raw"
    }
  ]

```

As you can see, there is nothing special in the file, but when you create your own version, pay attention to use your name for the bucket and the disk image that is stored inside the bucket. Name the file whatever you want, and use that name to call the import process:

```

[root@workstation deploy-1]# /usr/local/bin/aws ec2 import-image --description "Deploy 1" --disk-containers "file://deploy.json"

```

ImportImage				
Description	ImportTaskId	Progress	Status	StatusMessage
Deploy 1	import-ami-0954ba7ec30026b59	2	active	pending

SnapshotDetails	
DiskImageSize	Format
0_0	RAW

UserBucket	
S3Bucket	S3Key
importkvm	deploy1.raw

Figure 13.27 – Final step – virtual machine deployment to AWS

Now you wait for the process to finish. What happens in this step is both the import and conversion of the image and the operating system you uploaded. AWS is not going to run your image as is; the system is going to change quite a few things to make sure your image can run on the infrastructure. Some users will also receive some changes, but more on that later.

The task will run in the background, and will not notify you when it completes; it is up to you to check on it. Luckily, there is a command that can be used in the AWS CLI called `describe-import-image-tasks` and this is the output:

```
DescribeImportImageTasks
+-----+-----+-----+-----+-----+-----+-----+
| Architecture | Description | ImageId | ImportTaskId | LicenseType | Platform | Status |
+-----+-----+-----+-----+-----+-----+-----+
| x86_64 | Deploy 1 | ami-06487af0f1c31c829 | import-ami-0954ba7ec30026b59 | BYOL | Linux | completed |
+-----+-----+-----+-----+-----+-----+-----+
SnapshotDetails
+-----+-----+-----+-----+-----+-----+-----+
| Description | DeviceName | DiskImageSize | Format | SnapshotId | Status |
+-----+-----+-----+-----+-----+-----+-----+
| Test deployment | /dev/sda1 | 8589934592.0 | RAW | snap-0e3cce75ff798ad46 | completed |
+-----+-----+-----+-----+-----+-----+-----+
UserBucket
+-----+-----+-----+-----+-----+-----+-----+
| S3Bucket | S3Key |
+-----+-----+-----+-----+-----+-----+-----+
| importkvm | deploy1.raw |
+-----+-----+-----+-----+-----+-----+-----+
[root@workstation deploy-1]#
```

Figure 13.28 – Checking the status of our upload process

What this means is that we successfully imported our machine. Great! But the machine is still not running. Now it has become something called an **Amazon Machine Image (AMI)**. Let's check how to use that:

1. Go to your EC2 console. You should be able to find the image under **AMIs** on the left side:

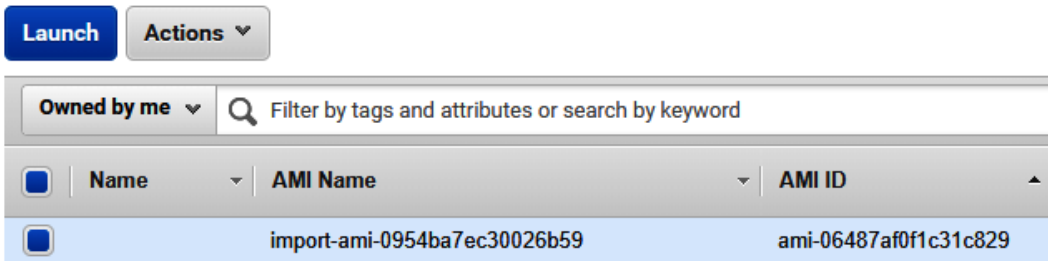


Figure 13.29 – Our AMI has been uploaded successfully and we can see it in the EC2 console

Now click the big blue **Launch** button. There are a couple of steps you need to finish before your instance is running, but we are almost there. First, you need to choose your instance type. This means choosing what configuration fits your needs, according to how much of everything (CPU, memory, and storage) you need.

- If you are using a region that is not overcrowded, you should be able to spin a *free tier* instance type that is usually called `t2.micro` and is clearly marked. In your free part of the account, you have enough processing credits to enable you to run this machine completely free:

Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances

Filter by: All instance types Current generation [Show/Hide Columns](#)

Currently selected: t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, EBS only)

	Family	Type
<input type="checkbox"/>	General purpose	t2.nano
<input checked="" type="checkbox"/>	General purpose	t2.micro Free tier eligible
<input type="checkbox"/>	General purpose	t2.small
<input type="checkbox"/>	General purpose	t2.medium
<input type="checkbox"/>	General purpose	t2.large
<input type="checkbox"/>	General purpose	t2.xlarge
<input type="checkbox"/>	General purpose	t2.2xlarge
<input type="checkbox"/>	General purpose	t3a.nano
<input type="checkbox"/>	General purpose	t3a.micro

Figure 13.30 – Selecting an instance type

And now for some security. Amazon changed your machine and has implemented passwordless login to the administrator account using a key pair. Since we don't have a key yet, we will also need to create the key pair.

- EC2 is going to put this key into the appropriate accounts on the machine you are just creating (all of them), so you can log in without using the password. A key pair is generated if you choose to do so, but Amazon will not store it – you have to do that:

Select an existing key pair or create a new key pair
✕

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. [Learn more about removing existing key pairs from a public AMI.](#)

Create a new key pair
▼

Key pair name

deploy1key

Download Key Pair

...

You have to download the **private key file** (*.pem file) before you can continue. **Store it in a secure and accessible location.** You will not be able to download the file again after it's created.

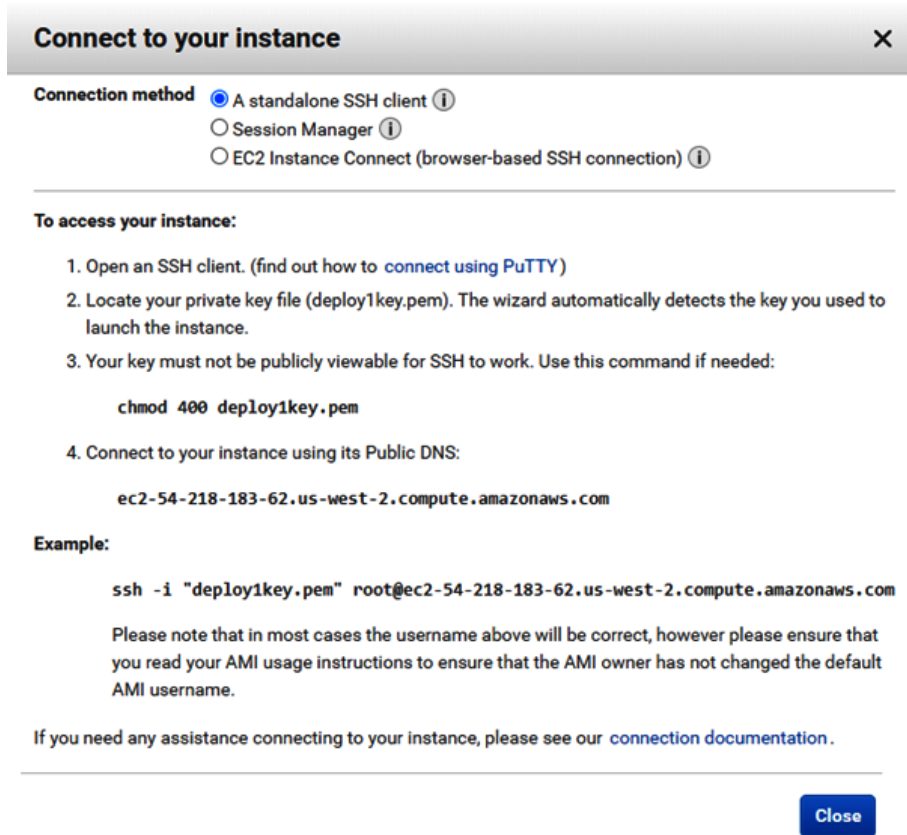
Cancel

Launch Instances

Figure 13.31 – Selecting an existing key or creating a new one

That's it, your VM should now take a couple of minutes to launch. Just wait for the confirmation window. Once it is ready, connect to it using the context menu. You will get to the list of instances by clicking **View Instances** at the bottom right.

To connect, you need to use the key pair provided to you, and you need an `ssh` client. Alternatively, you can use the embedded `ssh` that AWS provides. In any case, you need the outside address of the machine, and AWS also provides that, along with simple instructions:



Connect to your instance ✕

Connection method A standalone SSH client ⓘ
 Session Manager ⓘ
 EC2 Instance Connect (browser-based SSH connection) ⓘ

To access your instance:

1. Open an SSH client. (find out how to [connect using PuTTY](#))
2. Locate your private key file (deploy1key.pem). The wizard automatically detects the key you used to launch the instance.
3. Your key must not be publicly viewable for SSH to work. Use this command if needed:


```
chmod 400 deploy1key.pem
```
4. Connect to your instance using its Public DNS:


```
ec2-54-218-183-62.us-west-2.compute.amazonaws.com
```

Example:

```
ssh -i "deploy1key.pem" root@ec2-54-218-183-62.us-west-2.compute.amazonaws.com
```

Please note that in most cases the username above will be correct, however please ensure that you read your AMI usage instructions to ensure that the AMI owner has not changed the default AMI username.

If you need any assistance connecting to your instance, please see our [connection documentation](#).

[Close](#)

Figure 13.32 – Connect to your instance instructions

So, going back to our workstation, we can use the `ssh` command mentioned in the previous screenshot to connect to our newly started instance:

```
[root@workstation ~]# ssh -i deploy1key.pem centos@ec2-54-218-183-62.us-west-2.compute.amazonaws.com
The authenticity of host 'ec2-54-218-183-62.us-west-2.compute.amazonaws.com (54.218.183.62)' can't be established.
ECDSA key fingerprint is SHA256:WRucACTXNTbAlvwfuynPEgqo6FjJoLas6bLKymPJrEQ.
ECDSA key fingerprint is MD5:44:b5:04:e8:87:ad:24:19:01:a3:e9:8d:a7:0e:42:34.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-54-218-183-62.us-west-2.compute.amazonaws.com,54.218.183.62' (ECDSA) to the list of known hosts.
Last login: Sat Apr 11 11:03:17 2020 from
[centos@ip-172-31-21-125 ~]$
```

Figure 13.33 – Connecting to our instance via SSH

That's it. You have successfully connected to your machine. You can even keep it running. But be aware, if you have accounts or services that are on by default or have no password – you have, after all, pulled a VM out of your safe, home sandbox and stuck it on the big, bad internet. And one last thing: after you have your VM running, delete the file in the bucket to save you some resources (and money). After conversion, this file is no longer needed.

The next topic on our list is how to extend our local cloud environments into hybrid cloud environments by using an application called Eucalyptus. This is a hugely popular process that a lot of enterprise companies go through as they scale their infrastructure beyond their local infrastructure. Also, this offers benefits in terms of scalability when needed – for example, when a company needs to scale its testing environment so an application that its employees are working on can be load-tested. Let's see how it's done via Eucalyptus and AWS.

Building hybrid KVM clouds with Eucalyptus

Eucalyptus is a strange beast, and by that, we do not mean the plant. Created as a project to bridge the gap between private cloud services and AWS, Eucalyptus tries to recreate almost all AWS functionalities in a local environment. Running it is almost like having a small local cloud that is compatible with AWS, and that in turn uses almost the same commands as AWS. It even uses the same names for things as AWS does, so it works with buckets and all of that. This is on purpose, and with consent from Amazon. Having an environment like this is a great thing for everybody since it creates a safe space for developers and companies to deploy and test their instances.

Eucalyptus consists of several parts:

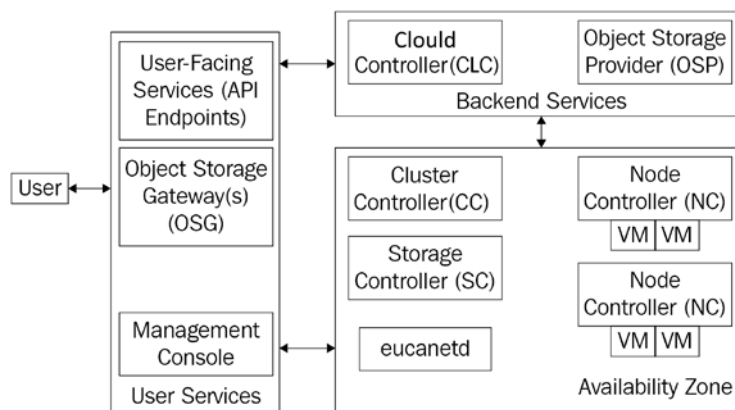


Figure 13.34 – Eucalyptus architecture (<http://eucalyptus.cloud>, official documentation)

As can be seen from the diagram, Eucalyptus is highly scalable.

An **availability zone** is one segment that can hold multiple nodes controlled by a cluster controller. **Zones** are then combined into the cloud itself, and this is controlled by the **Cloud Controller**. Connected to all this is the user services part that enables interaction between the user and the whole Eucalyptus stack.

All in all, Eucalyptus uses five components that are sometimes referred to by their names from the diagram, and sometimes by their project names, much like OpenStack does:

- **Cloud Controller (CLC)** is the central point of the system. It provides both the EC2 and the web interfaces and routes every task to itself. It is there to provide scheduling, allocation of resources, and accounting. There is one of these per cloud.
- **Cluster Controller (CC)** is the part that manages each individual node and controls VMs and their execution. One is running in each availability zone.
- **Storage Controller (SC)** is there to provide block-level storage, and to provide support for instances and snapshots but within the cluster. It is similar to EBS storage from AWS.
- **Node Controller (NC)** hosts instances and their endpoints. One is running for each node.
- **euca-netd** is a service Eucalyptus uses to manage cloud networking, as we are talking about extending your local networks to the AWS cloud, at the end of the day.

When you get to know Eucalyptus, you'll notice that it has a vast array of capabilities. It can do the following:

- Work with volumes, instances, key pairs, snapshots, buckets, images, network objects, tags, and IAM.
- Work with load balancers.
- Work with AWS as an AWS integration tool.

These are just some of the features worth mentioning at the start of your Eucalyptus journey. Eucalyptus has an additional command-line interface called **Euca2ools**, available as a package for all the major Linux distributions. Euca2ools is an additional tool that provides full API and CLI compatibility between AWS and Eucalyptus. This means that you can use a single tool to manage both and to perform *hybrid cloud* migrations. The tool is written in Python, so it is more or less platform-independent. If you want to learn more about this interface, make sure that you visit <https://wiki.debian.org/euca2ools>.

How do you install it?

Installing Eucalyptus is easy, if you are installing a test machine and *following the instructions*, as we'll describe in the last chapter of the book, *Chapter 16, Troubleshooting Guideline for the KVM Platform*, which deals with KVM troubleshooting. We are going to do just that – install a single machine that will hold all the nodes and part of the whole cloud. This is, of course, not even close to what is needed for a production environment, so on the Eucalyptus website, there are separate guides for this single-machine-does-all situation, and for installing production-level clouds. Make sure that you check the following link: <https://docs.eucalyptus.cloud/eucalyptus/4.4.5/install-guide-4.4.5.pdf>.

Installation is simple – just provide a minimally installed CentOS 7 system that has at least 120 GB of disk space and 16 GB of RAM. These are the minimums. If you go below them, you will have two kinds of problems:

- If you try to install on a machine that has less than 16 GB of RAM, the installation will probably fail.
- The installation will, however, succeed on a machine with a smaller disk size than the minimum recommended, but you will almost immediately run out of disk space as soon as you start getting the deployment images installed.

For production, everything changes – the minimums are 160 GB for the storage, or 500 GB of storage for nodes that are going to run Walrus and SC services. Nodes must run on bare metal; nested virtualization is not supported. Or, to be more precise, it will work but will negate any positive effect that the cloud can provide.

Having said all that, we have another point to make before you start installing – check for the availability of a new version, and have in mind that it is quite possible that there is a newer release than the one that we are working on in this book.

Important note

At the time of writing, the current version was 4.4.5, with version 5 being actively worked on and close to being released.

Having installed your base operating system – and it has to be a core system without a GUI, it's time to do the actual Eucalyptus installation. The whole system is installed using **FastStart**, so the only thing we have to do is to run the installer from the internet. The link is helpfully given on the front page of the following URL for the project – <https://eucalyptus.cloud>.

There are some prerequisites for successful Eucalyptus installation:

- You have to be connected to the internet. There is no way to do a local installation this way, because everything is downloaded on the fly.
- You also have to have some IP addresses available for the system to use when installed. The minimum is 10, and they will get installed along with the cloud. The installer will ask for the range and will try to do everything without intervention.
- The only other prerequisites are a working DNS and some time.

Let's start the installation by using the following command:

```
# bash <(curl -Ls https://eucalyptus.cloud/install)
```

The installation looks strange if you're seeing it for the first time. It kind of reminds us of some text-based games and services that we used in the 1990s (MUD, IRC):

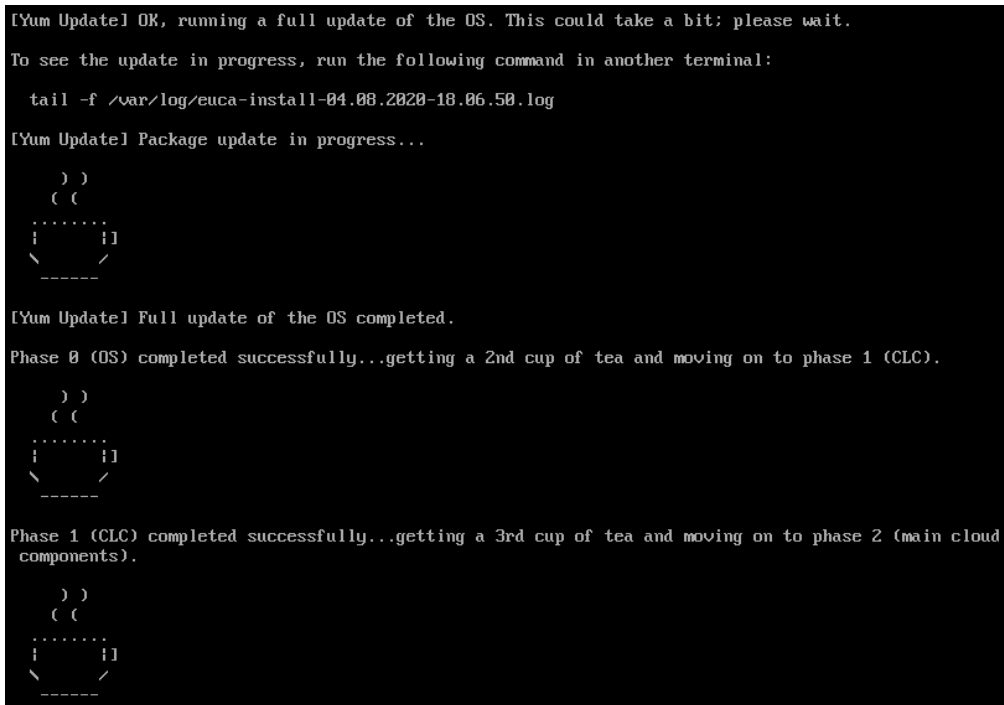


Figure 13.35 – Eucalyptus text-mode installation

The information on the screen will tell you which log to follow if you want to see what is actually happening; otherwise, you can look at the installer and wait for the tea on the screen to get cold. In all honesty, on a decent machine, the installation will probably take around 15 minutes, or 10 minutes more if you install all the packages.

Once installed, Eucalyptus will provide you with a default set of credentials:

- **Account name:** eucalyptus
- **Username:** admin
- **Password:** password

Important note

In the event that the current installer breaks, the solutions to the subsequent problems are in the CiA video here: <video_URL>. There are known bugs, and may or may not be solved before this book hits the stores. Make sure that you check <https://eucalyptus.cloud> and documentation before installation.

The information is case sensitive. Having finished the installation, you can connect to the machine using a web browser and log in. The IP address you are going to use is the IP address of the machine you just installed:

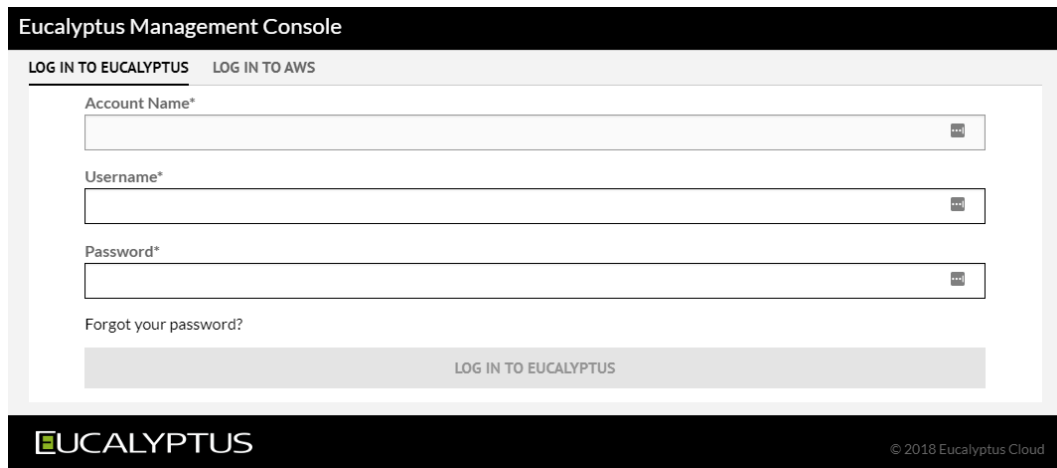


Figure 13.36 – Eucalyptus login screen

Once the system has finished installing, on the console of the newly installed system, you are going to be met with an instruction that will say to run the master tutorial contained on the system. The tutorial itself is a great way to get to know how the system looks, what the key concepts are, and how to use the command line. The only problem you may have is that the tutorial is a set of scripts that have some information hardcoded. One of the things you will notice straight away is that the links to the cloud versions of the image templates will not work unless you fix them – the links point to expired addresses. This is easy to solve but will catch you off guard.

On the other hand, by the time you read this, maybe the problem will be fixed. The tutorial on how to do this and all of its parts is offered in plain text mode on the machine Eucalyptus is running on. It's not available in the GUI:

```

root@euca tutorials]# ./master-tutorial.sh

****

Welcome to the Getting Started Tutorial. We will walk you through
some of the key concepts of managing your new Eucalyptus cloud.
It is strongly recommended for first-time users of Eucalyptus.

Would you like to walk through the Getting Started tutorial? [Y/n]

```

Figure 13.37 – Starting a text-mode Eucalyptus master tutorial

The tutorial is extremely rudimentary in its appearance, but we liked it because it gave us a short but important overview of everything that Eucalyptus offers:

```

Remember: when using Eucalyptus, you must "log in".
When using euca2ools, the way to "log in" is to use
the euca2ools configuration credentials file located
under /root/.euca. By default, Faststart sets this
configuration file up for you. Once this has been
set up, with each euca2ools command, the
"--region" option must be used. For FastStart,
the region option will contain the value
"admin@192.168.5.48.nip.io". For example:

euca-describe-availability-zones --region admin@192.168.5.48.nip.io

To learn more about using euca2ools configuration file, please refer to
the Euca2ools Guide section entitled "Working with Euca2ools Configuration Files":
https://docs.eucalyptus.com/eucalyptus/4.4.2/index.html#shared/euca2ools\_working\_with\_config\_files.html

Hit Enter to continue.

The euca2ools command for listing images is euca-describe-images.
If you have ever worked with Amazon Web Services, you will
notice that the command, and the output from the command, is
nearly identical to the comparable AWS command; this is by design.
Press Enter to run euca-describe-images --region admin@192.168.5.48.nip.io now.

+ euca-describe-images --region admin@192.168.5.48.nip.io
IMAGE emi-0142c1c3 default/default.img.manifest.xml 000020543277 available public x86_64 machine
vm

Now let's review some of the key output of that command:

emi-0142c1c3 is the image ID, which is used
to refer to the image by most other commands.

default/default.img.manifest.xml is the image path.

public is the permission for this image. Images that
are accessible to all users of this cloud are marked public; images that can
only be run by the owner of the image are marked private.

To learn more about the euca-describe-images command, check out the documentaion:
http://docs.hpcloud.com/eucalyptus/4.2.0/#euca2ools-guide/euca-describe-images.html

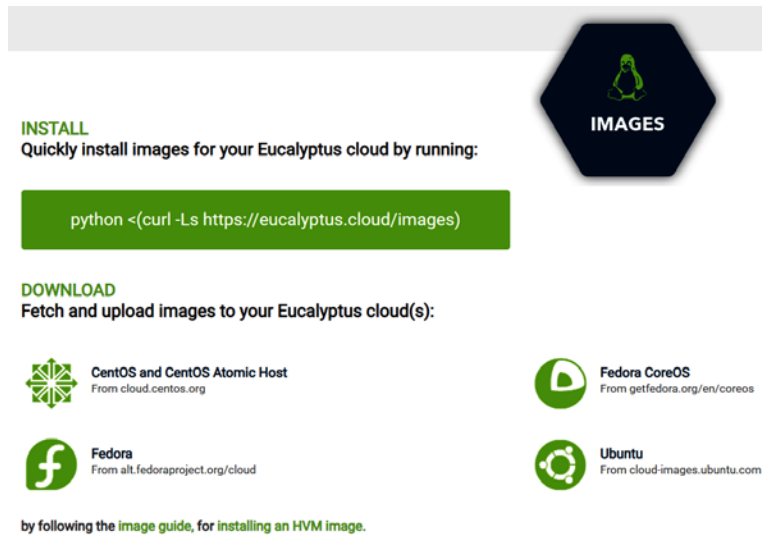
Installing Images

Continue with Installing Images Tutorial? (Y/n)

```

Figure 13.38 – Using the master tutorial to learn how to configure Eucalyptus





As you can see, everything is explained in detail, so you can really learn key concepts in a short amount of time. Go through the tutorial – it is well worth it. Another thing you can do from the command line as soon as you start up the system is to download a couple of new template images. The script for this is also started from the web, and is written in big letters on the official site, literally on the landing page located at the following URL (make sure that you scroll down a bit) – <https://www.eucalyptus.cloud/>:



INSTALL
Quickly install images for your Eucalyptus cloud by running:

```
python <(curl -Ls https://eucalyptus.cloud/images)>
```

DOWNLOAD
Fetch and upload images to your Eucalyptus cloud(s):

-  **CentOS and CentOS Atomic Host**
From cloud.centos.org
-  **Fedora CoreOS**
From getfedora.org/en/coreos
-  **Fedora**
From alt.fedoraproject.org/cloud
-  **Ubuntu**
From cloud-images.ubuntu.com

by following the [image guide](#), for installing an HVM image.

Figure 13.39 – Downloading images to our Eucalyptus cloud

Copy-paste this into the root prompt and, shortly, there will be a menu that will enable you to download images you may use. This is one of the simplest and most bullet-proof installations of templates we have ever seen, short of them being included in the initial download:

```
Complete!
Select an image Id from the following table:

ID   Format  Updates  Login   Description
1    qcow2   yes      ubuntu  Ubuntu 16.04/Xenial
2    qcow2   yes      ubuntu  Ubuntu 18.04/Bionic
3    qcow2   yes      ubuntu  Ubuntu 19.10/Edoan
4    raw     no       fedora  Fedora 31
5    vmdk    no       core    Fedora CoreOS Stable
6    raw     no       centos  CentOS 7
7    qcow2   no       centos  CentOS Atomic Host 7
8    qcow2   no       centos  CentOS 8
9    qcow2   yes      core    Flatcar Container Linux Stable

