Community Experience Distilled

OpenStack Essentials

Demystify the cloud by building your own private OpenStack cloud

Dan Radez

[PACKT] PUBLISHING open source* community experience distilled

www.allitebooks.com

OpenStack Essentials

Demystify the cloud by building your own private OpenStack cloud

Dan Radez

BIRMINGHAM - MUMBAI

www.allitebooks.com

OpenStack Essentials

Copyright © 2015 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, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

First published: May 2015

Production reference: 1190515

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-708-5

www.packtpub.com

Cover image by Bartosz Chucherko (chucherko@gmx.com)

www.allitebooks.com

Credits

Author
Dan Radez

Reviewers
Will Foster
Mostafa A. Hamid
Alvaro Lopez Ortega
Clay Shelor

Acquisition Editors
Sam Wood
Purav Motiwalla

Content Development Editor
Rohit Singh

Technical Editor
Siddhesh Patil

Copy Editor
Sarang Chari

Project Coordinator
Mary Alex

Proofreaders
Stephen Copestake
Safis Editing

Indexer
Mariammal Chettiyar

Production Coordinator
Alwin Roy

Cover Work
Alwin Roy

www.allitebooks.com

About the Author

Dan Radez joined the OpenStack community in 2012 in an operator role. His experience has centered around installing, maintaining, and integrating OpenStack clusters. He has been extended offers internationally to present OpenStack content to a range of experts. Dan's other experience includes web application programming, systems release engineering, virtualization product development, and network function virtualization. Most of these roles have had an open source community focus to them. In his spare time, Dan enjoys spending time with his wife and three boys, training for and racing triathlons, and tinkering with electronics projects.

I would like to thank Packt Publishing for giving me the opportunity to write my first book. A big thank you goes to my wife for her encouragement and support throughout the time I was writing this book. She takes excellent care of me and my kids. Thanks also to Chris Alfonso for referring Packt's inquiry to me and for his hospitality during the month my family ransacked his house. I'd also like to thank my friends and colleagues, Clay Shelor, Alvero Lopez Ortega, and Will Foster. These gentlemen provided feedback and reviews invaluable to my content being properly written and coherent for your consumption. Finally, I'd like to thank the Lord for the life and breath given to His creation for the purpose of His glory.

www.allitebooks.com

About the Reviewers

Will Foster is originally from Raleigh, North Carolina. He attended The Citadel, The Military College of South Carolina, in 1996, to pursue a degree in english. He was a performing member of the Summerall Guards, the elite close order Prussian drill unit, as well as a cadet officer within the Tango Company class of 2000. He also holds a degree in technical writing from Appalachian State University and is a Red Hat Certified Engineer.

Since 2000, Will has been working as a UNIX/Linux systems administrator involved in mission-critical, customer-facing production business environments. A lifelong skateboard enthusiast, Will had a brief stint as a snowboard instructor during 2000-2001.

Will has been working at Red Hat since 2007 as a senior systems administrator / DevOps engineer managing enterprise IT storage and core infrastructure. Currently, he works in the OpenStack deployment team. This team designs, architects, and builds laboratories and infrastructure to test and vet real-world customer deployments and cloud scenarios. They also collaborate with the upstream development community and partners to improve and build upon the OpenStack platform.

Will currently resides in Dublin, Ireland, and works in the same development operations deployment team as the author, Dan Radez.

www.allitebooks.com

Mostafa A. Hamid is an information systems engineer from State University of New York (SUNY), Potsdam. He is a Certified Information Systems Security Professional (CISSP), Rational Unified Process (RUP) architect, and has a Linux Professional Institute Certification (LPIC). Besides these, he has certifications in JavaScript, PHP, Backbone.js, and ethical hacking from SUNY Potsdam. He is also a certified Java programmer from American University, Cairo.

Mostafa has worked with Manon Systems. He has also worked as a technical support engineer for United Systems, TP-LINK, and Hilton Worldwide. He was employed as an ICT teacher at MOIS and is currently working as a software developer at Wassaq. Mostafa has contributed to PHP classes and was nominated for an award. He currently contributes to United Nations, Launchpad.net, and Stackoverflow.org.

I would like to thank Manon Niazi, whom I met in college—she means a lot to me, the Deutschlander; my mother and my family for their help at home; Mary Alex for her coordination of the project activities; Siddhesh Patil for his assistance and instructions on the technical part; and all the employees at Packt Publishing—thank you everyone for giving me an opportunity to review this book. Special thanks to the author of this book, Dan Radez. The reviewing process was a cherishable experience.

Alvaro Lopez Ortega is a well-known leader in the open source community. He is member of the GNU project and a contributor to OpenStack. He's also a former GNOME developer and OpenSolaris core contributor. He is a veteran speaker at open source conferences worldwide.

Currently, Alvaro works as an engineering manager for OpenStack R&D at Red Hat. During 15 years of his professional career, Alvaro held several leader positions with technology companies around the open source ecosystem, including product strategy engineering management at Canonical and OpenSolaris technical lead at Sun Microsystems.

www.allitebooks.com

Clay Shelor has worked as an English teacher, in network operations, and as a team leader doing IT staff augmentation. He loves to gather information, put the pieces together, implement a project, and then write about it for others to learn. When not at work, he enjoys time with the family, reading, music, and tug of war with the family dog.

Many thanks to Dan Radez for sharing his lifework with me and allowing me to come along for the ride on this project. Dan is exemplary in his work and a great friend. A big thank you goes to Mary Alex for the encouragement to keep me going.

---

www.allitebooks.com

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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 www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.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.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

www.allitebooks.com

Table of Contents

Preface **v**

Chapter 1: Architecture and Component Overview **1**

- **OpenStack architecture** **1**
- **Dashboard** **2**
- **Keystone** **2**
- **Glance** **4**
- **Neutron** **5**
- **Nova** **5**
- **Cinder** **6**
- **Swift** **7**
- **Ceilometer** **7**
- **Heat** **7**
- **Summary** **8**

Chapter 2: RDO Installation **9**

- **Installing RDO using Packstack** **10**
  - Preparing nodes for installation 11
  - Installing Packstack and generating an answer file 12
- **Summary** **17**

Chapter 3: Identity Management **19**

- **Services and endpoints** **19**
  - Hierarchy of users, tenants, and roles 20
  - Creating a user 21
  - Creating a tenant 22
  - Granting a role 22
  - Logging in with the new user 23
- **Interacting with Keystone in the dashboard** **24**
- **Endpoints in the dashboard** **26**
- **Summary** **27**

www.allitebooks.com

Chapter 4: Image Management 29
Glance as a registry of images 29
Downloading and registering an image 30
Using the web interface 32
Building an image 34
Summary 36
Chapter 5: Network Management 37
Networking and Neutron 37
Network fabric 38
Open vSwitch configuration 38
VLAN 39
GRE tunnels 39
VXLAN tunnels 40
Creating a network 40
Web interface management 42
External network access 46
Preparing a network 46
Creating an external network 50
Web interface external network setup 51
Summary 54
Chapter 6: Instance Management 55
Managing flavors 55
Managing key pairs 56
Launching an instance 58
Managing floating IP addresses 59
Managing security groups 60
Communicating with the instance 61
Launching an instance using the web interface 62
Summary 67
Chapter 7: Block Storage 69
Use case 69
Creating and using block storage 69
Attaching the block storage to an instance 70
Managing Cinder volumes in the web interface 72
Backing storage 75
Cinder types 75
GlusterFS setup 76
Summary 79

Chapter 8: Object Storage **81**
Use case **81**
Architecture of a Swift cluster **81**
Creating and using object storage **82**
Object file management in the web interface **83**
Using object storage on an instance **85**
Ring files **86**
Creating ring files 86
Summary **88**
Chapter 9: Telemetry **89**
Understanding the data store **89**
Definitions of Ceilometer's configuration terms **90**
Pipelines 90
Meters 90
Samples 91
Statistics 91
Alarms 92
Graphing the data **93**
Summary **96**
Chapter 10: Orchestration **97**
About orchestration **97**
Writing templates **97**
The AWS CloudFormation format 98
The Heat Orchestration Template (HOT) format 99
Launching a stack **99**
Autoscaling instances with Heat **102**
LBaaS setup **102**
Web interface **110**
Summary **113**
Chapter 11: Scaling Horizontally **115**
Scaling compute nodes **115**
Installing more control and networking **117**
Scaling control and network services **119**
Load-balancing keystone **119**
Additional Keystone tuning 122
Glance load balancing **122**
Scaling other services **124**
High availability **124**
Highly available database and message bus **126**
Summary **126**

Chapter 12: Monitoring 127

Monitoring defined 127
Installing Nagios 128
Adding Nagios host checks 128
Nagios commands 129
Monitoring methods 131
Non-OpenStack service checks 133
Monitoring control services 134
Monitoring network services 137
Monitoring compute services 139
Summary 140

Chapter 13: Troubleshooting 141

The debug command line option 141
Tail the server logs 142
Troubleshooting Keystone and authentication 143
Troubleshooting Glance image management 145
Troubleshooting Neutron networking 145
Troubleshooting Nova launching instances 149
Troubleshooting post-boot metadata 150
Troubleshooting console access 152
Troubleshooting Cinder block storage 152
Troubleshooting Swift object storage 153
Troubleshooting Ceilometer Telemetry 153
Troubleshooting Heat orchestration 153
Getting more help 154
Summary 154

Index 155

Preface

The cloud has risen in popularity and function in the past few years. Storing data and consuming computing resources on a third party's hardware reduces the overhead of operations by keeping the number of people and owned assets low. For a small company, this could be an opportunity to expand operations, whereas for a large company, this could help to streamline costs. The cloud not only abstracts the management of the hardware that an end user consumes, it also creates an on-demand provisioning capability that was previously not available to consumers. Traditionally, provisioning new hardware or virtualized hardware was a fairly manual process that would often lead to a backlog of requests, thus stigmatizing this way of provisioning resources as a slow process.

The cloud grew in popularity mostly as a public offering in the form of services accessible to anyone on the Internet and operated by a third party. This paradigm has implications for how data is handled and stored and requires a link that travels over the public Internet for a company to access the resources they are using. These implications translate into questions of security for some use cases. As the adoption of the public cloud increased in demand, a private cloud was birthed as a response to addressing these security implications. A private cloud is a cloud platform operated without a public connection, inside a private network. By operating a private cloud, the speed of on-demand visualization and provisioning could be achieved without the risk of operating over the Internet, paying for some kind of private connection to a third party, or the concern of private data being stored by a third-party provider.

Enter OpenStack, a cloud platform. OpenStack began as a joint project between NASA and Rackspace. It was originally intended to be an open source alternative that has compatibility with the Amazon Elastic Compute Cloud (EC2) cloud offering. Today, OpenStack has become a key player in the cloud platform industry. It is in its fifth year of release, and it continues to grow and gain adoption both in its open source community and the enterprise market.

In this book, we will explore the components of OpenStack. Today, OpenStack offers virtualization of compute, storage, networking, and many other resources. We will walk though installation, use, and troubleshooting of each of the pieces that make up an OpenStack installation. By the end of this book, you should not only recognize OpenStack as a growing and maturing cloud platform, but also have gained confidence in setting up and operating your own OpenStack cluster.

What this book covers

Chapter 1, *Architecture and Component Overview*, outlines a list of components that make up an OpenStack installation and what they do. The items described in this chapter will be the outline for most of the rest of the book.

Chapter 2, *RDO Installation*, is a step-by-step walkthrough to install OpenStack using the RDO distribution.

Chapter 3, *Identity Management*, is about Keystone, the OpenStack component that manages identity and authentication within OpenStack. The use of Keystone on the command line and through the web interface is covered in this chapter.

Chapter 4, *Image Management*, is about Glance, the OpenStack component that stores and distributes disk images for instances to boot from. The use of Glance on the command line and through the web interface is covered in this chapter.

Chapter 5, *Network Management*, talks about Neutron, the OpenStack component that manages networking resources. The use of Neutron on the command line and through the web interface is covered in this chapter.

Chapter 6, *Instance Management*, discusses Nova, the OpenStack component that manages virtual machine instances. The use of Nova on the command line and through the web interface is covered in this chapter.

Chapter 7, *Block Storage*, talks about Cinder, the OpenStack component that manages block storage. The use of Cinder on the command line and through the web interface is covered in this chapter.

Chapter 8, *Object Storage*, discusses Swift, the OpenStack component that manages object storage. The use of Swift on the command line and through the web interface is covered in this chapter.

Chapter 9, *Telemetry*, discusses Ceilometer, the OpenStack component that collects telemetry data. Swift's command-line usage and basic graph generation are discussed in this chapter.

Chapter 10, *Orchestration*, is about Heat, the OpenStack component that can orchestrate resource creation within an OpenStack cloud. The templates used to launch stacks will be reviewed. The use of Heat on the command line and through the web interface is covered in this chapter.

Chapter 11, *Scaling Horizontally*, discusses building OpenStack to be run on off-the-shelf hardware. Ways to expand an OpenStack cloud's capacity are also covered in this chapter.

Chapter 12, *Monitoring*, introduces one option to use to monitor your cloud's health, considering the fact that there are a large number of moving parts to a running OpenStack cloud.

Chapter 13, *Troubleshooting*, says that things break and OpenStack is no exception. Each component that has been covered is revisited to offer some tips on how to troubleshoot your cloud when something is not working the way it is expected to.

What you need for this book

You will need to have basic skills on a Linux command line, a computer (physical or virtualized) to run an installation on, and an Internet connection to access OpenStack installation resources. Exercises in this book will work off a Fedora installation and will use three computers. While three are used as an example, an all-in-one installation of OpenStack on a single machine is also a very practical deployment to use to learn OpenStack.

Who this book is for

This book is for those that are interested in learning more about OpenStack as a cloud platform. This book starts at the beginner's level and is intended as a getting-started guide. Understand that it starts at the beginning of OpenStack and assumes a basic knowledge of system administration and virtualization.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `keystonerc_admin` file thus becomes much more than just a storage place for the user's credentials."