Enter the image ID you would like to install:
```

Figure 13.40 – Simple menu asking us to select which image we want to install

Choose one at a time, and they will get included in the image list.

Now let's switch to the web interface to see how it works. Log in using the credentials written above. You will be greeted with a well-designed dashboard. On the right, there are groups of functionalities that are most commonly used. The left part is reserved for the menu that holds links to all the services. The menu will autohide as soon as you move your mouse away from it, leaving only the most essential icons:

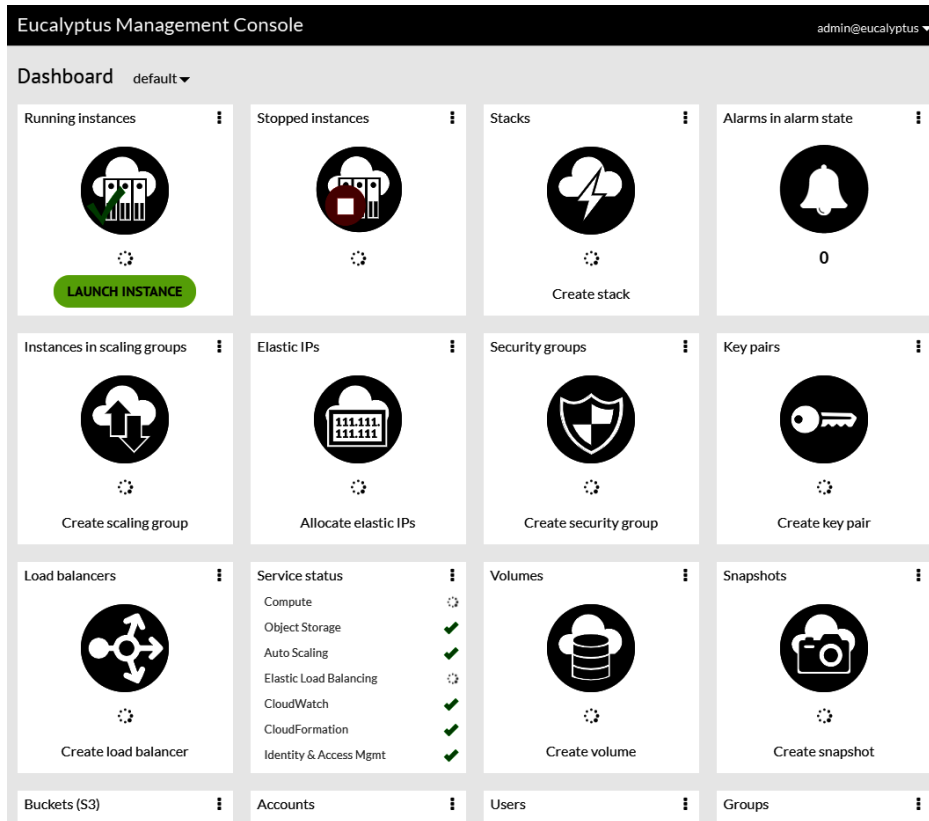


Figure 13.41 – Eucalyptus management console

We already discussed most of the things on this page – just check the content in this chapter related to AWS and you'll be in very familiar territory. Let's try using this console. We are going to launch a new instance, just to get a feel for how Eucalyptus works:

1. In the left part of the stack of services, there is an inviting green button labeled **Launch instance** – click on it. A list of the images that are available on the system will appear. We already used the script to grab some cloud images, so we have something to choose from:

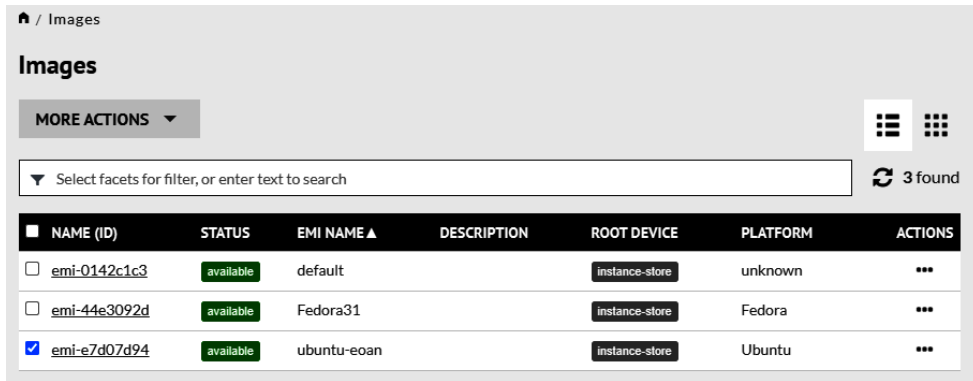


Figure 13.42 – Selecting an image to run in the Eucalyptus cloud

We chose to run Ubuntu from the cloud image. Choose **Launch** from the drop-down menu after you have selected the image you want. A new window opens, permitting you to create your virtual machine. In the dropdown or the instance type, we chose a machine that looked powerful enough to run our Ubuntu, but basically, any instance with over 1 GB of RAM will do fine. There is not much to change since we are preparing just one instance:

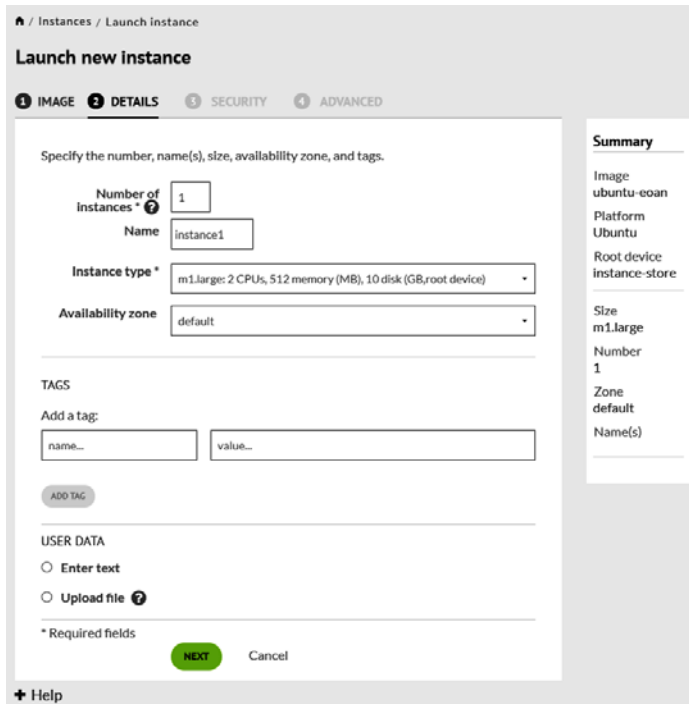


Figure 13.43 – Launch a new instance wizard

The next configuration screen is related to security.

2. We have a choice of using the default key pair that was created on the Eucalyptus cloud or creating a new one. Only the public part of the key is stored in Eucalyptus, so we can use this key pair for authentication only if we downloaded the keys when we installed them. The process of creating keys is completely identical to the one used for AWS:

Home / Instances / Launch instance

Launch new instance

1 IMAGE 2 DETAILS 3 SECURITY 4 ADVANCED

Specify key pair and security group.

Key name *

Or: Create key pair

Security group *

Or: Create security group

+ Rules for default

Specify an IAM role if you would like to give this instance special access privileges.

Role

* Required fields

LAUNCH INSTANCE Cancel

Or: Select advanced options

+ Help

Summary

Image
ubuntu-eoan

Platform
Ubuntu

Root device
instance-store

Size
m1.large

Number
1

Zone
no preference

Name(s)

Network

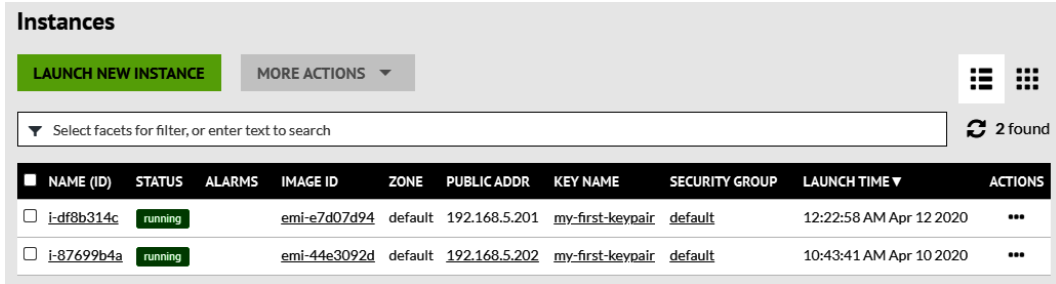
Subnet

Key
my-first-keypair

Security group
default

Figure 13.44 – Security configuration – selecting keys and an IAM role

After clicking the **LAUNCH INSTANCE** button, your machine should boot. For testing purposes, we already started another machine earlier, so right now we have two of them running:



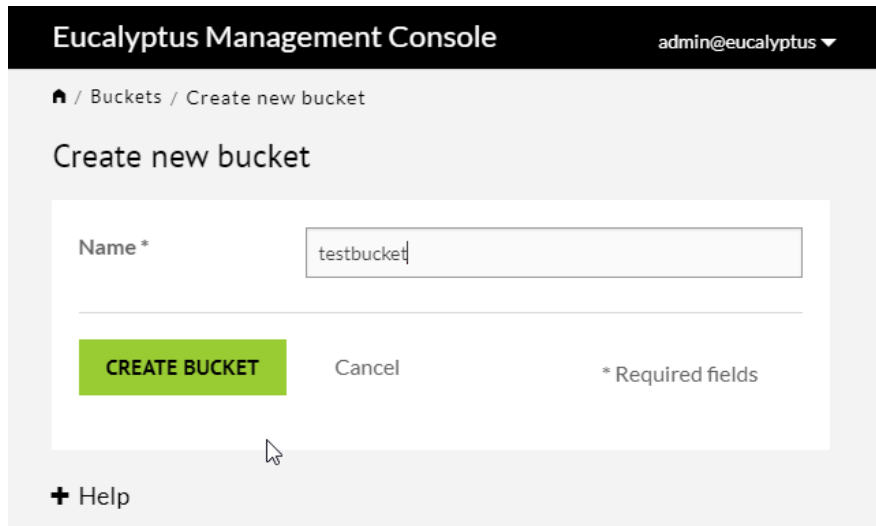
The screenshot shows the 'Instances' section of the Eucalyptus Management Console. At the top, there is a green 'LAUNCH NEW INSTANCE' button and a 'MORE ACTIONS' dropdown. Below this is a search bar with the text 'Select facets for filter, or enter text to search' and a refresh icon with '2 found' next to it. The main content is a table with the following columns: NAME (ID), STATUS, ALARMS, IMAGE ID, ZONE, PUBLIC ADDR, KEY NAME, SECURITY GROUP, LAUNCH TIME, and ACTIONS. Two instances are listed, both with a 'running' status.

NAME (ID)	STATUS	ALARMS	IMAGE ID	ZONE	PUBLIC ADDR	KEY NAME	SECURITY GROUP	LAUNCH TIME	ACTIONS
<input type="checkbox"/> i-df8b314c	running		emi-e7d07d94	default	192.168.5.201	my-first-keypair	default	12:22:58 AM Apr 12 2020	...
<input type="checkbox"/> i-87699b4a	running		emi-44e3092d	default	192.168.5.202	my-first-keypair	default	10:43:41 AM Apr 10 2020	...

Figure 13.45 – A couple of launched instances in the Eucalyptus cloud

The next step is trying to create a storage bucket.

3. Creating a storage bucket is easy, and looks very similar to what AWS enables you to do since Eucalyptus tries to be as similar to AWS as possible:



The screenshot shows the 'Eucalyptus Management Console' interface. The top navigation bar includes the title 'Eucalyptus Management Console' and the user 'admin@eucalyptus'. The breadcrumb trail is 'Buckets / Create new bucket'. The main heading is 'Create new bucket'. Below this is a form with a 'Name *' field containing the text 'testbucket'. At the bottom of the form are two buttons: 'CREATE BUCKET' (green) and 'Cancel'. A note '* Required fields' is visible to the right of the buttons. A '+ Help' link is located at the bottom left of the form area.

Figure 13.46 – Creating a bucket

Since Eucalyptus is not as complex as AWS, especially in regard to policy and security, the security tab for the bucket is smaller, but has some very powerful tools, as you can see in the following screenshot:

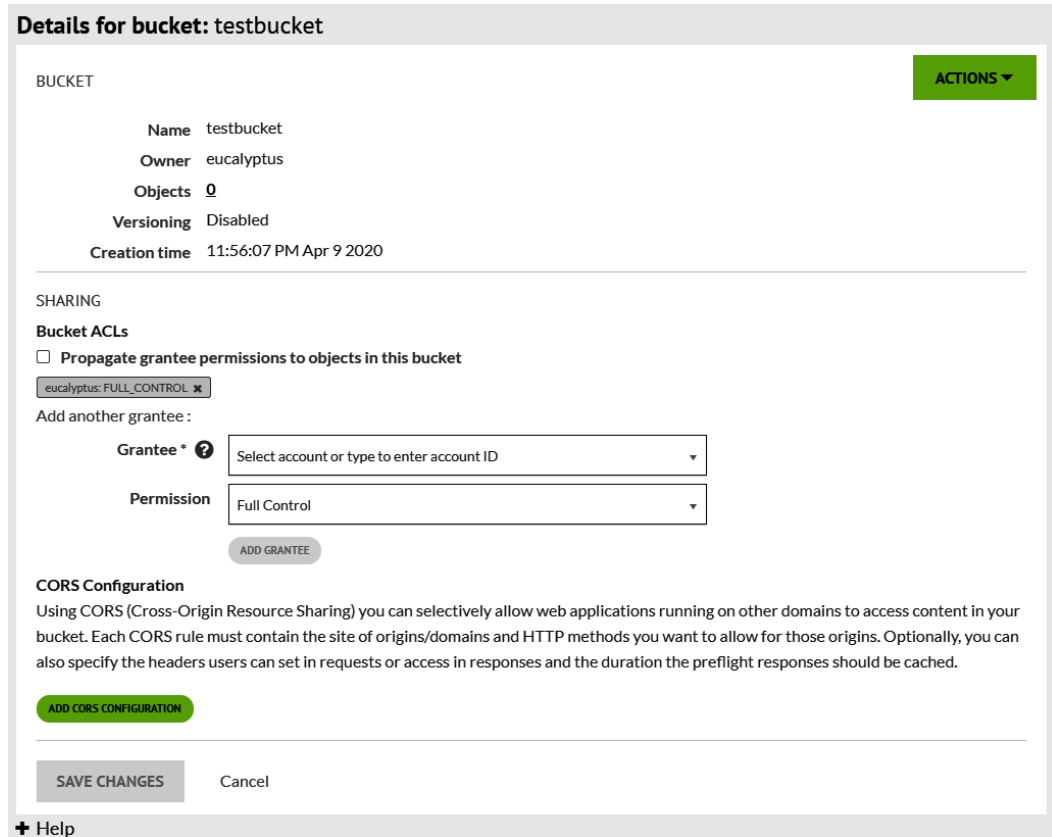


Figure 13.47 – Bucket security configuration

Now that we have installed, configured, and used Eucalyptus, it's time to move on to the next topic of our chapter, which is scaling out our Eucalyptus-based cloud to AWS.

Using Eucalyptus for AWS control

Do you remember the initial screen that showed you the login credentials, and we mentioned that you can also log in to AWS? Log out of your Eucalyptus console and get to the login screen. Click on **LOG IN TO AWS** this time:

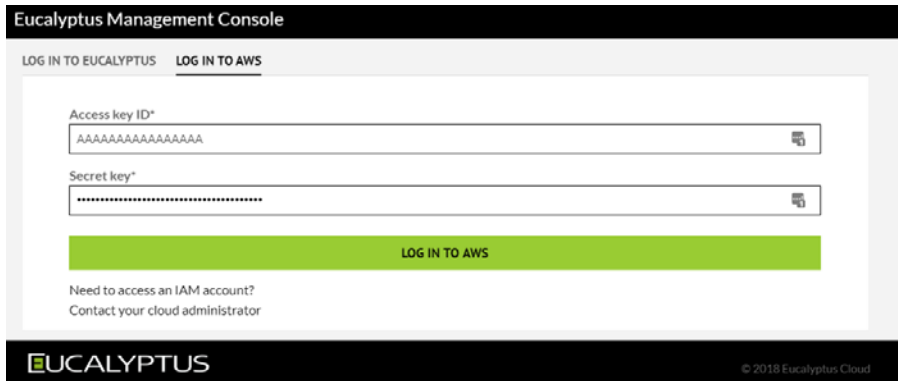


Figure 13.48 – Log on to AWS via Eucalyptus

Try and use the auth key we created in the *Uploading an image to EC2* section, or create a new one for the Administrator user in AWS IAM. Copy and paste it into the AWS credentials, and you will have a fully working interface connected to your AWS account. Your dashboard will look almost the same, but will show the status of your AWS account:

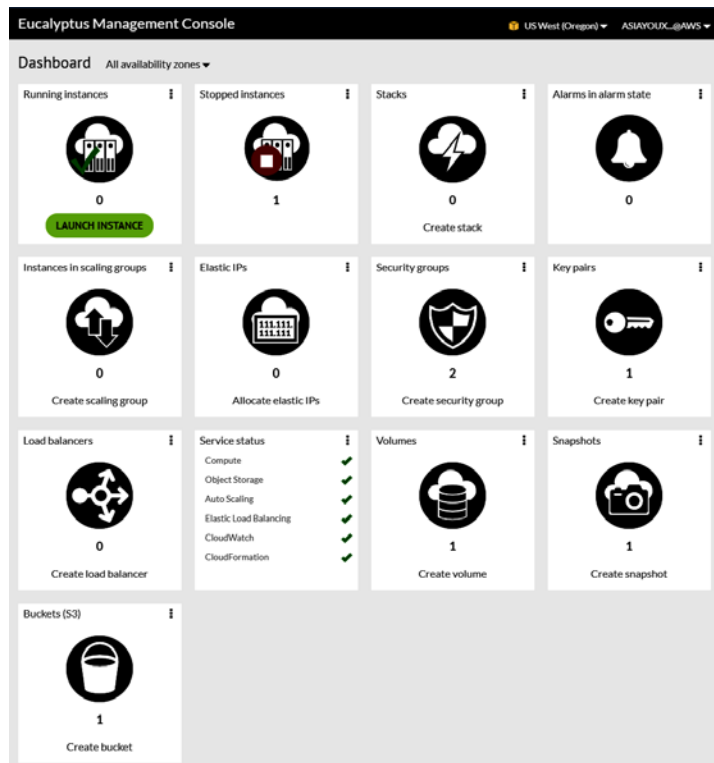


Figure 13.49 – Eucalyptus Management Console for AWS

Let's check we can see our buckets:

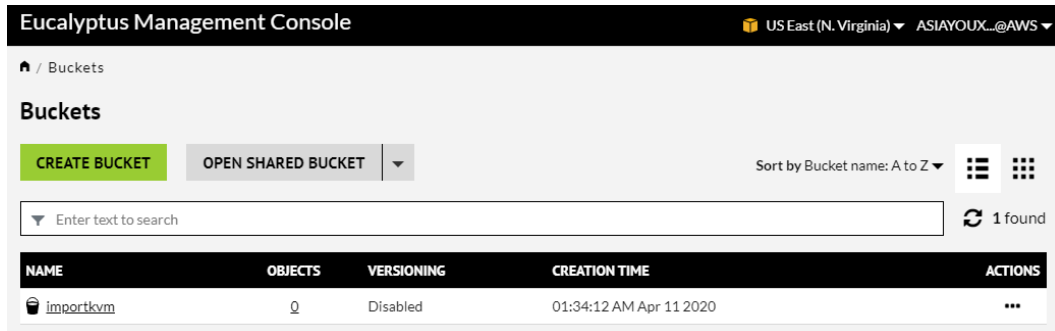


Figure 13.50 – Checking our AWS buckets

Notice that we not only see the bucket we used to test the AWS KVM import, but we also see the region we are running in, in the upper-right corner. Your account is given by its key name, not the actual user; this is simply because we are in fact logged in *programmatically*. Everything we click gets translated to API calls, and the returned data is then parsed and displayed to the user.

Our currently stopped instance is also here, but keep in mind that you will see it only if you choose the region you initially imported your instance into. In our case, it was US West, so our instance is there:



Figure 13.51 – Checking our AWS instances

As you have probably noticed, Eucalyptus is a multi-faceted tool that's able to provide us with hybrid-cloud services. Basically, one of the key points of Eucalyptus is the fact that it gets you to an AWS-compatible level. So, if you start using it as a private solution and sometime in the future start thinking about moving to AWS, Eucalyptus has you covered. It's a de-facto standard solution for KVM-based virtual machines for that purpose.

We will stop here with the AWS integration. The point of this chapter was, after all, to get you to see how Eucalyptus connects to AWS. You might see that this interface lacks functionality that AWS has, but can at the same time be more than enough to control a basic mid-size infrastructure – buckets, images, and instances from one place. Having tested the 5.0 beta 1 version, we can definitely tell you that the full 5.0 version should be quite a substantial upgrade when it comes out. The beta version already has many additional options and we're rather excited to see when the full release comes out.

Summary

In this chapter, we covered a lot of topics. We introduced AWS as a cloud solution and did some cool things with it – we converted our virtual machine so that we can run in it, and made sure that everything works. Then we moved to Eucalyptus, to check how we can use it as a management application for our local cloud environment, and how to use it to extend our existing environment to AWS.

The next chapter takes us into the world of monitoring KVM virtualization by using the ELK stack. It's a hugely important topic, especially as companies and infrastructures grow in size – you just can't keep up with the organic growth of IT services by manually monitoring all possible services. The ELK stack will help you with that – just how much, you'll learn in the next chapter.

Questions

1. What is AWS?
2. What are EC2, S3, and IAM?
3. What is an S3 bucket?
4. How do we migrate a virtual machine to AWS?
5. Which tool do we use to upload a raw image to AWS?
6. How do we authenticate ourselves to AWS as a user?
7. What is Eucalyptus?
8. What are the key services in Eucalyptus?
9. What are availability zones? What are fault domains?
10. What are the fundamental problems of delivering Tier-0 storage services for virtualization, cloud, and HPC environments?

Further reading

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

- Amazon AWS documentation: <https://docs.aws.amazon.com/>
- Amazon EC2 documentation: https://docs.aws.amazon.com/ec2/?id=docs_gateway
- Amazon S3 documentation: https://docs.aws.amazon.com/s3/?id=docs_gateway
- Amazon IAM documentation: https://docs.aws.amazon.com/iam/?id=docs_gateway
- Eucalyptus installation guide: <https://docs.eucalyptus.cloud/eucalyptus/4.4.5/install-guide/index.html>
- Eucalyptus administration guide: <https://docs.eucalyptus.cloud/eucalyptus/4.4.5/admin-guide/index.html>
- Eucalyptus console guide: <https://docs.eucalyptus.cloud/eucalyptus/4.4.5/console-guide/index.html>
- Euca2ools guide: <https://docs.eucalyptus.cloud/eucalyptus/4.4.5/euca2ools-guide/index.html>

14

Monitoring the KVM Virtualization Platform

When you move away from an environment that only has a couple of objects to manage (for example, KVM hosts) to an environment that has hundreds of objects to manage, you start asking yourself very important questions. One of the most prominent questions is, *How am I going to monitor my hundreds of objects without doing a lot of manual work and with some GUI reporting options?* And the answer to that question is the **Elasticsearch, Logstash, Kibana (ELK)** stack. In this chapter, we'll see what these software solutions can do for you and your KVM-based environment.

Behind those cryptic names are technologies that are here to solve a lot of problems you might have when running more than one server. Although you can run the ELK stack to monitor one service, it makes no sense to do so. The advice and solutions provided in this chapter are applicable to all projects involving multiple devices and servers, not only those running on KVM but, in essence, anything that is capable of producing any kind of logging. We will start with the basics of how to monitor KVM as a virtualization platform in general. Then, we'll move on to the ELK stack, including its building blocks and installation, before moving on to its advanced configuration and customization.

In this chapter, we will cover the following topics:

- Monitoring the KVM virtualization platform
- Introduction to the open source ELK solution
- Setting up and integrating the ELK stack
- Configuring the data collector and aggregator
- Creating custom utilization reports
- Let's get started!

Monitoring the KVM virtualization platform

When we talk about running a system that is performing any kind of processing, we quickly come to the problem of monitoring and making sure that our system runs inside a given set of parameters.

When we create a system that is running a workload, it will inevitably produce some kind of data on everything that is happening. This data can be almost infinite in its scope – a server that is just online, without a single *useful* task running will create some kind of log or service data, such as the amount of used memory, services that are starting or stopping, the amount of disk space left, devices that are connecting and disconnecting, and more.

When we start running any useful task, the logs will only get larger.

Having a good and verbose log means that we can find what is going on at this instant with the system; is it running correctly and do we need to do something to make it run better? If something unexpected happens, logs can help us determine what is actually wrong and point us in the direction of the solution. Correctly configured logs can even help us spot errors before they start to create problems.

Suppose you have a system that is getting slower and slower week after week. Let's further suppose that our problem is with the memory allocation of an app we installed on the system. But let's also suppose that this memory allocation is not constant, and instead varies with the number of users using the system. If you take a look at any point in time, you may notice the number of users and memory allocated. But if you just take measurements at different times, you will have a hard time understanding what kind of correlation there is between the memory and the number of users – will the amount of memory allocated be linear to the number of users or will it behave exponentially? If we can see that 100 users are using 100 MB of memory, does that mean that 1,000 users will use 1,000 MB?

But let's suppose that we are logging the amount of memory and the number of users at equally spaced intervals.

We are not doing anything complicated; every couple of seconds, we are writing down the time of the measurement, the amount of memory allocated, and the number of users using the system. We are creating something called a dataset, consisting of **data points**. Using data points is no different than what we did in the preceding example, but once we have a dataset, we can do trend analysis. Basically, instead of looking at a slice of the problem, we can analyze different time segments and compare the number of users and what the amount of memory they were using actually was. That will give us important information about how our system is actually using our memory and at what point we had a problem, even if we don't have a problem right now.

This approach can even help us find and troubleshoot problems that are non-obvious, such as a backup that is taking too long to finish once a month and works normally the rest of the time. This kind of capability that enables us to spot trends and analyze data and system performance is what logging is all about.

Put simply, any kind of monitoring boils down to two things: collecting data from the thing we are trying to monitor and analyzing that data.

Monitoring can be either online or offline. Online monitoring is useful when we are trying to create some sort of alerting system or when we are trying to establish the self-correcting system that will be able to respond to changes in the process. Then, we can either try to correct problems or shut down or restart the system. Online monitoring is usually used by the operations team in order to make sure that everything is running smoothly and that the problems the system may have are logged.

Offline monitoring is much more complicated. Offline monitoring enables us to gather all the data into logs, analyze these logs later, and extrapolate trends and figure out what can be done to the system to make it better. But the fact of the matter is that it's always *delayed* in terms of real-time activity since the offline methodology requires us to *download* and then *analyze* the logs. That's why we prefer real-time log ingestion, which is something that needs to be done online. That's why learning about the ELK stack is so important.

By fitting all these small pieces – real-time log ingestion, search, analytics, and reports – into one larger stack, ELK makes it easier for us to monitor our environment in real time. Let's learn how.

Introduction to the open source ELK solution

We mentioned previously that ELK stands for Elasticsearch, Logstash, and Kibana because these three applications or systems are the building blocks of a complete monitoring and reporting solution. Each part has its own purpose and functions it performs – Logstash gathers all the data into a consistent database, Elasticsearch is able to quickly go through all the data that Logstash stored, and Kibana is here to turn search results into something that is both informational and visually appealing. Having said all this, ELK recently changed its name. Although it is still referred to as the ELK Stack, and almost the entirety of the internet will call it that, the ELK stack is now named the Elastic Stack, for the sole reason that, at the time of writing, there is another fourth component included in the stack. This component is called Beats, and it represents a significant addition to the whole system.

But let's start from the beginning and try to describe the whole system the way its creators describe it.

Elasticsearch

The first component that was created and that got traction in the community was Elasticsearch, created to be a flexible, scalable system for indexing and searching large datasets. Elasticsearch was used for thousands of different purposes, including searching for specific content in documents, websites, or logs. Its main selling point and the reason a lot of people started using it is that it is both flexible and scalable, and at the same time extremely fast.

When we think of searching, we usually think about creating some kind of query and then waiting for the database to give us back some form of answer. In complex searches, the problem is usually the waiting since it is exhausting having to tweak our queries and wait for them to produce results. Since a lot of modern data science relies on the concept of non-structured data, meaning that a lot of data that we need to search has no fixed structure, or no structure at all, creating a fast way to search inside this pool of data is a tough problem.

Imagine you need to find a certain book in a library. Also, imagine you do not have a database of all the books, authors, publishing information, and everything else that a normal library has; you are only allowed to search through all the books themselves.

Having a tool that is able to recognize patterns in those books and that can tell you the answer to questions such as *who wrote this book?* or *how many times is KVM mentioned in all the books that are longer than 200 pages?* is a really useful thing. This is what a good search solution does.

Being able to search for a machine that is running the Apache web server and has problems with a certain page requested by a certain IP address is essential if we want to quickly and efficiently administer a cluster or a multitude of clusters of physical and virtual servers.

The same goes for system information when we are monitoring even a single point of data, such as memory allocation across hundreds of hosts. Even presenting that data is a problem and searching for it in real time is almost impossible without the right tool.

Elasticsearch does exactly that: it creates a way for us to quickly go through enormous amounts of barely structured data and then comes up with results that make sense. What makes Elasticsearch different is its ability to scale, which means you can use it to create search queries on your laptop, and later just run them on a multi-node instance that searches through a petabyte of data.

Elasticsearch is also fast, and this is not something that only saves time. Having the ability to get search results faster gives you a way to learn more about your data by creating and modifying queries and then understanding their results.

Since this is just a simple introduction to what ELK actually does, we will switch to the next component, Logstash, and come back to searching a bit later.

Logstash

Logstash has a simple purpose. It is designed to be able to digest any number of logs and events that generate data and store them for future use. After storing them, it can export them in multiple formats such as email, files, HTTP, and others.

What is important about how Logstash works is its versatility in accepting different input streams. It is not limited to using only logs; it can even accept things such as Twitter feeds.

Kibana

The last part of the old ELK stack is Kibana. If Logstash is storage and Elasticsearch is for computing, then Kibana is the output engine. Simply put, Kibana is a way to use the results of Elasticsearch queries to create visually impressive and highly customizable layouts. Although the output of Kibana is usually some kind of a dashboard, its output can be many things, depending on the user's ability to create new layouts and visualize data. Having said all this, don't be afraid – the internet offers at least a partial, if not full solution, to almost every imaginable scenario.

Next, what we will do is go through the basic installation of the ELK stack, show what it can do, point you in the right direction, and demonstrate one of the most popular *beats* – **metricbeat**.

Using the ELK stack is, in many ways, identical to *running* a server – what you need to do depends on what you actually want to accomplish; it takes only a couple of minutes to get the ELK stack running, but the real effort only starts there.

Of course, for us to fully understand how the ELK stack is used in a live environment, we need to deploy it and set it up first. We'll do that next.

Setting up and integrating the ELK stack

Thankfully, almost everything that we need to install is already prepared by the Elasticsearch team. Aside from Java, everything is nicely sorted and documented on their site.

The first thing you need to do is install Java – ELK depends on Java to run, so we need to have it installed. Java has two different install candidates: the official one from Oracle and the open source OpenJDK. Since we are trying to stay in the open source ecosystem, we'll install OpenJDK. In this book, we are using CentOS 8 as our platform, so the `yum` package manager will be used extensively.

Let's start with the prerequisite packages. The only prerequisite package we need in order to install Java is the `java-11-openjdk-devel` package (substitute "11" with the current version of OpenJDK). So, here, we need to run the following command:

```
yum install java-11-openjdk-devel
```

After issuing that command, you should get a result like this:

```
[root@localhost yum.repos.d]# yum install java-11-openjdk-devel
Updating Subscription Management repositories.
Last metadata expiration check: 0:36:07 ago on Sat 27 Jul 2019 04:37:07 PM EDT.
Dependencies resolved.
=====
Package      Arch   Version                Repository              Size
=====
Installing:
java-11-openjdk-devel
    x86_64 1:11.0.4.11-0.el8_0  rhel-8-for-x86_64-appstream-rpms 3.4 M
Installing dependencies:
javapackages-filesystem
    noarch 5.3.0-1.module+el8+2447+6f56d9a6
    rhel-8-for-x86_64-appstream-rpms 30 k
xorg-x11-fonts-Type1
    noarch 7.5-19.el8             rhel-8-for-x86_64-appstream-rpms 522 k
copy-jdk-configs
    noarch 3.7-1.el8             rhel-8-for-x86_64-appstream-rpms 27 k
ttmkfdir     x86_64 3.0.9-54.el8         rhel-8-for-x86_64-appstream-rpms 62 k
java-11-openjdk-headless
    x86_64 1:11.0.4.11-0.el8_0  rhel-8-for-x86_64-appstream-rpms 39 M
tzdata-java  noarch 2019b-1.el8          rhel-8-for-x86_64-appstream-rpms 189 k
java-11-openjdk
    x86_64 1:11.0.4.11-0.el8_0  rhel-8-for-x86_64-appstream-rpms 227 k
lkscpt-tools x86_64 1.0.18-3.el8        rhel-8-for-x86_64-baseos-rpms 100 k
Enabling module streams:
javapackages-runtime
    201801

Transaction Summary
=====
Install 9 Packages

Total download size: 44 M
Installed size: 180 M
Is this ok [y/N]: y
```

Figure 14.1 – Installing one of the main prerequisites – Java

Once installed, we can verify whether the setup was successful and whether Java is working properly by running the following command:

```
java -version
```

This is the expected output:

```
root@localhost yum.repos.d]# java -version
penjdk version "11.0.4" 2019-07-16 LTS
penJDK Runtime Environment 18.9 (build 11.0.4+11-LTS)
penJDK 64-Bit Server VM 18.9 (build 11.0.4+11-LTS, mixed mode, sharing)
root@localhost yum.repos.d]#
```

Figure 14.2 – Checking Java's version

The output should be the current version of Java and no errors. Other than verifying whether Java works, this step is important in order to verify that the path to Java is correctly set – if you are running on some other distributions, you may have to set the path manually.

Now that java is installed and ready to go, we can continue with the installation of the ELK stack. The next step is to configure the install source for Elasticsearch and other services:

1. We need to create a file in `/etc/yum.repos.d/` named `elasticsearch.repo` that will contain all the information about our repository:

```
[Elasticsearch-7.x]
name=Elasticsearch repository for 7.x packages
baseurl=https://artifacts.elastic.co/packages/7.x/yum
gpgcheck=1
gpgkey=https://artifacts.elastic.co/GPG-KEY-Elasticsearch
enabled=1
autorefresh=1
type=rpm-md
```

Save the file. The important thing here is that the repository is GPG-signed, so we need to import its key and apply it so that the packages can be verified when they're downloaded.

The files that you are going to install are not free software. Elasticsearch has two distinct free versions and a paid subscription model. What you are going to get using the files in this repository is the subscription-based install that is going to run in *basic* mode, which is free. At the time of writing, Elastic has four subscription models – one is open source, based on the Apache License 2.0, and free; the rest of them are closed source but offer additional functionalities. Currently, these subscriptions are named Basic, Gold, and Platinum. Basic is free, while the other models require a monthly paid subscription.

You will inevitably ask why you should choose open source over Basic, or vice versa since they are both free. While both of them have the same core, Basic is more advanced as it offers core security features and more things that can be important in everyday use, especially if you are after Kibana visualizations.

2. Let's continue with the installation and import the necessary GPG key:

```
rpm --import https://artifacts.elastic.co/GPG-KEY-
elasticsearch
```

3. Now, we are ready to do some housekeeping on the system side and grab all the changes in the repository system:

```
sudo yum clean all
```

```
sudo yum makecache
```

If everything is okay, we can now install `elasticsearch` by running this command:

```
sudo yum install elasticsearch
```

Neither `elasticsearch` nor any of the other services are going to be started or enabled automatically. We must do this manually for each of them. Let's do that now.

4. The procedure to start and enable services is standard and is the same for all three services:

```
sudo systemctl daemon-reload
```

```
sudo systemctl enable elasticsearch.service
```

```
sudo systemctl start elasticsearch.service
```

```
sudo systemctl status elasticsearch.service
```

```
sudo yum install kibana
```

```
sudo systemctl status kibana.service
```

```
sudo systemctl enable kibana.service
```

```
sudo systemctl start kibana.service
```

```
sudo yum install logstash
```

```
sudo systemctl start logstash.service
```

```
sudo systemctl enable logstash.service
```

The last thing to do is installing *beats*, which are services that are usually installed on the monitored servers, and which can be configured to create and send important metrics on the system. Let's do that now.

5. For the purpose of this demonstration, we will install them all, although we are not going to use all of them:

```
sudo yum install filebeat metricbeat packetbeat  
heartbeat-elastic auditbeat
```

After this, we should have a functional system. Let's have a quick overview.

Kibana and Elasticsearch are both running as web services, on different ports. We are going to interact with Kibana via the web browser (using the URLs `http://localhost:9200` and `http://localhost:5601`) since this is where the visualization happens:

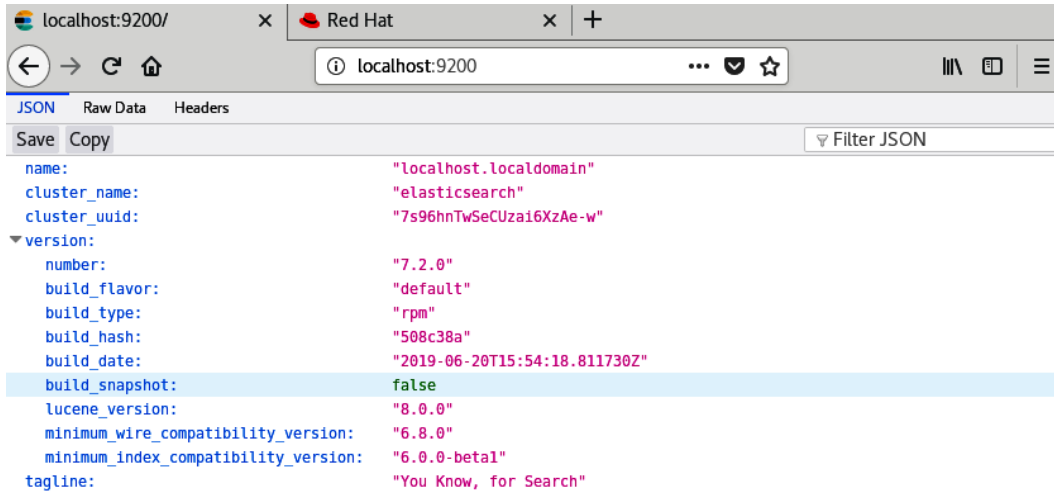


Figure 14.3 – Checking the Elasticsearch service

Now, we can connect to Kibana on port 5601:

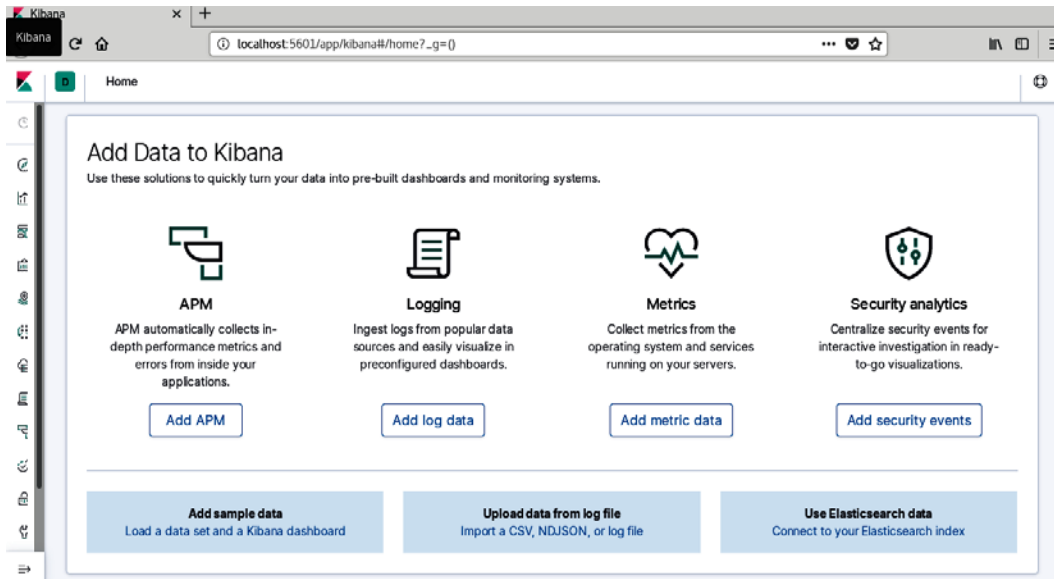


Figure 14.4 – Successful connection to Kibana

With that, the deployment process was finished successfully. Our logical next step would be to create a workflow. Let's do that now.

Workflow

In this section, we are going to establish a workflow – we are going to create logs and metrics that are going to be ingested into Logstash, queried via Elasticsearch, and then visually represented in Kibana.

By default, Kibana runs on port 5601, which can be changed in the configuration.

But what does this mean for me? What does this mean for KVM?

The biggest selling point for using Elastic stack is flexibility and ease of presentation. It doesn't matter if we are running one, 10, or 1,000 machines inside dozens of KVM hosts; we can treat them the same in production and establish a stable monitoring workflow. Using extremely simple scripts, we can create completely custom metrics and quickly display them, we can watch for trends, and we can even create a near-real-time monitoring system. All this, essentially for free.

Let's create a simple monitor that is going to dump system metrics for the host system that is running ELK. We've already installed Metricbeat, so the only thing left is to configure the service to send the data to Elasticsearch. Data is sent to Elasticsearch, not Logstash, and this is simply because of the way that the services interoperate. It is possible to send both to Logstash and Elasticsearch, so we need to do a quick bit of explaining here.

Logstash is, by definition, a service that stores data that's sent to it. Elasticsearch searches that data and communicates with Logstash. If we send the data to Logstash, we are not doing anything wrong; we are just dumping data for later analysis. But sending to Elasticsearch gives us one more feature – we can send not only data but also information about the data in the form of templates.

On the other hand, Logstash has the ability to perform data transformation right after it receives it and before data is stored, so if we need to do things such as parse GeoIP information, change the names of hosts, and so on, we will probably use Logstash as our primary destination. Keeping that in mind, do not set Metricbeat so that it sends data both to Elasticsearch and Logstash; you will only get duplicate data stored in the database.

Using ELK is simple, and we've got this far into the installation without any real effort. When we start analyzing the data is when the real problems start. Even simple and perfectly formatted data that comes out of Metricbeat can be complex to visualize, especially if we are doing it for the first time. Having premade templates both for Elasticsearch and Kibana saves a lot of time.

Take a look at the following screenshot:

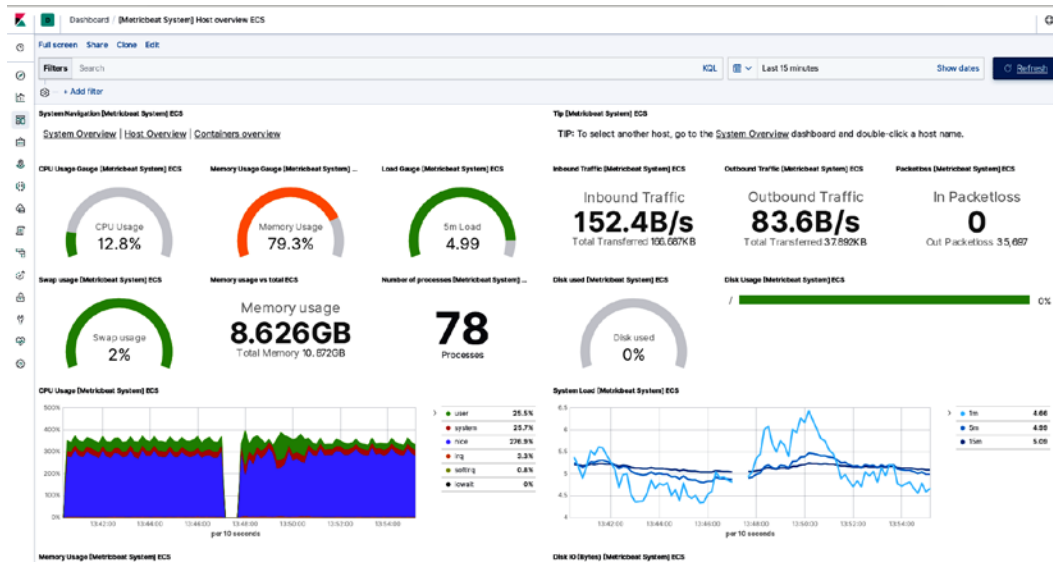


Figure 14.5 – Metricbeat dashboard

It takes no more than 10 minutes of setup to get a complete dashboard like this one. Let's go through this step by step.

We already have Metricbeat installed and only need to configure it, but before that, we need to configure Logstash. We only need to define one *pipeline*.

So, how can data be transformed?

Up until now, we did not go into details regarding how Logstash functions, but to create our first set of data, we need to know some of the inner workings of Logstash. Logstash uses a concept of a pipeline to define what happens to data once it's received, and before that data is sent to Elasticsearch.

Each pipeline has two required, and one optional, elements:

- The input is always the first in the pipeline and is designed to receive data from the source.
- The output is the last element in the pipeline, and it outputs the data.
- The filter is an optional element and stands between the input and output in order to modify the data in accordance with the rules that we can define.

All these elements can be chosen from a list of plugins in order for us to create an optimal pipeline adjusted for a specific purpose. Let's go through this step by step.

What we need to do is just uncomment the one pipeline that is defined in the configuration file, located in the `/etc/logstash` folder.

The whole stack uses YAML as the standard for the configuration file structure, so every configuration file ends with the `.yaml` extension. This is important in order to understand that all the files that do not have this extension are here as either a sample or some kind of template for the configuration; only files with the `.yaml` extension will get parsed.

To configure Logstash, just open `logstash.yaml` and uncomment all the lines that are related to the first pipeline, called `main`. We don't need to do anything else. The file itself is located in the `/etc/logstash` folder, and should look something like this after you make these changes:

```
#
  pipeline.id: main
#
# Set the number of workers that will, in parallel, execute the filters+outputs
# stage of the pipeline.
#
# This defaults to the number of the host's CPU cores.
#
  pipeline.workers: 2
#
# How many events to retrieve from inputs before sending to filters+workers
#
  pipeline.batch.size: 125
#
# How long to wait in milliseconds while polling for the next event
# before dispatching an undersized batch to filters+outputs
#
  pipeline.batch.delay: 50
#
# Force Logstash to exit during shutdown even if there are still inflight
# events in memory. By default, logstash will refuse to quit until all
# received events have been pushed to the outputs.
#
```

Figure 14.6 – The `logstash.yaml` file

The next thing we need to do is configure Metricbeat.

Configuring data collector and aggregator

In the previous steps, we managed to deploy Metricbeat. Now, we need to start the actual configuration. So, let's go through the configuration procedure, step by step:

1. Go to `/etc/metricbeat` and open `metricbeat.yml`.

Uncomment the lines that define `elasticsearch` as the target for Metricbeat. Now, we need to change one more thing. Find the line containing the following:

```
setup.dashboards.enabled: false
```

Change the preceding line to the following:

```
setup.dashboards.enabled: true
```

We need to do this to load dashboards so that we can use them.

2. The rest of the configuration is done from the command line. Metricbeat has a couple of commands that can be run, but the most important is the following one:

```
metricbeat setup
```

This command will go through the initial setup. This part of the setup is probably the most important thing in the whole initial configuration – pushing the dashboard templates to Kibana. These templates will enable you to get up and running in a couple of clicks, as opposed to learning how to do visualization and configuring it from scratch. You will have to do this eventually but for this example, we want to get things running as quickly as possible.

3. One more command that you need right now is the following one:

```
metricbeat modules list
```

This will give you a list of all the modules that Metricbeat already has prepared for different services. Go ahead and enable two of them, `logstash` and `kvm`:

```
metricbeat modules enable kvm
```

```
metricbeat modules enable logstash
```

The `logstash` module is confusingly named since it is not intended to push data to Logstash; instead, its main purpose is to report the Logstash service and enable you to monitor it through Logstash. Sound confusing? Let's rephrase this: this module enables Logstash to monitor itself. Or to be more precise, it enables beats to monitor part of the Elastic stack.

The `KVM` module is a template that will enable you to gather different KVM-related metrics.

This should be it. As a precaution, type in the following command to check Metricbeat's configuration:

```
metricbeat test config
```

If the preceding command runs okay, start the Metricbeat service using the following command:

```
systemctl start metricbeat
```

You now have a running service that is gathering data on your host – the same one that is running KVM and dumping that data into Elasticsearch. This is essential since we are going to use all that data to create visualizations and dashboards.

Creating charts in Kibana

Now, open Kibana in a browser using `localhost:5601` as the address. There should be an icon-based menu on the left-hand side of the screen. Go to **Stack management** and take a look at **Elasticsearch index management**.

There should be an active index named `metricbeat-<somenum>`. In this particular example, `<somenum>` will be the current version of metricbeat and the date of the first entry in the log file. This is completely arbitrary and is just a default that ensures you know when this instance was started.

In the same line as this name, there should be some numbers: what we are interested in is the docs count – the number of objects that database holds. For the time being, if it's not zero, we are okay.

Now, go to the **Dashboard** page and open the **Metricbeat System Overview ECS** dashboard. It will show a lot of visual widgets representing CPU, memory, disk, and network usage:

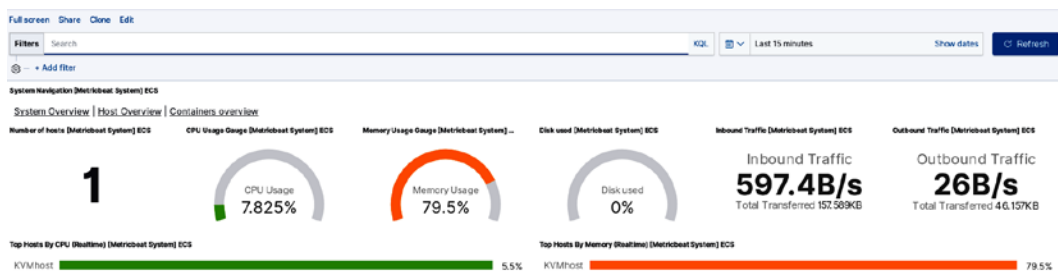


Figure 14.7 – Overview of the ECS dashboard

Now, you can click on **Host Overview** and view even more data about your system. Try playing with the dashboard and different settings. One of the most interesting items on this dashboard is the one in the upper-right part of the screen – the one that defines the timespan that we are interested in. We can either create our own or use one of the presets, such as `last 15 minutes`. After you click the **Refresh** button, new data should show on the page.

With that, you now know enough about Kibana to get started, but we still are unable to visualize KVM data. The next step is to create a dashboard that will cover that.

But before we do that, think about what you can do with only what we've learned so far. Not only can you monitor the local system that has your KVM stack installed, but you can also monitor any system that is able to run Metricbeat. The only thing that you need to know is the IP address of the ELK stack, so that you can send data to it. Kibana will automatically deal with visualizing all the different data from different systems, as we will see later.

Creating custom utilization reports

Since version 7, Elastic stack has introduced mandatory checks that are designed to ensure minimum security and functionality compliance, especially once we start using ELK in production.

At first glance, these checks may confuse you – the installation that we guided you through will work, and suddenly, as you try to configure some settings, everything will fail. This is intentional.

In previous versions, these checks were performed but were flagged as warnings if a configuration item was missed or misconfigured. Starting from version 7, these checks will trigger an error when the system is in production and not configured correctly. This state automatically means that your installation will not work if it's not configured properly.

ELK has two distinct modes of operation: *development* and *production*. On the first installation, it is assumed that you are in development mode, so most of the functionality simply works out of the box.

Things change a lot once you go into production mode – security settings and other configuration options need to be explicitly set in order for the stack to function.

The trick is that there is no explicit mode change – production settings and checks associated with them are triggered by some settings in the configuration. The idea is that once you reconfigure something that can be important from a security standpoint, you need to reconfigure everything correctly. This will prevent you from forgetting something that can be a big problem in production and force you to have at least a stable configuration to start from. There is a switch to disable checks, but it is not recommended in any circumstances.

The main thing to pay attention to is the binding interface – the default installation binds everything to `localhost` or a local loopback interface, which is completely fine for production. Once your Elasticsearch is capable of forming a cluster and it can be triggered by simply reconfiguring the network address for HTTP and transport communication, you have to pay attention to the checks and reconfigure the whole system in order to make it work. Please consult the documentation available on <https://www.elastic.co/> for more information, starting with <https://www.elastic.co/guide/index.html>.

For example, configuring clusters in the Elastic stack and all that it entails is way out of the scope of this book – we are going to stay within the realm of a *single-node cluster* in our configuration. This solution was specifically created for situations that can work with a single node or, more precisely, a single machine instance that covers all the functionality of a stack. In a normal deployment, you will run Elastic stack in a cluster, but implementation details will be something determined by your configuration and its needs.

We need to warn you of two crucial points – firewall and SELinux settings are up to you. All the services use standard TCP to communicate. Don't forget that for the services to run, the network has to be configured correctly.

Now that we've gotten that out of the way, let's answer one simple question: what do we need to do to make the Elastic stack work with more than one server? Let's discuss this scenario, bit by bit.

Elasticsearch

Go to the configuration file (`/etc/elasticsearch/elasticsearch.yml`) and add a line in the discovery section:

```
discovery.type: single-node
```

Using this section is not mandatory, but it helps when you must go back to the configuration later.

This option will tell Elasticsearch that you will have only one node in the cluster, and it will make Elasticsearch ignore all the checks associated with the cluster and its network. This setting will also make this node the master node automatically since Elasticsearch depends on having master nodes that control everything in the cluster.

Change the setting under `network.host`: so that it points to the IP address of the interface Elasticsearch is going to be available on. By default, it points to localhost and is not visible from the network.

Restart the Elasticsearch service and make sure it is running and not generating errors:

```
sudo systemctl restart elasticsearch.service
```

Once you have it working, check whether the service is behaving normally from the local machine. The easiest way is to do this is as follows:

```
curl -XGET <ip_address>:9200
```

The response should be `.json` formatted text containing information about the server.

Important note

The Elastic stack has three (or four) parts or *services*. In all our examples, three of them (Logstash, Elasticsearch, and Kibana) were running on the same server, so no additional configuration was necessary to accommodate network communication. In a normal configuration, these services would probably run on independent servers and in multiple instances, depending on the workload and configuration of the service we are trying to monitor.

Logstash

The default installation for Logstash is a file named `logstash-sample.conf` in the `/etc/logstash` folder. This contains a simple Logstash pipeline to be used when we are using Logstash as the primary destination for beats. We will come to this later, but for the time being, copy this file to `/etc/logstash/conf.d/logstash.conf` and change the address of the Elasticsearch server in the file you just copied. It should look something like this:

```
hosts => ["http://localhost:9200"].
```

Change `localhost` to the correct IP address of your server. This will make Logstash listen on port 5044 and forward the data to Elasticsearch. Restart the service and verify that it runs:

```
sudo systemctl restart logstash.service
```

Now, let's learn how to configure Kibana.

Kibana

Kibana also has some settings that need to be changed, but when doing so, there are a couple of things to remember about this service:

- By itself, Kibana is a service that serves visualizations and data over the HTTP protocol (or HTTPS, depending on the configuration).
- At the same time, Kibana uses Elasticsearch as its backend in order to get and work with data. This means that there are two IP addresses that we must care about:
 - a) The first one is the address that will be used to show Kibana pages. By default, this is localhost on port 5601.
 - b) The other IP address is the Elasticsearch service that will deal with the queries. The default for this is also localhost, but it needs to be changed to the IP address of the Elasticsearch server.

The file that contains configuration details is `/etc/kibana/kibana.yml` and you need to at least make the following changes:

- `server.host`: This needs to point to the IP address where Kibana is going to have its pages.
- `elasticsearch.hosts`: This needs to point to the host (or a cluster, or multiple hosts) that are going to perform queries.

Restart the service, and that's it. Now, log into Kibana and test whether everything works.

To get you even more familiarized with Kibana, we will try and establish some basic system monitoring and show how we can monitor multiple hosts. We are going to configure two *beats*: Metricbeat and Filebeat.

We already configured Metricbeat, but it was for localhost, so let's fix that first. In the `/etc/metricbeat/metricbeat.yml` file, reconfigure the output in order to send data to the `elasticsearch` address. You only need to change the host IP address since everything else stays the same:

```
# Array of hosts to connect to
Hosts: ["Your-host-IP-address:9200"]
```

Make sure that you change `Your-host-IP-address` to the IP address you're using.

Configuring filebeat is mostly the same; we need to use `/etc/filebeat/filebeat.yml` to configure it. Since all the beats use the same concepts, both filebeat and metricbeat (as well as other beats) use modules to provide functionality. In both, the core module is named `system`, so enable it using the following command in filebeat:

```
filebeat modules enable system
```

Use the following command for metricbeat:

```
metricbeat modules enable system
```

We mentioned this previously, in the first example, but you can test your configuration by running the following command:

```
filebeat test config
```

You can also use the following command:

```
metricbeat test config
```

Both beats should say that the config is ok.

Also, you can check the output settings, which will show you what the output settings actually are and how they work. If you are configuring the system using only this book, you should have a warning come up to remind you there is no TLS protection for the connection, but otherwise, the outputs should work on the IP address that you set in the configuration file.

To test the outputs, use the following command:

```
filebeat test output
```

You can also use the following command:

```
metricbeat test output
```

Repeat all this for every system that you intend to monitor. In our example, we have two systems: one that is running KVM and another that is running Kibana. We also have Kibana set up on the other system to test syslog and the way it notifies us of the problems it notices.

We need to configure filebeat and metricbeat to send data to Kibana. We'll edit the `filebeat.yml` and `metricbeat.yml` files for that purpose, by changing the following portion of both files:

```
setup.kibana
  host: "Your-Kibana-Host-IP:5601"
```

Before running beats, on a fresh installation, you need to upload dashboards to Kibana. You only need to do this once for each Kibana installation, and you only need to do this from one of the systems you are monitoring – templates will work, regardless of the system they were uploaded from; they will just deal with data that is coming into Elasticsearch.

To do this, use the following command:

```
filebeat setup
```

You also need to use the following command:

```
metricbeat setup
```

This will take a couple of seconds or even a minute, depending on your server and client. Once it says that it created the dashboards, it will display all the dashboards and settings it created.

Now, you are almost ready to go through all the data that Kibana will display:

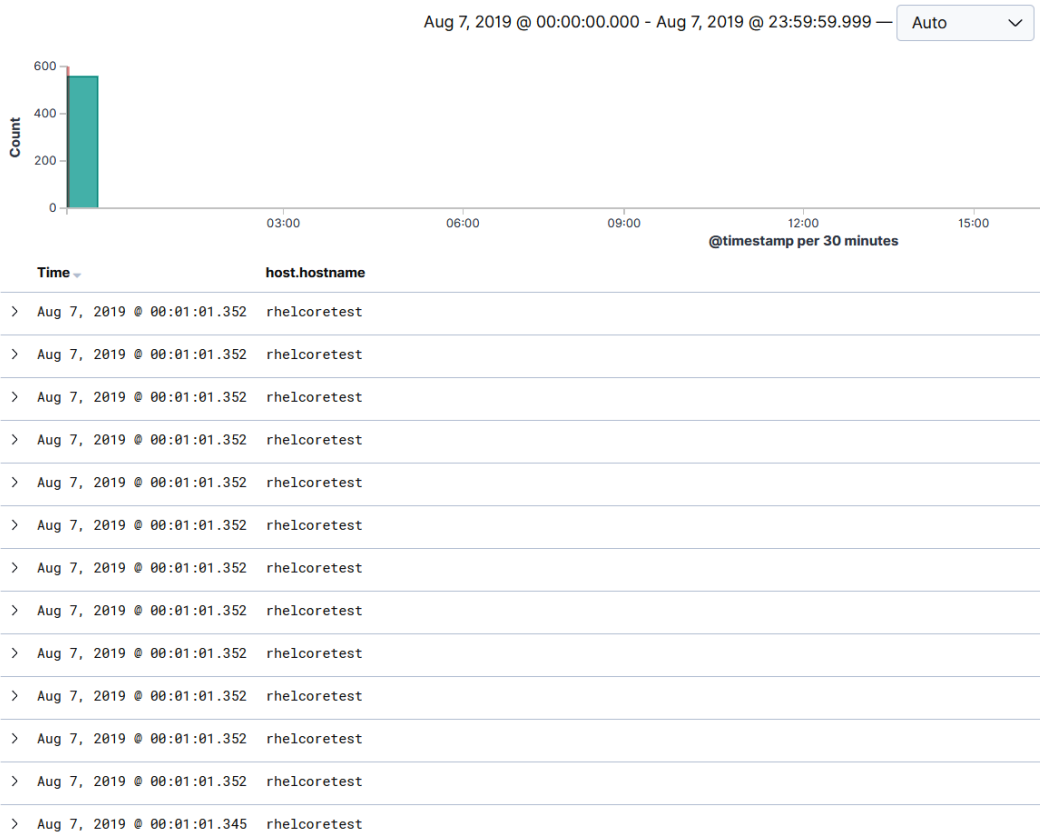


Figure 14.8 – Excerpt from the Kibana dashboard

Before we start, there's something else you need to know about time and timestamps. The date/time picker in the top-right corner will let you choose either your own timespan or one of the predefined intervals:

Quick select

Last 15 minutes Apply

Commonly used

Today	This week
Last 15 minutes	Last 30 minutes
Last 1 hour	Last 24 hours
Last 7 days	Last 30 days
Last 90 days	Last 1 year

Recently used date ranges

- Last 24 hours
- Today
- Aug 6, 2019 @ 03:47:17.882 to Aug 6, 2019 @ 22:37:06.182
- Last 15 minutes
- Last 1 hour

Refresh every

0 seconds Start

Figure 14.9 – Date/time picker

Important note

Always remember that the time that's shown is *local* to the browser's/machine's time zone you are accessing Kibana from.

All the timestamps in the logs are *local* to the machine that is sending the logs. Kibana will try and match time zones and translate the resulting timestamps, but if there is a mismatch in the actual time settings on the machines you are monitoring, there is going to be a problem trying to establish a timeline of events.

Let's presume you got filebeat and metricbeat running. What can you do with these? As it turns out, a lot:

- The first thing is discovering what is in your data. Press the **Discover** button in Kibana (it looks like a small compass). Some data should show on the right if everything is okay.
- To the right of the icon you just clicked on, a vertical space will fill up with all the attributes that Kibana got from the data. If you do not see anything or something is missing, remember that the time span you select narrows down the data that will get shown in this view. Try readjusting the interval to **Last 24 hours** or **Last 30 days**.

Once the list of attributes shows up, you can quickly establish how many times each shows up in the data you just selected – just click on any attribute and select **Visualize**. Also note that once you click on the attribute, Kibana shows you the top five distinct values in the last 500 records. This is a very useful tool if you need to know, for example, which hosts are showing data, or how many different OS versions there are.

The visualization of particular attributes is just a start – notice how, once you hover over an attribute name, a button called **Add** appears? Try clicking it. A table will start forming on the right, filled with just the attributes you selected, sorted by timestamp. By default, these values are not auto-refreshed, so the timestamps will be fixed. You can choose as many attributes as you want and save this list or open it later.

The next thing we need to look at is individual visualizations. We are not going to go into too many details, but you can create your own visualizations out of the datasets using predefined visualization types. At the same time, you are not limited to using only predefined things – using JSON and scripting is also possible, for even more customization.

The next thing we need to learn about is dashboards.

Depending on a particular dataset, or to be more precise, on the particular set of machines you are monitoring, some of them will have attributes that cover things only a particular machine does or has. One example is virtual machines on AWS – they will have some information that is useful only in the context of AWS. This is not important in our configuration, but you need to understand that there may be some attributes in the data that are unique for a particular set of machines. For starters, choose one of the system metrics; either **System Navigation ECS** for metricbeat or **Dashboards ECS** for filebeat.

These dashboards show a lot of information about your systems in a lot of ways. Try clicking around and see what you can deduce.

The metricbeat dashboard is more oriented toward running systems and keeping an eye on memory and CPU allocation. You can click and filter a lot of information, and have it presented in different ways. The following is a screenshot of metricbeat so that you can get a rough idea of what it looks like:



Figure 14.10 – metricbeat dashboard

The filebeat dashboard is more oriented toward analyzing what happened and establishing trends. Let's check a couple of excerpts from the filebeat dashboard, starting with the syslog entries part:

Syslog logs [Filebeat System] ECS

Time	host.hostname	process.name
> Aug 6, 2019 @ 20:35:01.000	localhost	kibana
> Aug 6, 2019 @ 20:35:01.000	localhost	auditbeat
> Aug 6, 2019 @ 20:35:01.000	rhelcoretest	logstash
> Aug 6, 2019 @ 20:34:59.000	localhost	dbus-daemon
> Aug 6, 2019 @ 20:34:59.000	localhost	systemd
> Aug 6, 2019 @ 20:34:59.000	localhost	journal
> Aug 6, 2019 @ 20:34:59.000	localhost	dbus-daemon

Figure 14.11 – filebeat syslog entries part

At first glance, you can notice a couple of things. We are showing data for two systems, and the data is partial since it covers a part of the interval that we set. Also, we can see that some of the processes are running and generating logs more frequently than others. Even if we do not know anything about the particular system, we can now see there are some processes that show up in logs, and they probably shouldn't:

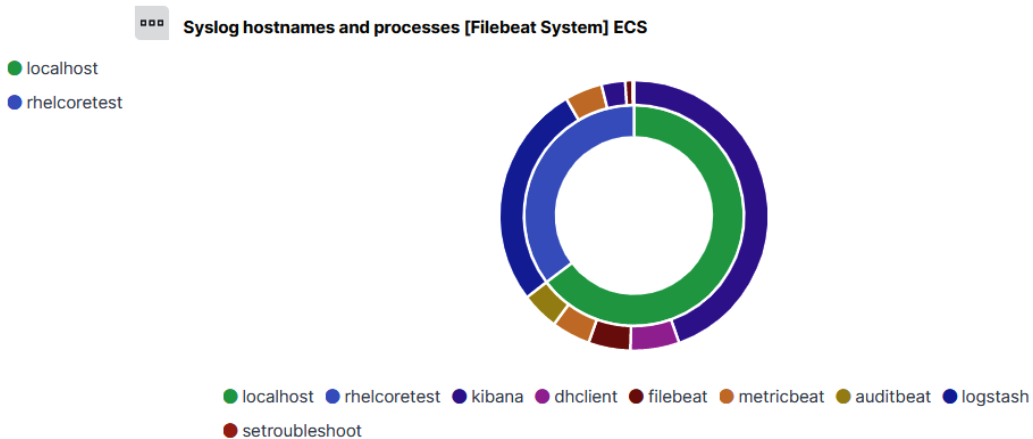


Figure 14.12 – filebeat interactive doughnut chart

Let's take a look at `setroubleshoot`. Click on the process name. In the window that opens, click on the magnifying glass. This isolates only this process and shows only its logs at the bottom of the screen.

We can quickly see on which host – including how often and why – `setroubleshoot` is writing logs to. This is a quick way to spot potential problems. In this particular case, some action should obviously be taken on this system to reconfigure SELinux since it generates exceptions and stops some applications from accessing files.

Let's move along the vertical navigation bar and point out some other interesting functionalities.

Going from top to bottom, the next big functionality is **Canvas** – it enables us to create live presentations using data from the dataset we are collecting. The interface is similar to what can be expected from other presentation programs, but the accent is on using data directly in slides and generating slides in almost real time.

The next is **Maps**. This is a new addition to version 7.0 and allows us to create a geographic presentation of data.

Machine learning is next – it enables you to manipulate data and use it to "train" filters and create pipelines out of them.

Infrastructure is also interesting – when we mentioned dashboards, we were talking about flexibility and customization. Infrastructure is a module that enables us to do real-time monitoring with minimal effort and observe important metrics. You can see important data either as a table, a balloon-like interface, or as a graph. Data can be averaged or presented in other ways, and all that is done through a highly intuitive interface.

Heartbeat is another of these highly specialized boards – as its name suggests, it is the easiest way to track and report on uptime data, and to quickly notice if something has gone offline. Inventory hosts require the Heartbeat service to be installed on each system we intend to monitor.

SIEM deserves a more thorough explanation: if we think about dashboards as being multipurpose, SIEM is the exact opposite; it is created to be able to track all the events on all the systems that can be categorized as security-related. This module will parse the data when searching for IPs, network events, sources, destinations, network flows, and all other data, and create simple to understand reports regarding what is happening on the machines you are monitoring. It even offers anomaly detection, a feature that enables the Elastic stack to serve as a solution for advanced security purposes. This feature is a paid one and requires the highest-paid tier to function.

Stack monitor is another notable board as it enables you to actually see what is happening in all the different parts of the Elastic stack. It will show the status of all the services, their resource allocation, and license status. The **Logs** feature is especially useful since it tracks how many logs of what type the stack is generating, and it can quickly point to problems if there are any.

This module also generates statistics for services, enabling us to understand how the system can be optimized.

Management, the last icon at the bottom, was already mentioned – it enables the management of the cluster and its parts. This is the place where we can see whether there are any indices we are expecting, whether the data flowing in, whether can we optimize something, and so on. This is also the place we can manage licenses and create snapshots of system configuration.

ELK and KVM

Last but not least, let's create a system gauge that is going to show us a parameter from a KVM hypervisor, and then visualize it in a couple of ways. The prerequisites for this are a running KVM hypervisor, metricbeat with the KVM module installed, and an Elastic stack configuration that supports receiving data from metricbeat. Let's go through ELK's configuration for this specific use case:

1. First, go to the hypervisor and open a `virsh` shell. List all the available domains, choose a domain, and use the `dommemstat --domain <domain_name>` command.

The result should be something like this:

```
virsh # dommemstat --domain cen
actual 1048576
swap_in 0
swap_out 0
major_fault 192
minor_fault 160373
unused 905060
available 1014904
usable 855520
last_update 1564957343
rss 580292