A block of code is set as follows:

```
export OS_USERNAME=danradez
export OS_TENANT_NAME=danradez
export OS_PASSWORD=supersecret
export OS_AUTH_URL=http://192.168.123.101:5000/v2.0/
export PS1='[\u@\h \W(keystone_danradez)]\$ '
```

Any command-line input or output is written as follows:

```
mylaptop$ ssh root@192.168.122.101
control# yum update -y
control# yum install -y http://rdo.fedorapeople.org/rdo-release.rpm
control# packstack --gen-answer-file myanswers.txt
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "When you click on the **Create Project** button, the **Create User** form will show up again with all your original data filled in for you and the new tenant's name populated for you."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Architecture and Component Overview

OpenStack has a very modular design, and because of this design, there are lots of moving parts. It's overwhelming to start walking through installing and using OpenStack without understanding the internal architecture of the components that make up OpenStack. In this chapter, we'll look at these components. Each component in OpenStack manages a different resource that can be virtualized for the end user. Separating each of the resources that can be virtualized into separate components makes the OpenStack architecture very modular. If a particular service or resource provided by a component is not required, then the component is optional to an OpenStack deployment. Let's start by outlining some simple categories to group these services into.

OpenStack architecture

Logically, the components of OpenStack can be divided into three groups:

- Control
- Network
- Compute

The control tier runs the **Application Programming Interfaces** (**API**) services, web interface, database, and message bus. The network tier runs network service agents for networking, and the compute node is the virtualization hypervisor. It has services and agents to handle virtual machines. All of the components use a database and/or a message bus. The database can be MySQL, MariaDB, or PostgreSQL. The most popular message buses are RabbitMQ, Qpid, and ActiveMQ. For smaller deployments, the database and messaging services usually run on the control node, but they could have their own nodes if required.

www.allitebooks.com

In a simple multi-node deployment, each of these groups is installed onto a separate server. OpenStack could be installed on one node or two nodes, but a good baseline for being able to scale out later is to put each of these groups on their own node. An OpenStack cluster can also scale far beyond three nodes, and we'll look at scaling beyond this basic deployment in *Chapter 11*, *Scaling Horizontally*.

Now that a base logical architecture of OpenStack is defined, let's look at what components make up this basic architecture. To do that, we'll first touch on the web interface and then work towards collecting the resources necessary to launch an instance. Finally, we will look at what components are available to add resources to a launched instance.

Dashboard

The OpenStack dashboard is the web interface component provided with OpenStack. You'll sometimes hear the terms dashboard and **Horizon** used interchangeably. Technically, they are not the same thing. This book will refer to the web interface as the dashboard. The team that develops the web interface maintains both the dashboard interface and the Horizon framework that the dashboard uses.

More important than getting these terms right is understanding the commitment that the team that maintains this code base has made to the OpenStack project. They have pledged to include support for all the officially accepted components that are included in OpenStack. Visit the OpenStack website (`http://www.openstack.org/`) to get an official list of OpenStack components.

The dashboard cannot do anything that the API cannot do. All the actions that are taken through the dashboard result in calls to the API to complete the task requested by the end user. Throughout this book, we will examine how to use the web interface and the API clients to execute tasks in an OpenStack cluster. Next, we will discuss both the dashboard and the underlying components that the dashboard makes calls to when creating OpenStack resources.

Keystone

Keystone is the identity management component. The first thing that needs to happen while connecting to an OpenStack deployment is authentication. In its most basic installation, Keystone will manage tenants, users, and roles and be a catalog of services and endpoints for all the components in the running cluster.

Everything in OpenStack must exist in a tenant. A tenant is simply a grouping of objects. Users, instances, and networks are examples of objects. They cannot exist outside of a tenant. Another name for a tenant is project. On the command line, the term tenant is used. In the web interface, the term project is used.

Users must be granted a role in a tenant. It's important to understand this relationship between the user and a tenant via a role. In *Chapter 3*, *Identity Management*, we will look at how to create the user and tenant and how to associate the user with a role in a tenant. For now, understand that a user cannot log in to the cluster unless they are members of a tenant. Even the administrator has a tenant. Even the users the OpenStack components use to communicate with each other have to be members of a tenant to be able to authenticate.

Keystone also keeps a catalog of services and endpoints of each of the OpenStack components in the cluster. This is advantageous because all of the components have different API endpoints. By registering them all with Keystone, an end user only needs to know the address of the Keystone server to interact with the cluster. When a call is made to connect to a component other than Keystone, the call will first have to be authenticated, so Keystone will be contacted regardless.

Within the communication to Keystone, the client also asks Keystone for the address of the component the user intended to connect to. This makes managing the endpoints easier. If all the endpoints were distributed to the end users, then it would be a complex process to distribute a change in one of the endpoints to all of the end users. By keeping the catalog of services and endpoints in Keystone, a change is easily distributed to end users as new requests are made to connect to the components.

By default, Keystone uses username/password authentication to request a token and **Public Key Infrastructure** (**PKI**) tokens for subsequent requests. The token has a user's roles and tenants encoded into it. All the components in the cluster can use the information in the token to verify the user and the user's access. Keystone can also be integrated into other common authentication systems instead of relying on the username and password authentication provided by Keystone. In *Chapter 3*, *Identity Management*, each of these resources will be explored. We'll walk through creating a user and a tenant and look at the service catalog.

Glance

Glance is the image management component. Once we're authenticated, there are a few resources that need to be available for an instance to launch. The first resource we'll look at is the disk image to launch from. Before a server is useful, it needs to have an operating system installed on it. This is a boilerplate task that cloud computing has streamlined by creating a registry of pre-installed disk images to boot from. Glance serves as this registry within an OpenStack deployment. In preparation for an instance to launch, a copy of a selected Glance image is first cached to the compute node where the instance is being launched. Then, a copy is made to the ephemeral disk location of the new instance. Subsequent instances launched on the same compute node using the same disk image will use the cached copy of the Glance image.

The images stored in Glance are sometimes called sealed-disk images. These images are disk images that have had the operating system installed but have had things such as **Secure Shell** (**SSH**) host key, and network device MAC addresses removed. This makes the disk images generic, so they can be reused and launched repeatedly without the running copies conflicting with each other. To do this, the host-specific information is provided or generated at boot. The provided information is passed in through a post-boot configuration facility called **cloud-init**.

The images can also be customized for special purposes beyond a base operating system install. If there was a specific purpose for which an instance would be launched many times, then some of the repetitive configuration tasks could be performed ahead of time and built into the disk image. For example, if a disk image was intended to be used to build a cluster of web servers, it would make sense to install a web server package on the disk image before it was used to launch an instance. It would save time and bandwidth to do it once before it is registered with Glance instead of doing this package installation and configuration over and over each time a web server instance is booted.

There are quite a few ways to build these disk images. The simplest way is to do a virtual machine install manually, make sure that the host-specific information is removed, and include cloud-init in the built image. Cloud-init is packaged in most major distributions; you should be able to simply add it to a package list. There are also tools to make this happen in a more autonomous fashion. Some of the more popular tools are **virt-install**, **Oz**, and **appliance-creator**. The most important thing about building a cloud image for OpenStack is to make sure that cloud-init is installed. Cloud-init is a script that should run post boot to connect back to the metadata service. An example build of a disk image will be done in *Chapter 4*, *Image Management*, when Glance is covered in greater detail.

Neutron

Neutron is the network management component. With Keystone, we're authenticated, and from Glance, a disk image will be provided. The next resource required for launch is a virtual network. Neutron is an API frontend (and a set of agents) that manages the **Software Defined Networking** (**SDN**) infrastructure for you. When an OpenStack deployment is using Neutron, it means that each of your tenants can create virtual isolated networks. Each of these isolated networks can be connected to virtual routers to create routes between the virtual networks. A virtual router can have an external gateway connected to it, and external access can be given to each instance by associating a floating IP on an external network with an instance. Neutron then puts all configuration in place to route the traffic sent to the floating IP address through these virtual network resources into a launched instance. This is also called **Networking as a Service** (**NaaS**). NaaS is the capability to provide networks and network resources on demand via software.

By default, the OpenStack distribution we will install uses **Open vSwitch** to orchestrate the underlying virtualized networking infrastructure. Open vSwitch is a virtual managed switch. As long as the nodes in your cluster have simple connectivity to each other, Open vSwitch can be the infrastructure configured to isolate the virtual networks for the tenants in OpenStack. There are also many vendor plugins that would allow you to replace Open vSwitch with a physical managed switch to handle the virtual networks. Neutron even has the capability to use multiple plugins to manage multiple network appliances. As an example, Open vSwitch and a vendor's appliance could be used in parallel to manage virtual networks in an OpenStack deployment. This is a great example of how OpenStack is built to provide flexibility and choice to its users.

Networking is the most complex component of OpenStack to configure and maintain. This is because Neutron is built around core networking concepts. To successfully deploy Neutron, you need to understand these core concepts and how they interact with one another. In *Chapter 5*, *Network Management*, we'll spend time covering these concepts while building the Neutron infrastructure for an OpenStack deployment.

Nova

Nova is the instance management component. An authenticated user who has access to a Glance image and has created a network for an instance to live on is almost ready to tie all of this together and launch an instance. The last resources that are required are a key pair and a security group. A key pair is simply an SSH key pair. OpenStack will allow you to import your own key pair or generate one to use. When the instance is launched, the public key is placed in the `authorized_keys` file so that a password-less SSH connection can be made to the running instance.

Before that SSH connection can be made, the security groups have to be opened to allow the connection to be made. A security group is a firewall at the cloud infrastructure layer. The OpenStack distribution we'll use will have a default security group with rules to allow instances to communicate with each other within the same security group, but rules will have to be added for **Internet Control Message Protocol** (**ICMP**), SSH, and other connections to be made from outside the security group.

Once there's an image, network, key pair, and security group available, an instance can be launched. The resource's identifiers are provided to Nova, and Nova looks at what resources are being used on which hypervisors, and schedules the instance to spawn on a compute node. The compute node gets the Glance image, creates the virtual network devices, and boots the instance. During the boot, cloud-init should run and connect to the metadata service. The metadata service provides the SSH public key needed for SSH login to the instance and, if provided, any post-boot configuration that needs to happen. This could be anything from a simple shell script to an invocation of a configuration management engine.

In *Chapter 6*, *Instance Management*, we'll walk through each of the pieces of Nova and see how to configure them so that instances can be launched and communicated with.

Cinder

Cinder is the block storage management component. Volumes can be created and attached to instances. Then, they are used on the instances as any other block device would be used. On the instance, the block device can be partitioned and a file system can be created and mounted. Cinder also handles snapshots. Snapshots can be taken of the block volumes or of instances. Instances can also use these snapshots as a boot source.

There is an extensive collection of storage backends that can be configured as the backing store for Cinder volumes and snapshots. By default, **Logical Volume Manager** (**LVM**) is configured. GlusterFS and Ceph are two popular software-based storage solutions. There are also many plugins for hardware appliances.

In *Chapter 7*, *Block Storage*, we'll take a look at creating and attaching volumes to instances, taking snapshots, and configuring additional storage backends to Cinder.

Swift

Swift is the object storage management component. Object storage is a simple content-only storage system. Files are stored without the metadata that a block filesystem has. These are simply containers and files. The files are simply content. Swift has two layers as part of its deployment: the proxy and the storage engine. The proxy is the API layer. It's the service that the end user communicates with. The proxy is configured to talk to the storage engine on the user's behalf. By default, the storage engine is the Swift storage engine. It's able to do software-based storage distribution and replication. GlusterFS and Ceph are also popular storage backends for Swift. They have similar distribution and replication capabilities to those of Swift storage.

In *Chapter 8*, *Object Storage*, we'll work with object content and the configuration involved in setting up an alternative storage backend for Swift.

Ceilometer

Ceilometer is the telemetry component. It collects resource measurements and is able to monitor the cluster. Ceilometer was originally designed as a metering system for billing users. As it was being built, there was a realization that it would be useful for more than just billing and turned into a general-purpose telemetry system.

Ceilometer meters measure the resources being used in an OpenStack deployment. When Ceilometer reads a meter, it's called a sample. These samples get recorded on a regular basis. A collection of samples is called a statistic. Telemetry statistics will give insights into how the resources of an OpenStack deployment are being used.

The samples can also be used for alarms. Alarms are nothing but monitors that watch for a certain criterion to be met. These alarms were originally designed for Heat autoscaling. We'll look more at getting statistics and setting alarms in *Chapter 9*, *Telemetry*. Let's finish listing out OpenStack components by talking about Heat.

Heat

Heat is the orchestration component. Orchestration is the process of launching multiple instances that are intended to work together. In orchestration, there is a file, known as a template, used to define what will be launched. In this template, there can also be ordering or dependencies set up between the instances. Data that needs to be passed between the instances for configuration can also be defined in these templates. Heat is also compatible with AWS CloudFormation templates and implements additional features in addition to the AWS CloudFormation template language.

To use Heat, one of these templates is written to define a set of instances that needs to be launched. When a template launches a collection of instances, it's called a stack. When a stack is spawned, the ordering and dependencies, shared conflagration data, and post-boot configuration are coordinated via Heat.

Heat is not configuration management. It is orchestration. It is intended to coordinate launching the instances, passing configuration data, and executing simple post-boot configuration. A very common post-boot configuration task is invoking an actual configuration management engine to execute more complex post-boot configuration. In *Chapter 10*, *Orchestration*, we'll explore creating a Heat template and launching a stack using Heat.

Summary

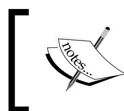
The list of components that have been covered is not the full list. This is just a small subset to get you started with using and understanding OpenStack. Now that we have introduced the OpenStack components, we will illustrate how they work together as a running OpenStack installation. To illustrate an OpenStack installation, we first need to install one. In the next chapter, we will use the RDO OpenStack distribution with its included installer to get OpenStack installed so that we can begin to investigate these components in more detail.

2
RDO Installation

We looked at the components that make up an OpenStack installation in the previous chapter; let's now take a look at what's involved in installing and configuring these components. In this chapter, we'll walk through the installation and configuration of a community-supported distribution of OpenStack called RDO using an installation tool called Packstack.

Manual installation and configuration of OpenStack involves installing, configuring, and registering each of the components we covered in the previous chapter and also multiple databases and a messaging system. It's a very involved, repetitive, error-prone, and sometimes-confusing process. Fortunately, there are a few distributions that include tools to automate this installation and configuration process.

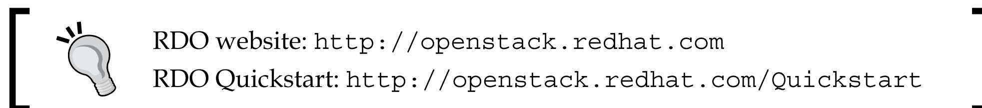
One such distribution is the RDO distribution. RDO, as a name, doesn't officially mean anything. It's just the name of Red Hat's community-supported distribution of OpenStack. Red Hat takes the upstream OpenStack code, packages its RPMs with several installation options, and provides documentation, forums, IRC, and other resources for the RDO community to use and support each other in running OpenStack on RPM-based systems. There are no modifications to the upstream OpenStack code in the RDO distribution. The RDO project packages the code that is in each of the upstream releases of OpenStack. This means that we'll use an open source, community-supported distribution of vanilla OpenStack for our example installation. RDO can be run on any RPM-based system, Fedora will be used for the operating system, and Packstack will be used for the install tool for this demonstrative installation. CentOS or other RPM Linux distributions should also work fine.



Other installation options available with RDO include **staypuft**, a plugin for the foreman and **triple-o**, which is short for OpenStack-on-OpenStack.

Installing RDO using Packstack

Packstack is an install tool for OpenStack intended for demonstration and proof of concept deployments. The other two installation tools mentioned are intended for longer term installations that need to be managed and maintained and are outside the scope of what we will accomplish in this book. Packstack uses SSH to connect to each of the nodes and invokes a puppet run (specifically a puppet apply) on each of the nodes to install and configure OpenStack.



RDO website: `http://openstack.redhat.com`
RDO Quickstart: `http://openstack.redhat.com/Quickstart`

RDO Quickstart gives instructions to install RDO using Packstack in three simple steps:

1. Update the system and install the RDO release rpm as follows:

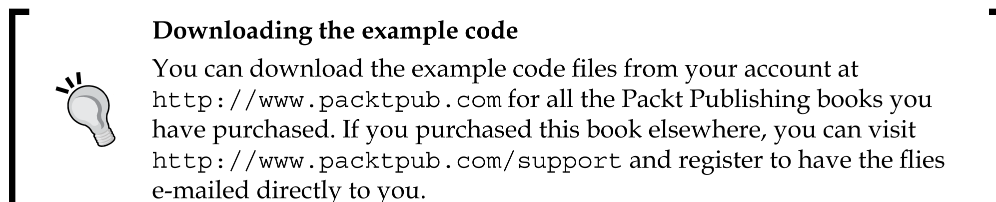
   ```
   sudo yum update -y
   sudo yum install -y http://rdo.fedorapeople.org/rdo-release.rpm
   ```

2. Install Packstack as shown in the following command:

   ```
   sudo yum install -y openstack-packstack
   ```

3. Run Packstack as shown in the following command:

   ```
   sudo packstack --allinone
   ```



Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the flies e-mailed directly to you.

The all in one installation method works well if you only have one server. It is a recommended way to get OpenStack running quickly; that's exactly what Packstack is for—building a deployment, tearing it down, and doing it again. In reality, however, a cluster will usually need more than one compute node to host all the instances that end users will spawn. Under the hood, the configuration of Neutron is slightly different for an all-in-one single-server installation as compared to a multinode installation. So, instead of boxing our example installation into a single server from the start as Quickstart does, we will work though a multinode installation.

Don't avoid doing an all-in-one installation; it really is as simple as the steps make it out to be, and there is value in getting an OpenStack installation done quickly. Getting an all-in-one installation is something that can be done easily, and it will be more beneficial for us to cover a multinode installation here.

The environment we will work through here will be useful in *Chapter 11*, *Scaling Horizontally*, to demonstrate adding compute nodes to OpenStack when scaling is covered. Packstack can do this multinode installation; it will just take additional configuration to pass to Packstack before the installation starts.

Compute nodes are the hypervisor nodes where the instances run. Neutron is the networking component.

Preparing nodes for installation

Before working on the extra configuration, let's define the architecture for our demonstration cloud. Let's use three nodes, one for each of our logical categories of OpenStack that were defined earlier:

- A control node
- A network node
- A compute node

Each node will have two network interfaces. The `eth0` interface on each node will be in the `192.168.123.0/24` subnet, and the `eth1` interface will be in the `192.168.122.0/24` subnet. We will assign IPs as shown in the following two paragraphs.

The `192.168.122.0/24` subnet represents the public network that the nodes are connected to, and the `192.168.123.0/24` subnet represents the private network. These networks represent the physical network that will carry communication in and out of the OpenStack cluster. In reality, more than two networks should exist. A recommended architecture still has the internal or data network for communication within OpenStack. The networking traffic for the instances and the storage traffic would also have their own segregated networks requiring four interfaces on each node for a more complete deployment. Our example will only use two networks to simplify the installation.

www.allitebooks.com

Make sure that your nodes have several CPUs and a minimum of 4 to 6 GB of RAM, and install CentOS or Fedora or another RPM-based Linux distribution of your choice. This will allow you to install OpenStack and launch a few small instances. If you're running low on resources, you could merge the control and network nodes into one node and initially run a two-node cluster.

Installing Packstack and generating an answer file

Now that we have an architecture defined for installation, let's take a look at the extra configuration that will be passed into Packstack using an answer file. The Packstack command has a parameter that can be passed to generate an initial answer file for you. This file is simply a text file full of key-value pairs that are initially generated with all the default values used for the all-in-one installation. The all-in-one installation actually generates the same file and uses it to complete the installation. To get started, log in to your control node. Set up the RDO repository, install Packstack, and generate a new answer file as follows:

```
mylaptop$ ssh root@192.168.122.101
control# yum update -y
control# yum install -y http://rdo.fedorapeople.org/rdo-release.rpm
control# packstack --gen-answer-file myanswers.txt
```

When you edit the generated file, you'll see an extensive list of key-value pairs that configure all the different OpenStack components.

Now that you have a Packstack file generated, let's start walking through customizing it. The first thing to notice is that Packstack has filled in all the `HOST` configuration options with an IP address. If the `192.168.123.101` address was not used, search and replace all of these values to ensure that the `123` network will be used. Here's the command to accomplish this:

```
control# sed -i 's/192.168.122.101/192.168.123.101/g' myanswers.txt
```

Next, we will update some of the sample host values to reflect the architecture just mapped out. Set the Neutron `HOST` values to the Neutron host IP address and the compute `HOST` value to the computer host. Also, update the Horizon `HOST` value to use the public IP of the host. This ensures that things get configured properly to expose the web interface on the public network. Here's how we accomplish this:

```
CONFIG_NEUTRON_SERVER_HOST=192.168.123.102
CONFIG_NEUTRON_L3_HOSTS=192.168.123.102
CONFIG_NEUTRON_DHCP_HOSTS=192.168.123.102
```

```
CONFIG_NEUTRON_METADATA_HOSTS=192.168.123.102
CONFIG_NOVA_COMPUTE_HOSTS=192.168.123.103
CONFIG_HORIZON_HOST=192.168.122.101
```

For networking to work properly in a multinode configuration, there are extra configuration options needed. We'll use **Virtual Extensible LAN** (**VXLAN**) tunneling. Update these configuration options to specify the VXLAN configuration:

```
CONFIG_NEUTRON_OVS_TENANT_NETWORK_TYPE=vxlan
CONFIG_NEUTRON_OVS_TUNNEL_RANGES=1:1000
CONFIG_NEUTRON_OVS_TUNNEL_IF=eth0
```

Finally, two of the components we'll cover are not installed by default in Packstack, so we will enable these as follows:

```
CONFIG_SWIFT_INSTALL=y
CONFIG_HEAT_INSTALL=y
```

Now that the extra hosts are configured, the extra components are added, and the networking configuration is updated, this file needs to be fed into Packstack to execute the installation. Packstack is invoked using the `--answer-file` parameter with the answer file as the argument value; here's how:

```
control# packstack --answer-file myanswers.txt
```

It's important to note here that when Packstack is run with this option, it is an idempotent run. So, if something fails in the Packstack run, you can correct it and rerun Packstack. All the other ways of invoking Packstack, all-in-one included, are not idempotent; only `--answer-file` is. This is very important because when a new answer file is generated, all new passwords get generated too. Consequently, if a previous Packstack run set up something that used one of the generated passwords, then using a newly generated answer file with new passwords will never succeed.

Let's look at how a successful Packstack run will look. When you execute Packstack and pass the answer file, the first section will ensure connectivity to the nodes and then generate manifest entries. There are a large number of lines in the output referring to these manifest entries, so the output here has been truncated. Where you see `{XYZ}` in this output, you can assume that you'll see the line repeated for all the different items that need manifests for installation. Manifest entries are files that are full of puppet classes. The manifest files are configuration definitions that will invoke puppet modules on the nodes when the puppet is run on them.

Next in this output, you will see pairs of lines that read `Applying {IP_ADDRESS}_{XYZ}.pp` and `{IP_ADDRESS}_{XYZ}.pp      [DONE]`. Each of the IP addresses in the answer file will be associated with the different items that need to be installed and configured for the OpenStack installation. When each of these tasks gets started, an Applying message is printed, and when each finishes, a `[DONE]` message is printed. Finally, if everything went successfully, a success message will be provided with any information important to the completed installation. Here's the output summary:

```
control# packstack --answer-file myanswers.txt
Welcome to Installer setup utility
Packstack changed given value y to required value n

Installing:
Clean Up                                             [ DONE ]
Setting up ssh keys                                  [ DONE ]
Discovering hosts' details                           [ DONE ]
Adding {XYZ} manifest entries                  [ DONE ]



Preparing servers                                    [ DONE ]
Installing Dependencies                              [ DONE ]
Copying Puppet modules and manifests                 [ DONE ]
Applying 192.168.123.103_{XYZ}.pp
Applying 192.168.123.101_{XYZ}.pp
Applying 192.168.123.102_{XYZ}.pp
Applying 192.168.122.101_{XYZ}.pp
192.168.123.102_{XYZ}.pp:                        [ DONE ]
192.168.123.103_{XYZ}.pp:                        [ DONE ]
192.168.123.101_{XYZ}.pp:                        [ DONE ]
192.168.122.101_{XYZ}.pp:                        [ DONE ]

Applying Puppet manifests                            [ DONE ]
Finalizing                                           [ DONE ]

 **** Installation completed successfully ******
```

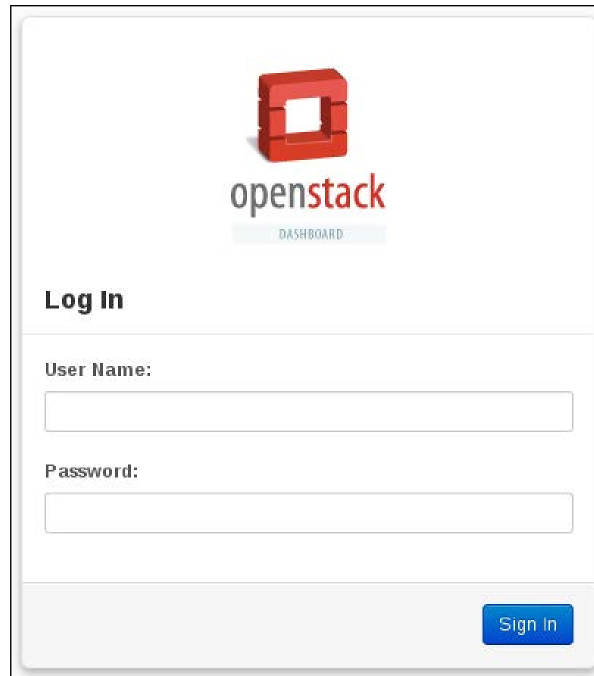
```
Additional information:
 * Time synchronization installation was skipped. Please note that
unsynchronized time on server instances might be problem for some
OpenStack components.
 * Did not create a cinder volume group, one already existed
 * File /root/keystonerc_admin has been created on OpenStack client host
192.168.123.101. To use the command line tools you need to source the
file.
 * To access the OpenStack Dashboard browse to http://192.168.122.101/
dashboard .
Please, find your login credentials stored in the keystonerc_admin in
your home directory.
 * To use Nagios, browse to http://192.168.123.101/nagios username :
nagiosadmin, password : 918aa228abe04e6d
 * Because of the kernel update the host 192.168.123.103 requires reboot.
 * Because of the kernel update the host 192.168.123.101 requires reboot.
 * Because of the kernel update the host 192.168.123.102 requires reboot.
 * Because of the kernel update the host 192.168.122.101 requires reboot.
 * The installation log file is available at: /var/tmp/
packstack/20140528-003206-reQmjV/openstack-setup.log
* The generated manifests are available at: /var/tmp/packstack/20140528-
003206-reQmjV/manifests
```

This installation run required a reboot of the nodes because of a kernel update. If this is indicated, make sure to do the reboot. In some cases, you may have got a new kernel that has added support for network namespaces required by the advanced networking.

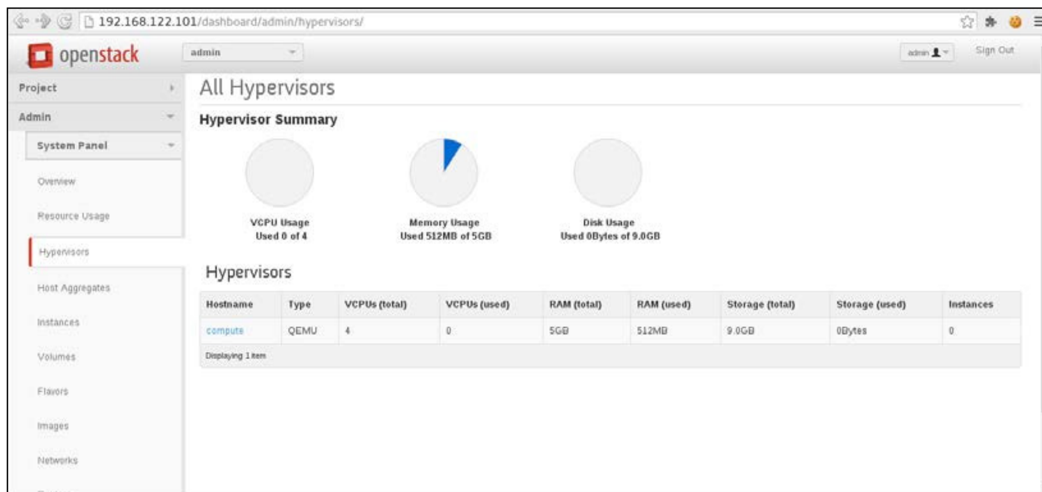
As part of the Packstack run, a file named `keystonerc_admin` is created on the control node with the administrative user's credentials. `Cat` this file to see its contents and get credentials to log in, as follows:

```
control# cat ~/keystonerc_admin
export OS_USERNAME=admin
export OS_TENANT_NAME=admin
export OS_PASSWORD=1ef82c52e0bd46d5
export OS_AUTH_URL=http://192.168.123.101:5000/v2.0/
export PS1='[\u@\h \W(keystone_admin)]\$ '
```

Now that you have an OpenStack installation and the credentials to log in, open your web browser and go to the IP address you used for your `CONFIG_HORIZON_HOST` configuration parameter. The demonstration installation configuration values would expose the web server as shown in the following screenshot of the page at `http://192.168.122.101/`.



Use the admin user and the generated password that came from the `keystonerc_admin` file to log in. If all went properly, you would be presented with the OpenStack dashboard web interface. Here's a screenshot of the hypervisor list showing the single hypervisor in the cluster built in this chapter:



Summary

Using Packstack, we now have OpenStack installed and running. Now that OpenStack is installed and running and you've logged in to the dashboard interface, let's walk through each of the components discussed in *Chapter 1*, *Architecture and Component Overview*, and you will learn how to use each of them. In the next chapter, we will take a look at Keystone to manage users, tenants, and roles used in managing identities within the OpenStack cluster.

3

Identity Management

In the previous chapter, we installed OpenStack using RDO. Now that OpenStack is ready for use, we will begin to investigate what was installed and how to use it, starting with identity management. Keystone is the identity management component in OpenStack. In this chapter, we will look at service registration and the relationship of users with tenants and the role of a user in a tenant.

Services and endpoints

Each of the components in an OpenStack cluster is registered with Keystone. Each of the services has endpoints and each of the services has a user. A service in Keystone is a record of another OpenStack component that will need to be contacted to manage virtual resources. Endpoints are the URLs to contact these services. Let's look at this on the command line. Remember the `keystonerc_admin` file? You'll need the information in that file to authenticate and interact with OpenStack. The information is as follows:

```
control# cat keystonerc_admin
export OS_USERNAME=admin
export OS_TENANT_NAME=admin
export OS_PASSWORD=1ef82c52e0bd46d5
export OS_AUTH_URL=http://192.168.123.101:5000/v2.0/
export PS1='[\u@\h \W(keystone_admin)]\$ '
control# keystone --os-username admin --os-tenant-name admin \
--os-password 1ef82c52e0bd46d5 --os-auth-url \
http://192.168.123.101:5000/v2.0/ service-list
```

Manually entering Keystone arguments is a real challenge and prone to error. The `keystonerc_admin` file thus becomes much more than just a storage place for the user's credentials. If you source the file, then those values are automatically placed in the shell's environment. OpenStack's Python clients know to look at the shell's environment to get these values when they aren't passed as arguments. For example, execute the `service-list` command again with the `keystonerc` file sourced, as follows:

```
control# source keystonerc_admin
control# keystone service-list
```

As you will see, it is much more manageable to issue this command and subsequent commands now. This list shows all the components that are registered with this OpenStack cluster. Now list the endpoints as follows:

```
control# keystone endpoint-list
```

The hashes in the service_id column will match the hashes from the `service-list` command you just executed. Each of the services has a public, private, and admin endpoint URL. These are used by the components and API clients to know how to connect to the different components. An end user or a component within the cluster can always ask Keystone for the endpoint of a component to connect to. This makes it manageable to update the endpoint and be certain that new clients are connecting to the correct endpoint. The only endpoint that needs to be known ahead of time is the Keystone endpoint. Registration of a service and a set of endpoints only allows us to know about a service and how to connect to it. Each of these services also has a user. The services' users are used for inter-component communication. Each of the services authenticate with Keystone to communicate with each other.

Hierarchy of users, tenants, and roles

A user is granted a role in a tenant. A tenant is simply a grouping of resources. A user can have a role in multiple tenants. Without a role in a tenant, a user cannot create virtual resources in an OpenStack cluster. A user is useless without a role in a tenant. All virtual resources created in OpenStack must exist in a tenant. Virtual resources are the virtual infrastructure that OpenStack manages. Among others, instances, networks, storage, and disk images must exist in a tenant. Recall the services that were just introduced; they all have a user that has a role in a tenant. If you list the users and tenants in your OpenStack installation, you will see a user for each of the components installed in the installed cluster. Then, list one of the user roles in the services tenant. Let's use Nova as an example; here's the output summary after you hit the following commands:

```
control# keystone user-list
control# keystone tenant-list
control# keystone user-role-list --user nova --tenant services
+---------------+-------+-------------------+--------------------+
|      id       | name  |      user_id      |     tenant_id      |
+---------------+-------+-------------------+--------------------+
|   {role_id}   | admin |     {user_id}     |    {tenant_id}     |
+---------------+-------+-------------------+--------------------+
```

Now recall that when we authenticated the admin user earlier, the admin user was authenticating to itself. A common convention for creating tenant names is to use the same name as that of the user that will be using it unless it is used by a group. If there are multiple users that have roles in a tenant, a more descriptive name is used for the tenant's name. Take the admin and services tenants as examples of using the user's name or a more descriptive name. There are multiple users in the services tenant. It's a tenant for all the users of services. There is only one user that uses the admin tenant—the admin user. Each user that will use an OpenStack deployment will need a user to log in and a tenant to operate out of. Let's walk through creating a user and tenant and giving that user a role in the tenant.

Creating a user

We will start by creating a user. There are a handful of subcommands for user management. Run the Keystone client without any arguments and look through the list of subcommands that start with `user-`. To create a user, use the `user-create` subcommand as follows:

```
control# keystone user-create --name danradez
```

A user now exists that has my first and last name as its username. There are other properties that can be set when a user is created. Use the `help` in the command-line client to get more information about these properties, as follows:

```
control# keystone help user-create
```

All of OpenStack's command-line clients use this syntax convention to display help. In any of the component's clients, you can use the subcommand help and pass it the subcommand's name that you want help on, and a list of arguments and their descriptions will be displayed. An e-mail or a password could have been set when the user was created. Except for passwords, all these properties can also be updated using the user-update subcommand. Let's update the new user's e-mail as an example:

```
control# keystone user-update --email danradez@example.com danradez
```


www.allitebooks.com

Here, the new user has been updated to have an e-mail address. To set a password for this, the user uses the `user-password-update` subcommand, as follows:

```
control# keystone user-password-update danradez --pass supersecret
```

In this example, the `--pass` argument was given; the client can be left to prompt you for the password.

Creating a tenant

Now that we have a user, we need a tenant for the user to store some virtual resources. Similar to the subcommands for user management, all the subcommands for tenant management begin with `tenant-`. The following `tenant-create` subcommand will create a new tenant for the new user:

```
control# keystone tenant-create --name danradez
```

In this example, the tenant is created using the convention mentioned earlier, with the username as the name of the tenant. A tenant also has a description property; use `keystone help tenant-create` or `keystone help tenant-update` to get the syntax to set the tenant's description.

Granting a role

Now that we have a user and a tenant, they need to be associated with each other. To do this, the user, the tenant, and a role need to be passed to the `user-role-add` command. Before this is executed, using the `role-list` command, get `role_id` of the member, as shown in the following code:

```
control# keystone role-list

control# keystone user-role-add --user danradez --tenant danradez \ -
-role {member_role_id}
```

This long command associates the user, the tenant, and the role with each other. This association can now be displayed using the `user-role-list` subcommand used earlier, as follows:

```
control# keystone user-role-list --user danradez --tenant danradez
```

That command will show you that the new user was granted the member role in the new tenant. Now that we have a new user that has a role in a tenant, we can use this user's password to make command-line API calls in the same way it was done with the admin user.

Logging in with the new user

The easiest way to start using the new user is to make a copy of an existing `keystonerc` file, update the values in it, and source the file. We conveniently already have an existing `keystonerc` file that was used for the admin user. Make a copy of it and edit it so that its contents have values respective to your new user, as follows:

```
control# cp keystonerc_admin keystonerc_danradez
```

Here are the contents of the new file:

```
export OS_USERNAME=danradez
export OS_TENANT_NAME=danradez
export OS_PASSWORD=supersecret
export OS_AUTH_URL=http://192.168.123.101:5000/v2.0/
export PS1='[\u@\h \W(keystone_danradez)]\$ '
```

`AUTH_URL` here is pointing to the internal URL; the public URL is also a fine choice for this value.

Remember to use Keystone's `service-list` and `endpoint-list` commands if you want to use a different Keystone endpoint. Next, we must source the new `keystonerc` file. A simple authentication verification is to issue a `token-get` command. If it returns an excessive amount of content, then you have received the contents of a **Public Key Infrastructure** (**PKI**) token for the user. If you get an error, it means that authentication failed.

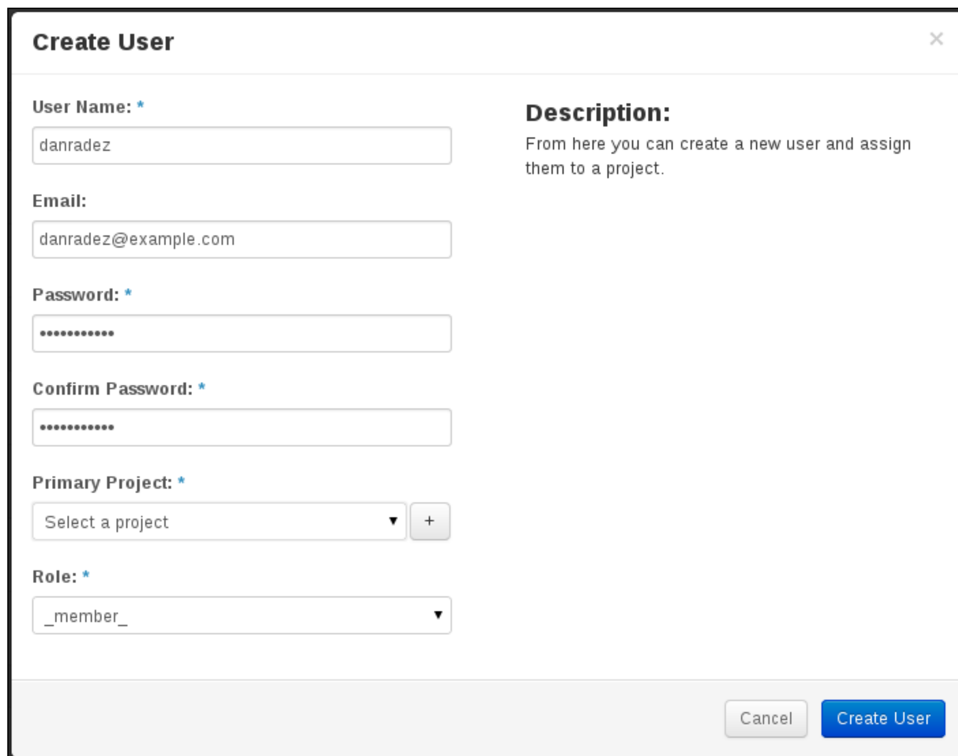
The following commands encapsulate the discussion in the preceding paragraph:

```
control# source keystonerc_danradez
control# keystone token-get
```

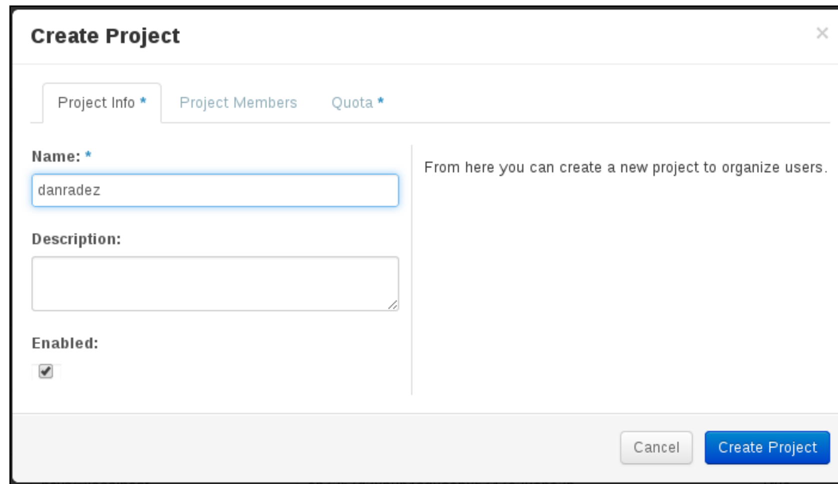
Once you are able to authenticate, you can start to build your virtual infrastructure as a non-administrative user and create more accounts for other non-administrative users.

Interacting with Keystone in the dashboard

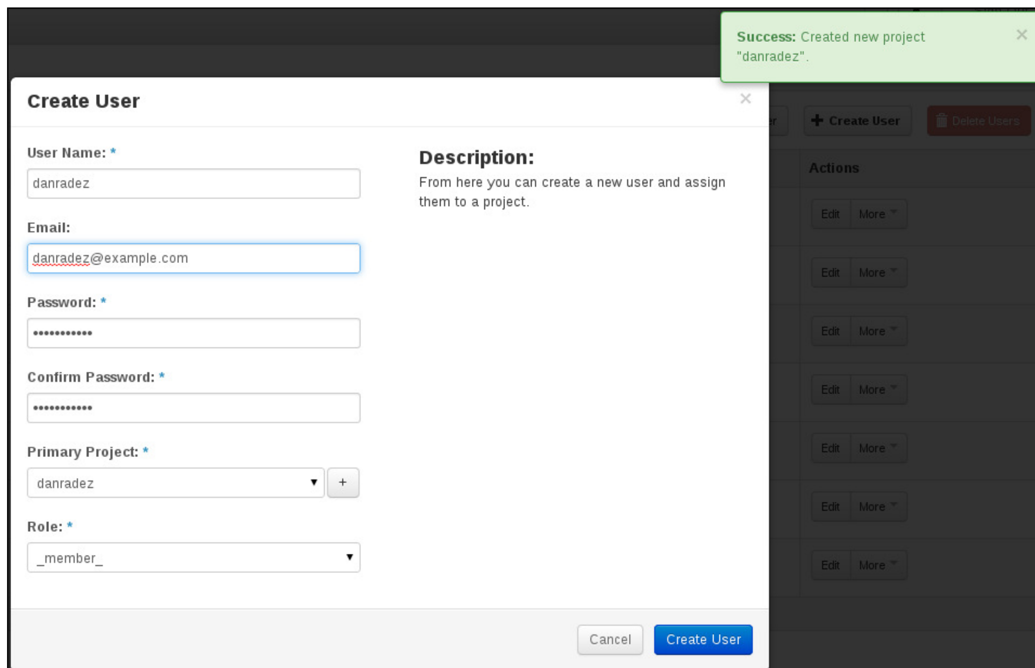
Now that we have worked through managing Keystone resources on the command line, let's take a look at how to do the same through the web interface. Log in as the admin user, select the **Admin** menu, and then select the identity submenu. Here, you'll see menu options to manage projects and users. A project and a tenant are the same. You'll see tenant used on the command line and project used in the web interface. Go ahead and select the **Users** menu. You'll see the same list of users from the Keystone `user-list` command on the command line. In the web interface, tenants can be created inline of a user creation. Select the **Create User** button in the top-right corner of the user management panel. Fill in the form as appropriate:



Before you can create the user, you'll have to select a project. If there isn't one that you want to add the new user to in the existing list, you can create one. Click the button next to the project selection dropdown. A **Create Project** form will show up as follows; fill this one in as appropriate:



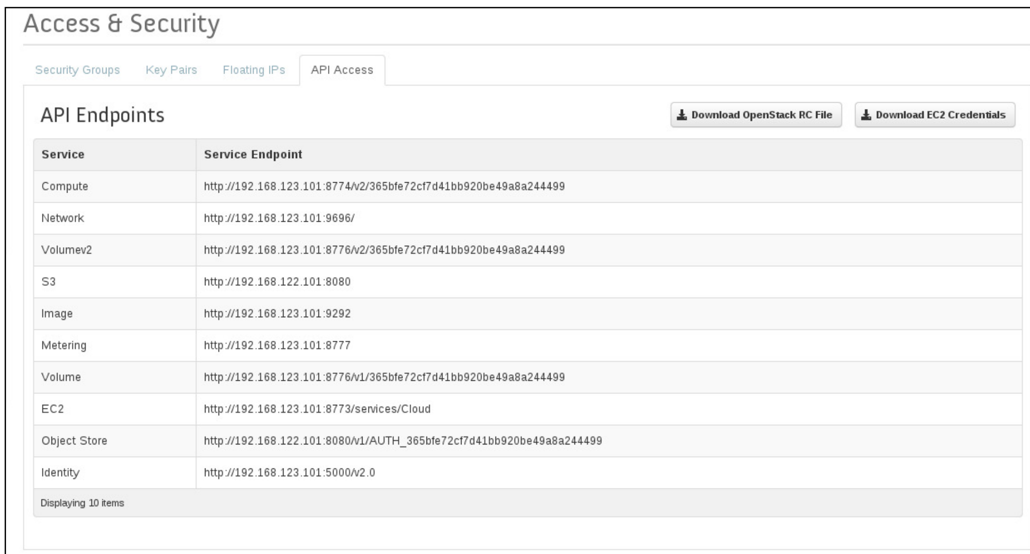
When you click on the **Create Project** button, the **Create User** form will show up again with all your original data filled in for you and the new tenant's name populated for you:



Now the user can be created. Click on **Create User**, and you're ready to start using the user's login and the new tenant. Remember that you can select an existing tenant instead of creating a new one. This just gives multiple users access to the resources in a tenant.

Endpoints in the dashboard

We've looked at user management in the dashboard; now let's look at service and endpoints in the web interface. The dashboard doesn't provide a way to add or update services and endpoints. This is something reserved for the command line because it's usually done once and doesn't need more management. The dashboard does provide a slightly better display of the services and endpoints than the command line does. Click on the **Project** menu and the **Access and Security** submenu. There will be a set of tabs to select from across the top of the screen, as shown in the following screenshot. Select **API Access**. Does this look familiar?



Summary

In this chapter, we looked at managing services, endpoints, users, tenants, and roles through both the command line and the dashboard. Now that we have created users and given them tenants to manage virtual resources, let's start collecting the resources needed to launch an instance. The first resource that is needed before an instance can be launched is a disk image for that instance to launch from. In the next chapter, we will look at Glance, the image management component, and how to import and build images to launch instances.

4
Image Management

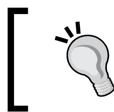
In the preceding chapter, we looked at how identities are managed in OpenStack and how to authenticate to an OpenStack cluster. In this chapter, we will start to gather the resources necessary to launch an instance. The first resource we will work with is the image that an instance will use as its disk image when it is launched. Glance is the image management component in OpenStack. In this chapter, we'll look at how to register images with the image registry and how to build a custom cloud image.

Glance as a registry of images

At launch, a generic virtual machine requires a prebuilt disk image to boot from—some kind of storage that holds the operating system using which the virtual machine will run. Traditionally, a new virtual machine is created with a form of installation media accessible to it. This could take the form of an ISO, optical device, or maybe some form of network-accessible media. Whatever media is provided, an operating system installation is the next step in this scenario. One of the purposes of cloud computing is to be able to quickly create disposable virtual instances. The tasks of running an operating system installation and spawning a virtual machine fast are polar opposites of each other. Cloud computing has removed the need for a per-instance operating system installation by creating what has come to be known as cloud images. Cloud images are simply pre-installed bootable disk images that have been sealed. A sealed disk image is a sparse file containing file system and an underlying operating system that has had its identifiable host-specific metadata removed. Host-specific items include things such as SSH host keys, MAC addresses, static IP addresses, persistent udev rules, and any other identifiers that would conflict if used by two of the same servers. Do you see where this is going? These images are imported into the Glance registry and then copied out to the compute nodes for the instances to launch with. We are going to first look at downloading a prebaked image and registering it with Glance. Then, we'll look at what's needed to build your own custom image.

Downloading and registering an image

If you search the Internet for cloud image, you'll most likely get a link to a place to download a disk image from and import into Glance; most of the major distributions out there have one already built and ready to go for you. In general, they are distributed as **qcow2** or raw images, and for the vast majority of cases, either of them will work fine. You'll have to research them on your own to decide whether one or the other fits your use case better. There's also a test distribution, which is extra-super small, called **CirrOS**. If you visit `http://download.cirros-cloud.net` download the `.img` file from the latest version is available.

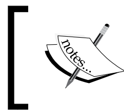


Don't use CirrOS for anything other than testing. It is built with libraries that make it insecure for anything other than demonstration and testing.

To demonstrate using Glance, we will use the Fedora qcow cloud image downloaded from `https://getfedora.org/`; let's start with the command line. To interact with Glance, you'll need to be sure that you've sourced a `keystonerc` file; refer to *Chapter 3*, *Identity Management*, if you need a refresher on this. You will just get an authentication error message if a `keystonerc` file is not currently sourced. Go ahead and list the images registered in Glance, as shown in the following command:

```
control# glance image-list
```

This should return nothing since there aren't any images in Glance yet.



It is important to note here that this command would only list the images in the tenant to which the user is authenticating.

If you have sourced your `keystone_admin` file, you would list the Glance images in the admin tenant. If you sourced your non-admin user's `keystonerc` file, you would get the Glance images for that user's tenant. If you're authenticating as the admin user and want to see all tenants' Glance images, you can pass the following argument to see them all:

```
control# glance image-list --all-tenants
```

If this should still return nothing, let's upload an image to Glance so that there's an image in Glance for us to list. To do this, use the `image-create` command. It is important to understand that you are not creating the disk image with this command. You need to have an already built image. This `image-create` command is creating a record of the image you're uploading in the Glance registry:

```
control# glance image-create --name Fedora --is-public true --disk-format qcow2 --container-format bare --file Fedora-x86_64-disk.qcow2
```

You will notice that you can give your image a name other than the filename of the file that is being uploaded. The disk format and the container format are specific to the image file format that is being uploaded. There are other options for these parameters that you can read about in the Glance documentation at `http://docs.openstack.org/cli-reference/content/glanceclient_commands.html`.

The public flag sets whether this image can be used across all tenants or is private to the tenant it is uploaded to. Now use the `image-list` command to list the image you just uploaded. Two images can have the same name; however, two images cannot have the same ID. There is also an argument that will protect the image from deletion and indicate that the image is protected (`--is-protected`). Administrators can't delete an image that is protected without first turning the protected property to `false`. Let's use the `image-update` command to set the image as protected. The following command captures the discussion in this paragraph:

```
control# glance image-update --is-protected true Fedora
```

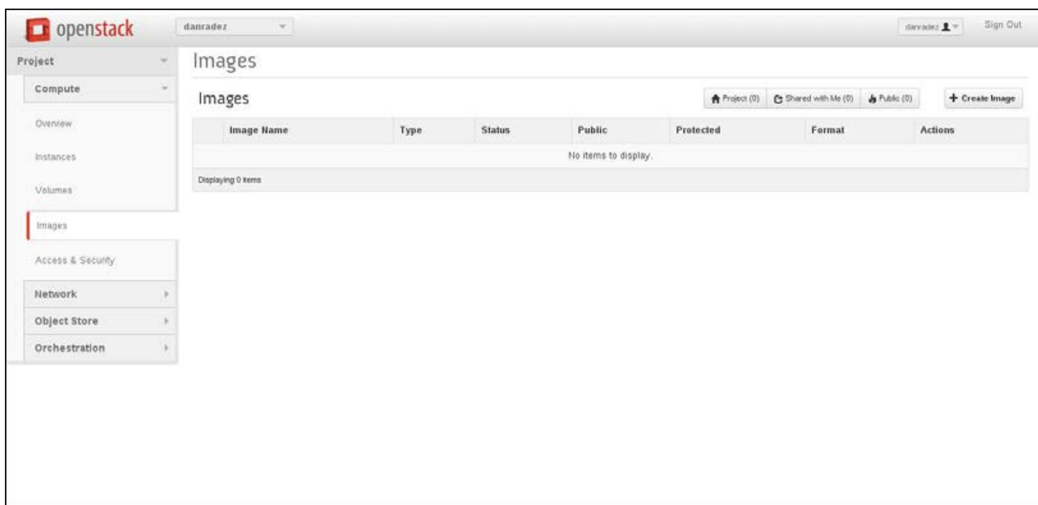
In that example, the image's name was used to set the image as protected. It was mentioned that two images can have the same name; if they do, then the image's ID will have to be used instead of the image name. The properties for the images can be passed to `image-create` or `image-update`. Now that we've worked through using the command line to register a disk image with Glance, let's take a look at using the web interface.

www.allitebooks.com

Using the web interface

Next, let's use the web interface to add an image to the Glance image registry. Images can be managed by administrators and non-privileged users. On the command line, an image was added as the administrator. In the web interface, we will use the non-privileged user you have created. The following are the steps to add an image to the Glance image registry:

1. Log in to your web interface using the user you created in *Chapter 3, Identity Management*. Then, select **Images** from the menu. The following screenshot shows the **Images** page:



2. Once you are logged in, click on the **Create An Image** button and fill out the form that appears (as shown in the following screenshot). All the options that were available on the command line are available in the web form.

3. Once the file has been uploaded and registered, it will show up in the list of images, as shown in the following screenshot:

4. If you log back in as the admin user, you'll see all the imported images listed in the **Images** list under the admin menu. As the administrator, you can also pass the `--all-tenants` argument on the command line to see all the images that have been uploaded to the Glance registry.

Building an image

Now that we've looked at getting a disk image into Glance, let's investigate how a cloud image is built. A cloud image is just a sealed disk image with cloud-init included. A sealed disk image is a file that has an operating system installed in it and has had all the host-specific items removed from it. Cloud-init is a post-boot process that checks the metadata service of OpenStack and asks for post-boot commands that should be run on the launched instance. We'll see cloud-init's use cases in *Chapter 6, Instance Management*, and *Chapter 10, Orchestration*; for now, we'll just make sure it's included in the cloud image we build. To build the image, we'll use `virt-install`. There are quite a few other options. If you're familiar with a different disk image-building tool, use that if you like. This is just one example of how to build one of these images. Go ahead and make sure `virt-install` is installed. The following command accomplishes this:

```
build-host# yum install -y virt-install httpd
```

`Httpd` was installed here too because we need a web server to serve the kickstart. Apache is not needed if you have an alternate web server to serve your kickstart. An automated Fedora install is accomplished via the kickstart. A great place to get a baseline kickstart is from the collection of kickstarts at `https://git.fedorahosted.org/cgit/cloud-kickstarts.git/tree/generic/` that Fedora uses to build cloud images.

These could even be adapted to build a different rpm-based distribution cloud image. Pull down one of those kickstart files and place it in `/var/www/html/`. Also, make sure that Apache is running. Issue the following command to accomplish this:

```
build-host# service httpd start
```

Now that we have something to build with and a kickstart to define what should be built, let's kick off a cloud image build, as follows:

```
build-host# qemu-img create -f qcow2 my_cloudimage.img 10G
build-host# sudo virt-install -n my_cloud_image -r 2048 --vcpus=2 \
  --network=default --graphics=spice --noautoconsole \
  --noreboot -v --disk=path=my_cloudimage.img,format=qcow2 \
  -l http://dl.fedoraproject.org/pub/linux/releases/20/Fedora/x86_64/os/
\
  -x "ks=http://192.168.122.1/my_kickstart_file.ks"
```

The first command creates an empty `qcow2` formatted disk image. The second line spawns a virtual machine in `libvirt` named `my_cloud_image` with 2 GB of RAM and 2 vCPUs using the default `libvirt` network. The virtual machine boots using the kernel and the RAM disk in the install tree from the `dl.fedoraproject.org` URL. The `ks=` option is a kernel parameter. In this example, the kernel pulled from `dl.fedoraproject.org` knows how to pull down the kickstart being served from the local Apache instance on the libvirt network's gateway IP address. Once the installation is complete, the virtual machine can be torn down and the disk image that you created is now an installed cloud image. A final optional step is to sparsify the disk image. There is plenty of documentation on the Internet that can explain what it means to sparsify a disk image better than I can. Use your Internet-searching expertise to read more about what this command does and its benefits. To reiterate, this is optional and will not prevent the final image from being useful. Issue the following command to sparsify the disk image:

```
build-host# virt-sparsify --compress my_cloudimage.img
sparsified.qcow2
```

If you sparsified, the resulting sparsified disk image is what is imported into Glance. If you didn't sparsify, then just import the resulting disk image from `virt-installer`. Note that the sparsify command used the `.img` extension and the `.qcow2` extension. You can use these interchangeably. All the commands you run on these disk images don't really care what the file extension is as they inspect the contents of the disk image to complete their operations.