virsh # █
```

Figure 14.13 – dommemtest for a domain

2. Open Kibana and log in, go to the **Discover** tab, and select `metric*` as the index we are working with. The left column should populate with attributes that are in the dataset that metricbeat sends to this Kibana instance. Now, look at the attributes and select a couple of them:

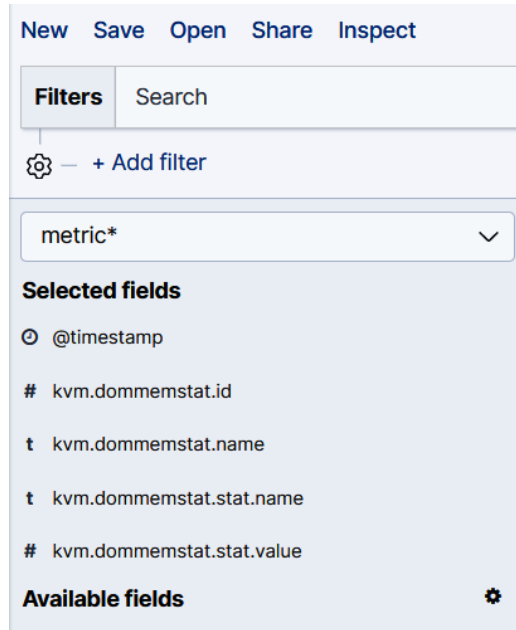


Figure 14.14 – Selecting attributes in Kibana

You can select fields using a button that will show up as soon as you hover the mouse cursor over any field. The same goes for deselecting them:

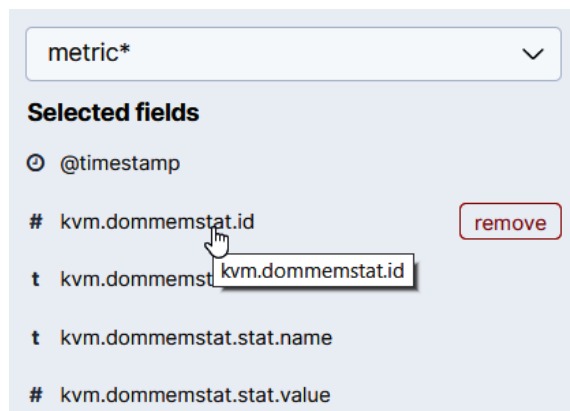


Figure 14.15 – Adding attributes in Kibana

3. For now, let's stick with the ones we selected. To the right of the column, a table is formed that contains only the fields you selected, enabling you to check the data that the system is receiving. You may need to scroll down to see the actual information since this table will display all the data that was received that has at least one item that has a value. Since one of the fields is always a timestamp, there will be a lot of rows that will not contain any useful data for our analysis:

Aug 8, 2019 @ 23:22:49.313	Aug 8, 2019 @ 23:22:49.313	2	cen	actualballoon	1,048,576
Aug 8, 2019 @ 23:22:49.313	Aug 8, 2019 @ 23:22:49.313	2	cen	swapi	0
Aug 8, 2019 @ 23:22:49.313	Aug 8, 2019 @ 23:22:49.313	2	cen	swapout	0
Aug 8, 2019 @ 23:22:49.313	Aug 8, 2019 @ 23:22:49.313	2	cen	majorfault	192
Aug 8, 2019 @ 23:22:49.313	Aug 8, 2019 @ 23:22:49.313	2	cen	minorfault	160,373
Aug 8, 2019 @ 23:22:49.313	Aug 8, 2019 @ 23:22:49.313	2	cen	unused	985,060
Aug 8, 2019 @ 23:22:49.313	Aug 8, 2019 @ 23:22:49.313	2	cen	available	1,014,904
Aug 8, 2019 @ 23:22:49.313	Aug 8, 2019 @ 23:22:49.313	2	cen	usable	855,520
Aug 8, 2019 @ 23:22:49.313	Aug 8, 2019 @ 23:22:49.313	2	cen	lastupdate	1,564,957,343
Aug 8, 2019 @ 23:22:49.313	Aug 8, 2019 @ 23:22:49.313	2	cen	rss	580,292

Figure 14.16 – Checking the selected fields

What we can see here is that we get the same data that was the result of running the command line on the monitored server.

What we need is a way to use these results as data to show our graphs. Click on the **Save** button at the top left of the screen. Use a name that you will be able to find later; we used `dommemstat`. Save the search.

- Now, let's build a gauge that will show us the real-time data and a quick visualization of one of the values. Go to **Visualize** and click **Create new visualization**:

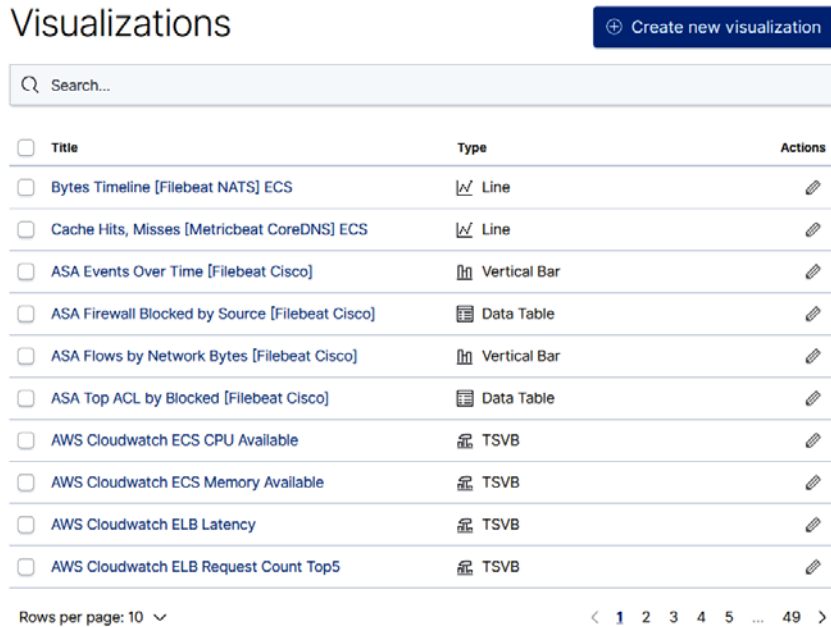


Figure 14.17 – Creating a new visualization

Choose the area graph. Then, on the next screen, find and select our source for the data:

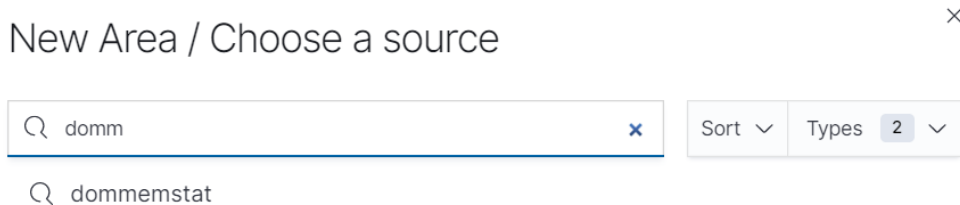


Figure 14.18 – Selecting a visualization source

This is going to create a window that has all the settings on the left and the end result on the right. At the moment, what we can see makes no sense, so let's configure what we need in order to show our data. There are a couple of ways to accomplish what we want: we are going to use a histogram and filters to quickly display how our value for unused memory changed over time.

5. We are going to configure the y axis to show average data for `kvm.dommemstat.stat.value`, which is the attribute that holds our data. Select **Average** under **Aggregation** and `kvm.dommemstat.stat.value` as the field we are aggregating. You can create a custom label if you want to:

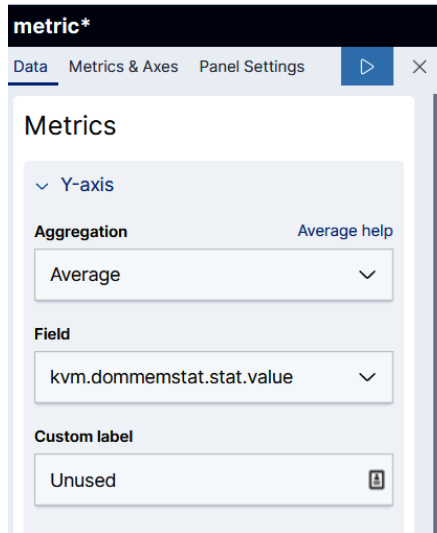


Figure 18.19 – Selecting metric properties

This is still not right; we need to add a timestamp to see how our data changed over time. We need to add a **Date Histogram** type to the x axis and use it:

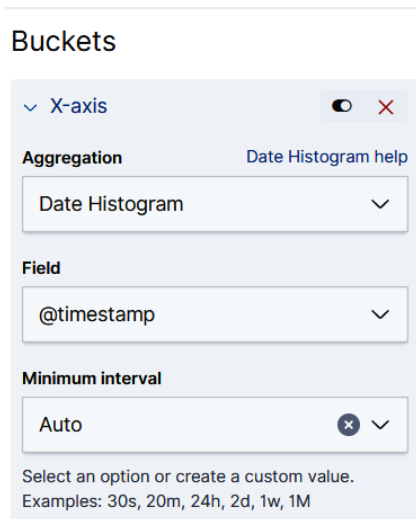


Figure 14.20 – Choosing an aggregation type

- Before we finish this visualization, we need to add a filter. The problem with data that is received from the KVM metricbeat module is that it uses one attribute to hold different data – if we want to know what the number in the file we are displaying actually means, we need to read its name from `kvm.dommemstat.stat.name`. To accomplish this, just create a filter called `kvm.dommemstat.stat.name:"unused"`.

After we refresh the visualization, our data should be correctly visualized on the right:

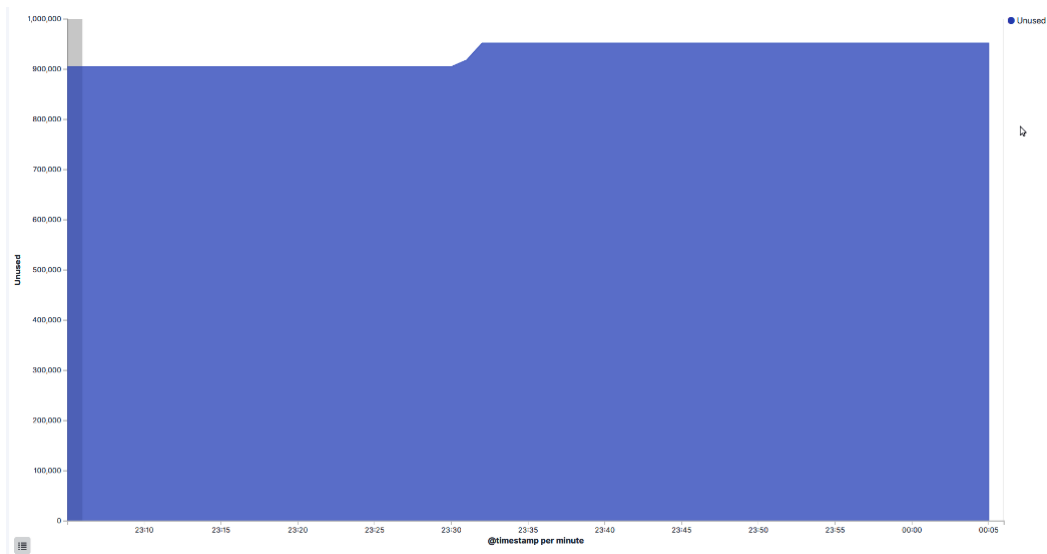


Figure 14.21 – Correct visualization

- We need to save this visualization using the **Save** button, give it a name that we will be able to find later, and repeat this process but instead of filtering for `unused`, filter for `usable`. Leave all the settings identical to the first visualization.

Let's build a dashboard. Open the **Dashboard** tab and click on **Add new dashboard** on the first screen. Now, add our two visualizations to this dashboard. You just need to find the right visualization and click on it; it will show up on the dashboard.

As a result, we have a couple of simple dashboards running:

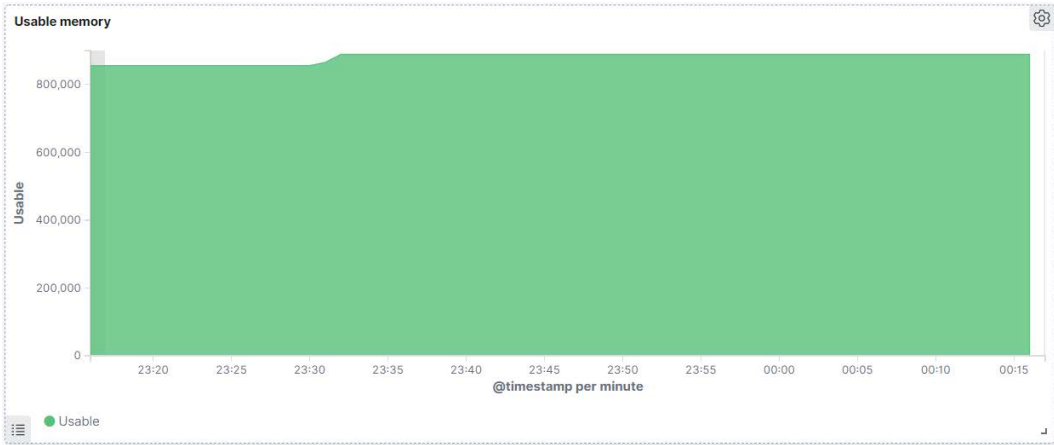


Figure 14.22 – Finished dashboard showing usable memory

The second dashboard – in the UI, which is actually right next to the first one – is the unused memory dashboard:

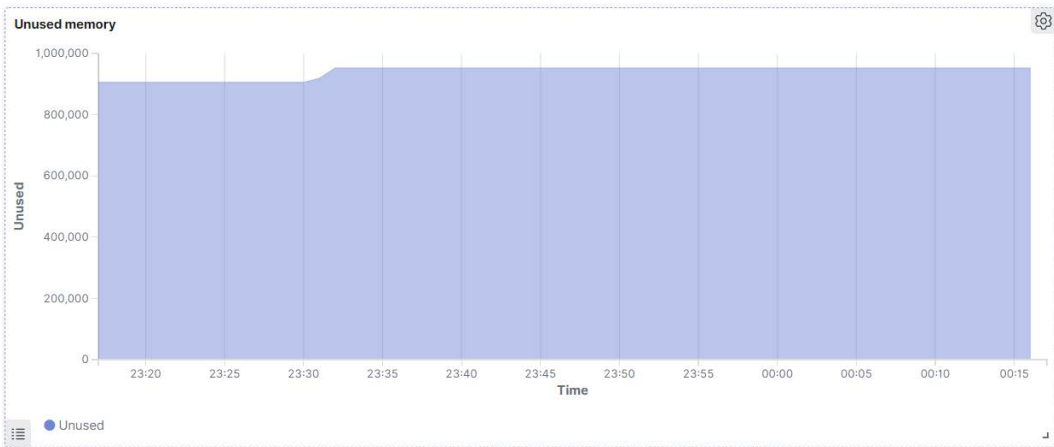


Figure 14.23 – Finished dashboard showing unused memory

8. Save this dashboard so that you can use it later. All the elements of the dashboard can be customized, and the dashboard can consist of any number of visualizations. Kibana lets you customize almost everything you see and combine a lot of data on one screen for easy monitoring. There is only one thing we need to change to make this a good monitoring dashboard, and that is to make it autorefresh. Click on the calendar icon on the right-hand side of the screen and select the auto refresh interval. We decided on 5 seconds:

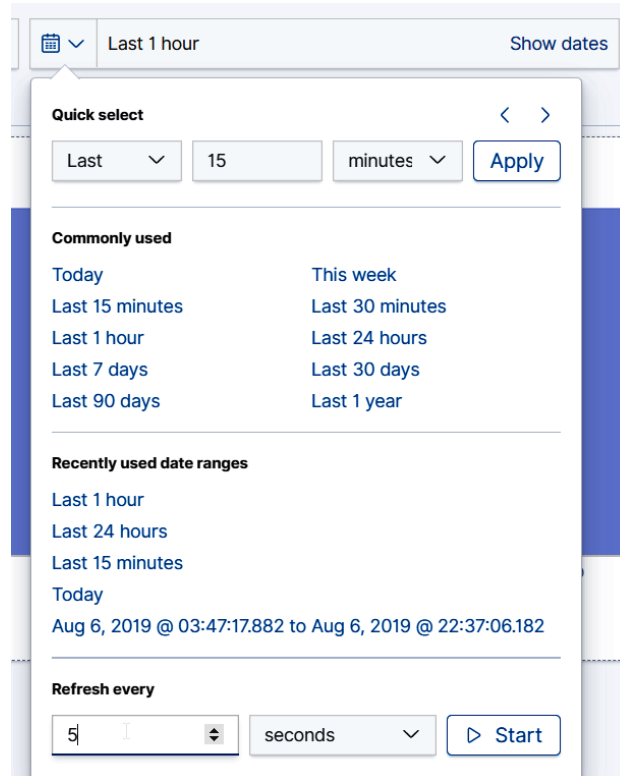


Figure 14.24 – Selecting time-related parameters

Now that we've done this, we can reflect on the fact that building this dashboard was really straightforward and easy. This only took us a couple of minutes, and it's easy to read. Imagine going through hundreds of megabytes of log files in text mode compared to this. There's really no contest, as we were able to use our previously deployed ELK stack to monitor information about KVM, which was the whole point of this chapter.

Summary

What Kibana enables you to do is create custom dashboards that can show you data for different machines side by side, so KVM is just one of the many options we have. Depending on your needs, you can display, for example, disk usage for a KVM hypervisor and all the hosts running on it, or some other metric. The Elastic stack is a flexible tool, but as with all things, it requires time to master. This chapter only covered the bare basics of Elastic configuration, so we strongly recommend further reading on this topic – alongside KVM, ELK can be used to monitor almost everything that produces any kind of data.

The next chapter is all about performance tuning and optimization for KVM virtual machines, a subject that we didn't really touch upon. There's quite a lot to be discussed – virtual machine compute sizes, optimizing performance, disks, storage access and multipathing, optimizing kernels, and virtual machine settings, just to name a few. All these subjects will be more important the larger our environment becomes.

Questions

1. What do we use metricbeat for?
2. Why do we use Kibana?
3. What is the basic prerequisite before installing the ELK stack?
4. How do we add data to Kibana?

Further reading

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

- ELK stack: <https://www.elastic.co/what-is/elk-stack>
- ELK stack documentation: <https://www.elastic.co/guide/index.html>
- Kibana documentation: <https://www.elastic.co/guide/en/kibana/current/index.html>
- Metricbeat documentation: <https://www.elastic.co/guide/en/beats/metricbeat/current/index.html>

15

Performance Tuning and Optimization for KVM VMs

When we're thinking about virtualization, there are always questions that keep coming up. Some of them might be simple enough, such as what are we going to get out of virtualization? Does it simplify things? Is it easier to back up? But there are also much more complex questions that start coming up once we've used virtualization for a while. How do we speed things up on a compute level? Is there a way to do more optimization? What can we tune additionally to get some more speed out of our storage or network? Can we introduce some configuration changes that will enable us to get more out of the existing infrastructure without investing a serious amount of money in it?

That's why performance tuning and optimization is so important to our virtualized environments. As we will find out in this chapter, there are loads of different parameters to consider – especially if we didn't design things properly from the very start, which is usually the case. So, we're going to cover the subject of design first, explain why it shouldn't be just a pure trial-and-error process, and then move on to disassembling that thought process through different devices and subsystems.

In this chapter, we will cover the following topics:

- Tuning VM CPU and memory performance – NUMA
- Kernel same-page merging
- Virtio device tuning
- Block I/O tuning
- Network I/O tuning

It's all about design

There are some fundamental patterns that we constantly repeat in many other aspects of our lives. We usually do so in IT, too. It's completely normal for us not to be good at something when we just start doing it. For example, when we start training in any kind of sport, we're usually not as good as we become after a couple of years of sticking with it. When we start musical training, we're usually much better at it after a couple of years of attending musical school. The same principle applies to IT – when we start doing IT, we're nowhere near as good at it as we become with time and – primarily – *experience*.

We as humans are really good at putting *intellectual defenses* in the way of our learning. We're really good at saying *I'm going to learn through my mistakes* – and we usually combine that with *leave me alone*.

The thing is – there's so much knowledge out there already, it would be silly not to use it. So many people already went through the same or similar process as we did; it would be a pointless exercise in futility *not* to use that experience to our advantage. Furthermore, why waste time on this whole *I'm going to learn through my mistakes* thing when we can learn much more from people with much more experience than us?

When we start using virtualization, we usually start small. For example, we start by installing a hosted-virtualization solution, such as VMware Player, Oracle VirtualBox, or something like that. Then, as time goes by, we move on to a hypervisor with a couple of **Virtual Machines (VMs)**. As the infrastructure around us grows, we start following linear patterns in trying to make infrastructure work *as it used to, when it was smaller*, which is a mistake. Nothing in IT is linear – growth, cost, the time spent on administration... absolutely nothing. It's actually rather simple to deconstruct that – as environments grow, there are more co-dependencies, which means that one thing influences another, which influences another, and so on. This endless matrix of influences is something that people often forget, especially in the design phase.

Important note:

It's really simple: linear design will get you nowhere, and proper design is the basis of performance tuning, which leaves much less work to be done on performance tuning afterward.

Earlier on in this book (in *Chapter 2, KVM as a Virtualization Solution*), we mentioned **Non-Uniform Memory Access (NUMA)**. Specifically, we mentioned that the NUMA configuration options are a *very important part of VM configuration, especially if you're designing an environment that hosts loads of virtualized servers*. Let's use a couple of examples to elaborate on this point further. These examples will give us a good basis to take a *mile-high view* of the biggest problem in performance tuning and optimization and describe how to use good design principles to get us out of many different types of trouble. We're going to use Microsoft-based solutions as examples on purpose – not because we're religious about using them, but because of a simple fact. We have a lot of widely available documentation that we can use to our advantage – design documents, best practices, shorter articles, and so on. So, let's use them.

General hardware design

Let's say that you've just started to design your new virtualized environment. When you order servers today from your channel partners – whichever they are – you need to select a model from a big list. It doesn't really matter which brand – there are a lot of models on offer. You can go with 1U (so-called *pizza box*) servers, which mostly have either one or two CPUs, depending on the model. Then, you can select a 2U server, a 3U server...the list gets exponentially bigger. Let's say that you selected a 2U server with one CPU.

In the next step, you select the amount of memory – let's say 96 GB or 128 GB. You place your order, and a couple of days or weeks later, your server gets delivered. You open it up, and you realize something – all of the RAM is connected to CPU1 memory channels. You put that in your memory bank, forget about it, and move on to the next phase.

Then, the question becomes about the micro-management of some very pedestrian settings. The BIOS version of the server, the drivers on the hypervisor level, and the BIOS settings (power management, C-states, Turbo Boost, hyperthreading, various memory-related settings, not allowing cores to turn themselves off, and so on) can have a vast influence on the performance of our VMs running on a hypervisor. Therefore, it's definitely best practice to first check whether there are any newer BIOS/firmware versions for our hardware, and check the manufacturer and other relevant documentation to make sure that the BIOS settings are as optimized as possible. Then, and only then, we can start *checkboxing* some physical and deployment procedures – deploying our server in a rack, installing an OS and everything that we need, and start using it.

Let's say that after a while, you realize that you need to do some upgrades and order some PCI Express cards – two single-port Fibre Channel 8 Gbit/s host-based adapters, two single-port 10 Gbit/s Ethernet cards, and two PCI Express NVMe SSDs. For example, by ordering these cards, you want to add some capabilities – to access Fibre Channel storage and to speed up your backup process and VM migrations by switching both of these functionalities from 1 Gbit/s to 10 Gbit/s networking. You place your order, and a couple of days or weeks later, your new PCI Express cards are delivered. You open them up, shut down your server, take it out of the rack, and install these cards. 2U servers usually have space for two or even three PCI Express riser cards, which are effectively used for connecting additional PCI Express devices. Let's say that you use the first PCI Express riser to deploy the first two cards – the Fibre Channel controllers and 10 Gbit/s Ethernet cards. Then, noticing that you don't have enough PCI Express connectors to connect everything to the first PCI Express riser, you use the second PCI Express riser to install your two PCI Express NVMe SSDs. You screw everything down, close the server cover, put the server back in your rack, and power it back on. Then, you go back to your laptop and connect to your server in a vain attempt to format your PCI Express NVMe SSDs and use them for new VM storage. You realize that your server doesn't recognize these SSDs. You ask yourself – what's going on here? Do I have a bad server?



Figure 15.1 – A PCI Express riser for DL380p G8 – you have to insert your PCI Express cards into its slots

You call up your sales rep, and tell them that you think the server is malfunctioning as it can't recognize these new SSDs. Your sales rep connects you to the pre-sales tech; you hear a small chuckle from the other side and the following information: "Well, you see, you can't do it that way. If you want to use the second PCI Express riser on your server, you have to have a CPU kit (CPU plus heatsink) in your second CPU socket, and memory for that second CPU, as well. Order these two things, put them in your server, and your PCI Express NVMe SSDs will work without any problems."

You end your phone conversation and are left with a question mark over your head – *what is going on here? Why do I need to have a second CPU and memory connected to its memory controllers to use some PCI Express cards?*

This is actually related to two things:

- You can't use the memory slots of an uninstalled CPU, as that memory needs a memory controller, which is inside the CPU.
- You can't use PCI Express on an uninstalled CPU, as the PCI Express lanes that connect PCI Express risers' cards to the CPU aren't necessarily provided by the chipset – the CPU can also be used for PCI Express lanes, and it often is, especially for the fastest connections, as you'll learn in a minute.

We know this is confusing; we can feel your pain as we've been there. Sadly, you'll have to stay with us for a little bit longer, as it gets even more confusing.

In *Chapter 4, Libvirt Networking*, we learned how to configure SR-IOV by using an Intel X540-AT2 network controller. We mentioned that we were using the HP ProLiant DL380p G8 server when configuring SR-IOV, so let's use that server for our example here, as well. If you take a look at specifications for that server, you'll notice that it uses an *Intel C600* chipset. If you then go to Intel's ARK website (<https://ark.intel.com>) and search for information about C600, you'll notice that it has five different versions (C602, C602J, C604, C606, and C608), but the most curious part of it is the fact that all of them only support eight PCI Express 2.0 lanes. Keeping in mind that the server specifications clearly state that this server supports PCI Express 3.0, it gets really confusing. How can that be and what kind of trickery is being used here? Yes, PCI Express 3.0 cards can almost always work at PCI Express 2.0 speeds, but it would be misleading at best to flat-out say that *this server supports PCI Express 3.0*, and then discover that it supports it by delivering PCI Express 2.0 levels of performance (twice as slow per PCI Express lane).

It's only when you go to the HP ProLiant DL380p G8 QuickSpecs document and find the specific part of that document (the *Expansions Slots* part, with descriptions of three different types of PCI Express risers that you can use) where all the information that we need is actually spelled out for us. Let's use all of the PCI Express riser details for reference and explanation. Basically, the primary riser has two PCI Express v3.0 slots that are provided by processor 1 (x16 plus x8), and the third slot (PCI Express 2.0 x8) is provided by the chipset. For the optional riser, it says that all of the slots are provided by the CPU (x16 plus x8 times two). There are actually some models that can have three PCI Express risers, and for that third riser, all of the PCI Express lanes (x16 times two) are also provided by processor 2.

This is all *very important*. It's a huge factor in performance bottlenecks for many scenarios, which is why we centered our example around the idea of two PCI Express NVMe SSDs. We wanted to go through the whole journey with you.

So, at this point, we can have an educated discussion about what should be the de facto standard hardware design of our example server. If our intention is to use these PCI Express NVMe SSDs for local storage for our VMs, then most of us would treat that as a priority. That would mean that we'd absolutely want to connect these devices to the PCI Express 3.0 slot so that they aren't bottlenecked by PCI Express 2.0 speeds. If we have two CPUs, we're probably better off using the *first PCI Express slot* in both of our PCI Express risers for that specific purpose. The reasoning is simple – they're *PCI Express 3.0 compatible* and they're *provided by the CPU*. Again, that's *very important* – it means that they're *directly connected* to the CPU, without the *added latency* of going through the chipset. Because, at the end of the day, the CPU is the central hub for everything, and data going from VMs to SSDs and back will go through the CPU. From a design standpoint, we should absolutely use the fact that we know this to our advantage and connect our PCI Express NVMe SSDs *locally* to our CPUs.

The next step is related to Fibre Channel controllers and 10 Gbit/s Ethernet controllers. The vast load of 8 Gbit/s Fibre Channel controllers are PCI Express 2.0 compatible. The same thing applies to 10 Gbit/s Ethernet adapters. So, it's again a matter of priority. If you're using Fibre Channel storage a lot from our example server, logic dictates that you'd want to put your new and shiny Fibre Channel controllers in the fastest possible place. That would be the second PCI Express slot in both of our PCI Express risers. Again, second PCI Express slots are both provided by CPUs – processor 1 and processor 2. So now, we're just left with 10 Gbit/s Ethernet adapters. We said in our example scenario that we're going to be using these adapters for backup and VM migration. The backup won't suffer all that much if it's done via a network adapter that's on the chipset. VM migration might be a tad sensitive to that. So, you connect your first 10 Gbit/s Ethernet adapter to the third PCI Express slot on the primary riser (for backup, provided by the chipset). Then, you also connect your second 10 Gbit/s Ethernet adapter to the third PCI Express slot on the secondary riser (PCI Express lanes provided by processor 2).

We've barely started on the subject of design with the hardware aspect of it, and already we have such a wealth of information to process. Let's now move on to the second phase of our design – which relates to VM design. Specifically, we're going to discuss how to create new VMs that are designed properly from scratch. However, if we're going to do that, we need to know which application this VM is going to be created for. For that matter, we're going to create a scenario. We're going to use a VM that we're creating to host a node in a Microsoft SQL database cluster on top of a VM running Windows Server 2019. The VM will be installed on a KVM host, of course. This is a task given to us by a client. As we already did the general hardware design, we're going to focus on VM design now.

VM design

Creating a VM is easy – we can just go to `virt-manager`, click a couple of times, and we're done. The same applies to oVirt, RedHat Enterprise Virtualization Manager, OpenStack, VMware, and Microsoft virtualization solutions... it's more or less the same everywhere. The problem is designing VMs properly. Specifically, the problem is creating a VM that's going to be pre-tuned to run an application on a very high level, which then only leaves a small number of configuration steps that we can take on the server or VM side to improve performance – the premise being that most of the optimization process later will be done on the OS or application level.

So, people usually start creating a VM in one of two ways – either by creating a VM from scratch with XYZ amount of resources added to the VM, or by using a template, which – as we explained in *Chapter 8, Creating and Modifying VM Disks, Templates, and Snapshots* – will save a lot of time. Whichever way we use, there's a certain amount of resources that will be configured for our VM. We then remember what we're going to use this VM for (SQL), so we increase the amount of CPUs to, for example, four, and the amount of memory to 16 GB. We put that VM in the local storage of our server, pool it up, and start deploying updates, configuring the network, and rebooting and generally preparing the VM for the final installation step, which is actually installing our application (SQL Server 2016) and some updates to go along with it. After we're done with that, we start creating our databases and move on to the next set of tasks that need to be done.

Let's take a look at this process from a design and tuning perspective next.

Tuning the VM CPU and memory performance

There are some pretty straightforward issues with the aforementioned process. Some are just engineering issues, while some are more procedural issues. Let's discuss them for a second:

- There is no *one-size-fits-all* solution to almost anything in IT. Every VM of every single client has a different set of circumstances and is in a different environment that consists of different devices, servers, and so on. Don't try to speed up the process to *impress* someone, as it will most definitely become a problem later.
- When you're done with deployment, stop. Learn the practice of breathe in, breathe out, and stop for a second and think – or wait for an hour or even a day. Remember what you're designing a VM for.
- Before allowing a VM to be used in production, check its configuration. The number of virtual CPUs, the memory, the storage placement, the network options, the drivers, software updates – everything.

- A lot of pre-configuration can be done before the installation phase or during the template phase, before you clone the VM. If it's an existing environment that you're migrating to a new one, *collect information about the old environment*. Find out what the database sizes are, what storage is being used, and how happy people are with the performance of their database server and the applications using them.

At the end of the whole process, learn to take a *mile-high perspective* on the IT-related work that you do. From a quality assurance standpoint, IT should be a highly structured, procedural type of work. If you've done something before, learn to document the things that you did while installing things and the changes that you made. Documentation – as it stands now – is one of the biggest Achilles' heels of IT. Writing documentation will make it easier for you to repeat the process in the future when faced with the same (less often) or a similar (much more often) scenario. Learn from the greats – just as an example, we would know much less about Beethoven, for example, if he didn't keep detailed notes of the things he did day in, day out. Yes, he was born in 1770 and this year will mark 250 years since he was born, and that was a long time ago, but that doesn't mean that 250-year-old routines are bad.

So now, your VM is configured and in production, and a couple of days or weeks later, you get a call from the company and they ask why the performance is *not all that great*. Why isn't it working just like on a physical server?

As a rule of thumb, when you're looking for performance issues on Microsoft SQL, they can be roughly divided into four categories:

- Your SQL database is memory-limited.
- Your SQL database is storage-limited.
- Your SQL database is just misconfigured.
- Your SQL database is CPU-limited.

In our experience, the first and second category can easily account for 80–85% of SQL performance issues. The third would probably account for 10%, while the last one is rather rare, but it still happens. Keeping that in mind, from an infrastructure standpoint, when you're designing a database VM, you should always look into VM memory and storage configuration first, as they are by far the most common reasons. The problems just kind of accumulate and snowball from there. Specifically, some of the most common key reasons for sub-par SQL VM performance is the memory location, looking at it from a CPU perspective, and storage issues – latencies/IOPS and bandwidth being the problem. So, let's describe these one by one.

The first issue that we need to tackle is related to – funnily enough – *geography*. It's very important for a database to have its memory content as close as possible to the CPU cores assigned to its VMs. This is what NUMA is all about. We can easily overcome this specific issue on KVM with a bit of configuration. Let's say that we chose that our VM uses four virtual CPUs. Our test server has Intel Xeon E5-2660v2 processors, which have 10 physical cores each. Keeping in mind that our server has two of these Xeon processors, we have 20 cores at our disposal overall.

We have two basic questions to answer:

- How do these four cores for our VM correlate to 20 physical cores below?
- How does that relate to the VM's memory and how can we optimize that?

The answer to both of these questions is that it depends on *our* configuration. By default, our VM might use two cores from two physical processors each and spread itself in terms of memory across both of them or 3+1. None of these configuration examples are good. What you want is to have all the virtual CPU cores on *one* physical processor, and you want those virtual CPU cores to use memory that's local to those four physical cores – directly connected to the underlying physical processor's memory controller. What we just described is the basic idea behind NUMA – to have nodes (consisting of CPU cores) that act as building compute blocks for your VMs with local memory.

If at all possible, you want to reserve all the memory for that VM so that it doesn't swap somewhere outside of the VM. In KVM, that *outside of the VM* would be in the KVM host swap space. Having access to real RAM memory all of the time is a performance and SLA-related configuration option. If the VM uses a bit of underlying swap partition that acts as its memory, it will not have the same performance. Remember, swapping is usually done on some sort of local RAID array, an SD card, or a similar medium, which are many orders of magnitude slower in terms of bandwidth and latency compared to real RAM memory. If you want a high-level statement about this – avoid memory overcommitment on KVM hosts at all costs. The same goes for the CPU, and this is a commonly used best practice on any other kind of virtualization solution, not just on KVM.

Furthermore, for critical resources, such as a database VM, it definitely makes sense to *pin* vCPUs to specific physical cores. That means that we can use specific physical cores to run a VM, and we should configure other VMs running on the same host *not* to use those cores. That way, we're *reserving* these CPU cores specifically for a single VM, thus configuring everything for maximum performance not to be influenced by other VMs running on the physical server.

Yes, sometimes managers and company owners won't like you because of this best practice (as if you're to blame), as it requires proper planning and enough resources. But that's something that they have to live with – or not, whichever they prefer. Our job is to make the IT system run as best as it possibly can.

VM design has its basic principles, such as the CPU and memory design, NUMA configuration, configuring devices, storage and network configuration, and so on. Let's go through all of these topics step by step, starting with an advanced CPU-based feature that can really help make our systems run as best as possible if used properly – CPU pinning.

CPU pinning

CPU pinning is nothing but the process of setting the *affinity* between the vCPU and the physical CPU core of the host so that the vCPU will be executing on that physical CPU core only. We can use the `virsh vcpupin` command to bind a vCPU to a physical CPU core or to a subset of physical CPU cores.

There are a couple of best practices when doing vCPU pinning:

- If the number of guest vCPUs is more than the single NUMA node CPUs, don't go for the default pinning option.
- If the physical CPUs are spread across different NUMA nodes, it is always better to create multiple guests and pin the vCPUs of each guest to physical CPUs in the same NUMA node. This is because accessing different NUMA nodes, or running across multiple NUMA nodes, has a negative impact on performance, especially for memory-intensive applications.

Let's look at the steps of vCPU pinning:

1. Execute `virsh nodeinfo` to gather details about the host CPU configuration:

```
[root@packtphy02 ~]# virsh nodeinfo
CPU model:      x86_64
CPU(s):         10
CPU frequency:  2099 MHz
CPU socket(s):  1
Core(s) per socket: 10
Thread(s) per core: 1
NUMA cell(s):  1
Memory size:    4193784 KiB
```

Figure 15.2 – Information about our KVM node

- The next step is to get the CPU topology by executing the `virsh capabilities` command and check the section tagged `<topology>`:

```

root@packtphy02:~
File Edit View Search Terminal Help
<topology>
  <cells num='1'>
    <cell id='0'>
      <memory unit='KiB'>4193784</memory>
      <pages unit='KiB' size='4'>1048446</pages>
      <pages unit='KiB' size='2048'>0</pages>
      <pages unit='KiB' size='1048576'>0</pages>
      <distances>
        <sibling id='0' value='10' />
      </distances>
      <cpus num='10'>
        <cpu id='0' socket_id='0' core_id='0' siblings='0' />
        <cpu id='1' socket_id='0' core_id='1' siblings='1' />
        <cpu id='2' socket_id='0' core_id='2' siblings='2' />
        <cpu id='3' socket_id='0' core_id='3' siblings='3' />
        <cpu id='4' socket_id='0' core_id='4' siblings='4' />
        <cpu id='5' socket_id='0' core_id='5' siblings='5' />
        <cpu id='6' socket_id='0' core_id='6' siblings='6' />
        <cpu id='7' socket_id='0' core_id='7' siblings='7' />
        <cpu id='8' socket_id='0' core_id='8' siblings='8' />
        <cpu id='9' socket_id='0' core_id='9' siblings='9' />
      </cpus>
    </cell>
  </cells>
</topology>

```

Figure 15.3 – The `virsh capabilities` output with all the visible physical CPU cores

Once we have identified the topology of our host, the next step is to start pinning the vCPUs.

- Let's first check the current affinity or pinning configuration with the guest named `SQLForNuma`, which has four vCPUs:

```

[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma
VCPU: CPU Affinity
-----
 0: 0-9
 1: 0-9
 2: 0-9
 3: 0-9
[root@PacktPhy02 ~]#

```

Figure 15.4 – Checking the default `vcpupin` settings

Let's change that by using CPU pinning.

4. Let's pin vCPU0 to physical core 0, vCPU1 to physical core 1, vCPU2 to physical core 2, and vCPU3 to physical core 3:

```
[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma 0 0
[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma 1 1
[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma 2 2
[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma 3 3
[root@PacktPhy02 ~]# virsh vcpupin SQLForNuma
VCPU: CPU Affinity
-----
 0: 0
 1: 1
 2: 2
 3: 3
[root@PacktPhy02 ~]#
```

Figure 15.5 – Configuring CPU pinning

By using `virsh vcpupin`, we changed a fixed virtual CPU allocation for this VM.

5. Let's use `virsh dumpxml` on this VM to check the configuration change:

```
<vcpu placement='static'>4</vcpu>
<cputune>
  <vcpupin vcpu='0' cpuset='0' />
  <vcpupin vcpu='1' cpuset='1' />
  <vcpupin vcpu='2' cpuset='2' />
  <vcpupin vcpu='3' cpuset='3' />
</cputune>
```

Figure 15.6 – CPU pinning VM configuration changes

Notice the CPU affinity listed in the `virsh` command and the `<cputune>` tag in the XML dump of the running guest. As the XML tag says, this comes under the CPU tuning section of the guest. It is also possible to configure a set of physical CPUs for a particular vCPU instead of a single physical CPU.

There are a couple of things to remember. vCPU pinning can improve performance; however, this depends on the host configuration and the other settings on the system. Make sure you do enough tests and validate the settings.

You can also make use of `virsh vcpuinfo` to verify the pinning. The output of the `virsh vcpuinfo` command is as follows:

```
root@packtphy02:~  
File Edit View Search Terminal Help  
[root@PacktPhy02 ~]# virsh vcpuinfo SQLForNuma  
VCPU:      0  
CPU:       0  
State:     running  
CPU time:  2.8s  
CPU Affinity: y-----  
  
VCPU:      1  
CPU:       1  
State:     running  
CPU time:  0.0s  
CPU Affinity: -y-----  
  
VCPU:      2  
CPU:       2  
State:     running  
CPU time:  0.0s  
CPU Affinity: --y-----  
  
VCPU:      3  
CPU:       3  
State:     running  
CPU time:  0.0s  
CPU Affinity: ---y-----  
[root@PacktPhy02 ~]#
```

Figure 15.7 – `virsh vcpuinfo` for our VM

If we're doing this on a busy host, it will have consequences. Sometimes, we literally won't be able to start our SQL machine because of these settings. So, for the greater good (the SQL VM working instead of not wanting to start), we can change the memory mode configuration from `strict` to `interleave` or `preferred`, which will relax the insistence on using strictly local memory for this VM.

Let's now explore the memory tuning options as they are the next logical thing to discuss.

Working with memory

Memory is a precious resource for most environments, isn't it? Thus, the efficient use of memory should be achieved by tuning it. The first rule in optimizing KVM memory performance is not to allocate more resources to a guest during setup than it will use.

We will discuss the following in greater detail:

- Memory allocation
- Memory tuning
- Memory backing

Let's start by explaining how to configure memory allocation for a virtual system or guest.

Memory allocation

To make the allocation process simple, we will consider the `virt-manager` libvirt client again. Memory allocation can be done from the window shown in the following screenshot:

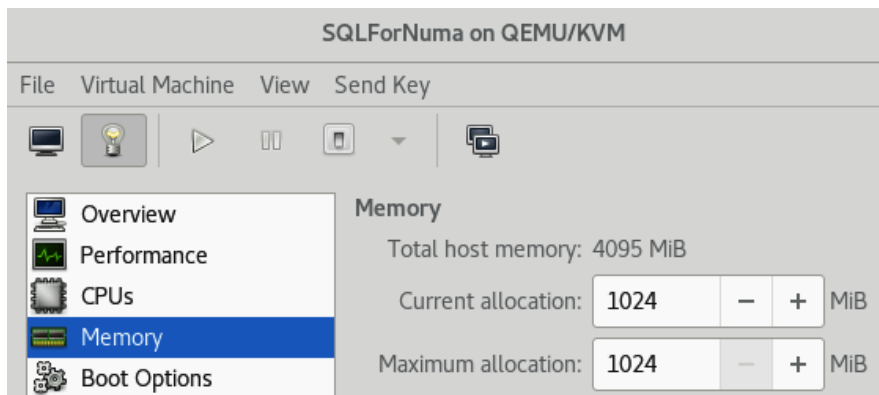


Figure 15.8 – VM memory options

As you can see in the preceding screenshot, there are two main options: **Current allocation** and **Maximum allocation**:

- **Maximum allocation:** The runtime maximum memory allocation of the guest. This is the maximum memory that can be allocated to the guest when it's running.
- **Current allocation:** How much memory a guest always uses. For memory ballooning reasons, we can have this value lower than the maximum.

The `virsh` command can be used to tune these parameters. The relevant `virsh` command options are `setmem` and `setmaxmem`.

Memory tuning

The memory tuning options are added under `<memtune>` of the guest configuration file.

Additional memory tuning options can be found at <http://libvirt.org/formatdomain.html#elementsMemoryTuning>.

The admin can configure the memory settings of a guest manually. If the `<memtune>` configuration is omitted, the default memory settings apply for a guest. The `virsh` command at play here is as follows:

```
# virsh memtune <virtual_machine> --parameter size parameter
```

It can have any of the following values; this best practice is well documented in the man page:

<code>--hard-limit</code>	The maximum memory the guest can use.
<code>--soft-limit</code>	The memory limit to enforce during memory contention.
<code>--swap-hard-limit</code>	The maximum memory plus swap the guest can use. This has to be more than hard-limit value provided.
<code>--min-guarantee</code>	The guaranteed minimum memory allocation for the guest.

The default/current values that are set for the `memtune` parameter can be fetched as shown:

```

root@packtphy02:~
File Edit View Search Terminal Help
[root@PacktPhy02 ~]# virsh list
  Id   Name           State
  ----  -----
  5    SQLForNuma     running

[root@PacktPhy02 ~]# virsh memtune SQLForNuma
hard_limit      : unlimited
soft_limit      : unlimited
swap_hard_limit: unlimited

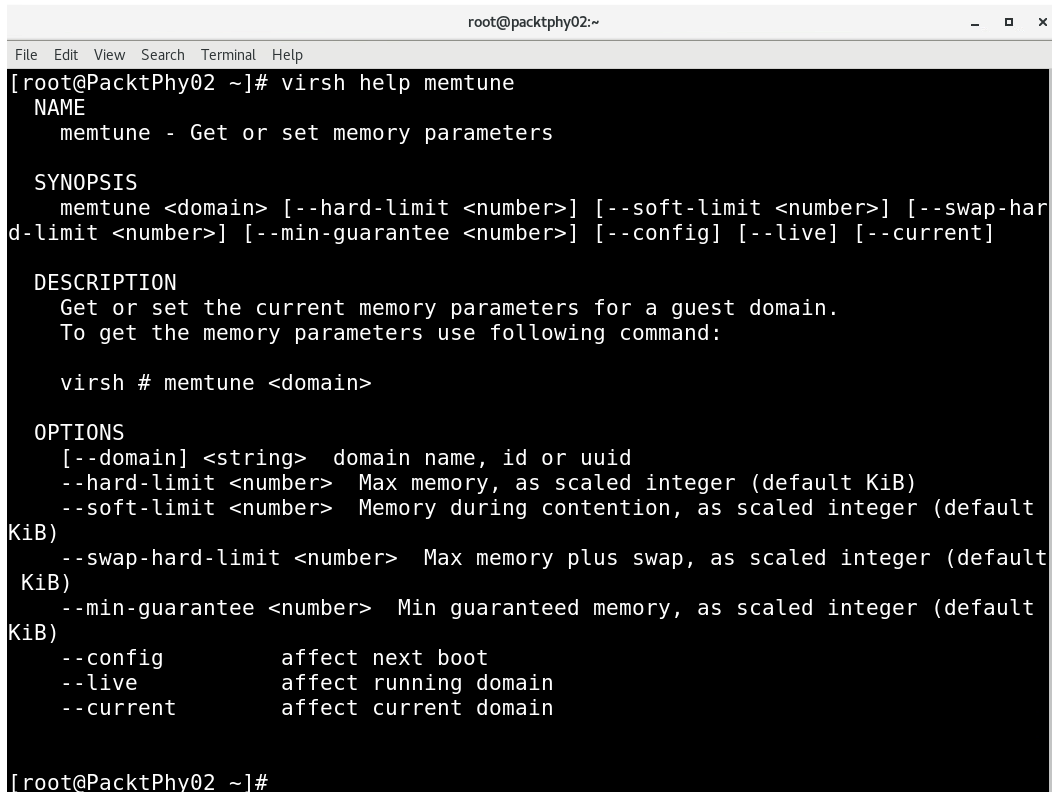
[root@PacktPhy02 ~]#

```

Figure 15.9 – Checking the `memtune` settings for the VM

When setting `hard_limit`, you should not set this value too low. This might lead to a situation in which a VM is terminated by the kernel. That's why determining the correct amount of resources for a VM (or any other process) is such a design problem. Sometimes, designing things properly seems like dark arts.

To learn more about how to set these parameters, please see the help output for the `memtune` command in the following screenshot:



```
root@packtphy02:~  
File Edit View Search Terminal Help  
[root@PacktPhy02 ~]# virsh help memtune  
NAME  
  memtune - Get or set memory parameters  
  
SYNOPSIS  
  memtune <domain> [--hard-limit <number>] [--soft-limit <number>] [--swap-hard-limit <number>] [--min-guarantee <number>] [--config] [--live] [--current]  
  
DESCRIPTION  
  Get or set the current memory parameters for a guest domain.  
  To get the memory parameters use following command:  
  
  virsh # memtune <domain>  
  
OPTIONS  
  [--domain] <string> domain name, id or uuid  
  --hard-limit <number> Max memory, as scaled integer (default KiB)  
  --soft-limit <number> Memory during contention, as scaled integer (default KiB)  
  --swap-hard-limit <number> Max memory plus swap, as scaled integer (default KiB)  
  --min-guarantee <number> Min guaranteed memory, as scaled integer (default KiB)  
  --config          affect next boot  
  --live           affect running domain  
  --current        affect current domain  
  
[root@PacktPhy02 ~]#
```

Figure 15.10 – Checking `virsh help memtune`

As we have covered memory allocation and tuning, the final option is memory backing.

Memory backing

The following is the guest XML representation of memory backing:

```
<domain>    ...
  <memoryBacking>
    <hugepages>
      <page size="1" unit="G" nodeset="0-3,5"/>
      <page size="2" unit="M" nodeset="4"/>
    </hugepages>
    <nosharepages/>
    <locked/>
  </memoryBacking>    ...
</domain>
```

You may have noticed that there are three main options for memory backing: `locked`, `nosharepages`, and `hugepages`. Let's go through them one by one, starting with `locked`.

locked

In KVM virtualization, guest memory lies in the process address space of the `qemu-kvm` process in the KVM host. These guest memory pages can be swapped out by the Linux kernel at any time, based on the requirement that the host has, and this is where `locked` can help. If you set the memory backing option of the guest to `locked`, the host will not swap out memory pages that belong to the virtual system or guest. The virtual memory pages in the host system memory are locked when this option is enabled:

```
<memoryBacking>
  <locked/>
</memoryBacking>
```

We need to use `<memtune>` to set `hard_limit`. The calculus is simple – whatever the amount of memory for the guest we need plus overhead.

nosharepages

The following is the XML representation of `nosharepages` from the guest configuration file:

```
<memoryBacking>
  <nosharepages/>
</memoryBacking>
```

There are different mechanisms that can enable the sharing of memory when the memory pages are identical. Techniques such as **Kernel Same-Page Merging (KSM)** share pages among guest systems. The `nosharepages` option instructs the hypervisor to disable shared pages for this guest – that is, setting this option will prevent the host from deduplicating memory between guests.

hugepages

The third and final option is `hugepages`, which can be represented in XML format, as follows:

```
<memoryBacking>
</hugepages>
</memoryBacking>
```

HugePages were introduced in the Linux kernel to improve the performance of memory management. Memory is managed in blocks known as pages. Different architectures (i386, ia64) support different page sizes. We don't necessarily have to use the default setting for x86 CPUs (4 KB memory pages), as we can use larger memory pages (2 MB to 1 GB), a feature that's called HugePages. A part of the CPU called the **Memory Management Unit (MMU)** manages these pages by using a list. The pages are referenced through page tables, and each page has a reference in the page table. When a system wants to handle a huge amount of memory, there are mainly two options. One of them involves increasing the number of page table entries in the hardware MMU. The second method increases the default page size. If we opt for the first method of increasing the page table entries, it is really expensive.

The second and more efficient method when dealing with large amounts of memory is using HugePages or increased page sizes by using HugePages. The different amounts of memory that each and every server has means that there is a need for different page sizes. The default values are okay for most situations, while huge memory pages (for example, 1 GB) are more efficient if we have large amounts of memory (hundreds of gigabytes or even terabytes). This means less *administrative* work in terms of referencing memory pages and more time spent actually getting the content of these memory pages, which can lead to a significant performance boost. Most of the known Linux distributions can use HugePages to manage large memory amounts. A process can use HugePages memory support to improve performance by increasing the CPU cache hits against the **Translation LookAside Buffer (TLB)**, as explained in *Chapter 2, KVM as a Virtualization Solution*. You already know that guest systems are simply processes in a Linux system, thus the KVM guests are eligible to do the same.

Before we move on, we should also mention **Transparent HugePages (THP)**. THP is an abstraction layer that automates the HugePages size allocation based on the application request. THP support can be entirely disabled, can only be enabled inside `MADV_HUGEPAGE` regions (to avoid the risk of consuming more memory resources), or enabled system-wide. There are three main options for configuring THP in a system: `always`, `madvise`, and `never`:

```
# cat /sys/kernel/mm/transparent_hugepage/enabled [always]
madvise never
```

From the preceding output, we can see that the current THP setting in our server is `madvise`. Other options can be enabled by using one of the following commands:

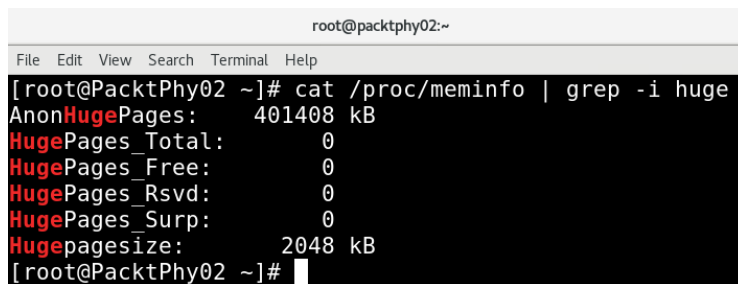
```
echo always >/sys/kernel/mm/transparent_hugepage/enabled
echo madvise >/sys/kernel/mm/transparent_hugepage/enabled
echo never >/sys/kernel/mm/transparent_hugepage/enabled
```

In short, what these values mean is the following:

- `always`: Always use THP.
- `madvise`: Use HugePages only in **Virtual Memory Areas (VMAs)** marked with `MADV_HUGEPAGE`.
- `never`: Disable the feature.

The system settings for performance are automatically optimized by THP. We can have performance benefits by using memory as cache. It is possible to use static HugePages when THP is in place or in other words THP won't prevent it from using a static method. If we don't configure our KVM hypervisor to use static HugePages, it will use 4 Kb transparent HugePages. The advantages we get from using HugePages for a KVM guest's memory are that less memory is used for page tables and TLB misses are reduced; obviously, this increases performance. But keep in mind that when using HugePages for guest memory, you can no longer swap or balloon guest memory.

Let's have a quick look at how to use static HugePages in your KVM setup. First, let's check the current system configuration – it's clear that the HugePages size in this system is currently set at 2 MB:



```

root@packtphy02:~
File Edit View Search Terminal Help
[root@PacktPhy02 ~]# cat /proc/meminfo | grep -i huge
AnonHugePages:      401408 kB
HugePages_Total:    0
HugePages_Free:     0
HugePages_Rsvd:     0
HugePages_Surp:     0
Hugepagesize:       2048 kB
[root@PacktPhy02 ~]#

```

Figure 15.11 – Checking the HugePages settings

We're primarily talking about all the attributes starting with HugePages, but it's worth mentioning what the AnonHugePages attribute is. The AnonHugePages attribute tells us the current THP usage on the system level.

Now, let's configure KVM to use a custom HugePages size:

1. View the current explicit hugepages value by running the following command or fetch it from `sysfs`, as shown:

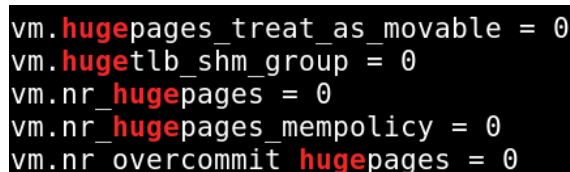


```

# cat /proc/sys/vm/nr_hugepages
0

```

2. We can also use the `sysctl -a | grep huge` command:



```

vm.hugepages_treat_as_movable = 0
vm.hugeltlb_shm_group = 0
vm.nr_hugepages = 0
vm.nr_hugepages_mempolicy = 0
vm.nr_overcommit_hugepages = 0

```

Figure 15.12 – The sysctl hugepages settings

3. As the HugePage size is 2 MB, we can set hugepages in increments of 2 MB. To set the number of hugepages to 2,000, use the following command:

```
# echo 2000 > /proc/sys/vm/nr_hugepages
```

The total memory assigned for hugepages cannot be used by applications that are not hugepage-aware – that is, if you over-allocate hugepages, normal operations of the host system can be affected. In our examples, 2048*2 MB would equal 4,096 MB of memory, which we should have available when we do this configuration.

4. We need to tell the system that this type of configuration is actually OK and configure `/etc/security/limits.conf` to reflect that. Otherwise, the system might refuse to give us access to 2,048 hugepages times 2 MB of memory. We need to add two lines to that file:

```
soft memlock <value>
```

```
hard memlock <value>
```

The `<value>` parameter will depend on the configuration we want to do. If we want to configure everything according to our 2048*2 MB example, `<value>` would be 4,194,304 (or $4096*1024$).

5. To make it persistent, you can use the following:

```
# sysctl -w vm.nr_hugepages=<number of hugepages>
```

6. Then, mount the `fs` hugepages, reconfigure the VM, and restart the host:

```
# mount -t hugetlbfs hugetlbfs /dev/hugepages
```

Reconfigure the HugePage-configured guest by adding the following settings in the VM configuration file:

```
<memoryBacking>
```

```
</hugepages>
```

```
</ memoryBacking>
```

It's time to shut down the VM and reboot the host. Inside the VM, do the following:

```
# systemctl poweroff
```

On the host, do the following:

```
# systemctl reboot
```

After the host reboot and the restart of the VM, it will now start using the hugepages.

The next topic is related to sharing memory content between multiple VMs, referred to as KSM. This technology is heavily used to *save* memory. At any given moment, when multiple VMs are powered on the virtualization host, there's a big statistical chance that those VMs have blocks of memory contents that are the same (they have the same contents). Then, there's no reason to store the same contents multiple times. Usually, we refer to KSM as a deduplication process being applied to memory. Let's learn how to use and configure KSM.

Getting acquainted with KSM

KSM is a feature that allows the sharing of identical pages between the different processes running on a system. We might presume that the identical pages exist due to certain reasons—for example, if there are multiple processes spawned from the same binary or something similar. There is no rule such as this though. KSM scans these identical memory pages and consolidates a **Copy-on-Write (COW)** shared page. COW is nothing but a mechanism where when there is an attempt to change a memory region that is shared and common to more than one process, the process that requests the change gets a new copy and the changes are saved in it.

Even though the consolidated COW shared page is accessible by all the processes, whenever a process tries to change the content (write to that page), the process gets a new copy with all of the changes. By now, you will have understood that, by using KSM, we can reduce physical memory consumption. In the KVM context, this can really add value, because guest systems are `qemu-kvm` processes in the system, and there is a huge possibility that all the VM processes will have a good amount of similar memory.

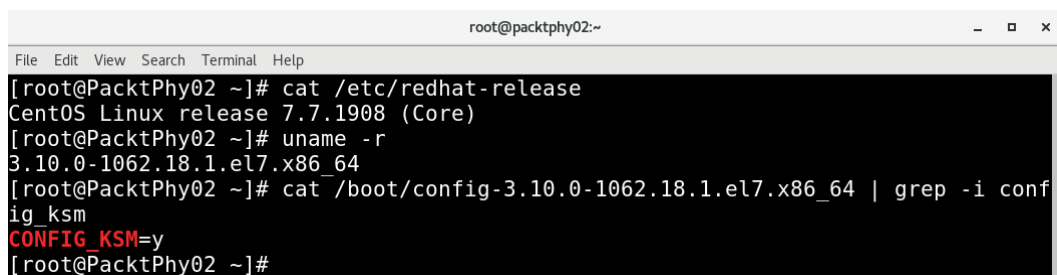
For KSM to work, the process/application has to register its memory pages with KSM. In KVM-land, KSM allows guests to share identical memory pages, thus achieving an improvement in memory consumption. That might be some kind of application data, a library, or anything else that's used frequently. This shared page or memory is marked as `copy on write`. In short, KSM avoids memory duplication and it's really useful when similar guest OSes are present in a KVM environment.

By using the theory of prediction, KSM can provide enhanced memory speed and utilization. Mostly, this common shared data is stored in cache or main memory, which causes fewer cache misses for the KVM guests. Also, KSM can reduce the overall guest memory footprint so that, in a way, it allows the user to do memory overcommitting in a KVM setup, thus supplying the greater utilization of available resources. However, we have to keep in mind that KSM requires more CPU resources to identify the duplicate pages and to perform tasks such as sharing/merging.

Previously, we mentioned that the processes have to mark the *pages* to show that they are eligible candidates for KSM to operate. The marking can be done by a process based on the `MADV_MERGEABLE` flag, which we will discuss in the next section. You can explore the use of this flag in the `madvise` man page:

```
# man 2 madvise
MADV_MERGEABLE (since Linux 2.6.32)
Enable Kernel Samepage Merging (KSM) for the pages in the
range specified by addr and length. The kernel regularly scans
those areas of user memory that have been marked as mergeable,
looking for pages with identical content. These are replaced
by a single write-protected page (that is automatically copied
if a process later wants to update the content of the page).
KSM merges only private anonymous pages (see mmap(2)).
The KSM feature is intended for applications that generate many
instances of the same data (e.g., virtualization systems such
as KVM). It can consume a lot of processing power; use with
care. See the Linux kernel source file Documentation/ vm/ksm.
txt for more details.
The MADV_MERGEABLE and MADV_UNMERGEABLE operations are
available only if the kernel was configured with CONFIG_KSM.
```

So, the kernel has to be configured with KSM, as follows:



```
root@packtphy02:~
File Edit View Search Terminal Help
[root@PacktPhy02 ~]# cat /etc/redhat-release
CentOS Linux release 7.7.1908 (Core)
[root@PacktPhy02 ~]# uname -r
3.10.0-1062.18.1.el7.x86_64
[root@PacktPhy02 ~]# cat /boot/config-3.10.0-1062.18.1.el7.x86_64 | grep -i conf
ig_ksm
CONFIG_KSM=y
[root@PacktPhy02 ~]#
```

Figure 15.13 – Checking the KSM settings

KSM gets deployed as a part of the `qemu-kvm` package. Information about the KSM service can be fetched from the `sysfs` filesystem, in the `/sys` directory. There are different files available in this location, reflecting the current KSM status. These are updated dynamically by the kernel, and it has a precise record of the KSM usage and statistics:

```
root@packtphy02:~  
File Edit View Search Terminal Help  
[root@PacktPhy02 ~]# ls /sys/kernel/mm/ksm/*  
/sys/kernel/mm/ksm/full_scans  
/sys/kernel/mm/ksm/max_page_sharing  
/sys/kernel/mm/ksm/merge_across_nodes  
/sys/kernel/mm/ksm/pages_shared  
/sys/kernel/mm/ksm/pages_sharing  
/sys/kernel/mm/ksm/pages_to_scan  
/sys/kernel/mm/ksm/pages_unshared  
/sys/kernel/mm/ksm/pages_volatile  
/sys/kernel/mm/ksm/run  
/sys/kernel/mm/ksm/sleep_millisecs  
/sys/kernel/mm/ksm/stable_node_chains  
/sys/kernel/mm/ksm/stable_node_chains_prune_millisecs  
/sys/kernel/mm/ksm/stable_node_dups  
[root@PacktPhy02 ~]#
```

Figure 15.14 – The KSM settings in `sysfs`

In an upcoming section, we will discuss the `ksmtuned` service and its configuration variables. As `ksmtuned` is a service to control KSM, its configuration variables are analogous to the files we see in the `sysfs` filesystem. For more details, you can check out <https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html>.

It is also possible to tune these parameters with the `virsh` command. The `virsh node-memory-tune` command does this job for us. For example, the following command specifies the number of pages to scan before the shared memory service goes to sleep:

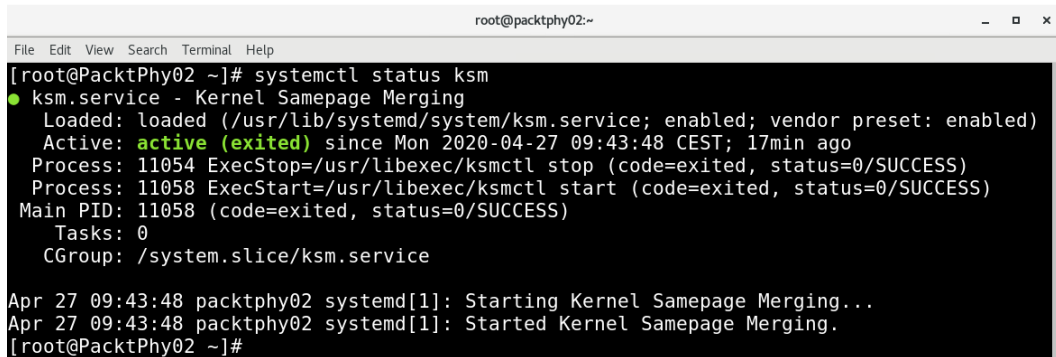
```
# virsh node-memory-tune --shm-pages-to-scan number
```

As with any other service, the `ksmtuned` service also has logs stored in a log file, `/var/log/ksmtuned`. If we add `DEBUG=1` to `/etc/ksmtuned.conf`, we will have logging from any kind of KSM tuning actions. Refer to <https://www.kernel.org/doc/Documentation/vm/ksm.txt> for more details.

Once we start the KSM service, as shown next, you can watch the values change depending on the KSM service in action:

```
# systemctl start ksm
```

We can then check the status of the `ksm` service like this:



```

root@packtphy02:~
File Edit View Search Terminal Help
[root@PacktPhy02 ~]# systemctl status ksm
● ksm.service - Kernel Samepage Merging
   Loaded: loaded (/usr/lib/systemd/system/ksm.service; enabled; vendor preset: enabled)
   Active: active (exited) since Mon 2020-04-27 09:43:48 CEST; 17min ago
   Process: 11054 ExecStop=/usr/libexec/ksmctl stop (code=exited, status=0/SUCCESS)
   Process: 11058 ExecStart=/usr/libexec/ksmctl start (code=exited, status=0/SUCCESS)
  Main PID: 11058 (code=exited, status=0/SUCCESS)
     Tasks: 0
    CGroup: /system.slice/ksm.service

Apr 27 09:43:48 packtphy02 systemd[1]: Starting Kernel Samepage Merging...
Apr 27 09:43:48 packtphy02 systemd[1]: Started Kernel Samepage Merging.
[root@PacktPhy02 ~]#

```

Figure 15.15 – The `ksm` service command and the `ps` command output

Once the KSM service is started and we have multiple VMs running on our host, we can check the changes by querying `sysfs` by using the following command multiple times:

```
cat /sys/kernel/mm/ksm/*
```

Let's explore the `ksmtuned` service in more detail. The `ksmtuned` service is designed so that it goes through a cycle of actions and adjusts KSM. This cycle of actions continues its work in a loop. Whenever a guest system is created or destroyed, `libvirt` will notify the `ksmtuned` service.

The `/etc/ksmtuned.conf` file is the configuration file for the `ksmtuned` service. Here is a brief explanation of the configuration parameters available. You can see these configuration parameters match with the KSM files in `sysfs`:

```

# Configuration file for ksmtuned.
# How long ksmtuned should sleep between tuning adjustments
# KSM_MONITOR_INTERVAL=60
# Millisecond sleep between ksm scans for 16Gb server.
# Smaller servers sleep more, bigger sleep less.
# KSM_SLEEP_MSEC=10

```

```
# KSM_NPAGES_BOOST - is added to the `npages` value, when `free
memory` is less than `thres`.
# KSM_NPAGES_BOOST=300
# KSM_NPAGES_DECAY - is the value given is subtracted to the
`npages` value, when `free memory` is greater than `thres`.
# KSM_NPAGES_DECAY=-50
# KSM_NPAGES_MIN - is the lower limit for the `npages` value.
# KSM_NPAGES_MIN=64
# KSM_NPAGES_MAX - is the upper limit for the `npages` value.
# KSM_NPAGES_MAX=1250
# KSM_THRES_COEF - is the RAM percentage to be calculated in
parameter `thres`.
# KSM_THRES_COEF=20
# KSM_THRES_CONST - If this is a low memory system, and the
`thres` value is less than `KSM_THRES_CONST`, then reset
`thres` value to `KSM_THRES_CONST` value.
# KSM_THRES_CONST=2048
```

KSM is designed to improve performance and allow memory overcommits. It serves this purpose in most environments; however, KSM may introduce a performance overhead in some setups or environments – for example, if you have a few VMs that have similar memory content when you start them and loads of memory-intensive operations afterward. This will create issues as KSM will first work very hard to reduce the memory footprint, and then lose time to cover for all of the memory content differences between multiple VMs. Also, there is a concern that KSM may open a channel that could potentially be used to leak information across guests, as has been well documented in the past couple of years. If you have these concerns or if you see/experience KSM not helping to improve the performance of your workload, it can be disabled.

To disable KSM, stop the `ksmtuned` and `kvm` services in your system by executing the following:

```
# systemctl stop kvm
# systemctl stop ksmtuned
```

We have gone through the different tuning options for CPU and memory. The next big subject that we need to cover is NUMA configuration, where both CPU and memory configuration become a part of a larger story or context.

Tuning the CPU and memory with NUMA

Before we start tuning the CPU and memory for NUMA-capable systems, let's see what NUMA is and how it works.

Think of NUMA as a system where you have more than one system bus, each serving a small set of processors and associated memory. Each group of processors has its own memory and possibly its own I/O channels. It may not be possible to stop or prevent running VM access across these groups. Each of these groups is known as a **NUMA node**.

In this concept, if a process/thread is running on a NUMA node, the memory on the same node is called local memory and memory residing on a different node is known as foreign/remote memory. This implementation is different from the **Symmetric Multiprocessor System (SMP)**, where the access time for all of the memory is the same for all the CPUs, as memory access happens through a centralized bus.

An important subject in discussing NUMA is the NUMA ratio. The NUMA ratio is a measure of how quickly a CPU can access local memory compared to how quickly it can access remote/foreign memory. For example, if the NUMA ratio is 2.0, then it takes twice as long for the CPU to access remote memory. If the NUMA ratio is 1, that means that we're using SMP. The bigger the ratio, the bigger the latency price (overhead) that a VM memory operation will have to pay before getting the necessary data (or saving it). Before we explore tuning in more depth, let's discuss exploring the NUMA topology of a system. One of the easiest ways to show the current NUMA topology is via the `numactl` command:

```
root@packtphy02:~  
File Edit View Search Terminal Help  
[root@PacktPhy02 ~]# numactl -H  
available: 1 nodes (0)  
node 0 cpus: 0 1 2 3 4 5 6 7 8 9  
node 0 size: 4095 MB  
node 0 free: 889 MB  
node distances:  
node 0  
  0: 10  
[root@PacktPhy02 ~]#
```

Figure 15.16 – The `numactl -H` output

The preceding `numactl` output conveys that there are 10 CPUs in the system and they belong to a single NUMA node. It also lists the memory associated with each NUMA node and the node distance. When we discussed CPU pinning, we displayed the topology of the system using the `virsh` capabilities. To get a graphical view of the NUMA topology, you can make use of a command called `lstopo`, which is available with the `hwloc` package in CentOS-/Red Hat-based systems:

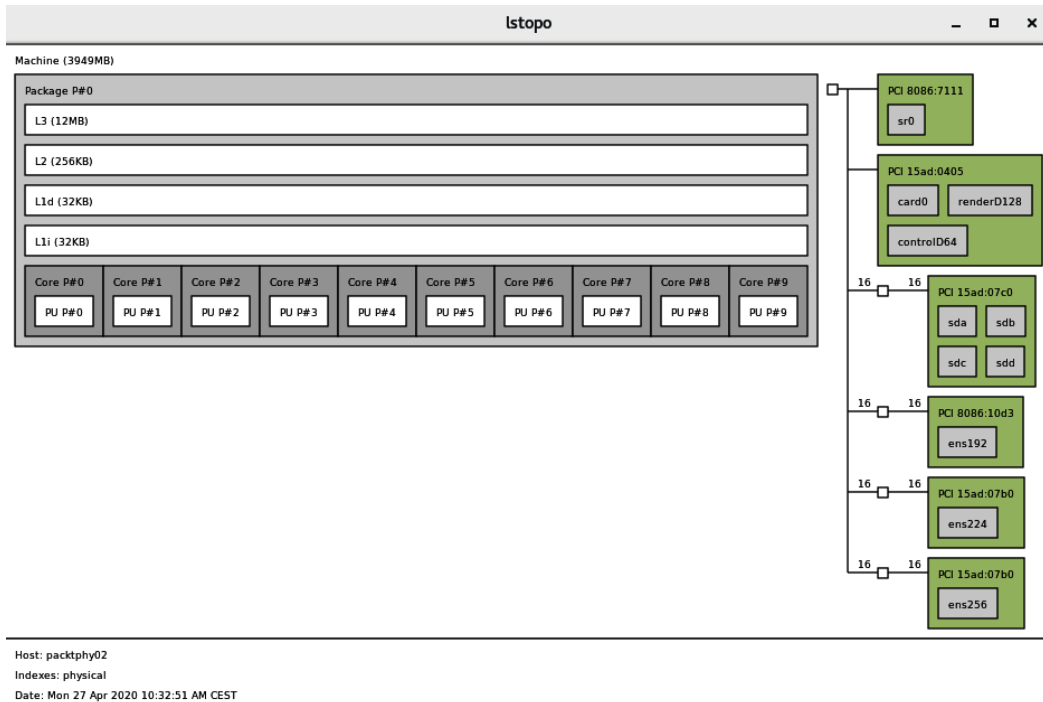


Figure 15.17 – The `lstopo` command to visualize the NUMA topology

This screenshot also shows the PCI devices associated with the NUMA nodes. For example, `ens*` (network interface) devices are attached to NUMA node 0. Once we have the NUMA topology of the system and understand it, we can start tuning it, specially for the KVM virtualized setup.

NUMA memory allocation policies

By modifying the VM XML configuration file, we can do NUMA tuning. Tuning NUMA introduces a new element tag called `numatune`:

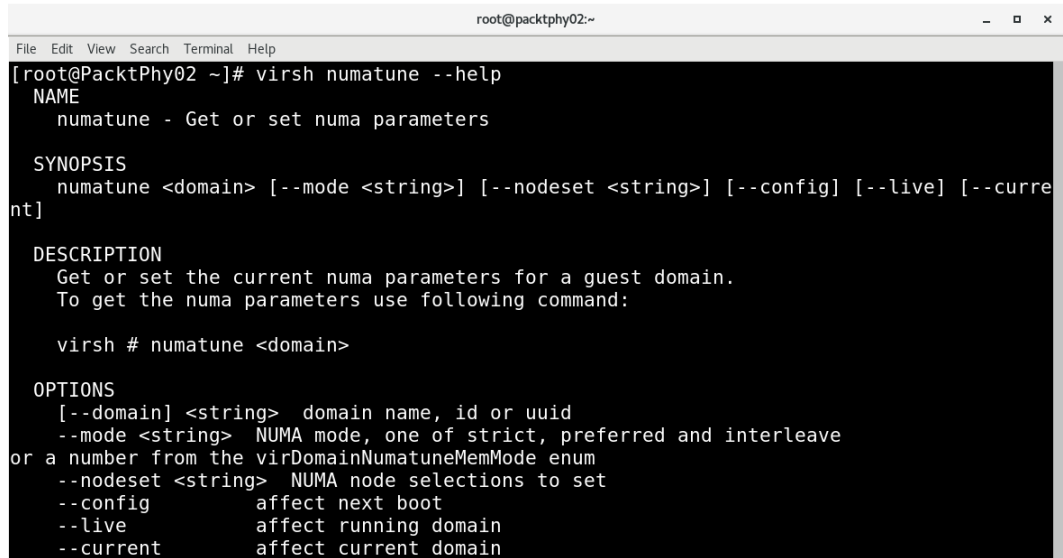
```
<domain> ...
  <numatune>
```

```

    <memory mode="strict" nodeset="1-4,^3"/>
  </numatune>    ...
</domain>

```

This is also configurable via the `virsh` command, as shown:



```

root@packtphy02:~
File Edit View Search Terminal Help
[root@PacktPhy02 ~]# virsh numatune --help
NAME
  numatune - Get or set numa parameters

SYNOPSIS
  numatune <domain> [--mode <string>] [--nodeset <string>] [--config] [--live] [--current]

DESCRIPTION
  Get or set the current numa parameters for a guest domain.
  To get the numa parameters use following command:

  virsh # numatune <domain>

OPTIONS
  [--domain] <string> domain name, id or uuid
  --mode <string> NUMA mode, one of strict, preferred and interleave
  or a number from the virDomainNumatuneMemMode enum
  --nodeset <string> NUMA node selections to set
  --config          affect next boot
  --live            affect running domain
  --current         affect current domain

```

Figure 15.18 – Using `virsh numatune` to configure the NUMA settings

The XML representation of this tag is as follows:

```

<domain>
...
  <numatune>
    <memory mode="strict" nodeset="1-4,^3"/>
    <memnode cellid="0" mode="strict" nodeset="1"/>
    <memnode cellid="2" mode="preferred" nodeset="2"/>
  </numatune>    ...
</domain>

```

Even though the element called `numatune` is optional, it is provided to tune the performance of the NUMA host by controlling the NUMA policy for the domain process. The main sub-tags of this optional element are `memory` and `nodeset`. Some notes on these sub-tags are as follows:

- `memory`: This element describes the memory allocation process on the NUMA node. There are three policies that govern memory allocation for NUMA nodes:
 - a) `strict`: When a VM tries to allocate memory and that memory isn't available, allocation will fail.
 - b) `interleave`: `nodeset`-defined round-robin allocation across NUMA nodes.
 - c) `preferred`: The VM tries to allocate memory from a preferred node. If that node doesn't have enough memory, it can allocate memory from the remaining NUMA nodes.
- `nodeset`: Specifies a NUMA node list available on the server.

One of the important attributes here is *placement*, as explained at the following URL – <https://libvirt.org/formatdomain.html>:

"Attribute placement can be used to indicate the memory placement mode for domain process, its value can be either "static" or "auto", defaults to placement of vCPU, or "static" if nodeset is specified. "auto" indicates the domain process will only allocate memory from the advisory nodeset returned from querying numad, and the value of attribute nodeset will be ignored if it's specified. If placement of vCPU is 'auto', and numatune is not specified, a default numatune with placement 'auto' and mode 'strict' will be added implicitly."

We need to be careful with these declarations, as there are inheritance rules that apply. For example, the `<numatune>` and `<vcpu>` elements default to the same value if we specify the `<nodeset>` element. So, we can absolutely configure different CPU and memory tuning options, but also be aware of the fact that these options can be inherited.

There are some more things to consider when thinking about CPU pinning in the NUMA context. We discussed the basis of CPU pinning earlier in this chapter, as it gives us better, predictable performance for our VMs and can increase cache efficiency. Just as an example, let's say that we want to run a VM as fast as possible. It would be prudent to run it on the fastest storage available, which would be on a PCI Express bus on the CPU socket where we pinned the CPU cores. If we're not using an NVMe SSD local to that VM, we can use a storage controller to achieve the same thing. However, if the storage controller that we're using to access VM storage is physically connected to another CPU socket, that will lead to latency. For latency-sensitive applications, that will mean a big performance hit.

However, we also need to be aware of the other extreme – if we do too much pinning, it can create other problems in the future. For example, if our servers are not architecturally the same (having the same amount of cores and memory), migrating VMs might become problematic. We can create a scenario where we're migrating a VM with CPU cores pinned to cores that don't exist on the target server of our migration process. So, we always need to be careful about what we do with the configuration of our environments so that we don't take it too far.

The next subject on our list is `emulatorpin`, which can be used to pin our `qemu-kvm` emulator to a specific CPU core so that it doesn't influence the performance of our VM cores. Let's learn how to configure that.

Understanding `emulatorpin`

The `emulatorpin` option also falls into the CPU tuning category. The XML representation of this would be as follows:

```
<domain>    ...
  <cputune>  ... .    <emulatorpin cpuset="1-3"/>
  ... .
  </cputune>  ...
</domain>
```

The `emulatorpin` element is optional and is used to pin the emulator (`qemu-kvm`) to a host physical CPU. This does not include the vCPU or IO threads from the VM. If this is omitted, the emulator is pinned to all the physical CPUs of the host system by default.

Important note:

Please note that `<vcpupin>`, `<numatune>`, and `<emulatorpin>` should be configured together to achieve optimal, deterministic performance when you tune a NUMA-capable system.

Before we leave this section, there are a couple more things to cover: the guest system NUMA topology and hugepage memory backing with NUMA.

Guest NUMA topology can be specified using the `<numa>` element in the guest XML configuration; some call this virtual NUMA:

```
<cpu>      ...
  <numa>
    <cell id='0' cpus='0-3' memory='512000' unit='KiB' />
    <cell id='1' cpus='4-7' memory='512000' unit='KiB' />
  </numa>   ...
</cpu>
```

The `cell id` element tells the VM which NUMA node to use, while the `cpus` element configures a specific core (or cores). The `memory` element assigns the amount of memory per node. Each NUMA node is indexed by number, starting from 0.

Previously, we discussed the `memorybacking` element, which can be specified to use hugepages in guest configurations. When NUMA is present in a setup, the `nodeset` attribute can be used to configure the specific hugepage size per NUMA node, which may come in handy as it ties a given guest's NUMA nodes to certain hugepage sizes:

```
<memoryBacking>
  <hugepages>
    <page size="1" unit="G" nodeset="0-2,4" />
    <page size="4" unit="M" nodeset="3" />
  </hugepages>
</memoryBacking>
```

This type of configuration can optimize the memory performance, as guest NUMA nodes can be moved to host NUMA nodes as required, while the guest can continue to use the hugepages allocated by the host.

NUMA tuning also has to consider the NUMA node locality for PCI devices, especially when a PCI device is being passed through to the guest from the host. If the relevant PCI device is affiliated to a remote NUMA node, this can affect data transfer and thus hurt the performance.

The easiest way to display the NUMA topology and PCI device affiliation is by using the `lstopo` command that we discussed earlier. The non-graphic form of the same command can also be used to discover this configuration. Please refer to the earlier sections.

KSM and NUMA

We discussed KSM in enough detail in previous sections. KSM is NUMA-aware, and it can manage KSM processes happening on multiple NUMA nodes. If you remember, we encountered a `sysfs` entry called `merge_across_node` when we fetched KSM entries from `sysfs`. That's the parameter that we can use to manage this process:

```
# cat /sys/kernel/mm/ksm/merge_across_nodes
1
```

If this parameter is set to 0, KSM only merges memory pages from the same NUMA node. If it's set to 1 (as is the case here), it will merge *across* the NUMA nodes. That means that the VM CPUs that are running on the remote NUMA node will experience latency when accessing a KSM-merged page.

Obviously, you know the guest XML entry (the `memorybacking` element) for asking the hypervisor to disable shared pages for the guest. If you don't remember, please refer back to the memory tuning section for details of this element. Even though we can configure NUMA manually, there is something called automatic NUMA balancing. We did mention it earlier, but let's see what this concept involves.

Automatic NUMA balancing

The main aim of automatic NUMA balancing is to improve the performance of different applications running in a NUMA-aware system. The strategy behind its design is simple: if an application is using local memory to the NUMA node where vCPUs are running, it will have better performance. By using automatic NUMA balancing, KVM tries to shift vCPUs around so that they are local (as much as possible) to the memory addresses that the vCPUs are using. This is all done automatically by the kernel when automatic NUMA balancing is active. Automatic NUMA balancing will be enabled when booted on the hardware with NUMA properties. The main conditions or criteria are as follows:

- `numactl --hardware`: Shows multiple nodes
- `cat /sys/kernel/debug/sched_features`: Shows NUMA in the flags

To illustrate the second point, see the following code block:

```
# cat /sys/kernel/debug/sched_features
GENTLE_FAIR_SLEEPERS START_DEBIT NO_NEXT_BUDDY LAST_BUDDY
CACHE_HOT_BUDDY
WAKEUP_PREEMPTION ARCH_POWER NO_HRTICK NO_DOUBLE_TICK LB_BIAS
NONTASK_
POWER TTWU_QUEUE NO_FORCE_SD_OVERLAP RT_RUNTIME_SHARE NO_LB_MIN
NUMA
NUMA_FAVOUR_HIGHER NO_NUMA_RESIST_LOWER
```

We can check whether it is enabled in the system via the following method:

```
# cat /proc/sys/kernel/numa_balancing
1
```

Obviously, we can disable automatic NUMA balancing via the following:

```
# echo 0 > /proc/sys/kernel/numa_balancing
```

The automatic NUMA balancing mechanism works based on the number of algorithms and data structures. The internals of this method are based on the following:

- NUMA hinting page faults
- NUMA page migration
- Pseudo-interleaving
- Fault statistics

- Task placement
- Task grouping

One of the best practices or recommendations for a KVM guest is to limit its resource to the amount of resources on a single NUMA node. Put simply, this avoids the unnecessary splitting of VMs across NUMA nodes, which can degrade the performance. Let's start by checking the current NUMA configuration. There are multiple available options to do this. Let's start with the `numactl` command, NUMA daemon, and `numastat`, and then go back to using a well-known command, `virsh`.

The numactl command

The first option to confirm NUMA availability uses the `numactl` command, as shown:

```
[root@PacktPhy02 ~]# numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9
node 0 size: 4095 MB
node 0 free: 2226 MB
node distances:
node 0
  0: 10
[root@PacktPhy02 ~]#
```

Figure 15.19 – The numactl hardware output

This lists only one node. Even though this conveys the unavailability of NUMA, further clarification can be done by running the following command:

```
# cat /sys/kernel/debug/sched_features
```

This will *not* list NUMA flags if the system is not NUMA-aware.

Generally, don't make VMs *wider* than what a single NUMA node can provide. Even if the NUMA is available, the vCPUs are bound to the NUMA node and not to a particular physical CPU.

Understanding numad and numastat

The numad man page states the following:

numad is a daemon to control efficient use of CPU and memory on systems with NUMA topology.

numad is also known as the automatic **NUMA Affinity Management Daemon**. It constantly monitors NUMA resources on a system in order to dynamically improve NUMA performance. Again, the numad man page states the following:

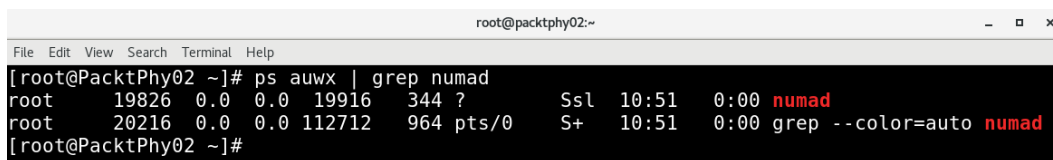
"numad is a user-level daemon that provides placement advice and process management for efficient use of CPUs and memory on systems with NUMA topology."

numad is a system daemon that monitors the NUMA topology and resource usage. It will attempt to locate processes for efficient NUMA locality and affinity, dynamically adjusting to changing system conditions. numad also provides guidance to assist management applications with the initial manual binding of CPU and memory resources for their processes. Note that numad is primarily intended for server consolidation environments, where there might be multiple applications or multiple virtual guests running on the same server system. numad is most likely to have a positive effect when processes can be localized in a subset of the system's NUMA nodes. If the entire system is dedicated to a large in-memory database application, for example, especially if memory accesses will likely remain unpredictable, numad will probably not improve performance.

To adjust and align the CPUs and memory resources automatically according to the NUMA topology, we need to run numad. To use numad as an executable, just run the following:

```
# numad
```

You can check whether this is started as shown:



```
root@packtphy02:~  
File Edit View Search Terminal Help  
[root@PacktPhy02 ~]# ps auwx | grep numad  
root    19826  0.0  0.0 19916  344 ?        Ssl  10:51   0:00 numad  
root    20216  0.0  0.0 112712  964 pts/0    S+   10:51   0:00 grep --color=auto numad  
[root@PacktPhy02 ~]#
```

Figure 15.20 – Checking whether numad is active

Once the `numad` binary is executed, it will start the alignment, as shown in the following screenshot. In our system, we have the following VM running:

```

root@packtphy02:~
File Edit View Search Terminal Help
[root@PacktPhy02 ~]# virsh list
 Id      Name                State
-----
 5      SQLForNuma          running
[root@PacktPhy02 ~]#

```

Figure 15.21 – Listing running VMs

You can use the `numastat` command, covered in an upcoming section, to monitor the difference before and after running the `numad` service. It will run continuously by using the following command:

```
# numad -i 0
```

We can always stop it, but that will not change the NUMA affinity state that was configured by `numad`. Now let's move on to `numastat`.

The `numactl` package provides the `numactl` binary/command and the `numad` package provides the `numad` binary/command:

```

root@packtphy02:~
File Edit View Search Terminal Help
[root@PacktPhy02 ~]#
[root@PacktPhy02 ~]#
[root@PacktPhy02 ~]# numad -i 0
[root@PacktPhy02 ~]# numastat -c qemu-kvm

Per-node process memory usage (in MBs) for PID 32372 (qemu-kvm)
Node 0 Total
-----
Huge      0    0
Heap     17   17
Stack     0    0
Private   30   30
-----
Total     47   47
[root@PacktPhy02 ~]#

```

Figure 15.22 – The `numastat` command output for the `qemu-kvm` process

Important note:

The numerous memory tuning options that we have used have to be thoroughly tested using different workloads before moving the VM to production.

Before we jump on to the next topic, we'd just like to remind you of a point we made earlier in this chapter. Live-migrating a VM with pinned resources might be complicated, as you have to have some form of compatible resources (and their amount) on the target host. For example, the target host's NUMA topology doesn't have to be aligned with the source host's NUMA topology. You should consider this fact when you tune a KVM environment. Automatic NUMA balancing may help, to a certain extent, the need for manually pinning guest resources, though.

Virtio device tuning

In the virtualization world, a comparison is always made with bare-metal systems. Paravirtualized drivers enhance the performance of guests and try to retain near-bare-metal performance. It is recommended to use paravirtualized drivers for fully virtualized guests, especially when the guest is running with I/O-heavy tasks and applications.

Virtio is an API for virtual IO and was developed by Rusty Russell in support of his own virtualization solution, called `lguest`. Virtio was introduced to achieve a common framework for hypervisors for IO virtualization.

In short, when we use paravirtualized drivers, the VM OS knows that there's a hypervisor beneath it, and therefore uses frontend drivers to access it. The frontend drivers are part of the guest system. When there are emulated devices and someone wants to implement backend drivers for these devices, hypervisors do this job. The frontend and backend drivers communicate through a virtio-based path. Virtio drivers are what KVM uses as paravirtualized device drivers. The basic architecture looks like this:

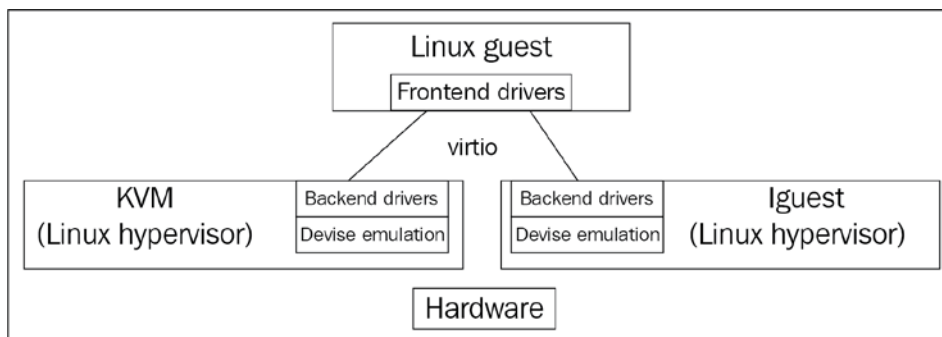


Figure 15.23 – The Virtio architecture

There are mainly two layers (virt queue and virtual ring) to support communication between the guest and the hypervisor.

Virt queue and **virtual ring (vring)** are the transport mechanism implementations in virtio. Virt queue (virtio) is the queue interface that attaches the frontend and backend drivers. Each virtio device has its own virt queues and requests from guest systems are put into these virt queues. Each virt queue has its own ring, called a vring, which is where the memory is mapped between QEMU and the guest. There are different virtio drivers available for use in a KVM guest.

The devices are emulated in QEMU, and the drivers are part of the Linux kernel, or an extra package for Windows guests. Some examples of device/driver pairs are as follows:

- `virtio-net`: The virtio network device is a virtual Ethernet card. `virtio-net` provides the driver for this.
- `virtio-blk`: The virtio block device is a simple virtual block device (that is, a disk). `virtio-blk` provides the block device driver for the virtual block device.
- `virtio-balloon`: The virtio memory balloon device is a device for managing guest memory.
- `virtio-scsi`: The virtio SCSI host device groups together one or more disks and allows communicating to them using the SCSI protocol.
- `virtio-console`: The virtio console device is a simple device for data input and output between the guest and host userspace.
- `virtio-rng`: The virtio entropy device supplies high-quality randomness for guest use, and so on.

In general, you should make use of these virtio devices in your KVM setup for better performance.

Block I/O tuning

Going back to basics – a virtual disk of a VM can be either a block device or an image file. For better VM performance, a block device-based virtual disk is preferred over an image file that resides on a remote filesystem such as NFS, GlusterFS, and so on. However, we cannot ignore that the file backend helps the virt admin to better manage guest disks and it is immensely helpful in some scenarios. From our experience, we have noticed most users make use of disk image files, especially when performance is not much of a concern. Keep in mind that the total number of virtual disks that can be attached to a VM has a limit. At the same time, there is no restriction on mixing and using block devices and files and using them as storage disks for the same guest.

A guest treats the virtual disk as its storage. When an application inside a guest OS writes data to the local storage of the guest system, it has to pass through a couple of layers. That said, this I/O request has to traverse through the filesystem on the storage and the I/O subsystem of the guest OS. After that, the `qemu-kvm` process passes it to the hypervisor from the guest OS. Once the I/O is within the realm of the hypervisor, it starts processing the I/O like any other applications running in the host OS. Here, you can see the number of layers that the I/O has to pass through to complete an I/O operation. Hence, the block device backend performs better than the image file backend.

The following are our observations on disk backends and file- or image-based virtual disks:

- A file image is part of the host filesystem and it creates an additional resource demand for I/O operations compared to the block device backend.
- Using sparse image files helps to over allocate host storage but its usage will reduce the performance of the virtual disk.
- The improper partitioning of guest storage when using disk image files can cause unnecessary I/O operations. Here, we are mentioning the alignment of standard partition units.

At the start of this chapter, we discussed virtio drivers, which give better performance. So, it's recommended that you use the virtio disk bus when configuring the disk, rather than the IDE bus. The `virtio_blk` driver uses the virtio API to provide high performance for storage I/O device, thus increasing storage performance, especially in large enterprise storage systems. We discussed the different storage formats available in *Chapter 5, Libvirt Storage*; however, the main ones are the `raw` and `qcow` formats. The best performance will be achieved when you are using the `raw` format. There is obviously a performance overhead delivered by the format layer when using `qcow`. Because the format layer has to perform some operations at times, for example, if you want to grow a `qcow` image, it has to allocate the new cluster and so on. However, `qcow` would be an option if you want to make use of features such as snapshots. These extra facilities are provided with the image format, `qcow`. Some performance comparisons can be found at <http://www.Linux-kvm.org/page/Qcow2>.

There are three options that can be considered for I/O tuning, which we discussed in *Chapter 7, Virtual Machine – Installation, Configuration, and Life Cycle Management*:

- Cache mode
- I/O mode
- I/O tuning

Let's briefly go through some XML settings so that we can implement them on our VMs.

The cache option settings can reflect in the guest XML, as follows:

```
<disk type='file' device='disk'>  
<driver name='qemu' type='raw' cache='writeback' />
```

The XML representation of I/O mode configuration is similar to the following:

```
<disk type='file' device='disk'>  
<driver name='qemu' type='raw' io='threads' />
```

In terms of I/O tuning, a couple of additional remarks:

- Limiting the disk I/O of each guest may be required, especially when multiple guests exist in our setup.
- If one guest is keeping the host system busy with the number of disk I/Os generated from it (noisy neighbor problem), that's not fair to the other guests.

Generally speaking, it is the system/virt administrator's responsibility to ensure all the running guests get enough resources to work on—in other words, the **Quality of Service (QOS)**.

Even though the disk I/O is not the only resource that has to be considered to guarantee QoS, this has some importance. Tuning I/O can prevent a guest system from monopolizing shared resources and lowering the performance of other guests running on the same host. This is really a requirement, especially when the host system is serving a **Virtual Private Server (VPS)** or a similar kind of service. KVM gives the flexibility to do I/O throttling on various levels – throughput and I/O amount, and we can do it per block device. This can be achieved via the `virsh blkdeiotune` command. The different options that can be set using this command are displayed as follows:

```

OPTIONS
[--domain] <string> domain name, id or uuid
[--device] <string> block device
--total-bytes-sec <number> total throughput limit, as scaled integer (default bytes)
--read-bytes-sec <number> read throughput limit, as scaled integer (default bytes)
--write-bytes-sec <number> write throughput limit, as scaled integer (default bytes)
--total-iops-sec <number> total I/O operations limit per second
--read-iops-sec <number> read I/O operations limit per second
--write-iops-sec <number> write I/O operations limit per second
--total-bytes-sec-max <number> total max, as scaled integer (default bytes)
--read-bytes-sec-max <number> read max, as scaled integer (default bytes)
--write-bytes-sec-max <number> write max, as scaled integer (default bytes)
--total-iops-sec-max <number> total I/O operations max
--read-iops-sec-max <number> read I/O operations max
--write-iops-sec-max <number> write I/O operations max
--size-iops-sec <number> I/O size in bytes
--group-name <string> group name to share I/O quota between multiple drives
--total-bytes-sec-max-length <number> duration in seconds to allow total max bytes
--read-bytes-sec-max-length <number> duration in seconds to allow read max bytes
--write-bytes-sec-max-length <number> duration in seconds to allow write max bytes
--total-iops-sec-max-length <number> duration in seconds to allow total I/O operations max
--read-iops-sec-max-length <number> duration in seconds to allow read I/O operations max
--write-iops-sec-max-length <number> duration in seconds to allow write I/O operations max
--config          affect next boot
--live            affect running domain
--current        affect current domain

```

Figure 15.24 – Excerpt from the `virsh blkdeiotune --help` command

Details about parameters such as `total-bytes-sec`, `read-bytes-sec`, `writebytes-sec`, `total-iops-sec`, and so on are easy to understand from the preceding command output. They are also documented in the `virsh` command man page.

For example, to throttle the `vdb` disk on a VM called `SQLForNuma` to 200 I/O operations per second and 50 MB-per-second throughput, run this command:

```
# virsh blkdeiotune SQLForNuma vdb --total-iops-sec 200
--total-bytes-sec 52428800
```

Next, we are going to look at network I/O tuning.

Network I/O tuning

What we've seen in most KVM environments is that all the network traffic from a guest will take a single network path. There won't be any traffic segregation, which causes congestion in most KVM setups. As a first step for network tuning, we'd advise trying different networks or dedicated networks for management, backups, or live migration. But when you have more than one network interface for your traffic, please try to avoid multiple network interfaces for the same network or segment. If this is at all in play, apply some network tuning that is common for such setups; for example, use `arp_filter` to control ARP Flux. ARP Flux happens when a VM has more than one network interface and is using them actively to reply to ARP requests, so we should do the following:

```
echo 1 > /proc/sys/net/ipv4/conf/all/arp_filter
```

After that, you need to edit `/etc/sysctl.conf` to make this setting persistent.

For more information on ARP Flux, please refer to <http://linux-ip.net/html/ether-arp.html#ether-arp-flux>.

Additional tuning can be done on the driver level; that said, by now we know that virtio drivers give better performance compared to emulated device APIs. So, obviously, using the `virtio_net` driver in guest systems should be taken into account. When we use the `virtio_net` driver, it has a backend driver in `qemu` that takes care of the communication initiated from the guest network. Even if this was performing better, some more enhancements in this area introduced a new driver called `vhost_net`, which provides in-kernel virtio devices for KVM. Even though `vhost` is a common framework that can be used by different drivers, the network driver, `vhost_net`, was one of the first drivers. The following diagram will make this clearer:

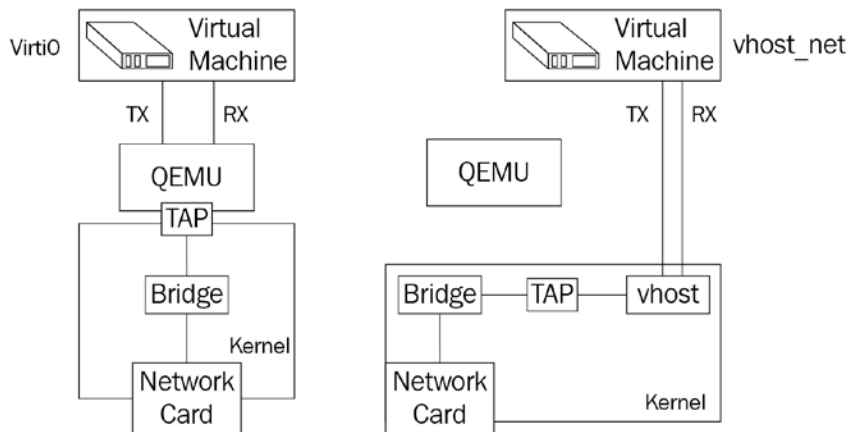


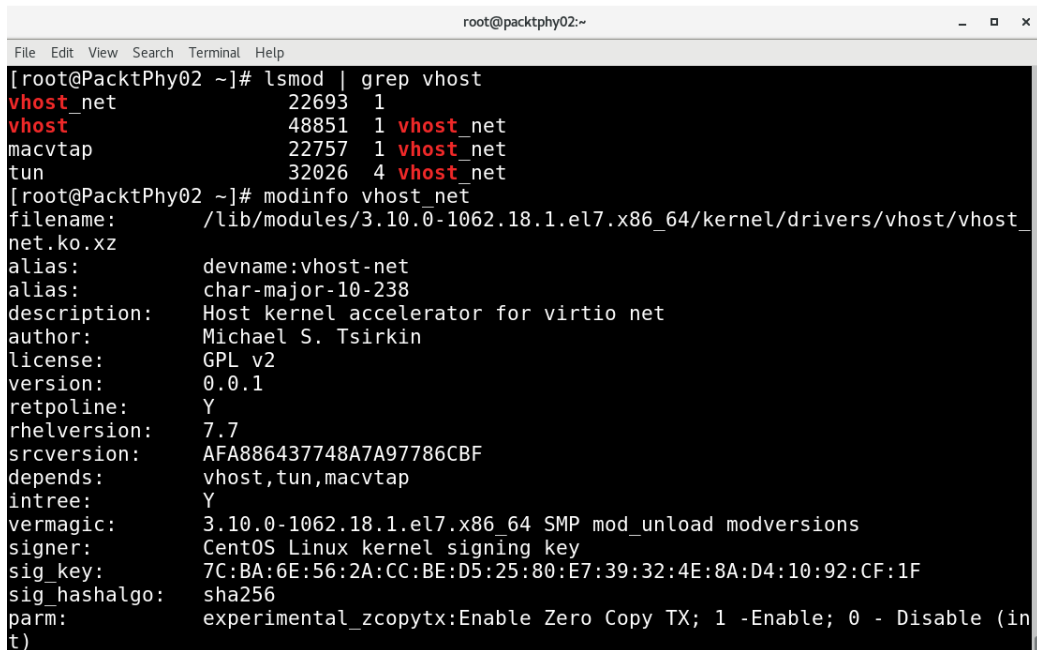
Figure 15.25 – The vhost_net architecture

As you may have noticed, the number of context switches is really reduced with the new path of communication. The good news is that there is no extra configuration required in guest systems to support vhost because there is no change to the frontend driver.

vhost_net reduces copy operations, lowers latency and CPU usage, and thus yields better performance. First of all, the kernel module called vhost_net (refer to the screenshot in the next section) has to be loaded in the system. As this is a character device inside the host system, it creates a device file called `/dev/vhost-net` on the host.

How to turn it on

When QEMU is launched with `-netdev tap,vhost=on`, it will instantiate the vhost-net interface by using `ioctl()` calls. This initialization process binds qemu with a vhost-net instance, along with other operations such as feature negotiations and so on:

A terminal window titled 'root@packtphy02:~' showing the output of two commands. The first command is 'lsmod | grep vhost', which lists loaded kernel modules: vhost_net (22693 1), vhost (48851 1), macvtap (22757 1), and tun (32026 4). The second command is 'modinfo vhost_net', which displays detailed metadata for the vhost_net module, including its filename, alias, description, author, license, version, and dependencies.

```
root@packtphy02:~  
File Edit View Search Terminal Help  
[root@PacktPhy02 ~]# lsmod | grep vhost  
vhost_net      22693  1  
vhost          48851  1 vhost_net  
macvtap        22757  1 vhost_net  
tun            32026  4 vhost_net  
[root@PacktPhy02 ~]# modinfo vhost_net  
filename:      /lib/modules/3.10.0-1062.18.1.el7.x86_64/kernel/drivers/vhost/vhost_  
net.ko.xz  
alias:         devname:vhost-net  
alias:         char-major-10-238  
description:   Host kernel accelerator for virtio net  
author:        Michael S. Tsirkin  
license:       GPL v2  
version:       0.0.1  
retpoline:    Y  
rhelversion:   7.7  
srcversion:    AFA886437748A7A97786CBF  
depends:        vhost,tun,macvtap  
intree:        Y  
vermagic:     3.10.0-1062.18.1.el7.x86_64 SMP mod_unload modversions  
signer:        CentOS Linux kernel signing key  
sig_key:       7C:BA:6E:56:2A:CC:BE:D5:25:80:E7:39:32:4E:8A:D4:10:92:CF:1F  
sig_hashalgo: sha256  
parm:         experimental_zcopytx:Enable Zero Copy TX; 1 -Enable; 0 - Disable (in  
t)
```

Figure 15.26 – Checking vhost kernel modules

One of the parameters available with the `vhost_net` module is `experimental_zcopytx`. What does it do? This parameter controls something called bridge zero copy transmit. Let's see what this means (as stated on <http://www.google.com/patents/US20110126195>):

"A system for providing a zero copy transmission in virtualization environment includes a hypervisor that receives a guest operating system (OS) request pertaining to a data packet associated with a guest application, where the data packet resides in a buffer of the guest OS or a buffer of the guest application and has at least a partial header created during the networking stack processing. The hypervisor further sends, to a network device driver, a request to transfer the data packet over a network via a network device, where the request identifies the data packet residing in the buffer of the guest OS or the buffer of the guest application, and the hypervisor refrains from copying the data packet to a hypervisor buffer."

If your environment uses large packet sizes, configuring this parameter may have a noticeable effect. The host CPU overhead is reduced by configuring this parameter when the guest communicates to the external network. This does not affect the performance in the following scenarios:

- Guest-to-guest communication
- Guest-to-host communication
- Small packet workloads

Also, the performance improvement can be obtained by enabling multi queue `virtio-net`. For additional information, check out https://fedoraproject.org/wiki/Features/MQ_virtio_net.

One of the bottlenecks when using `virtio-net` was its single RX and TX queue. Even though there are more vCPUs, the networking throughput was affected by this limitation. `virtio-net` is a single-queue type of queue, so multi-queue `virtio-net` was developed. Before this option was introduced, virtual NICs could not utilize the multi-queue support that is available in the Linux kernel.

This bottleneck is lifted by introducing multi-queue support in both frontend and backend drivers. This also helps guests scale with more vCPUs. To start a guest with two queues, you could specify the `queues` parameters to both `tap` and `virtio-net`, as follows:

```
# qemu-kvm -netdev tap,queues=2,... -device virtio-net-pci,queues=2,...
```

The equivalent guest XML is as follows:

```
<interface type='network'>
  <source network='default' />
  <model type='virtio' />
  <driver name='vhost' queues='M' />
</interface>
```

Here, M can be 1 to 8, as the kernel supports up to eight queues for a multi-queue tap device. Once it's configured for `qemu`, inside the guest, we need to enable multi-queue support with the `ethtool` command. Enable the multi-queue through `ethtool` (where the value of K is from 1 to M), as follows:

```
# ethtool -L eth0 combined 'K'
```

You can check the following link to see when multi-queue `virtio-net` provides the greatest performance benefit: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-networking-techniques.

Don't use the options mentioned on the aforementioned URL blindly – please test the impact on your setup, because the CPU consumption will be greater in this scenario even though the network throughput is impressive.

KVM guest time-keeping best practices

There are different mechanisms for time-keeping. One of the best-known techniques is **Network Time Protocol (NTP)**. By using NTP, we can synchronize clocks to great accuracy, even when using networks that have jitter (variable latency). One thing that needs to be considered in a virtualization environment is the maxim that the guest time should be in sync with the hypervisor/host, because it affects a lot of guest operations and can cause unpredictable results if they are not in sync.

There are different ways to achieve time sync, however; it depends on the setup you have. We have seen people using NTP, setting the system clock from the hardware clock using `hwclock -s`, and so on. The first thing that needs to be considered here is trying to make the KVM host time in sync and stable. You can use NTP-like protocols to achieve this. Once it's in place, the guest time has to be kept in sync. Even though there are different mechanisms for doing that, the best option would be using `kvm-clock`.

kvm-clock

`kvm-clock` is also known as a virtualization-aware (paravirtualized) clock device. When `kvm-clock` is in use, the guest asks the hypervisor about the current time, guaranteeing both stable and accurate timekeeping. The functionality is achieved by the guest registering a page and sharing the address with the hypervisor. This is a shared page between the guest and the hypervisor. The hypervisor keeps updating this page unless it is asked to stop. The guest can simply read this page whenever it wants time information. However, please note that the hypervisor should support `kvm-clock` for the guest to use it. For more details, you can check out <https://lkml.org/lkml/2010/4/15/355>.

By default, most of the newer Linux distributions use **Time Stamp Counter (TSC)**, a CPU register, as a clock source. You can verify whether TSC or `kvm_clock` are configured inside the guest via the following method:

```
[root@kvmguest ]$ cat /sys/devices/system/clocksource/
clocksource0/current_clocksource
tsc
```

You can also use `ntpd` or `chrony` as your clock sources on Linux, which requires minimal configuration. In your Linux VM, edit `/etc/ntp.conf` or `/etc/chronyd.conf` and modify the *server* configuration lines to point to your NTP servers by IP address. Then, just enable and start the service that you're using (we're using `chrony` as an example here):

```
systemctl enable chronyd
systemctl start chronyd
```

There's another, a bit newer, protocol that's being heavily pushed for time synchronization, which is called the **Precision Time Protocol (PTP)**. Nowadays, this is becoming the de facto standard service to be used on the host level. This protocol is directly supported in hardware (as in network interface cards) for many of the current network cards available on the market. As it's basically hardware-based, it should be even more accurate than `ntpd` or `chronyd`. It uses timestamping on the network interface, and external sources, and the computer's system clock for synchronization.

Installing all of the necessary pre-requisites is just a matter of one `yum` command to enable and start a service:

```
yum -y install linuxptp
systemctl enable ptp4l
systemctl start ptp4l
```

By default, the `ptp4l` service will use the `/etc/sysconfig/ptp4l` configuration file, which is usually bound to the first network interface. If you want to use some other network interface, the simplest thing to do would be to edit the configuration file, change the interface name, and restart the service via `systemctl`.

Now, from the perspective of VMs, we can help them time sync by doing a little bit of configuration. We can add the `ptp_kvm` module to the global KVM host configuration, which is going to make our PTP as a service available to `chronyd` as a clock source. This way, we don't have to do a lot of additional configuration. So, just add `ptp_kvm` as a string to the default KVM configuration, as follows:

```
echo ptp_kvm > /etc/modules-load.d/kvm-chrony.conf
modprobe ptp_kvm
```

By doing this, a `ptp` device will be created in the `/dev` directory, which we can then use as a `chrony` time source. Add the following line to `/etc/chrony.conf` and restart `chronyd`:

```
refclock PHC /dev/ptp0 poll 3 dpoll -2 offset 0
systemctl restart chronyd
```

By using an API call, all Linux VMs are capable of then getting their time from the physical host running them.

Now that we've covered a whole bunch of VM configuration options in terms of performance tuning and optimization, it's time to finally step away from all of these micro-steps and focus on the bigger picture. Everything that we've covered so far in terms of VM design (related to the CPU, memory, NUMA, virtio, block, network, and time configuration) is only as important as what we're using it for. Going back to our original scenario – a SQL VM – let's see how we're going to configure our VM properly in terms of the software that we're going to run on it.

Software-based design

Remember our initial scenario, involving a Windows Server 2019-based VM that should be a node in a Microsoft SQL Server cluster? We covered a lot of the settings in terms of tuning, but there's more to do – much more. We need to be asking some questions. The sooner we ask these questions, the better, as they're going to have a key influence on our design.

Some questions we may ask are as follows:

- Excuse me, dear customer, when you say *cluster*, what do you mean specifically, as there are different SQL Server clustering methodologies?
- Which SQL licenses do you have or are you planning to buy?
- Do you need active-active, active-passive, a backup solution, or something else?
- Is this a single-site or a multi-site cluster?
- Which SQL features do you need exactly?
- Which licenses do you have and how much are you willing to spend on them?
- Is your application capable of working with a SQL cluster (for example, in a multi-site scenario)?
- What kind of storage system do you have?
- What amount of IOPS can your storage system provide?
- How are latencies on your storage?
- Do you have a storage subsystem with different tiers?
- What are the service levels of these tiers in terms of IOPS and latency?
- If you have multiple storage tiers, can we create SQL VMs in accordance with the best practices – for example, place data files and log files on separate virtual disks?
- Do you have enough disk capacity to meet your requirements?

These are just licensing, clustering, and storage-related questions, and they are not going to go away. They need to be asked, without hesitation, and we need to get real answers before deploying things. We have just mentioned 14 questions, but there are actually many more.

Furthermore, we need to think about other aspects of VM design. It would be prudent to ask some questions such as the following:

- How much memory can you give for SQL VMs?
- Which servers do you have, which processors are they using, and how much memory do you have per socket?
- Are you using any latest-gen technologies, such as persistent memory?

- Do you have any information about the scale and/or amount of queries that you're designing this SQL infrastructure for?
- Is money a big deciding factor in this project (as it will influence a number of design decisions as SQL is licensed per core)? There's also the question of Standard versus Enterprise pricing.

This stack of questions actually points to one very, very important part of VM design, which is related to memory, memory locality, the relationship between CPU and memory, and also one of the most fundamental questions of database design – latency. A big part of that is related to correct VM storage design – the correct storage controller, storage system, cache settings, and so on, and VM compute design – which is all about NUMA. We've explained all of those settings in this chapter. So, to configure our SQL VM properly, here's a list of the high-level steps that we should follow:

- Configure a VM with the correct NUMA settings and local memory. Start with four vCPUs for licensing reasons and then figure out whether you need more (such as if your VM becomes CPU-limited, which you will see from performance graphs and SQL-based performance monitoring tools).
- If you want to reserve CPU capacity, make use of CPU pinning so that specific CPU cores on the physical server's CPU is always used for the SQL VM, and only that. Isolate other VMs to the *remaining* cores.
- Reserve memory for the SQL VM so that it doesn't swap, as only using real RAM memory will guarantee smooth performance that's not influenced by noisy neighbors.
- Configure KSM per VM if necessary and avoid using it on SQL VMs as it might introduce latency. In the design phase, make sure you buy as much RAM memory as possible so that memory doesn't become an issue as it will be a very costly issue in terms of performance if a server doesn't have enough of it. Don't *ever* overcommit memory.
- Configure the VM with multiple virtual hard disks and put those hard disks in storage that can provide levels of service needed in terms of latency, overhead, and caching. Remember, an OS disk doesn't necessarily need write caching, but database and log disks will benefit from it.
- Use separate physical connections from your hosts to your storage devices and tune storage to get as much performance out of it as possible. Don't oversubscribe – both on the links level (too many VMs going through the same infrastructure to the *same* storage device) and the datastore level (don't put one datastore on a storage device and store all VMs on it as it will negatively impact performance – isolate workloads, create multiple targets via multiple links, and use masking and zoning).

- Configure multipathing, load balancing, and failover – to get as much performance out of your storage, yes, but also to have redundancy.
- Install the correct virtio drivers, use vhost drivers or SR-IOV if necessary, and minimize the overhead on every level.
- Tune the VM guest OS – turn off unnecessary services, switch the power profile to `High Performance` (most Microsoft OSes have a default setting that puts the power profile into `Balanced` mode for some reason). Tune the BIOS settings and check the firmware and OS updates – everything – from top to bottom. Take notes, measure, benchmark, and use previous benchmarks as baselines when updating and changing the configuration so that you know which way you're going.
- When using iSCSI, configure jumbo frames as in most use cases, this will have a positive influence on the storage performance, and make sure that you check the storage device vendor's documentation for any best practices in that regard.

The takeaway of this chapter is the following – don't just blindly install an application just because a client asks you to install it. It will come to haunt you later on, and it will be much, much more difficult to resolve any kind of problems and complaints. Take your time and do it right. Prepare for the whole process by reading the documentation, as it's widely available.

Summary

In this chapter, we did some digging, going deep into the land of KVM performance tuning and optimization. We discussed many different techniques, varying from simple ones, such as CPU pinning, to much more complex ones, such as NUMA and proper NUMA configuration. Don't be put off by this, as learning design is a process, and designing things correctly is a craft that can always be improved with learning and experience. Think of it this way – when architects were designing the highest skyscrapers in the world, didn't they move the goalposts farther and farther with each new highest building?

In the next chapter – the final chapter of this book - we will discuss troubleshooting your environments. It's at least partially related to this chapter, as we will be troubleshooting some issues related to performance as well. Go through this chapter multiple times before switching to the troubleshooting chapter – it will be very, very beneficial for your overall learning process.

Questions

1. What is CPU pinning?
2. What does KSM do?
3. How do we enhance the performance of block devices?
4. How do we tune the performance of network devices?
5. How can we synchronize clocks in virtualized environments?
6. How do we configure NUMA?
7. How do we configure NUMA and KSM to work together?

Further reading

Please refer to the following links for more information:

- RedHat Enterprise Linux 7 – installing, configuring, and managing VMs on a RHEL physical machine: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/index
- vCPU pinning: <http://libvirt.org/formatdomain.html#elementsCPUTuning>
- KSM kernel documentation: <https://www.kernel.org/doc/Documentation/vm/ksm.txt>
- Placement: <http://libvirt.org/formatdomain.html#elementsNUMATuning>
- Automatic NUMA balancing: https://www.redhat.com/files/summit/2014/summit2014_riel_chegu_w_0340_automatic_numa_balancing.pdf
- Virtio 1.1 specification: <http://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html>
- ARP Flux: <http://Linux-ip.net/html/ether-arp.html#ether-arp-flux>

- MQ virtio: https://fedoraproject.org/wiki/Features/MQ_virtio_net
- libvirt NUMA tuning on RHEL 7: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-numa-numa_and_libvirt

16

Troubleshooting Guidelines for the KVM Platform

If you've followed this book all the way from *Chapter 1, Understanding Linux Virtualization*, then you'll know we went through *a lot* together in this book – hundreds and hundreds of pages of concepts and practical aspects, including configuration examples, files and commands – everything. 700 or so pages of it. So far, we've almost completely ignored troubleshooting as part of that journey. We didn't do this on the premise that everything just *works* in Linux and that we didn't have any issues at all and that we achieved a state of *nirvana* while going through this book cover to cover.

It was a journey riddled with various types of issues. Some of them aren't worth mentioning as they were our own mistakes. Mistakes like the ones we made (and you will surely make more too) mostly come from the fact that we mistyped something (in a command or configuration file). Basically, humans play a big role in IT. But some of these issues were rather frustrating. For example, implementing SR-IOV required a lot of time as we had to find different types of problems at the hardware, software, and configuration levels to make it work. oVirt was quite quirky, as we'll soon explain. Eucalyptus was interesting, to put it mildly. Although we used it *a lot* before, cloudbase-init was really complicated and required a lot of our time and attention, which turned out not to be due to something we did – it was just the cloudbase-init version. But overall, this just further proved a general point from our previous chapter – reading about various IT subjects in books, articles, and blog posts – is a really good approach to configuring a lot of things correctly from the start. But even then, you'll still need a bit of troubleshooting to make everything picture perfect.

Everything is great and amazing once you install a service and start using it, but it seldom happens that way the first time round. Everything we used in this book was actually installed to enable us to test different configurations and grab the necessary screenshots, but at the same time, we wanted to make sure that they can actually be installed and configured in a more structured, procedural way.

So, let's start with some simple things related to services, packages, and logging. Then, we'll move on to more advanced concepts and tools for troubleshooting, described through various examples that we have covered along the way.

In this chapter, we will cover the following topics:

- Verifying the KVM service status
- KVM service logging
- Enabling debug mode logging
- Advanced troubleshooting tools
- Best practices for troubleshooting KVM issues

Verifying the KVM service status

We're starting off with the simplest of all examples – verifying the KVM service status and some of its normal influence on host configuration.

In *Chapter 3, Installing a KVM Hypervisor, libvirt, and ovirt*, we did a basic installation of the overall KVM stack by installing `virt` module and using the `dnf` command to deploy various packages. There are a couple of reasons why this might not end up being a good idea:

- A lot of servers, desktops, workstations, and laptops come pre-configured with virtualization turned off in BIOS. If you're using an Intel-based CPU, make sure that you find all the VT-based options and enable them (VT, VT-d, VT I/O). If you're using an AMD-based CPU, make sure that you turn on AMD-V. There's a simple test that you can do to check if virtualization is enabled. If you boot any Linux live distribution, go to the shell and type in the following command:

```
cat /proc/cpuinfo | egrep "vmx|svm"
```

You can also use the following command, if you already installed your Linux host and the appropriate packages that we mentioned in *Chapter 3, Installing a KVM hypervisor, libvirt, and ovirt*:

```
virt-host-validate
```

If you don't get any output from this command, your system either doesn't support virtualization (less likely) or doesn't have virtualization features turned on. Make sure that you check your BIOS settings.

- Your networking configuration and/or package repository configuration might not be set up correctly. As we'll repeatedly state in this chapter, please, start with the simplest things – don't go off on a journey of trying to find some super complex reason why something isn't working. Keep it simple. For network tests, try using the `ping` command on some well-known server, such as `google.com`. For repository problems, make sure that you check your `/etc/yum.repos.d` directory. Try using the `yum clean all` and `yum update` commands. Repository problems are more likely to happen on some other distributions than CentOS/Red Hat, but still, they can happen.
- After the deployment process has finished successfully, make sure that you start and enable KVM services by using the following commands:

```
systemctl enable libvirtd libvirt-guests
```

```
systemctl start libvirtd libvirt-guests
```

Often, we forget to start and enable the `libvirt-guests` service, and then we get very surprised after we reboot our host. The result of `libvirt-guests` not being enabled is simple. When started, it suspends your virtual machines when you initiate shutdown and resumes them on the next boot. In other words, if you don't enable them, your virtual machines won't resume after the next reboot. Also, check out its configuration file, `/etc/sysconfig/libvirt-guests`. It's a simple text configuration file that enables you to configure at least three very important settings: `ON_SHUTDOWN`, `ON_BOOT`, and `START_DELAY`. Let's explain these:

- a) By using the `ON_SHUTDOWN` setting, we can select what happens with the virtual machine when we shut down your host since it accepts values such as `shutdown` and `suspend`.
- b) The `ON_BOOT` option does the opposite – it tells `libvirtd` whether it needs to start all the virtual machines on host boot, whatever their `autostart` settings are. It accepts values such as `start` and `ignore`.
- c) The third option, `START_DELAY`, allows you to set a timeout value (in seconds) between multiple virtual machine power-on actions while the host is booting. It accepts numeric values, with 0 being the value for parallel startup and *all other (positive) numbers* being the number of seconds it waits before it starts the next virtual machine.

Considering this, there are at least three things to remember:

- Make sure that these two services are actually running by typing in the following commands:

```
systemctl status libvirtd
systemctl status libvirt-guests
```

At least `libvirtd` needs to be started for us to be able to create or run a KVM virtual machine.

- If you're configuring more advanced settings such as SR-IOV, make sure that you read your server's manual to select a correct slot that is SR-IOV compatible. On top of that, make sure that you have a compatible PCI Express card and BIOS that's configured correctly. Otherwise, you won't be able to make it work.

- When you start the libvirt service, it usually comes with some sort of pre-defined firewall configuration. Keep that in mind in case you ever decide to disable libvirt services as the firewall rules will almost always still be there. That might require a bit of additional configuration.

The next step in your troubleshooting journey will be checking through some of the log files. And there are plenty to choose from – KVM has its own, oVirt has its own, as does Eucalyptus, ELK, and so on. So, make sure that you know these services well so that you can check the correct log files for the situation you're trying to troubleshoot. Let's start with KVM services logging.

KVM services logging

When discussing KVM services logging, there are a couple of locations that we need to be aware of:

- Let's say that you're logged in as root in the GUI and that you started virt-manager. This means that you have a `virt-manager.log` file located in the `/root/.cache/virt-manager` directory. It's really verbose, so be patient when reading through it.
- The `/etc/libvirt/libvirtd.conf` file is libvirtd's configuration file and contains a lot of interesting options, but some of the most important options are actually located almost at the end of the file and are related to auditing. You can select the commented-out options (`audit_level` and `audit_logging`) to suit your needs.
- The `/var/log/libvirt/qemu` directory contains logs and rotated logs for all of the virtual machines that were ever created on our KVM host.

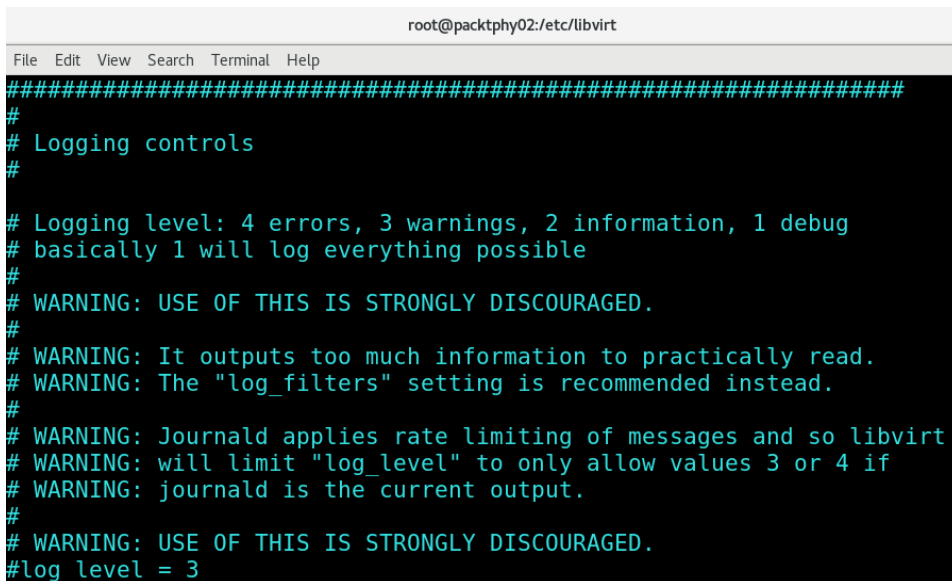
Also, be sure to check out a command called `auvirt`. It's really handy as it tells you basic information about the virtual machines on your KVM host – both virtual machines that are still there and/or successfully running and the virtual machines that we tried to install and failed at doing so. It pulls its data from audit logs, and you can use it to display information about a specific virtual machine we need as well. It also has a very debug-level option called `--all-events`, if you want to check every single little detail about any virtual machine that was – or still is – an object on the KVM host.

Enabling debug mode logging

There's another approach to logging in KVM: configuring debug logging. In the `libvirtd` configuration file that we just mentioned, there are additional settings you can use to configure this very option. So, if we scroll down to the `Logging controls` part, these are the settings that we can work with:

- `log_level`
- `log_filters`
- `log_outputs`

Let's explain them step by step. The first option – `log_level` – describes log verbosity. This option has been deprecated since `libvirt` version 4.4.0. In the `Logging controls` section of the file, there's additional documentation hardcoded into the file to make things easier. For this specific option, this is what the documentation says:



```
root@packtphy02:/etc/libvirt
File Edit View Search Terminal Help
#####
#
# Logging controls
#
# Logging level: 4 errors, 3 warnings, 2 information, 1 debug
# basically 1 will log everything possible
#
# WARNING: USE OF THIS IS STRONGLY DISCOURAGED.
#
# WARNING: It outputs too much information to practically read.
# WARNING: The "log_filters" setting is recommended instead.
#
# WARNING: Journald applies rate limiting of messages and so libvirt
# WARNING: will limit "log_level" to only allow values 3 or 4 if
# WARNING: journald is the current output.
#
# WARNING: USE OF THIS IS STRONGLY DISCOURAGED.
#log_level = 3
```

Figure 16.1 – Logging controls in `libvirtd.conf`

What people usually do is see the first part of this output (Logging level description), go to the last line (`log_level`), set it to 1, save, restart the `libvirtd` service, and be done with it. The problem is the text part in-between. It specifically says that `journald` does rate limiting so that it doesn't get hammered with logs from one service only and instructs us to use the `log_filters` setting instead.

Let's do that, then – let's use `log_filters`. A bit lower in the configuration file, there's a section that looks like this:

```

root@packtphy02:/etc/libvirt
File Edit View Search Terminal Help
# 1: DEBUG
# 2: INFO
# 3: WARNING
# 4: ERROR
#
# Multiple filters can be defined in a single @log_filters, they just need
# to be separated by spaces. Note that libvirt performs "first" match, i.e.
# if there are concurrent filters, the first one that matches will be applied,
# given the order in @log_filters.
#
# A typical need is to capture information from a hypervisor driver,
# public API entrypoints and some of the utility code. Some utility
# code is very verbose and is generally not desired. Taking the QEMU
# hypervisor as an example, a suitable filter string for debugging
# might be to turn off object, json & event logging, but enable the
# rest of the util code:
#
#log_filters="1:qemu 1:libvirt 4:object 4:json 4:event 1:util"

```

Figure 16.2 – Logging filters options in libvirtd.conf

This gives us various options we can use to set different logging options per object types, which is great. It gives us options to increase the verbosity of things that we're interested in at a desired level, while keeping the verbosity of other object types to a minimum. What we need to do is remove the comment part of the last line (`#log_filters="1:qemu 1:libvirt 4:object 4:json 4:event 1:util"` should become `log_filters="1:qemu 1:libvirt 4:object 4:json 4:event 1:util"`) and configure its settings so that they match our requirements.

The third option relates to where we want our debug logging output file to be:

```

root@packtphy02:/etc/libvirt
File Edit View Search Terminal Help
# Logging outputs:
# An output is one of the places to save logging information
# The format for an output can be:
#   level:stderr
#     output goes to stderr
#   level:syslog:name
#     use syslog for the output and use the given name as the ident
#   level:file:file_path
#     output to a file, with the given filepath
#   level:journald
#     output to journald logging system
# In all cases 'level' is the minimal priority, acting as a filter
# 1: DEBUG
# 2: INFO
# 3: WARNING
# 4: ERROR
#
# Multiple outputs can be defined, they just need to be separated by spaces.
# e.g. to log all warnings and errors to syslog under the libvirtd ident:
#log_outputs="3:syslog:libvirtd"
#

```

Figure 16.3 – Logging outputs options in libvirtd.conf

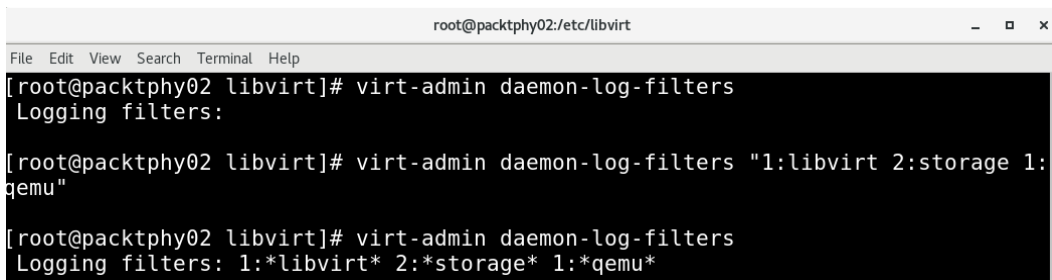
Important Note

After changing any of these settings, we need to make sure that we restart the `libvirtd` service by typing in the `systemctl restart libvirtd` command.

If we're only interested in client logs, we need to set an environment variable called `LIBVIRT_LOG_OUTPUTS` to something like this (let's say we want DEBUG-level logging):

```
export LIBVIRT_LOG_OUTPUTS="1:file:/var/log/libvirt_guests.log"
```

All these options are valid until the next `libvirtd` service restart, which is quite handy for permanent settings. However, there's a runtime option that we can use when we need to debug a bit on the fly, without resorting to permanent configuration. That's why we have a command called `virt-admin`. We can use it to set our own settings. For example, let's see how we can use it to get our current settings, and then how to use it to set temporary settings:



```

root@packtphy02:/etc/libvirt
File Edit View Search Terminal Help
[root@packtphy02 libvirt]# virt-admin daemon-log-filters
Logging filters:

[root@packtphy02 libvirt]# virt-admin daemon-log-filters "1:libvirt 2:storage 1:qemu"

[root@packtphy02 libvirt]# virt-admin daemon-log-filters
Logging filters: 1:*libvirt* 2:*storage* 1:*qemu*

```

Figure 16.4 – Runtime libvirtd debugging options

We can also delete these settings by issuing the following command:

```
virt-admin daemon-log-filters ""
```

This is something that's definitely recommended after we're done debugging. We don't want to use our log space for nothing.

In terms of straight-up debugging virtual machines – apart from these logging options – we can also use serial console emulation to hook up to the virtual machine console. This is something that we'd do if we can't get access to a virtual machine in any other way, especially if we're not using a GUI in our environments, which is often the case in production environments. Accessing the console can be done as follows:

```
virsh console kvm_domain_name
```

In the preceding command, `kvm_domain_name` is the name of the virtual machine that we want to connect to via the serial console.

Advanced troubleshooting tools

Depending on the subject – networking, hardware and software problems, or specific application problems – there are different tools that we can use to troubleshoot problems in our environments. Let's briefly go over some of these methods, with the chapters of this book in mind as we troubleshoot:

- oVirt problems
- Problems with snapshots and templates
- Virtual machine customization issues
- Ansible issues
- OpenStack problems
- Eucalyptus and AWS combo problems
- ELK stack issues

Interestingly enough, one thing that we usually don't have problems with when dealing with KVM virtualization is networking. It's really well documented – from KVM bridges all the way to open vSwitch – and it's just the matter of following the documentation. The only exception is related to firewall rules, which can be a handful, especially when dealing with oVirt and remote database connections while keeping a minimal security footprint. If you're interested in this, make sure that you check out the following link: https://www.ovirt.org/documentation/installing_ovirt_as_a_standalone_manager_with_remote_databases/#dns-requirements_SM_remoteDB_deploy.

There's a big table of ports later in that article describing which port gets used for what and which protocols they use. Also, there's a table of ports that need to be configured at the oVirt host level. We recommend that you use this article if you're putting oVirt into production.

oVirt

There are two common problems that we often encounter when dealing with oVirt:

- *Installation problems*: We need to slow down when we're typing installation options into the engine setup and configure things correctly.
- *Update problems* : These can either be related to incorrectly updating oVirt or the underlying system.

Installation problems are fairly simple to troubleshoot as they usually happen when we're just starting to deploy oVirt. This means that we can afford the luxury of just stopping the installation process and starting from scratch. Everything else will just be too messy and complicated.

Update problems, however, deserve a special mention. Let's deal with both subsets of oVirt update issues and explain them in a bit more detail.

Updating the oVirt Engine itself requires doing the thing that most of us just dislike doing – reading through heaps and heaps of documentation. The first thing that we need to check is which version of oVirt are we running. If we're – for example – running version 4.3.0 and we want to upgrade to 4.3.7, this is a minor update path that's pretty straightforward. We need to back up our oVirt database first:

```
engine-backup --mode=backup --file=backupfile1 --log=backup.log
```

We do this just as a precaution. Then, later on, if something does get broken, we can use the following command:

```
engine-backup --mode=restore --log=backup.log  
--file=backupfile1 --provision-db --provision-dwh-db --no-  
restore-permissions
```

If you didn't deploy the DWH service and its database, you can ignore the `--provision-dwh-db` option. Then, we can do the standard procedure:

```
engine-upgrade-check  
yum update ovirt\*setup\  
engine-setup
```

This should take about 10 minutes and cause no harm at all. But it's still better to be safe than sorry and back up the database before doing that.

If we're, however, migrating from some older version of oVirt to the latest one – let's say, from version 4.0.0, or 4.1.0, or 4.2.0 to version 4.3.7 – that's a completely different procedure. We need to go to the ovirt.org website and read through the documentation. For example, let's say that we're updating from 4.0 to 4.3. There's documentation on ovirt.org that describes all these processes. You can start here: https://www.ovirt.org/documentation/upgrade_guide/.

This will give us 20 or so substeps that we need to complete to successfully upgrade. Please be careful and patient, as these steps are written in a very clear order and need to be implemented that way.

Now that we've covered oVirt troubleshooting in terms of upgrading, let's delve into *OS and package upgrades* as that's an entirely different discussion with much more to consider.

Keeping in mind that oVirt has its own prerequisites, ranging from CPU, memory, and storage requirements to firewall and repository requirements, we can't just blindly go and use a system-wide command such as the following:

```
yum -y update
```

We can't expect oVirt to be happy with that. It just won't, and this has happened to us many times, both in production environments and while writing this book. We need to check which packages are going to be deployed and check if they're in some co-dependent relationship to oVirt. If there are such packages, you need to make sure that you do the engine-backup procedure that we mentioned earlier in this chapter. It will save you from a lot of problems.

It's not only the oVirt Engine that can be a problem – *updating KVM hosts* that oVirt has in its inventory can also be quite a bit melodramatic. The oVirt agent (`vdsm`) that gets deployed on hosts either by the oVirt Engine or our manual installation procedures, as well as its components, also have their own co-dependencies that can be affected by a system-wide `yum -y update` command. So, put the handbrake on before just accepting upgrades as it might bring a lot of pain later. Make sure that you check the `vdsm` logs (usually located in the `/var/log/vdsm` directory). These log files are very helpful when you're trying to decipher what went wrong with `vdsm`.

oVirt and KVM storage problems

Most storage problems that we come across are usually related to either LUN or share presentation to hosts. Specifically, when you're dealing with block storage (Fibre Channel or iSCSI), we need to make sure that we don't zone out or mask out a LUN from the host, as the host won't see it. The same principle applies to NFS shares, Gluster, CEPH, or any other kind of storage that we're using.

The most common problem apart from these pre-configuration issues is related to failover – a scenario where a path toward a storage device fails. That's when we are very happy if we scaled out our storage or storage network infrastructure a bit – we added additional adapters, additional switches, configured multipathing (MPIO), and so on. Make sure that you check your storage device vendor's documentation and follow along with the best practices for a specific storage device. Believe us when we say this – iSCSI storage configuration and its default settings are a world apart from configuring Fibre Channel storage, especially when multipathing is concerned. For example, when using MPIO with iSCSI, it's much happier and snappier if you configure it properly. You'll find more details about this process in the *Further reading* section at the end of this chapter.

If you're using IP-based storage, make sure that multiple paths toward your storage device(s) use separate IP subnets as everything else is a bad idea. LACP-like technologies and iSCSI don't work in same sentence together and you'll be troubleshooting a technology that's not meant for storage connections and is working properly, while you're thinking that it's not. We need to know what we're troubleshooting; otherwise, troubleshooting makes no sense. Creating LACP for iSCSI equals still using one path for iSCSI connections, which means wasting network connectivity that doesn't actively get used except for in the case of a failover. And you don't really need LACP or similar technologies for that. One notable exception might be blade servers as you're really limited in terms of upgrade options on blades. But even then, the solution to the *we need more bandwidth from our host to storage* problem is to get a faster network or Fibre Channel adapter.

Problems with snapshots and templates – virtual machine customization

To be quite honest, over the years of working on various virtualization technologies, which covers Citrix, Microsoft, VMware, Oracle, and Red Hat, we've seen a lot of different issues with snapshots. But it's only when you start working in enterprise IT and see how complicated operational, security, and backup procedures are that you start realizing how hazardous a *simple* procedure such as creating a snapshot can be.

We've seen things such as the following:

- The backup application doesn't want to start because the virtual machine has a snapshot (a common one).
- A snapshot doesn't want to delete and assemble.
- Multiple snapshots don't want to delete and assemble.
- A snapshot crashes a virtual machine for quirky reasons.

- A snapshot crashes a virtual machine for valid reasons (lack of disk space on storage)
- A snapshot crashes an application running in a virtual machine as that application doesn't know how to tidy itself up before the snapshot and goes into a dirty state (VSS, sync problems)
- Snapshots get lightly misused, something happens, and we need to troubleshoot
- Snapshots get heavily misused, something always happens, and we need to troubleshoot

This last scenario occurs far more often than expected as people really tend to flex their muscles regarding the number of snapshots they have if they're given permission to. We've seen virtual machines with 20+ snapshots running on a production environment and people complaining that they're slow. All you can do in that situation is breathe in, breathe out, shrug, and ask, "What did you expect, that 20+ snapshots are going to increase the speed of your virtual machine"?

Through it all, what got us through all these issues was three basic principles:

- Really learning how snapshots work on any given technology.
- Making sure that, every time we even think of using snapshots, we first check the amount of available storage space on the datastore where the virtual machine is located, and then check if the virtual machine already has snapshots.
- Constantly repeating the mantra: *snapshots are not backups* to all of our clients, over, and over again, and hammering them with additional articles and links explaining why they need to lay off the snapshots, even if that means denying someone permission to even take a snapshot.

Actually, this last one has become a de facto policy in many environments we've encountered. We've even seen companies implementing a flat-out policy when dealing with snapshots, stating that the company policy is to have one or two snapshots, max, for a limited period of time. For example, in VMware environments, you can assign a virtual machine advanced property that sets the maximum number of snapshots to 1 (using a property called `snapshot.maxSnapshots`). In KVM, you're going to have to use storage-based snapshots for these situations and hope that the storage system has policy-based capabilities to set the snapshot number to something. However, this kind of goes against the idea of using storage-based snapshots in many environments.

Templating and virtual machine customization is another completely separate world of troubleshooting. Templating only rarely creates issues, apart from the warning we mentioned in *Chapter 8, Creating and Modifying VM Disks, Templates, and Snapshots*, related to the serial use of `sysprep` on Windows machines. Creating Linux templates is pretty straightforward nowadays, and people use either `virt-sysprep`, `sys-unconfig`, or custom scripts to do that. But the next step – related to virtual machine customization – is a completely different thing. This is especially true when using `cloudbase-init`, as `cloud-init` has been a standard method used for preconfiguring Linux virtual machines in cloud environments for years.

The following is a short list containing some of problems that we had with `cloudbase-init`:

- `Cloudbase-init` failed due to `Cannot load user profile: the device is not ready`.
- Domain join doesn't work reliably.
- Error during network setup.
- Resetting Windows passwords via `cloudbase-init`.
- Getting `cloudbase-init` to execute a PowerShell script from a specified directory.

The vast majority of these and other problems are related to the fact that `cloudbase-init` has documentation that's really bad. It does have some config file examples, but most of it is more related to APIs or the programmatic approach than actually explaining how to create some kind of configuration via examples. Furthermore, we had various issues with different versions, as we mentioned in *Chapter 10, Automated Windows guest deployment and customization*. We then settled on a pre-release version, which worked out-of-the-box with a configuration file that wasn't working on a stable release. But by and large, the biggest issue we had while trying to make it work was related to making it work with PowerShell properly. If we get it to execute PowerShell code properly, we can pretty much configure anything we want on a Windows-based system, so that was a big problem. Sometimes, it didn't want to execute a PowerShell script from a random directory on the Windows system disk.

Make sure that you use examples in this book for your starting points. We deliberately made examples in *Chapter 10, Automated Windows guest deployment and customization* as simple as possible, which includes the executed PowerShell code. Afterward, spread your wings and fly – do whatever needs to be done with it. PowerShell makes everything easier and more natural when you're working with Microsoft-based solutions, both local and hybrid ones.

Problems working with Ansible and OpenStack

Our first interaction with Ansible and OpenStack happened years ago – Ansible was introduced in 2012, and OpenStack in 2010. We always thought that both were (are) very cool pieces of kit, albeit with a few problems. Some of these small niggles were related to the fast pace of development (OpenStack), with a large number of bugs being solved from version to version.

In terms of Ansible, we had loads of fights with people over it – one day, the subject was related to the fact that *we're used to using Puppet, why do we need Ansible?!*; the next day it was *argh, this syntax is so complex*; the day after that it was something else, and something else... and it was usually just related to the fact that the Ansible architecture is much simpler than all of them in terms of architecture, and a bit more involved – at least initially – in terms of syntax. With Ansible, it's all about the syntax, as we're sure that you either know or will find out soon enough.

Troubleshooting Ansible playbooks is usually a process that has a 95% chance that we misspelled or mistyped something in the configuration file. We're talking about the initial phase in which you already had a chance to work with Ansible for a while. Make sure that you re-check outputs from Ansible commands and use their output for that. In that sense, it's really excellent. You don't need to do complex configuration (such as with `libvirt`, for example) to get usable output from your executed procedures and playbooks. And that makes our job a lot easier.

Troubleshooting OpenStack is a completely different can of worms. There are some well-documented OpenStack problems out there, which can also be related to a specific device. Let's use one example of that – check out the following link for issues when using NetApp storage: https://netapp-openstack-dev.github.io/openstack-docs/stein/appendices/section_common-problems.html.

The following are some examples:

- Creating volume fails
- Cloning volume fails
- Volume attachment fails
- Volume upload to image operation fails
- Volume backup and/or restore fails

Then, for example, check out these links:

- <https://docs.openstack.org/cinder/queens/configuration/block-storage/drivers/ibm-storwize-svc-driver.html>
- https://www.ibm.com/support/knowledgecenter/STHGUJ_8.2.1/com.ibm.storwize.v5100.821.doc/storwize_openstack_matrix.html

As you've probably deduced yourself, OpenStack is really, really picky when it comes to storage. That's why storage companies usually create reference architectures for their own storage devices to be used in OpenStack-based environments. Check out these two documents from HPE and Dell EMC as good examples of that approach:

- <https://www.redhat.com/cms/managed-files/cl-openstack-hpe-synergy-ceph-reference-architecture-f18012bf-201906-en.pdf>
- <https://docs.openstack.org/cinder/rocky/configuration/block-storage/drivers/dell-emc-unity-driver.html>

One last word of warning relates to the most difficult obstacle to surmount – OpenStack version upgrades. We can tell you loads of horror stories on this subject. That being said, we're also partially to blame here, because we, as users, deploy various third-party modules and utilities (vendor-based plugins, forks, untested solutions, and so on), forget about using them, and then we're really surprised and horrified when the upgrade procedure fails. This goes back to our multiple discussions about documenting environments that we had throughout this book. This is a subject that we'll revisit for one final time just a bit later in this chapter.

Dependencies

Every administrator is completely aware that almost every service has some dependencies – either the services that depend on this particular service running or services that our service needs to work. Dependencies are also a big thing when working with packages – the whole point of package managers is to strictly pay attention to what needs to be installed and what depends on it so that our system works as it should.

What most admins do wrong is forget that, in larger systems, dependencies can stretch across multiple systems, clusters, and even data centers.

Every single course that covers OpenStack has a dedicated lesson on starting, stopping, and verifying different OpenStack services. The reason for this is simple – OpenStack is usually run across a big number of nodes (hundreds, sometimes thousands). Some services must run on every node, some are needed by a set of nodes, some services are duplicated on every node instance, and some services can only exist as a single instance.

Understanding the basics of each service and how it falls into the whole OpenStack schema is not only essential when installing the whole system but is also the most important thing to know when debugging why something is not working on OpenStack. Read the documentation at least once to *connect the dots*. Again, the *Further reading* section at the end of this chapter contains links that will point you in the right direction regarding OpenStack.

OpenStack is one of those systems that includes *how do I properly reboot a machine running X?* in the documentation. The reason for this is as simple as the whole system is complex – each part of the system both has something it depends on and something that is depending on it – if something breaks, you need to not only understand how this particular part of the system works, but also how it affects everything else. But there is a silver lining through all this – in a properly configured system, a lot of it is redundant, so sometimes, the easiest way of repairing something is to reinstall it.

And this probably sums the whole troubleshooting story – trying to fix a simple system is usually more complicated and time-consuming than fixing a complex system. Understanding how each of them works is the most important part.

Troubleshooting Eucalyptus

It would be a lie to say that once we started the installation process, everything went according to the manual – most of it did, and we are reasonably sure that if you follow the steps we documented, you will end up with a working service or system, but at any point in time, there are things that can – and will – go wrong. This is when you need to do the most complicated thing imaginable – troubleshoot. But how do you do that? Believe it or not, there is a more or less systematic approach that will enable you to troubleshoot almost any problem, not just KVM/OpenStack/AWS/Eucalyptus-related ones.

Gathering information

Before we can do anything, we need to do some research. And this is the moment most people do the wrong thing, because the obvious answer is to go to the internet and search for the problem. Take a look at this screenshot:

```

- execute the ruby block Upload cluster keys Chef Server
* execute[Register User Facing 192.168.5.48] action run[2020-04-09T16:38:50-04:00] INFO: Processing
g execute[Register User Facing 192.168.5.48] action run (eucalyptus::register-components line 99)
[2020-04-09T16:38:50-04:00] INFO: Processing execute[Guard resource] action run (dynamically defined
)
[2020-04-09T16:38:56-04:00] INFO: execute[Register User Facing 192.168.5.48] ran successfully

- execute eval `cloudadmin-assume-system-credentials` && //usr/bin/euser-v-register-service -t use
r-api -h 192.168.5.48 API_192.168.5.48
* execute[Register Walrus] action run[2020-04-09T16:38:56-04:00] INFO: Processing execute[Register
Walrus] action run (eucalyptus::register-components line 110)
[2020-04-09T16:38:56-04:00] INFO: Processing execute[Guard resource] action run (dynamically defined
)
[2020-04-09T16:38:58-04:00] INFO: execute[Register Walrus] ran successfully

- execute eval `cloudadmin-assume-system-credentials` && //usr/bin/euser-v-register-service -t wal
rusbackend -h 192.168.5.48 walrus-0
Recipe: eucalyptus::walrus
* yum_package[eucalyptus-walrus] action upgrade[2020-04-09T16:38:58-04:00] INFO: Processing yum_pa
ckage[eucalyptus-walrus] action upgrade (eucalyptus::walrus line 23)
[2020-04-09T16:39:04-04:00] INFO: yum_package[eucalyptus-walrus] installing eucalyptus-walrus-4.4.5-
0.34.as.el7 from eucalyptus repository
[2020-04-09T16:39:07-04:00] INFO: yum_package[eucalyptus-walrus] upgraded eucalyptus-walrus to 4.4.5
-0.34.as.el7

- upgrade package eucalyptus-walrus from uninstalled to 4.4.5-0.34.as.el7
[2020-04-09T16:39:07-04:00] INFO: yum_package[eucalyptus-walrus] sending create action to template[e
ucalyptus.conf] (immediate)
Recipe: eucalyptus::storage-controller
* template[eucalyptus.conf] action create[2020-04-09T16:39:07-04:00] INFO: Processing template[euc
alyptus.conf] action create (eucalyptus::storage-controller line 50)
[2020-04-09T16:39:07-04:00] INFO: template[eucalyptus.conf] backed up to /root/.chef/local-mode-cach
e/backup/etc/eucalyptus/eucalyptus.conf.chef-20200409163907.862503
[2020-04-09T16:39:07-04:00] INFO: template[eucalyptus.conf] updated file contents //etc/eucalyptus/e
ucalyptus.conf

```

Figure 16.5 – Eucalyptus logs, part I – clean, crisp, and easy to read – every procedure that's been done in Eucalyptus clearly visible in the log

If you haven't noticed already, the internet is full of ready-made solutions to almost any imaginable problem, with a lot of them being wrong. There are two reasons why this is so: most of the people who worked on the solution didn't understand what the problem was, so as soon as they found any solution that solved their particular problem, they simply stopped solving it. In other words – a lot of people in IT try to picture a path from point A (problem) to point B (solution) as a laser beam – super flat, the shortest possible path, no obstacles along the way. Everything is nice and crisp and designed to mess with our troubleshooting thought process as soon as the laser beam principle stops working. This is because, in IT, things are rarely that simple.

Take, for example, any problem caused by DNS being misconfigured. Most of those can be *solved* by creating an entry in the *hosts* file. This solution usually works but is, at the same time, wrong on almost any level imaginable. The problem that is solved by this is solved on only one machine – the one that has the particular *hosts* file on it. And the DNS is still misconfigured; we just created a quick, undocumented workaround that will work in our particular case. Every other machine that has the same problem will need to be patched in this way, and there is a real possibility that our fix is going to create even more problems down the road.

The real solution would obviously be to get to the root of the problem itself and solve the issue with DNS, but solutions like this are few and far between on the internet. This happens mainly because the majority of the commenters on the internet are not familiar with a lot of services, and quick fixes are basically the only ones they are able to apply.

Another reason why the internet is mostly wrong is because of the famous *reinstall fixed the problem* solution. Linux has a better track record there as people who use it are less inclined to solve everything by wiping and reinstalling the system, but most of the solutions you will find for Windows problems are going to have at least one simple *reinstall fixed it*. Compared to just giving a random fix as the one that always works, this *reinstall* approach is far worse. Not only does it mean you are going to waste a lot of time reinstalling everything; it also means your problem may or may not be solved in the end, depending on what the problem actually was.

So, the first short piece of advice we will give is, *do not blindly trust the internet*.

OK, but what should you actually do? Let's take a look:

1. *Gather information about the problem.* Read the error message, read the logs (if the application has logs), and try to turn on debug mode if at all possible. Get some solid data. Find out what is crashing, how it is crashing, and what problems are causing it to crash. Take a look at the following screenshot:

```
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/tutorials/launch-instances.sh in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/bind-addr.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/node-template.json in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/nuke.sh in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/eucalyptus_helper.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/enterprise.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/midonet.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/create_riakcs_user.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/motherbrain.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/tutorials/install-image.sh in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/cloud-controller.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/ceph.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/cloud-service.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/install-nc.sh in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/metadata.json in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/cloud-in-a-box.sh in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/create-first-resources.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/cluster-controller.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/eucanetd.rb in the cache.
:~
```

Figure 16.6 – Eucalyptus logs, part II – again, clean, crisp, and easy to read – information messages about what was updated and where

2. *Read the documentation.* Is the thing you are trying to do even supported? What are the prerequisites for the functioning system? Are you missing something? A cache disk? Some amount of memory? A fundamental service that is a dependency for your particular system? A dependency that's a library or additional packages? A firmware upgrade?

Sometimes, you will run into an even bigger problem, especially in poorly written documentation – *some crucial system dependency may be mentioned in passing* and may cause your entire system to crash. Take, for example, an external identification service – maybe your directory uses a *wrong character set*, causing your system to crash when a particular user uses it in a particular way. Always make sure you understand how your systems are interconnected.

Next, check your system. If you are installing a new system, check the prerequisites. Do you have *enough disk space and memory*? Are all the services your application requires readily available and working properly?

Search the internet. We mentioned previously that the internet has a simple, incorrect solution to all possible problems, but it usually also has the right solution hidden somewhere among the wrong ones. Having armed yourself with a lot of data about your particular system and your specific problem, the internet will soon become your friend. Since you understand what the problem is, you will be able to understand what solutions have been offered to you are simply wrong.

Now, let's talk about a real-world problem we created while installing Eucalyptus on purpose, just to show you how important documentation is.

We showed you how to install Eucalyptus in *Chapter 13, Scaling Out KVM with AWS* – we not only went through the installation process but also how to use this amazing service. If you want to learn something about how not to do it, continue reading. We will present you with a deliberate scenario of an unsuccessful Eucalyptus installation that won't finish because we *creatively forgot to do some steps that we knew we needed to do*. Let's put it this way – we acted as humans and used the method of *browsing the documentation* instead of *actually sitting down and reading the documentation*. Does that sound familiar?

Installing Eucalyptus should be a straightforward task since its installation is, in essence, an exercise in applied scripting. Eucalyptus even says so on the front page of the project: *just run this script*.

But the truth is much more complicated – Eucalyptus can definitely be installed using only this script, but certain prerequisites must be met. Of course, in your rush to test the new service, you will probably neglect to read the documentation, as we did, since we already had experience with Eucalyptus.

We configured the system, we started the installation, and we ran into a problem. After confirming the initial configuration steps, our installation failed with an error that said it was unable to resolve a particular address: `192.168.1.1.nip.io`.

DNS is one of the primary sources of problems in the IT infrastructure, and we quickly started debugging – the first thing we wanted to see was what this particular address is. There's actually a saying in IT – *It's always DNS*. It looks like a local address, so we started pinging it, and it seemed fine. But why is DNS even involved with IP addresses? DNS should be resolving domain names, not IP addresses. Then, we turned to the documentation, but that didn't yield much. The only thing that we found was that DNS must work for the whole system to work.

Then, it was time to try and debug the DNS. First, we tried resolving it from the machine we were installing it on. The DNS returned a timeout. We tried this on another machine and we got back the response we didn't expect – `127.0.0.1.nip.io` resolved as `127.0.0.1`, which meant localhost. Basically, we asked a DNS on the internet to give us an address, and it directed us to our local system.

So, we had an error we didn't understand, an address that resolved to an IP address we hadn't expected, and two different systems exhibiting completely different behaviors for an identical command. We turned our attention to the machine we were installing on and realized it was misconfigured – there was no DNS configured at all. The machine not only failed to resolve our *strange* IP address but failed to resolve anything.

We fixed that by pointing to the right DNS server. Then, in true IT fashion, we restarted the installation so that we were able to go through with this part and everything was ok, or so it seemed. But what happened? Why is a local service resolving such strange names and why do they get resolved at all?

We turned to the internet and took a look at the name of the domain that our mystery name had at its end. What we found out is that the service, `nip.io`, actually does just the thing we observed it do – when asked for a particular name formed from an IP address in the local subnet range (as defined by RFC 1918), it returned that same IP.

Our next question was – why?

After some more reading, you will realize what the trick was here – Eucalyptus uses DNS names to talk to all of its components. The authors very wisely chose not to hardcode a single address into the application, so all the services and nodes of the system have to have a real DNS registered name. In a normal multi-node, multi-server installation, this works like a charm – every server and every node are first registered with their appropriate DNS server, and Eucalyptus will try and resolve them so it can communicate with the machine.

We are installing a single machine that has all the services on it, and that makes installing easier, but nodes do not have separate names, and even our machine may not be registered with the DNS. So, the installer does a little trick. It turns local IP addresses into completely valid domain names and makes sure we can resolve them.

So, now we know what happened (resolving process was not working) and why it happened (our DNS server settings were broken), but we also understood why DNS was needed in the first place.

This brings us to the next point – *do not presume anything*.

While we were troubleshooting and then following up on our DNS problem, our installation crashed. Eucalyptus is a complex system and its installation is a fairly complex thing – it automatically updates the machine you run it on, then it installs what seems like thousands of packages, and then it downloads, configures, and runs a small army of images and virtual packages. To keep things tidy, the user doesn't see everything that is happening, only the most important bits. The installer even has a nice ASCII graphic screen to keep you busy. Everything was OK up to a point, but suddenly, our installation completely crashed. All we got was a huge stack trace that looked like it belonged to the Python language. We reran the installation, but it failed again.

The problem at this point was that we had no idea why all this was happening since the installation calls for a minimal installation of CentOS 7. We were running our tests on a virtual machine, and we actually did a minimal install.

We retried installing from scratch. Reinstalling the whole machine took a couple of minutes, and we retried the installation. The result was the same – a failed installation that left us with an unusable system. But there was a possible solution – or to be more precise, a way to understand what happened.

As with all great installers of the IT universe, this one also has something reserved especially for this possibility: a log file. Take a look at the following screenshot:

```
[Yum Update] OK, running a full update of the OS. This could take a bit; please wait.

To see the update in progress, run the following command in another terminal:

tail -f /var/log/euca-install-04.08.2020-18.06.50.log

[Yum Update] Package update in progress...

  ) )
 ( (
 .....
 |         | |
 \         /
 -----

[Yum Update] Full update of the OS completed.

Phase 0 (OS) completed successfully...getting a 2nd cup of tea and moving on to phase 1 (CLC).

  ) )
 ( (
 .....
 |         | |
 \         /
 -----

Phase 1 (CLC) completed successfully...getting a 3rd cup of tea and moving on to phase 2 (main cloud
components).

  ) )
 ( (
 .....
 |         | |
 \         /
 -----
```

Figure 16.7 – The Eucalyptus installation process takes time when you don't read its documentation.

And then some more time... and some more...

This is the installation screen. We can't see any real information regarding what is happening, but the third line from the top contains the most important clue – the location of the log file. In order to stop your screen from being flooded with information, the installer shows this very nice figlet-coffee graphic (everyone who ever used IRC in the 1990s and 2000s will probably smile now), but also dumps everything that is happening into a log. By everything, we mean everything – every command, every input, and every output. This makes debugging easy – we just need to scroll to the end of this file and try to go from that point backward to see what broke. Once we did that, the solution was simple – we forgot to allocate enough memory for the machine. We gave it 8 GB of RAM, and officially it should have at least 16 GB to be able smoothly. There are reports of machines running with as little as 8 GB of RAM, but that makes absolutely no sense – we are running a virtualized environment after all.

AWS and its verbosity, which doesn't help

Another thing we wanted to mention is AWS and how to troubleshoot it. AWS is an amazing service, but it has one huge problem – its size. There are so many services, components, and service parts that you need to use to get something to run on AWS that simple tasks can get very complicated. Our scenario involved trying to put up an EC2 instance that we used as our example.

This task is relatively straightforward and demonstrates how a simple problem can have a simple solution that can, at the same time, be completely not obvious.

Let's go back to what we were trying to do. We had a machine that was on a local disk. We had to transfer it to the cloud and then create a running VM out of it. This is probably one of the simplest things to do.

For that, we created an S3 bucket and got our machine from the local machine into the cloud. But after we tried to run the machine, all we got was an error.

The biggest problem with a service like AWS is that it is enormous and that there is no way of understanding everything at once – you must build your knowledge block by block. So, we went back to the documentation. There are two kinds of documentation on AWS – extensive help that covers every command and every option on every service, and guided examples. Help is amazing, it really is, but if you have no idea what you are looking for, it will get you nowhere. Help in this form only works as long as you have a basic understanding of the concepts. If you are doing something for the first time, or you have a problem you haven't seen before, we suggest that you find an example of the task you are trying to do, and do the exercise.

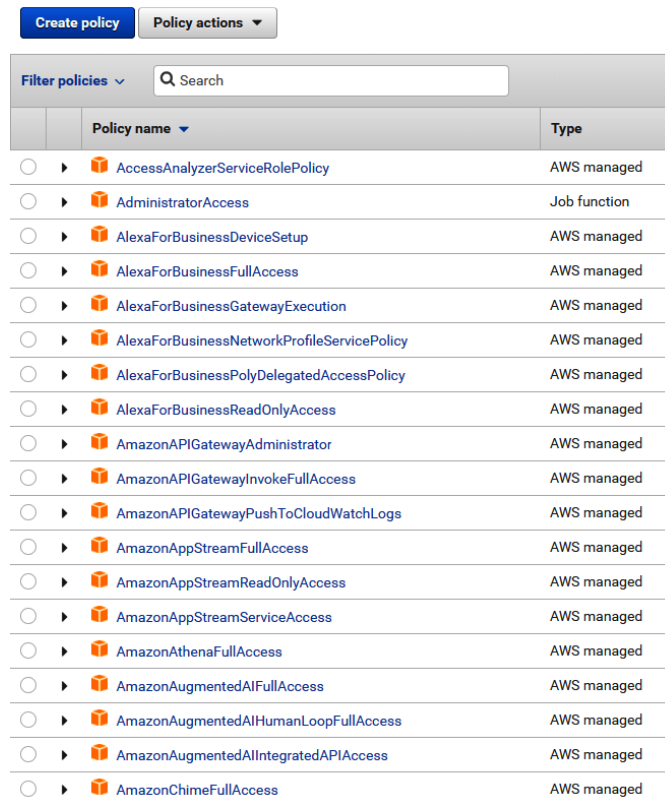
In our case, this was strange, since all we had to do was run a simple command. But our import was still failing. After a couple of hours of us banging our head against the wall, we decided to just behave like we knew nothing and went and did the *how do I import a VM into AWS?* example. Everything worked. Then, we tried importing our own machine; that didn't work. The commands were copy/pasted, but it still didn't work.

And then we realized the most important thing – *we need to pay attention to details*. Without this train of thought properly implemented and executed, we're inviting a world of problems upon ourselves.

Paying attention to details

To cut the story (that's way too long) short, what we did wrong was we misconfigured the identity service. In a cloud environment such as AWS, every service runs as an independent domain, completely separate from other services. When something needs to be done, the service doing it has to have some kind of authorization. There is a service that takes care of that – IAM – and the obvious default for every request from every service is to deny everything. Once we decide what needs to be done, it is our job to configure proper access and authorization. We knew that, and so we created all the roles and permissions for EC2 to access the files in S3. Even though that may sound strange, we had to give a service we are using the permission to get the files we uploaded. If you are new to, this you might expect this to be automatic, but it isn't.

Check out the following small excerpt, which is from the really long list of roles that AWS has predefined. Keep in mind that the complete list is much, much longer and that we've barely scratched the surface of all of the available roles. These are just roles that have names starting with the letter *A*:



The screenshot shows the AWS IAM console interface. At the top, there are buttons for 'Create policy' and 'Policy actions'. Below that is a search bar labeled 'Filter policies' and 'Search'. The main content is a table with two columns: 'Policy name' and 'Type'. Each row represents a predefined role, starting with a radio button and a right-pointing arrow. The roles listed are:

Policy name	Type
AccessAnalyzerServiceRolePolicy	AWS managed
AdministratorAccess	Job function
AlexaForBusinessDeviceSetup	AWS managed
AlexaForBusinessFullAccess	AWS managed
AlexaForBusinessGatewayExecution	AWS managed
AlexaForBusinessNetworkProfileServicePolicy	AWS managed
AlexaForBusinessPolyDelegatedAccessPolicy	AWS managed
AlexaForBusinessReadOnlyAccess	AWS managed
AmazonAPIGatewayAdministrator	AWS managed
AmazonAPIGatewayInvokeFullAccess	AWS managed
AmazonAPIGatewayPushToCloudWatchLogs	AWS managed
AmazonAppStreamFullAccess	AWS managed
AmazonAppStreamReadOnlyAccess	AWS managed
AmazonAppStreamServiceAccess	AWS managed
AmazonAthenaFullAccess	AWS managed
AmazonAugmentedAIFullAccess	AWS managed
AmazonAugmentedAIHumanLoopFullAccess	AWS managed
AmazonAugmentedAIIntegratedAPIAccess	AWS managed
AmazonChimeFullAccess	AWS managed

Figure 16.8 – AWS predefined roles

What we misconfigured was the name of the role – to import the VM into the EC2 instance, there needs to be a security role named `vmimport` giving EC2 the right permissions. We configured a role named `importvm` in our haste. When we completed the examples, we pasted the examples and everything was fine, but as soon as we started using our security settings, EC2 was failing to do its job. So, always *check the product documentation and read it carefully*.

Troubleshooting problems with the ELK stack

The ELK stack can be used to monitor our environment efficiently. It does require a bit of manual labor, additional configuration, and being a bit sneaky, but it can still offer reporting, automatic reporting, sending reports via email, and a whole lot of other valuable things.

Out of the box, you can't just send reports directly – you need to do some more snooping. You can use Watcher, but most of the functionality you need from it is commercial, so you'll have to spend some cash on it. There are some other methods, as well:

- Using snapshot for Kibana/Grafana – check out this URL: <https://github.com/parvez/snapshot>
- Using ElastAlert – check out this URL: <https://github.com/Yelp/elastalert>
- Use Elastic Stack Features (formerly X-Pack) – check out this URL: <https://www.elastic.co/guide/en/x-pack/current/installing-xpack.html>

Here's one more piece of advice: you can always centralize logs via `rsyslog` as it's a built-in feature. There are free applications out there for browsing through log files if you create a centralized log server (Adiscon LogAnalyzer, for example). If dealing with ELK seems like a bit too much to handle, but you're aware of the fact that you need something, start with something like that. It's very easy to install and configure and offers a free web-like interface with regular expression support so that you can browse through log entries.

Best practices for troubleshooting KVM issues

There are some common-sense best practices when approaching troubleshooting KVM issues. Let's list some of them:

- *Keep it simple, in configuration:* What good does a situation in which you deployed 50 OpenStack hosts across three subnets in one site do? Just because you can subnet to an inch of an IP range's life doesn't mean you should. Just because you have eight available connections on your server doesn't mean that you should LACP all of them to access iSCSI storage. Think about end-to-end configuration (for example, Jumbo Frames configuration for iSCSI networks). Simple configuration almost always means simpler troubleshooting.
- *Keep it simple, in troubleshooting:* Don't go chasing the super-complex scenarios first. Start simple. Start with log files. Check what's written there. With time, use your gut feeling as it will develop and you'll be able to trust it.
- *Use monitoring tools such as ELK stack:* Use something to monitor your environments constantly. Invest in some kind of large-screen display, hook it up to a separate computer, hang that display on a wall, and spend time configuring important dashboards for your environments.
- *Use reporting tools* to create multiple automated reports about the state of your environment: Kibana supports report generation, for example, in PDF format. As you monitor your environment, you will notice some of the *more sensitive* parts of your environments, such as storage. Monitor the amount of available space. Monitor path activity and network connections being dropped from host to storage. Create reports and send them automatically to your email. There's a whole world of options there, so use them.
- *Create notes while you configure your environment:* If nothing else, do this so that you have some starting point and/or a reference for future, as there will be many changes that are often done *on the fly*. And when the process of taking notes is finished, create documentation.
- *Create documentation:* Make these permanent, readable, and as simple as possible. Don't *remember* things, *write things down*. Make it a mission to write everything down, and try to spread that culture all around you.

Get used to having a large portion of <insert your favorite drink here> available at all times and many sleepless nights if you want to work in IT as administrator, engineer, or DevOps engineer. Coffee, Pepsi, Coca-Cola, lemon juice, orange juice.... whatever gets your intellectual mojo flowing. And sometimes, learn to walk away from a problem for a short period of time. Solutions often click in your head when you're thinking about something completely opposite to work.

And finally, remember to try and have fun while working. Otherwise, the whole ordeal of working with KVM or any other IT solution is just going to be an *Open Shortest Path First* to relentless frustration. And frustration is never fun. We prefer yelling at our computers or servers. It's therapeutic.

Summary

In this chapter, we tried to describe some basic troubleshooting steps that can be applied generally and when troubleshooting KVM. We also discussed some of the problems that we had to deal with while working with various subjects of this book – Eucalyptus, OpenStack, the ELK stack, cloudbase-init, storage, and more. Most of these issues were caused by misconfiguration, but there were quite a few where documentation was severely lacking. Whatever happens, don't give up. Troubleshoot, make it work, and celebrate when you do.

Questions

1. What do we need to check before deploying the KVM stack?
2. What do we need to configure after deploying the KVM stack in terms of making sure that virtual machines are going to run after reboot?
3. How do we check KVM guest log files?
4. How can we turn on and configure KVM debug logging permanently?
5. How can we turn on and configure KVM debug logging at runtime?
6. What's the best way to solve oVirt's installation problems?
7. What's the best way to solve oVirt's minor and major version upgrade problems?
8. What's the best way to manage the oVirt Engine and host updates?
9. Why do we need to be careful with snapshots?
10. What are the common problems with templates and cloudbase-init?
11. What should be our first step when installing Eucalyptus?
12. What kind of advanced capabilities for monitoring and reporting can we use with the ELK stack?
13. What are some of the best practices when troubleshooting KVM-based environments?

Further reading

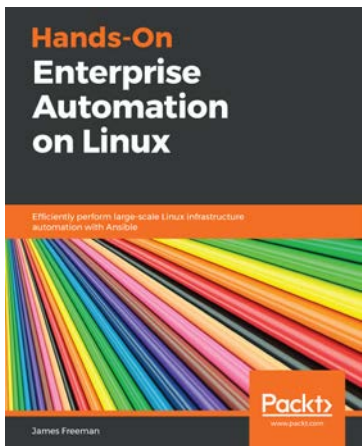
Please refer to the following links for more information regarding what was covered in this chapter:

- Working with KVM debug logging: <https://wiki.libvirt.org/page/DebugLogs>
- Firewall requirements for oVirt nodes and oVirt Engine: https://www.ovirt.org/documentation/installing_ovirt_as_a_standalone_manager_with_remote_databases/#dns-requirements_SM_remoteDB_deploy
- oVirt upgrade guide: https://www.ovirt.org/documentation/upgrade_guide/
- Common problems with NetApp and Openstack integration: https://netapp-openstack-dev.github.io/openstack-docs/stein/appendices/section_common-problems.html
- Integrating the IBM Storwize family and SVC driver in OpenStack: <https://docs.openstack.org/cinder/queens/configuration/block-storage/drivers/ibm-storwize-svc-driver.html>
- Integrating IBM Storwize and OpenStack: https://www.ibm.com/support/knowledgecenter/STHGuj_8.2.1/com.ibm.storwize.v5100.821.doc/storwize_openstack_matrix.html
- HPE Reference Architecture for the Red Hat OpenStack Platform on HPE Synergy with Ceph Storage: <https://www.redhat.com/cms/managed-files/cl-openstack-hpe-synergy-ceph-reference-architecture-f18012bf-201906-en.pdf>
- Integrating Dell EMC Unity and OpenStack: <https://docs.openstack.org/cinder/rocky/configuration/block-storage/drivers/dell-emc-unity-driver.html>
- DM-multipath configuration for Red Hat Enterprise Linux 7: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/dm_multipath/mpio_setup
- DM-multipath configuration for Red Hat Enterprise Linux 8: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/pdf/configuring_device_mapper_multipath/Red_Hat_Enterprise_Linux-8-Configuring_device_mapper_multipath-en-US.pdf

- Using snapshot for Kibana/Grafana: <https://github.com/parvez/snapshot>
- Using ElastAlert: <https://github.com/Yelp/elastalert>
- Using Elastic Stack Features (formerly X-Pack): <https://www.elastic.co/guide/en/elasticsearch/reference/current/setup-xpack.html>
- Troubleshooting OpenStack Networking: <https://docs.openstack.org/operations-guide/ops-network-troubleshooting.html>
- Troubleshooting OpenStack Compute: <https://docs.openstack.org/ocata/admin-guide/support-compute.html>
- Troubleshooting OpenStack Object Storage: <https://docs.openstack.org/ocata/admin-guide/objectstorage-troubleshoot.html>
- Troubleshooting OpenStack Block Storage: <https://docs.openstack.org/ocata/admin-guide/blockstorage-troubleshoot.html>
- Troubleshooting OpenStack Shared File Systems: <https://docs.openstack.org/ocata/admin-guide/shared-file-systems-troubleshoot.html>
- Troubleshooting a Bare Metal OpenStack service: <https://docs.openstack.org/ocata/admin-guide/baremetal.html#troubleshooting>

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

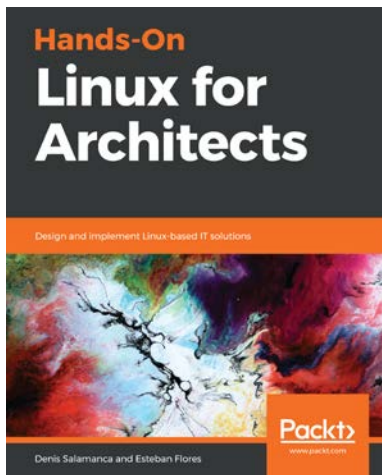


Hands-On Enterprise Automation on Linux

James Freeman

ISBN: 978-1-78913-161-1

- Perform large-scale automation of Linux environments in an enterprise
- Overcome common challenges and pitfalls of extensive automation
- Define the business processes needed to support a large-scale Linux environment
- Get well-versed with the most effective and reliable patch management strategies
- Automate a range of tasks from simple user account changes to complex security policy enforcement
- Learn best practices and procedures to make your Linux environment automatable



Hands-On Linux for Architects

Denis Salamanca , Esteban Flores

ISBN: 978-1-78953-410-8

- Study the basics of infrastructure design and the steps involved
- Expand your current design portfolio with Linux-based solutions
- Discover open source software-based solutions to optimize your architecture
- Understand the role of high availability and fault tolerance in a resilient design
- Identify the role of containers and how they improve your continuous integration and continuous deployment pipelines
- Gain insights into optimizing and making resilient and highly available designs by applying industry best practices

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

- Access Control List (ACL) 143
- Address Resolution Protocols (ARPs) 249
- ad hoc 388
- Advanced Host Controller
 - Interface (AHCI) 172
- Advanced Micro Devices (AMD) 247
- advanced troubleshooting tools
 - about 621
 - Ansible and OpenStack, problems
 - working with 627, 628
 - AWS 637
 - dependencies 628, 629
 - ELK stack, used for troubleshooting
 - problems 639
 - Eucalyptus, troubleshooting 629
 - KVM, storage problems 624
 - oVirt 622, 623
 - oVirt, storage problems 624
 - service, executing 638, 639
 - service, implementing 638, 639
 - snapshots and templates,
 - problems 624-626
 - agentless systems 369
- aggregator
 - configuring 536, 537
- Amazon Elastic Compute Cloud (EC2) 14
- Amazon Machine Image (AMI) 503
- Amazon Web Services (AWS)
 - about 14, 470, 637
 - big infrastructure 474, 475
 - cloud, approaching 470, 471
 - data centers 477, 478
 - image, uploading to EC2 498-507
 - key placement 478, 479
 - market share 474
 - multi-cloud 472
 - pricing 475-477
 - Shadow IT 473, 474
 - verbosity 637
 - virtual machines, converting 484
 - virtual machines, migrating 484-498
 - virtual machines, preparing 484
- AMD Virtualization (AMD-V) 21
- Ansible
 - about 366-372
 - approaches 366
 - deploying 382, 383
 - examples, with KVM 409, 410

- problems, working with 627, 628
- used, for integrating OpenStack 462-464
- using, for automation and orchestration 399-406

Ansible AWX 372

Apache Software Foundation (ASF) 14

App Engine 471

application programming interface (API) 259

Application Programming Interface (API) 30

application virtualization 6

ARK

- URL 563

ARP Flux

- reference link 601

automatic NUMA balancing 591-593

automation approaches, Ansible

- about 367
- agentless systems 369
- systems that use agents 368

avirt command 617

availability zone 508

AWS free tier

- reference link 480

AWS services

- about 480, 481, 482
- EC2 482
- IAM 483
- other services 483
- S3 482, 483

AWX

- about 371
- deploying 372-381
- reference link 373
- using 372-381

AWX project

- reference link 371

B

bare-metal hypervisors 10

basic KVM storage operations 162

blockcommit

- used, for merging external disk snapshots 302, 303

block I/O

- tuning 597-600

blockpull

- used, for merging external disk snapshots 304, 305

bridge zero copy 603

Brtrfs 127

C

CentOS 8 template

- preparing, with complete LAMP stack 266-270

central processing unit (CPU)

- about 215, 263
- tuning, with NUMA 585, 586

Ceph

- about 155
- using, as storage backend for KVM 155-162

ceph-mds 155

ceph-mon 155

ceph-osd 155

cloud

- Linux virtualization 14, 15

cloudbase-init

- used, for customizing Windows VMs 350-352

cloudbase-init customization

- examples 353-361
- issues, troubleshooting 361, 363

- cloud-config script
 - using, with cloud-init 323-329
- Cloud Controller 508
- CloudFront 481
- cloud-init
 - about 314, 399
 - architecture 315-317
 - configuring, at boot time 318
 - data sources 320
 - images 319, 320
 - installing 318
 - metadata, passing to 321, 322
 - modules 322
 - post installation 336-342
 - user data, passing to 321, 322
 - using, for automation and orchestration 399-406
- cloud-init boot, stages
 - config stage 316
 - final stage 316
 - generator 315
 - local phase 315
 - network phase 315
- cloud provider scalability, problems
 - compute problem 417
 - network problem 418
 - storage problem 418
- CloudStack 14
- cloud-utils 321
- Cockpit
 - used, for creating VM 223-225
- container-based virtualization 9
- Content Delivery Network (CDN) 471
- Controlled Replication Under Scalable Hashing (CRUSH) 155
- control node 383
- Copy-on-Write (COW) 265, 580

- core concept, Xen
 - Dom0 12
 - management utilities 12
 - virtual machines 12
 - Xen Hypervisor 12
- CPU pinning 568-571
- custom utilization reports
 - creating 538, 539

D

- dashboard 435
- data centers 477, 478
- data collector
 - configuring 536
- data points 525
- debug mode logging
 - enabling 618, 619, 620
- design 560, 561
- Designate 435, 436
- Desired State Configuration (DSC) 367
- desktop virtualization 6
- disk
 - attaching, with virsh 166
 - attaching, with virt-manager 164-166
- disk allocation methods
 - preallocated 163
 - thin-provisioned 163
- Distributed Resource Scheduler (DRS) 216
- Domain Name System (DNS) 240, 262

E

- Elastic Compute Cloud (EC2) 471, 481, 482
- Elasticsearch
 - about 526, 527

- configuring 539
 - Elasticsearch, Logstash, Kibana
 - (ELK) 7, 526
 - and KVM 549-556
 - ELK stack
 - development mode 538
 - integrating 528-533
 - production mode 538
 - setting up 528-532
 - used, for troubleshooting problems 639
 - workflow, creating 533-535
 - embedded hypervisors 10
 - emulator
 - QEMU, operating as 39
 - emulatorpin 589-591
 - Enterprise Virtualization
 - Hypervisor (RHEV-H) 10
 - ESX integrated (ESXi) 214, 257
 - euca2ools
 - reference link 508
 - Euca2ools 508
 - Eucalyptus
 - about 14, 507
 - information gathering 630-636
 - installation, pre-requisites 510
 - installing 509-518
 - troubleshooting 629
 - used, for building hybrid
 - KVM clouds 507, 508
 - using, for AWS control 518-521
 - Eucalyptus 4.4.5 installation guide
 - reference link 509
 - Eucalyptus, components
 - Cloud Controller (CLC) 508
 - Cluster Controller (CC) 508
 - eucanetd 508
 - Node Controller (SC) 508
 - Storage Controller (SC) 508
 - execution flow
 - of vCPU 59-64
 - Extended Page Tables (EPT) 22, 50
 - Extensible Markup Language (XML) 222
 - external disk snapshots
 - creating 293-296
 - deleting 302
 - merging, blockcommit used 302, 303
 - merging, blockpull used 304, 305
 - quiesce 296, 297
 - reverting to 298-301
 - working with 292, 293
 - external snapshot 286
- ## F
- FastCGI Process Manager (FPM) 269
 - FastStart 509
 - fault domains 477
 - Fiber Channel (FC) 239
 - File Transfer Protocol (FTP) 214
 - Front Side Bus (FSB) 174
 - full cloning method
 - about 265
 - used, for deploying VM 276, 277
 - full virtualization 8, 26, 27
 - Fully Automated Install (FAI) 313
- ## G
- gather_facts module 370
 - GDPR 479
 - General Public License (GPL) 38
 - GENEVE 425
 - gigabytes (GB) 213
 - Glance 434
 - Global Accelerator 481

Gluster
 using, as storage backend
 for KVM 150-154

GPU
 partitioning, with NVIDIA
 vGPU 187, 188, 189

GPU PCI passthrough
 enabling 189-193

graphical user interface (GUI) 286

guestfish
 using 259-262

H

hardware-assisted virtualization 28, 29

hardware-based approach 8

hardware design 561-564

high availability (HA) 226

Horizon 435

hosted hypervisors 11

host's NUMA topology 596

HugePages 576

hybrid KVM clouds
 building, with Eucalyptus 507, 508

hybrid virtualization 8

hypercalls 28

HyperText Transfer Protocol (HTTP) 214

HyperText Transfer Protocol
 Secure (HTTPS) 214

hypervisor
 type 1 hypervisor 10, 11
 type 2 hypervisor 10, 11
 using 9

Hyper-V Network Virtualization
 (HNV) 425

I

Identity and Access Management
 (IAM) 481, 483

image information
 obtaining 164

Infrastructure-as-a-Service (IaaS) 14, 416

input/output (I/O) 231

input/output (I/O)-intensive
 applications 285

Input/Output Memory Management
 Unit (IOMMU) 22

integrated drive electronics (IDE) 234

internal snapshot
 about 285
 creating 287
 creating, with custom name
 and description 288
 deleting 290, 291
 multiple snapshots, creating 288, 289
 reverting to 290
 working with 287

International Organization for
 Standardization (ISO) 213

Internet of Things (IoT) 182

Internet Protocol (IP) 212, 263

Internet Small Computer Systems
 Interface (iSCSI) 239

inventories 370

iothread 49

iSCSI
 using 136-145

ISO image library
 creating 167, 168

J

Just-In-Time (JIT) compiler 39

K

Keystone 436

Kibana

about 528

charts, creating 537, 538

configuring 541-548

kickstart file 77

KSM 580-584

and NUMA 591

Kernel-based Virtual Machine (KVM)

about 5, 13, 210

anonymous inodes 53, 54

Ansible, examples using 409

file structures 53, 54

guest mode 50

installing 393-399

internal working 49, 50

storage problems 623, 624

virtual machine, installing 76

Windows VMs, creating prerequisites 346

KVM APIs 52

kvm-clock 605, 606

KVM guest time-keeping

best practices 604

KVM internals 41

KVM issues

best practices, for troubleshooting 640

kvm_libvirt module

used, for provisioning virtual machine 383-387

KVM services logging 617

KVM service status

verifying 614-617

kvm structure 55-58

KVM virtualization platform

monitoring 524, 525

KVM VM

multi-tier application deployment, orchestrating 406, 408, 409

L

libguestfs tools

guestfish, using 259-262

used, for modifying VM images 256

virt-builder 281

virt-p2v, using 259

virt-v2v, using 257, 258

libvirt

about 70

installing 73, 75

internal working 30

URL 30

used, for starting virtual machine 83-86

libvirtd 30

libvirt isolated network 95-100

libvirt NAT network 93, 94

libvirt routed network 94, 95

libvirt storage pools 130

linear design 561

linked cloning method

about 265

used, for deploying VM 277-280

Linux, Apache, MySQL, and PHP (LAMP) 266

Linux bridging

implementing 103-105

- Linux virtualization
 - about 4, 5
 - in cloud 14, 15
- live migration, VM 247-252
- local.conf file 446
- local storage pools 128, 129
- logical unit number (LUN) 245
- Logical Volume Manager (LVM)
 - about 239
 - configuring 141-143
- Logstash
 - about 527
 - installing 540

M

- macvtap
 - about 118
 - bridge mode 120
 - passthrough mode 120
 - private mode 119
 - VEPA mode 118
- market share 474
- media access control (MAC) 266
- memory
 - tuning, with NUMA 585, 586
 - working with 572
- memory allocation 572
- memory backing 575
- memory backing, options
 - hugepages 576-580
 - locked 575
 - nosharepages 576
- Memory Management Unit (MMU) 576
- Memory Overcommitment Manager (MOM) 72

- memory performance
 - tuning 565-567
- memory tuning 573, 574
- metadata
 - about 320
 - passing, to cloud-init 321, 322
- metricbeat 528
- Microsoft Azure 471
- Microsoft SQL database
 - used, for preparing Windows Server 2019 template 273-275
- Modular Layer 2 (ML2) 437
- modules 367, 370
- multi-cloud
 - about 472
 - need for 472, 473
- multipathing 146
- multi-tier application deployment
 - orchestrating, on KVM VM 406-409

N

- native hypervisors 10
- Nested Paging Table (NPT) 50
- Network-as-a-Service (NaaS) 426
- Network File System (NFS) 240
- networking interfaces, Linux
 - Bond 92
 - Bridge 92
 - IPOIB 92
 - IPVLAN 92
 - MACVLAN 92
 - MACVTAP/IPVTAP 92
 - Team 92
 - VETH 92
 - VXLAN 92
- network interface controllers (NICs) 249

- network I/O
 - tuning 601, 602
 - Network Time Protocol (NTP) 240, 604
 - network virtualization 7
 - Neutron 437, 438
 - Neutron, network categories
 - project networks 438
 - provider networks 438
 - self-service networks 438
 - tenant networks 438
 - Neutron, network types
 - flat 438
 - local 438
 - VLAN 438
 - NFS server options
 - configuring 135
 - NFS storage pool
 - about 131, 132
 - configuration options 133
 - selecting 134
 - Non-Unified Memory Access (NUMA) 20
 - Non-Uniform Memory Access (NUMA)
 - about 231, 561
 - and KSM 591
 - CPU, tuning with 585, 586
 - memory allocation policies 586-589
 - memory, tuning with 585, 586
 - Non-Volatile Memory Express (NVMe) 172
 - Nova
 - about 430-432
 - Reference link 432
 - nova-api 432
 - nova-compute 433
 - nova-conductor 434
 - nova-network 433
 - nova-volume 433
 - noVNC
 - used, for obtaining display portability 202-206
 - NUMA Affinity Management
 - Daemon (numad) 594
 - numactl command 593
 - NUMA node 585
 - numastat command 595
 - numatune 586
 - NVIDIA vGPU
 - used, for partitioning GPU 187-189
 - NVMe over Fabrics (NVMe-OF) 175
- ## O
- Object Storage Daemon (OSD) 155
 - Office 365 471
 - offline migration, VM
 - about 243, 244
 - lockd, enabling 245-247
 - open source cloud projects
 - CloudStack 14
 - Eucalyptus 14
 - OpenStack 14
 - open source ELK solution
 - about 526
 - Elasticsearch 526, 527
 - Kibana 528
 - Logstash 527
 - open source virtualization projects
 - about 11
 - KVM 13
 - Xen 12, 13
 - OpenStack
 - about 14, 416-418
 - day-to-day administration 457-460
 - identity management 460, 461

- installing 445-449
- Packstack demo environment,
 - creating for 441-443
- problems, working with 627, 628
- URL 417, 465
- use cases 439, 440
- virtualization solutions 15
- OpenStack administration 449-456
- OpenStack-Ansible (OSA)
 - about 462
 - reference link 463
- OpenStack, components
 - about 426
 - Designate 436
 - Glance 434
 - Horizon 435
 - Keystone 436
 - Neutron 437, 438
 - Nova 430-432
 - Swift 427-429
 - Swift daemons 429, 430
- OpenStack Compute 426
- OpenStack environment
 - provisioning 443, 444
- OpenStack, integrating with Ansible
 - about 462-464
 - Ansible deployment server,
 - installing 464, 465
 - Ansible inventory, configuring 466
 - Ansible playbooks, running 467
- Open Virtualization Appliance (OVA) 257
- Open Virtualization Format (OVF) 257
- Open vSwitch
 - architecture 107
 - configuring 105-112
 - flow mode 108
 - installing, on CentOS 8 108, 110

- normal mode 108
- use cases 113, 114
- working 106, 108
- orchestration 416
- oVirt
 - about 71, 622, 623
 - architecture 73
 - functionalities 72
 - installing 73-82
 - storage problems 623, 624
 - URL 30
 - used, for creating VM 226-229
- oVirt Engine Administration Portal 83
- oVirt, example on iSCSI
 - using 146-149

P

- Packstack 441
- Packstack demo environment
 - creating, for OpenStack 441-443
- Packstack documentation
 - reference link 443
- paravirtualization
 - about 8, 27
 - URL 28
- partitioning 8
- PC over IP (PCoIP) 179
- Persistent Memory (PM) 174
- physical environment
 - versus virtualized environment 18, 19
- Physical Functions (PF) 23
- physical graphics cards
 - in VDI scenarios 185-187
- physical networking 90
- physical-to-virtual (P2V) conversion 256

- playbook
 - about 387
 - working with 387-393
- Point of Contact (POC) 478
- pong 390
- PowerShell 369, 383
- Preboot eXecution Environment (PXE) 213
- Precision Time Protocol (PTP) 240, 605
- pricing 475-477
- Privacy Shield 479
- Proof of Concept (POC) 441
- protection rings 24

Q

- qemu-img 222
- Quality of Service (QoS) 252, 599
- Quantum 437
- Quick Emulator (QEMU)
 - about 12, 235
 - data structures 42, 44, 45, 46
 - installing 73, 75
 - internal working 38
 - KVM internals 41
 - operating, as emulator 39
 - operating, as virtualizer 40, 41
 - threading models 48, 49
 - URL 39
 - used, for starting virtual machine 83-86
- Quick Path Interconnect (QPI) 174
- quiesce 296, 297

R

- Rapid Virtualization Indexing (RVI) 22
- Red Hat Enterprise Linux (RHEL) 216

- Red Hat Enterprise Linux (RHEL) 8.0
 - URL 259
- Red Hat Enterprise Virtualization Hypervisor (RHEV-H) 146
- redundancy 146
- Registry 434
- Reliable Autonomic Distributed Object Store (RADOS) 155
- remote desktop protocol (RDP) 179
- remote display protocols
 - about 193
 - history 193, 194
 - types 195
- replica 429
- risk zones 477
- Rundeck 372

S

- SAN storage
 - using 136
- scheduler 432
- scheduler, Nova
 - chance 433
 - simple scheduler 433
 - zone scheduling 433
- Second-Level Address
 - Translation (SLAT) 22
- Secure Copy Protocol (SCP) 248
- Secure Shell (SSH) 212, 263
- Secure Virtual Machine (SVM) 29
- Security-Enhanced Linux (SELinux) 243, 267
- Semaphore 372
- Server Message Block (SMB) 214
- server virtualization 6
- service-level agreement (SLA) 417

-
- Shadow IT 473, 474
 - Simple Protocol for Independent Computing Environments (SPICE)
 - about 198
 - reference link 198
 - Simple Storage Service (S3) 471, 481, 482
 - Single Point Of Failure (SPOF) 125
 - Single Root Input Output
 - Virtualization (SR/IOV) 22
 - small computer system
 - interface (SCSI) 234
 - snapshots
 - about 285
 - external disk snapshots,
 - working with 292
 - internal snapshots, working with 286
 - managing, with virt-manager 291, 292
 - problems 624-626
 - use cases 305
 - using, best practices 305
 - software-based approach 8
 - software-based design 606-609
 - software-defined networking (SDN)
 - about 7, 93, 418-420
 - GENEVE 425
 - VXLAN 420-422
 - Software-Defined Storage (SDS) 7
 - Solid State Drives (SSDs) 172
 - Spanning Tree Protocol (STP) 103
 - SPICE display protocol
 - using 198
 - SPICE graphics server
 - adding 198, 199
 - SR-IOV
 - about 114
 - using 115-118
 - SSH 369
 - storage
 - about 124, 125
 - latest developments 172-175
 - Storage Class Memory (SCM) 174
 - storage pool
 - about 126
 - deleting 169
 - libvirt storage pools 130
 - local storage pools 128, 129
 - types 126, 127
 - storage virtualization 7
 - storage volumes
 - creating 170
 - Stratis 128
 - Structured Query Language (SQL) 273
 - Swift
 - about 427-429
 - URL 427
 - Swift-account 429
 - Swift-container 430
 - Swift daemons 429, 430
 - Swift-object 430
 - Symmetric Multiprocessor System (SMP) 585
 - System Center Operations Manager (SCOM) 216
 - system gauge
 - creating 549-556
 - system identification number (SID) 274
 - System on a Chip (SoC) 182
 - System Preparation (sysprep) tool
 - URL 273
 - systems that use agents 368

T

- TAP devices
 - userspace networking, using
 - with 101, 102
- templates
 - creating 266
 - creating, examples 266-273
 - problems 624-626
 - virt-sysprep 270, 272
 - working with 266
- terabyte (TB) 222
- Time Stamp Counter (TSC) 605
- Tiny Code Generator (TCG)
 - about 39
 - reference link 39
- Translation Lookaside Buffer (TLB) 22, 577
- Transmission Control Protocol (TCP) 248
- Transparent Hugepages (THP) 577
- Trusted Platform Module (TPM) 23
- TUN devices
 - userspace networking, using
 - with 101, 102
- tunneling interfaces
 - GENEVE 92
 - GRE 92
 - GRETAP 92
 - IP6GRE 92
 - IP6GRETAP 92
 - ip6tnl 92
 - IPIP 92
 - SIT 92
- type 1 hypervisor
 - about 10, 11
 - advantages 10
- type 2 hypervisor 10, 11

U

- Uniform Resource Locator (URL) 214
- universally unique identifier (UUID) 160, 246
- user data
 - about 320
 - passing, to cloud-init 321, 322
- user mode Linux (UML) 15, 433
- userspace networking
 - using, with TAP devices 101, 102
 - using, with TUN devices 101, 102

V

- vCPU
 - execution flow 59-64
- VDI scenarios
 - physical graphics cards 185-187
 - virtual graphics cards 185-187
- vdsm 72
- vhost
 - turning on 602-604
- vhost kernel modules
 - checking 602-604
- virsh
 - about 30
 - used, for attaching disk 166
- virsh binary
 - using, for remote connection 31-38
- virsh command
 - used, for creating volume 171
 - used, for deleting volume 171
- virt-builder
 - about 281, 283
 - repositories 283-285
- virt-clone command 220, 221

- virt-* commands
 - using 217
- virt-convert command 222
- virt-install utility
 - used, for creating Windows VMs 347-349
- virtio-blk 597
- Virtio devices
 - tuning 596, 597
- virt-manager
 - URL 30
 - used, for attaching disk 164-166
 - used, for creating VM 210
 - used, for managing snapshots 291, 292
 - using 210-215
- virt-p2v
 - using 259
- Virt queue 597
- virt-sysprep 270, 272
- virtual CPU (vCPU) 215
- Virtual Desktop Infrastructure (VDI) 6, 179, 198, 265, 345
- Virtual Direct Graphics
 - Acceleration (vDGA) 186
- virtual disk images 162
- Virtual Ethernet Port Aggregator (VEPA) 118
- Virtual Functions (VF) 22
- virtual graphic card, use cases
 - cg3 181
 - cirrus 180
 - none 181
 - qxl 180
 - std 180
 - tcx 180
 - virtio 181
 - vmware 180
 - virtual graphics cards
 - in VDI scenarios 185-187
- virtual hardware
 - adding, in VM 236, 237
 - removing, from VM 236, 237
- virtualization
 - about 18
 - from CPU perspective 50-52
 - hardware requirements 21, 23
 - need for 20
 - software requirements 24-26
- Virtualization Technology (VT) 29
- virtualization, types
 - about 6, 7
 - application virtualization 6
 - container-based virtualization 9
 - desktop virtualization 6
 - full virtualization 8
 - hybrid virtualization 8
 - network virtualization 7
 - paravirtualization 8
 - partitioning 8
 - server virtualization 6
 - storage virtualization 7
- virtualized environment
 - versus physical environment 18, 19
- Virtualized Functions (VF) 187
- virtualized networking 90
- virtualizer
 - QEMU, operating as 40, 41
- virtual LAN (VLAN ID) 418
- virtual local area network (VLAN) 252, 263
- virtual machine
 - Ansible, using for automation and orchestration 399-406

- cloud-init, using for automation
 - and orchestration 399-406
- deploying 329-335
- installation, automating 77-80
- installing, in KVM 76
- KVM, installing 393-399
- playbook, working with 388-393
- provisioning, with `kvm_libvirt` module 383-387
- starting, with `libvirt` 83-86
- starting, with `QEMU` 83-86
- virtual machine console
 - accessing, methods 200, 201
- Virtual Machine Control
 - Block (VMCB) 50
- Virtual Machine Control Structure (VMCS) 50, 51
- virtual machine customization 624-626
 - need for 312, 313
- Virtual Machine Extensions (VMX) 21, 50
- virtual machine manager (VMM)
 - about 80
 - using 9
- Virtual Machine Monitor (VMM) 26
- virtual machines (VMs)
 - about 345, 560
 - configuring 230-235
 - converting, for AWS 483
 - creating, with `Cockpit` 223-225
 - creating, with `oVirt` 226-229
 - creating, with `virt-manager` 210
 - deploying, from template 275
 - deploying, full cloning
 - method used 276, 277
 - deploying, linked cloning
 - method used 277-280
 - preparing, for AWS 483
 - starting, accidentally on two hypervisors 244
- virtual machines (VMs), importing
 - with VM Import/Export reference link 485
- Virtual Memory Areas (VMAs) 577
- Virtual Network Computing (VNC) 218
- virtual networking 91
- Virtual Private Server (VPS) 600
- virtual ring (`vring`)
 - tuning 597
- Virtual Shared Graphics
 - Acceleration (vSGA) 186
- virtual switch 90, 91
- `virt-v2v`
 - using 257, 258
- `virt-viewer` command 218
- `virt-xml` command 219
- VM CPU
 - tuning 565-567
- VM design 565
- VM Disk (VMDK) 222
- VM display devices
 - using 180-184
- VM images
 - modifying, with `libguestfs` tools 256
- VM migration
 - about 238
 - benefits 239
 - environment, setting up 240-242
 - live migration 247-252
 - offline migration 243, 244
 - requisites, for production environment 239
- VM storage
 - configuring 215, 216

VM templates
 creating 263-265

VNC display protocol
 need for 197
 using 196, 197

volume
 creating, with virsh command 171
 deleting, with virsh command 171

VT-i 28

VXLAN
 about 420-422
 UDP, need for 421

VXLAN, features
 Layer 2 across sites, stretching 423, 424
 layer 2, bridging 423
 offloading technologies 423

VXLAN network identifiers (VNIs) 421

W

Web Server Gateway Interface
 (WSGI) 431

Windows Azure 471

Windows Remote Management
 (WinRM) 350, 369, 383

Windows Server 2019 template
 creating, with Microsoft SQL
 database 273-275

Windows VMs
 creating, on KVM prerequisites 346
 creating, with virt-install utility 347-349
 customizing, with
 cloudbase-init 350-352

X

Xen
 about 5, 12, 13
 URL 12

Y

yum module 395

Z

ZFS 127

Zones 508