```
control# glance image-create --name Fedora --is-public true --disk-format
qcow2 --
container-format bare --file sparsified.qcow2
```

Now, let's be frank here. All that really happened was that an operating system was installed into a standard `qcow2` disk image with cloud-init included in the package list and the host's networking was set to DHCP. That means that if you want to do this manually instead of using `virt-install`, you could absolutely launch a virtual machine and do a manual install. Then, make sure that cloud-init is installed and just before you shut down the machine, run commands to set the networking to DHCP and seal the image; somewhat like this:

```
cloud-image# cat > /etc/sysconfig/network-scripts/ifcfg-eth0 << EOF
DEVICE="eth0"
ONBOOT="yes"
BOOTPROTO="dhcp"
TYPE="Ethernet"
```

```
EOF
cloud-image# rm -f /etc/ssh/ssh_host*
cloud-image# rm /etc/udev/rules.d/70-persistent-net.rules
cloud-image# halt –p
```

The `udev` rule may not actually exist, but it doesn't hurt to make sure it's not there. What these commands do is remove any host-specific identification. The MAC address and ID are removed from the networking device configuration and the SSH host keys are removed. They're regenerated on boot if they don't exist, and the udev network persistence configuration is removed, which is also regenerated on boot if it's needed. This list is not exclusive. In the unlikely event that you come across other host-specific things, you should make sure that they are removed to make the image generic. However, on a fresh basic Fedora install, this list should work well to seal the image. Once you've run these commands and shut down the virtual machine, the disk image is ready to be imported into Glance. If you boot the virtual machine back up outside of OpenStack, you will have to partially reseal the image as some of the things you just deleted will be regenerated when you boot up using the disk image again. This does not apply to instances you boot in OpenStack. This only applies to manually spawning a virtual machine using the disk image outside of OpenStack. Once the image has been imported into Glance, OpenStack will handle things properly and not taint the Glance image. The imported image will be stored in the Glance registry and copied out to the compute nodes using which the instances will run. The instances will run using copies of the original disk images stored in Glance.

Summary

In this chapter, we looked at adding images to the Glance image registry for you to learn how to get a prebaked Glance image and how to build your own Glance image. With a disk image stored in Glance, there is now a disk that the instances can copy and use to boot from the time they are spawned. Now that we have created users and stored disk images to launch with, the final resource that needs to be created before we launch an OpenStack instance is a virtual network. In the next chapter, we will use Neutron to create a virtual network fabric for an instance to be connected to.

5
Network Management

In the previous chapter, we prepared to launch an instance by importing disk images into Glance. The next preparation required for launching an instance is to create a virtual network for the instance to use. Neutron is the network management component in OpenStack. In this chapter, we'll look at how to create virtual networks and routers for the OpenStack instances to use. We will also look at some of the underlying plumbing that is used to support the virtual networks.

Networking and Neutron

As I was learning Neutron networking and started to present my experiences to audiences, I coined a phrase that I continue to stand by: *Networking is hard*. Networking is the most complex component in OpenStack and for good reason. This is because networking is a complex part of computing. It takes time and hard work to understand networking. It's often left to the network administrators and neglected by others. Hats off to you network administrators. I spent seven years of my professional career avoiding learning some of the core concepts of networking and leaving it to the folks that did networking. OpenStack is where it caught up with me and bowled me over. To administer an OpenStack cloud that uses Neutron networking, you have to understand some of the core concepts used in networking. As we work through the rest of this book, I'll make sure to explain these concepts as we come across them so that if you're not a networking guru, you will hopefully come out on the other side with an understanding that will equip you to administer your cloud well.

Using Neutron, you enable what's referred to as per-tenant networking. This means that virtual isolated networks can be created in tenants. These networks only have routes to each other if you create them. These networks only have routes to the outside world if you create them, and there is next to nothing assumed about what an instance should be able to do on a network. This is important because it isolates the instances in different tenants from each other. It is a security risk for an instance in tenant A to have access to an instance in tenant B by default. So then, the default is for instances in the same network to have access to each other only until the network is configured differently. OpenStack was designed this way so that you would gain this security out of the box.

Network fabric

Neutron itself is an API that has a modular plugin architecture. The plugins interface with a networking fabric and manage that fabric for you. A networking fabric is just a fancy term for the underlying networking infrastructure and architecture that transports the data within a network. What this means is that Neutron by itself is kind of like a television remote by itself. Until the remote has a television that it can interface with and control, it's just a paperweight that your two-year-old likes to spend time developing his fine motor skills with by pushing the buttons over and over. Similarly, until Neutron has a networking fabric tied to it that it can manage for you, it is basically useless.

There is a broad collection of vendors that have written plugins for Neutron to allow you to manage their compatible networking appliances. If you have a preferred networking vendor, ask them about their support for Neutron. Investigating vendor support and configuration is beyond the scope of this book. Luckily, there is an open source virtual networking project that can meet the needs of our OpenStack networking installation.

Open vSwitch configuration

By default, the RDO installation you ran back in *Chapter 2*, *RDO Installation*, installed Open vSwitch, or OVS for short, and configured the Neutron Open vSwitch plugin for you. Open vSwitch is virtual-networking software that allows you to create virtual switches on your nodes and ties the virtual switches on your nodes together by way of a configured transport. A configured transport is a defined method for the virtual switches to talk to each other. As traffic comes out of an instance, it travels through these connections between each of the virtual switches. There are three common methods for configuring OVS, which are explained here.

VLAN

Virtual Local Area Network (**VLAN**) is the most complex to set up. This is because the hardware switch that carries your traffic must be configured properly to carry the VLAN tagging that is assigned to the traffic. When the network traffic is traveling through one of the virtual networks, it is assigned a VLAN tag, which is simply a numeric identifier. If the switch does not support this identifier to be attached to the traffic, it will not be carried from one virtual switch to another virtual switch, and the network separation is then lost. The benefit of this method is its efficiency. Because the VLAN tag is the only metadata being carried and is already a part of the packets being transferred, there is no additional overhead to using this method and it will provide you with the best performance.

GRE tunnels

Generic Routing Encapsulation (**GRE**) doesn't necessarily require special configuration in the physical switch connecting your nodes. This is because it encapsulates the traffic. Each node must have a direct established connection to every other node. This connection is a tunnel that hides the traffic being sent from the physical switch transporting it. This makes initial setup much easier, but it also comes with its own complexity. When networking traffic is transferred, it's divided into packets—just chunks of the network data being sent. This packet size by default is 1,500 bytes. This default size is known as the **Maximum Transmission Unit** (**MTU**). The data being sent in a packet can be smaller than the MTU. The complexity comes in because to encapsulate or to identify a packet as a GRE packet, an extra header has to be added to each packet. If the data being transferred in the packet together with the GRE header is less than the MTU, then the packet passes through without any trouble. If the packet's data and the header are larger than the MTU, then fragmentation occurs. Fragmentation means that it takes two packets to transfer one packet worth of data, and there is extra communication that has to happen to get the packets fragmented. In short , fragmentation is very bad for network load and throughput. Everything has an MTU. There are two ways to accommodate the GRE header, which are as follows:

- **Lower the instance's MTU**: If every instance boots with an MTU set on its network device that is low enough that when the GRE header is added, it doesn't exceed 1,500 bytes, then the rest of the network fabric can happily function at the default 1,500 MTU.

- **Enable jumbo frames**: This is the better, but more involved, option to configure. For the purpose of this book, we'll define jumbo frames as setting the MTU higher than the default 1,500 bytes. Using jumbo frames is preferable to modifying the instance's MTU because you have control over setting up jumbo frames. You will not have control over every instance that boots in your OpenStack cloud. You can try to use DHCP to send an MTU value for the instance to use, but not all operating systems will honor this value sent via DHCP. Jumbo frames have to be set up anywhere a GRE encapsulated packet will travel—mainly OVS and the physical switch connecting your OpenStack nodes.

VXLAN tunnels

VXLAN functions much like GRE. It's a tunnel that encapsulates the traffic by adding a header. The main difference is that it operates more like **User Datagram Protocol** (**UDP**) instead of like TCP. This eliminates some of the overhead of connection made between the nodes in your OpenStack cluster and is generally regarded as a more efficient tunneling approach than GRE. VXLAN requires the same accommodations for handling its headers that GRE does.

For simplicity's sake, we will lower the instance's MTU size to work around the header MTU size conflict in this book. We will do this by configuring DHCP to send a DHCP option to the instances telling them to use an MTU of 1,450. The header will fit comfortably in the 50 bytes of space we've created for it, and the packets will flow normally through the rest of the network that has GRE or VXLAN encapsulation. Be aware that this is not a 100 percent foolproof method. If the instance's operating system does not support accepting the DHCP option to lower the MTU, there is a chance that communication will not be established fully with the instance via its network device.

Creating a network

Now that we've explored some of the intricacies of what's happening under the hood, let's actually use Neutron to create a network by performing the following steps:

1. Log in to your control node and source your `keystonerc` file; use the non-administrative user for this. The command to create a virtual network is:

   ```
   control# neutron net-create internal
   control# neutron subnet-create internal 192.168.37.0/24
   ```

That's it. You just created a virtual network. I know that for the length of the introduction we just covered, that was pretty anticlimactic. Note that when you create the subnet, you're adding it to the network named `internal` that you just created. The final argument to the `subnet-create` command is the **Classless Inter-Domain Routing** (**CIDR**) notation. I'm not going to spend time on the CIDR notation here. You'll have to search the internet for an explanation of it. There are plenty of good ones. Also, search for the CIDR calculator; there are plenty of CIDR calculators on the Internet too.

Here are a couple of examples of CIDR calculators:

- `http://jodies.de/ipcalc`
- `http://www.subnet-calculator.com/cidr.php`

In a CIDR calculator, you can type in the CIDR mentioned earlier and it will give you the usable IP range that it signifies. The CIDR that I've used, `192.168.37.0/24`, identifies a range of IP addresses from `192.168.37.1` to `192.168.37.254` with `192.168.37.255` as the broadcast address. This means that we can allocate IP addresses in this range for things on our network.

2. Next, let's list the network that we just created; you can also list the subnet. Here's how:

```
control# neutron net-list
control# neutron subnet-list
```

The subnet could have been created with a name. If it was, we could have updated it by referring to its name. Since one wasn't passed, the subnet's ID will have to be used as follows:

```
control# neutron subnet-create internal 192.168.37.0/24 -name
internal_subnet
```

3. Let's update the subnet by adding a **Domain Name System** (**DNS**) name server. The properties of a subnet and a network can be passed at the time of creation or updated later. Refer to the Neutron command-line help for more details. Here's how we update the subnet by adding a DNS name:

```
control# neutron subnet-update {subnet-id-hash} --dns-
nameservers list=true 8.8.8.7 8.8.8.8
```


www.allitebooks.com

In *Chapter 3*, *Image Management*, we mentioned cloud-init. Cloud-init is the service that runs when an instance is booted and connects back to `169.254.169.254` to get metadata. SSH keys and post-boot scripts are two examples of what can be provided via metadata. This IP address is provided by a Neutron router and proxies the call from cloud-init to the metadata service. In that case, we need a router.

4. Let's create one and add the internal network as an interface to it:

```
control# neutron router-create my_router
control# neutron router-interface-add my_router {subnet-id-hash}
```

Here again, had we passed the `--name` argument and given the subnet a name, we could have used that name instead of the subnet ID. Now that the router has been created and attached to the subnet, the instances on this network will be able to talk to the metadata service on boot.

Web interface management

The web interface lets you create the network and subnet in the same dialog. Perform the following steps to obtain a network and a router:

1. Log in as your non-administrative user, select the **Network** menu, select the **Networks** submenu, and click on the **Create Network** button in the top-right corner, as shown here:

2. After you have filled in the network name, go to the next dialog screen and fill in the subnet information, as shown in the following screenshot:

Create Network

Network | Subnet * | Subnet Detail

Create Subnet:

Subnet Name:

192.168.37.0/24

Network Address:

192.168.37.0/24

IP Version: *

IPv4

Gateway IP:

Disable Gateway:

You can create a subnet associated with the new network, in which case "Network Address" must be specified. If you wish to create a network WITHOUT a subnet, uncheck the "Create Subnet" checkbox.

Network address in CIDR format (e.g. 192.168.0.0/24)

« Back | Next »

3. On the final dialog box add the DNS entries, as shown in the following screenshot:

Create Network

Network | Subnet * | Subnet Detail

Enable DHCP:

Allocation Pools:

DNS Name Servers:

8.8.8.7
8.8.8.8

Host Routes:

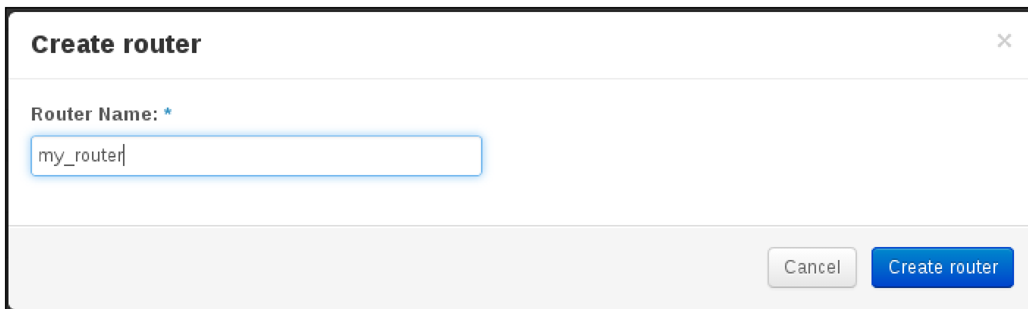
You can specify additional attributes for the subnet.

IP address list of DNS name servers for this subnet. One entry per line.

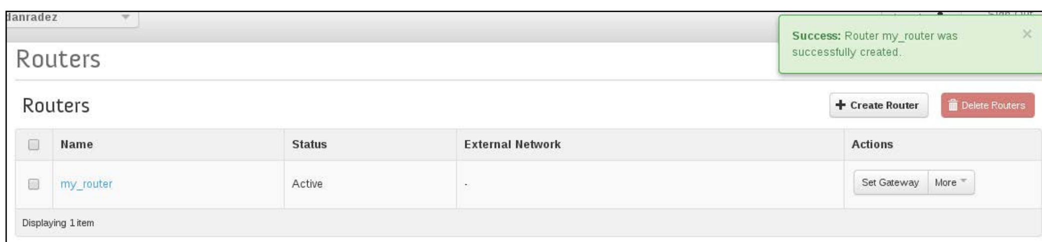
« Back | Create

4. When you've completed filling in the dialog, you'll end up with a network and a subnet that's associated with the network, as shown in the following screenshot:

5. Next, create the router. Select **Routers** from the **Network** menu, and click on **Create Router** in the top-right corner of the page, as shown in the following screenshot:



6. Once you've filled in the router name, click on **Create Router**, as shown in the following screenshot:



7. Next, click on the router's name and click on the **Add interface** button in the top-right corner, as shown in the following screenshot:

8. Select the subnet on the network you created and add it as an interface to the router. Once the router has been created, there will be a success message in the upper-right corner, as shown in the following screenshot:

Now that we have a network and a router available, an instance can be launched and attached to the network. When the launched instance runs cloud-init, it will be able to connect to the metadata service via the router. We'll launch the first instance when we get to *Chapter 6, Instance Management*. Before we do that, we have a little more networking to set up.

External network access

Every tenant will have at least one network to launch instances on, which will be built as we have just built a network. Whenever a new tenant is created, the steps that have just been performed will need to be performed for that new tenant. All tenants will share a network that provides external access to the outside world. Let's work through creating this external network.

Preparing a network

Earlier, we discussed how Neutron is an a API layer that manages virtual networking resources. The preparation for external network access will be different for different Neutron plugins. Talk to your networking vendor for your specific implementation. In general, what's being accomplished by this preparation is the connection of the networking node to a set of externally routeable IP addresses. External just means external to the OpenStack cluster. These may be a pool within your company's `10.0.0.0/8` network or a pool of IPs public to the Internet.
The tenant network IP addresses are not publicly routeable. The floating IP addresses allocated from the external network will be public and mapped to the tenant IP addresses on the instances to provide access to the instances outside of your OpenStack deployment. This is accomplished using the **Network Address Translation** (**NAT**) rules.

In future versions of Packstack, part of this process may already be completed for you. If you find some of it already completed by your installation, just use this section to gain an understanding of what has been done for you.

Since we are using Open vSwitch for this deployment, let's take a look at how to set up OVS. Let's start by looking at the virtual switches defined on the networking node as follows:

```
network# ovs-vsctl show
a621d2b2-a4cb-4cbd-8d4a-f3e802125445
    Bridge br-int
        Port patch-tun
            Interface patch-tun
                type: patch
                options: {peer=patch-int}
        Port br-int
            Interface br-int
                type: internal
    Bridge br-tun
```

```
            Port patch-int
                Interface patch-int
                    type: patch
                    options: {peer=patch-tun}
            Port br-tun
                Interface br-tun
                    type: internal
            Port "vxlan-2"
                Interface "vxlan-2"
                    type: vxlan
                    options: {in_key=flow, local_ip="192.168.123.102",
out_key=flow, remote_ip="192.168.123.103"}
        Bridge br-ex
            Port br-ex
                Interface br-ex
                    type: internal
        ovs_version: "2.0.1"
```

In this output, you can see three bridges. You can think of these exactly as you would think of a switch—as a network appliance that has a bunch of places to plugin Ethernet cables into. A port is just something plugged into one of these virtual switches. Each of these virtual switches has a port to itself; `br-int` is patched to `br-tun` and `br-tun` is patched to `br-int`. You can see the VXLAN tunnel established between the network node and the compute node on `br-tun`. `Br-ex` is just a switch that's not plugged into anything right now. `Br-int` is known as the integration bridge and is used for local attachments to OVS. `Br-tun` is the tunnel bridge used to establish tunnels between nodes, and `br-ex` is the external bridge, which is what we need to focus on.

The network node has interfaces for its actual network devices, which are probably `eth0` and `eth1`, or `em1` and `em2` depending on your distribution and device. What needs to happen is for the device on your network node that can route to the external pool of IP addresses to be plugged into `br-ex`. It is important when this happens to make sure that traffic flowing through the Ethernet device on the node communicates with OVS and not directly with the host itself. To make sure this happens, the IP address associated with the Ethernet device must be moved off the device and onto the OVS `br-ex`. To do this, we will create a network device configuration for `br-ex` and let Linux networking bring up this OVS device and the physical Ethernet device. Then, OVS will be used to bridge these two devices together. This is not a traditional Linux networking bridge; it is attaching the physical device to an OVS switch as a port.

Let's walk through what this looks like.

	Control node	Networking node	Compute node
eth0	192.168.123.101	192.168.123.102	192.168.123.103
eth1	192.168.122.101	192.168.122.102	192.168.122.103

First, look at the IP configuration and the configuration file for our nodes in the following table. Start by recalling the networking configuration defined in *Chapter 2, RDO Installation*. In this example, `192.168.122.0/24` is the external IP pool and `192.168.123.0/24` is the internal subnet for the OpenStack nodes to communicate. That means that the VXLAN tunnels will be established over `192.168.123.0/24`, as we saw in the OVS output, and the external floating IP addresses will be allocated from `192.168.122.0/24`. The network configuration file for `eth1` should look something like this:

```
network# cat /etc/sysconfig/network-scripts/ifcfg-eth1
DEVICE=eth1
BOOTPROTO=static
NM_CONTROLLED=no
ONBOOT=yes
IPADDR=192.168.122.100
NETMASK=255.255.255.0
GATEWAY=192.168.122.1
DNS1=192.168.122.1
```

A file for `br-ex` will not exist. A simple way to create one is to copy the file of eth1, as shown in the following command, because almost all of the configuration needed for `br-ex` is already in that file:

```
network# cd /etc/sysconfig/network-scripts/
network# cp ifcfg-eth1 ifcfg-br-ex
```

To complete the device configuration preparation, remove all of the IP addresses from the file of `eth1` and update the device name in the file of `br-ex`. The final result will look like this:

```
network# cat ifcfg-eth1
DEVICE=eth1
BOOTPROTO=static
NM_CONTROLLED=no
ONBOOT=yes
network# cat ifcfg-br-ex
DEVICE=br-ex
BOOTPROTO=static
NM_CONTROLLED=no
ONBOOT=yes
IPADDR=192.168.122.100
GATEWAY=192.168.122.1
NETMASK=255.255.255.0
DNS1=192.168.122.1
```

When networking is restarted, `eth1` will be brought up and operate at layer 2 only, and `br-ex` will be brought up ready to communicate on layer 3. If you are not familiar with the difference between layer 2 and layer 3, layer 2 is communication at the MAC address level and layer 3 is communication at the IP address level. The last piece of this puzzle is associating them together with OVS. When `eth1` gets plugged in as a port to `br-ex` in OVS, OVS will take control of the interface and traffic traveling on it will be interrupted until the devices are restarted. I am usually SSHed into a machine over my external device. To avoid this loss in connectivity, you can perform the following OVS command and the network restart in the same line; SSH will do a reconnect, and it will appear as though you never lost connection:

```
network# ovs-vsctl add-port br-ex eth1 && service network restart
Restarting network (via systemctl):                        [  OK  ]
network#
```

In the first command, you are adding the `eth1` port to the `br-ex` bridge or just plugging `eth1` into `br-ex`. When the prompt comes back, it means you have successfully prepared the underlying network infrastructure in OVS for an external OpenStack network.

Creating an external network

Now that OVS has connectivity to the externally routeable IP pool that will be managed by OpenStack, it's time to tell OpenStack about this set of resources it can manage. Because an external network is a general purpose resource, it must be created by the administrator.

Go ahead and source your `keystonerc_admin` file on your control node so that you can create the external network as a privileged user. Then, create the external network, as shown in the following commands:

```
control# neutron net-create --tenant-id services ext --
router:external=True --shared
```

```
control# neutron subnet-create --tenant-id services ext
192.168.122.0/24 --disable-dhcp --allocation_pool
start=192.168.122.2,end=192.168.122.99 --allocation-pool
start=192.168.122.110,end=192.168.122.254
```

You'll notice a few things here. First, the tenant that the network and subnet are created in is the services tenant. As mentioned in *Chapter 3*, *Identity Management*, everything is a member of a tenant. General purpose resources like these are no exception. They are put into the `services` tenant because users don't have access to networks in this tenant directly, so they would not have the ability to create instances and attach them directly to the external network. Things would not work if that was done because the underlying virtual networking infrastructure is not structured to allow this to work properly. Second, the network is marked as external and shared. Third, note the allocation pools; the nodes are 101, 102, and 103. So I've left out the IP addresses 100–109. This way, OpenStack won't allocate the IP addresses assigned to the nodes. Finally, DHCP is disabled. If DHCP was not disabled, OpenStack would try to start and attach a dnsmasq service for the external network. This should not happen because there may be a DHCP service running external to OpenStack that would conflict with the one that would have started if DHCP was enabled on the external network.

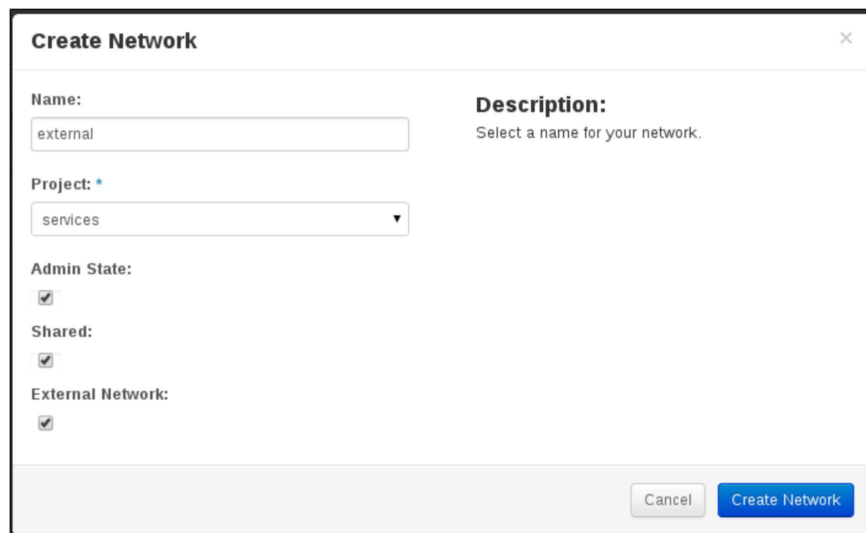
The final step to make this network accessible to the instances that will be launched on a tenant network is setting the tenant router's gateway to the external network. Let's do that for the router created earlier, as shown in the following command:

```
control# neutron router-gateway-set my_router ext
```

Web interface external network setup

Previously, there was preparation done to tie `eth1` and `br-ex` together. This preparation must be done on the command line. Because it is external configuration to OpenStack, it cannot be completed through the OpenStack web interface. You can, however, complete the external network creation through the web interface by performing the following steps:

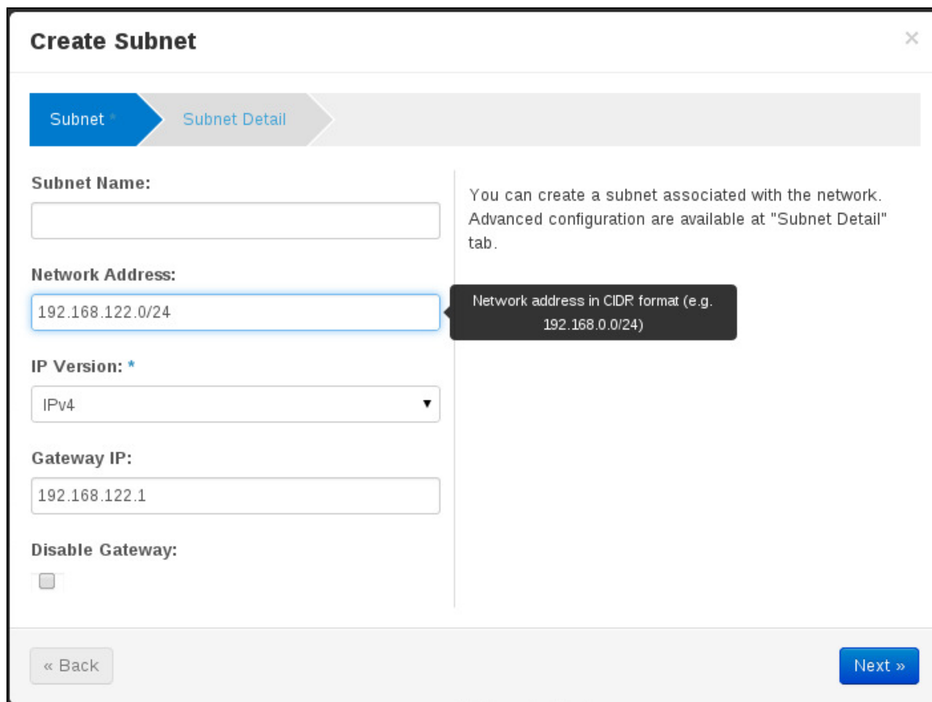
1. Start by logging in to the web interface at the **Admin** user to create the external network and subnet. Select the **Networks** submenu from the **Admin** menu and click on **Create Network**. Give the network a name and flag it as external and shared. This step is encapsulated in the following screenshot:



www.allitebooks.com

2. Once you have created the network, select the network by its name and click on **Create Subnet**, as shown in the following screenshot:

3. Fill out the form with the network information for the external pool of IP addresses. Make sure the correct gateway is specified. The following screenshot captures this step:



4. Move to the **Subnet Detail** dialog and make sure that you uncheck **Enable DHCP**. Fill in the allocation pool if necessary. This is only necessary when creating an external network as the administrative user. This step is amply illustrated in the following screenshot:

5. Once the subnet and network are created, log out of the admin account and log back in as the non-privileged user, as shown in the following screenshot:

6. Select the **Routers** menu options from the **Network** menu, click on **Set Gateway**, and select the external network you just created, as shown in the following screenshot:

7. Once the router's gateway is set to the external network, everything will be in place to assign a floating IP address to an instance once it's launched, as shown in the following screenshot:

Summary

In this chapter, we looked at creating networks, subnets, routers, and the preparation involved in using an external network with Open vSwitch. Using these resources, the necessary virtual networking fabric has been created for an instance to be launched on. Now that we have created virtual networks for the instances to attach to, let's get into launching instances. In the next chapter, we will do what we have been working towards—launch an instance. We will use Nova to launch an instance from the image that was imported and attach it to these virtual networking resources.

6
Instance Management

In the past few chapters, we collected resources that laid the foundation to launch an instance. We have created a tenant—a place for our resources to live in. We added a disk image that the instance will use as its boot device. We created a network for the instance. Now, it is time to launch the instance. Nova is the instance management component in OpenStack. In this chapter, we will look at managing:

- Flavors
- Key pairs
- Instances
- Floating IPs
- Security groups

Managing flavors

When an instance is launched, there has to be a definition of the amount of resources that will be allocated to the instance. In OpenStack, this is defined by what are called flavors. A flavor defines the quantum of virtual CPUs, RAM, and disk space that an instance will use when it launches. When Packstack installed your system earlier, it created a few flavors for you. Let's take a look at those. Go ahead and source a `keystonerc` file. If you don't have one sourced, then list the flavors, as follows:

```
control# nova flavor-list
```

You can create your own flavors if these don't fit your needs exactly. There's nothing magical about the ones that Packstack has created. They have been created close to what the rest of the cloud industry uses. We're not going to get too deep into flavors; we'll just use the preconfigured flavors that you have just listed.

Managing key pairs

As a cloud image is a copy of an already existing disk image with an operating system already installed, the root users are generally disabled, and if the root password is set, it is usually not distributed. To overcome the inability to authenticate without a password, OpenStack uses SSH key pairs. If you remember, in *Chapter 4*, *Image Management*, we discussed the need for cloud-init to be installed in a cloud image. Then, in *Chapter 5*, *Network Management*, we discussed how cloud-init would connect to the metadata service via the IP address provided by the router. One of the primary roles of this process is to pull down the public SSH key that will be used for authentication. OpenStack provides a facility for you to manage your SSH key pairs so that you can select which will be used when you launch an instance. Let's start by generating a new key pair and listing it, as shown in the following commands:

```
control# nova keypair-add my_keypair
-----BEGIN RSA PRIVATE KEY-----
{ truncated private key content }
-----END RSA PRIVATE KEY-----
control# nova keypair-list
```

This has generated an SSH public/private key pair and listed the record of the key pair. The content that has been put on standard output should end up in a file in your home directory's SSH directory with a mode of 600. OpenStack has generated the key pair, given you the private key, and stored the public key to place on a future running instance. You could always redirect the output to a file so that you don't have to copy and paste. This is an alternative way to generate that key pair. The only difference is that the private key ends up in a file instead of being printed in the terminal. Issue the following command to accomplish this:

```
control# nova keypair-add another_keypair > another.key
```

Once you have a file that contains the private key, for example, the other key file just created, you can drop this file into your `~/.ssh` directory with a mode of 600. Then, that file is referenced to log in to a running instance that has the respective public key.

SSH key pairs are not anything specific to OpenStack. They are very commonly used in the Linux world. OpenStack supports the importing of an already existing public key. Let's walk through generating an SSH key pair outside of OpenStack and importing the public key into OpenStack, as shown in the following commands:

```
laptop$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/dradez/.ssh/id_rsa):
/home/dradez/.ssh/openstack.rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in
/home/dradez/.ssh/openstack.rsa.
Your public key has been saved in
/home/dradez/.ssh/openstack.rsa.pub.
The key fingerprint is:
4f:62:ee:b9:0f:97:35:f7:8a:91:37:84:0b:b9:cb:05 dradez@laptop
dradez@laptop:~$ ls -l /home/dradez/.ssh/openstack*
-rw-------. 1 dradez dradez 1675 /home/dradez/.ssh/openstack.rsa
-rw-r--r--. 1 dradez dradez  411
/home/dradez/.ssh/openstack.rsa.pub
```

As illustrated, on my laptop, I have generated a public/private key pair. The private key has a mode of 600, and the public key is the file that will be imported into OpenStack. In the OpenStack cluster, we're using the control node to interact with the cluster. Copy the public key to your control node so that it can be imported, and import it into Nova, as shown in the following command:

```
control# nova keypair-add --pub_key openstack.rsa.pub keypair_name
```

You can also manage key pairs in the web interface. In the **Compute** menu, select the **Access & Security** submenu. On this page, there will be a **Key Pairs** tab. You can click on **Create Key Pair** or **Import Key Pair** to manage key pairs through the web interface instead of on the command line. The following screenshot captures how we can manage key pairs in the web interface:

Launching an instance

At this point, there has been what may seem like an excessive amount of groundwork laid to get to launching an instance. We now have a tenant for the instance to live in, an image using which it can run, a network for it to live in, and a key pair to authenticate with. These are all the necessary resources to create in order to launch an instance, and now that these resources have been created, they can be reused for future instances that will be launched. Without further delay, let's launch the first instance in this OpenStack environment as follows:

```
control# nova boot --flavor 2 --image Fedora --key-name openstack --
nic net-id={network id} ""My First Instance""
```

This launches an instance using the small flavor, the key pair we just imported, the Fedora image from *Chapter 4*, *Image Management*, and the tenant network from *Chapter 5*, *Network Management*. This instance will go through a few different states before it's ready to use. You can see the current state of your instances by listing them as follows:

```
control# nova list
```

This command will list the instances in your tenant. Once the instance boot process completes, the instance will settle in an active state. The first time an instance boots, it will take an extra minute or two because the image file has to be copied from Glance to the hypervisor. Successive instance launches should happen in less than a minute. Administrative users also have the ability to see all instances. If the admin's `keystonerc` file is sourced, pass the `-all-tenants` option to see all instances, as shown in the following command:

```
control# nova list --all-tenants
```

Initially, the only communication you have with the instance is getting console logs or connecting to the console via Nova, as follows:

```
control# nova console-log ""My First Instance""
control# nova get-vnc-console ""My First Instance"" novnc
```

The first command will print the console log of the instance if it's available. This is useful to help debug why an instance won't start or to find out if it got an IP address from DHCP. The second command will give you a URL that can be loaded into your browser to give you a VNC console to the running instance.

Managing floating IP addresses

Now that an instance is running, the next step is to communicate with it in a fashion other than with the console through a web browser. In the instance list you just saw, an IP address on the tenant network will be listed once it's been assigned. The IP address that's initially assigned to the instance is not a routeable IP address; to communicate with the instance, you will need to assign a floating IP address from the external network. The floating IP address will be mapped to the tenant network IP address, and you will be able to communicate with the instance by way of the floating IP address.

Before a floating IP address can be associated with an instance, it needs to be allocated to your tenant. Floating IP addresses are managed through Neutron, as follows:

```
control# neutron floatingip-create external
```

This allocates a floating IP address to the tenant. List it so that you can get its details, as follows:

```
control# neutron floatingip-list
```

Use the ID of the allocated IP address to assign it to the port of the running instance. To do that, you'll have to find the port ID of your instance. Use the IP address assigned to the instance from the list of instances to cross reference with a list of ports. Then, associate the floating IP address ID with the port ID, as follows:

```
control# neutron port-list | grep {ip address}
control# neutron floatingip-associate {floating ip id} {port id}
```

When the association is complete, you will see the floating IP address listed next to the tenant network IP address for your instance.

Managing security groups

At this point, you may think that you should be able to connect to your instance. Not quite yet. There is a layer of security built into OpenStack called security groups. Security groups are tenant-level firewalls. You can define multiple security groups; you can even assign multiple security groups to a running instance. A security group named default is created for each tenant when the tenant is created. Let's list that default group:

```
control# neutron security-group-list
```

To see the rules defined in a security group, list the rules. This command lists all the rules in the tenant. If you want to see the rules for a specific security group, you'll have to filter out the security group you are interested in; `grep` is a good tool for this. Here are the commands to accomplish this:

```
control# neutron security-group-rule-list
control# neutron security-group-rule-list | grep sec_group_name
```

As illustrated, the default rules added to the default security group are pretty basic and restrict all incoming traffic from the outside. Ingress is incoming traffic, and only incoming traffic from within the security group itself is allowed. Egress is outgoing traffic; all outgoing traffic is allowed by default. Let's add a few rules to allow some external traffic to connect to the instance:

```
control# neutron security-group-rule-create --protocol tcp --port-
range-min 22 --port-range-max 22 --remote-ip-prefix 0.0.0.0/0 default
```

This rule will allow all incoming SSH traffic on `port 22` to be passed to instances in the default security group. If you try to do this with the admin's `keystonerc` sourced, you'll get a message indicating that multiple security groups named default were found. This is because the admin user can see everyone's security group, so you'll have to use the security group ID instead. Let's add a rule to allow us to ping the host too:

```
control# neutron security-group-rule-create --protocol icmp --remote-
ip-prefix 0.0.0.0/0 default
```

As mentioned earlier, you can also have more than one security group. You can create a new security group with Neutron's `security-group-create` command, as follows:

```
control# neutron security-group-create new_secgroup
```

Then, other rules could be added to the group, for example, a rule to allow access to port 80 for web traffic:

```
control# neutron security-group-rule-create --protocol tcp --port-
range-min 80 --port-range-max 80 --remote-ip-prefix 0.0.0.0/0
new_secgroup
```

Now, when an instance is launched, the option `--security-groups` could be passed. The value given to it could be `default` or `new_secgroup` or `default,new_secgroup`. The respective traffic would be allowed based on what combination of security groups was assigned to the new instance being booted. If you don't pass this option, the default security group will automatically be the group assigned to the new instance.

Communicating with the instance

The instance we booted was assigned the default security group. Edits made to a security group are immediately applied to the instances operating in them. We just added the ping and SSH rules to allow incoming traffic to the instances running in the default security group, so you should be able to ping and SSH to the instance you launched. Here's the output summary:

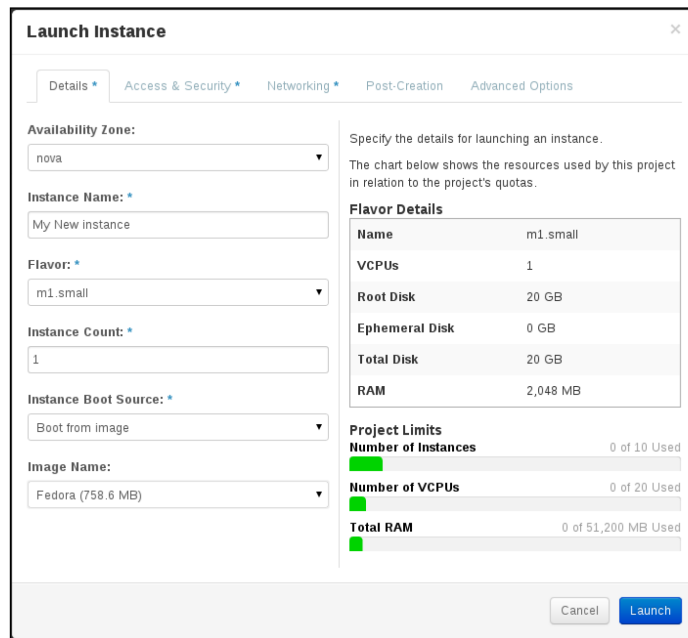
```
laptop# ping -c 3 192.168.122.3
PING 192.168.122.3 56(84) bytes of data.
64 bytes from 192.168.122.3: icmp_seq=1 ttl=64 time=0.040 ms
64 bytes from 192.168.122.3: icmp_seq=2 ttl=64 time=0.041 ms
64 bytes from 192.168.122.3: icmp_seq=3 ttl=64 time=0.040 ms

--- 192.168.122.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.040/0.040/0.041/0.005 ms
laptop# ssh fedora@192.168.122.3
The authenticity of host '' 192.168.122.3 (192.168.122.3)'' can''t be
established.
RSA key fingerprint is 83:d8:f4:7e:01:db:4e:50:8a:bd:f6:dc:77:2c:31:d7.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added ''192.168.122.3'' (RSA) to the list of
known hosts.
[fedora@test ~]$
```

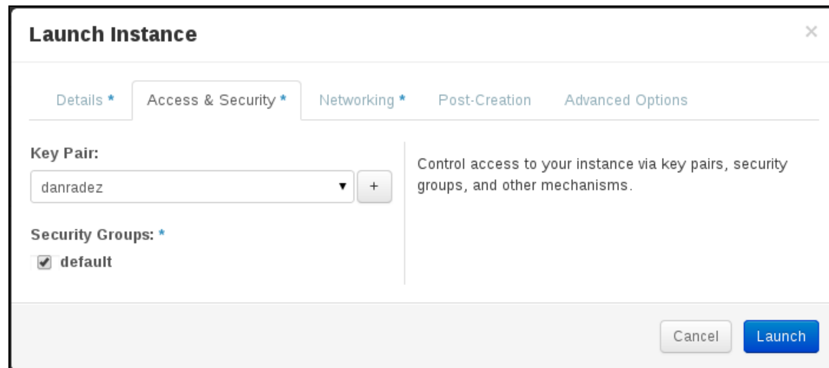
Launching an instance using the web interface

Now that we've booted an instance on the command line, let's take a look at doing the same thing in the web interface:

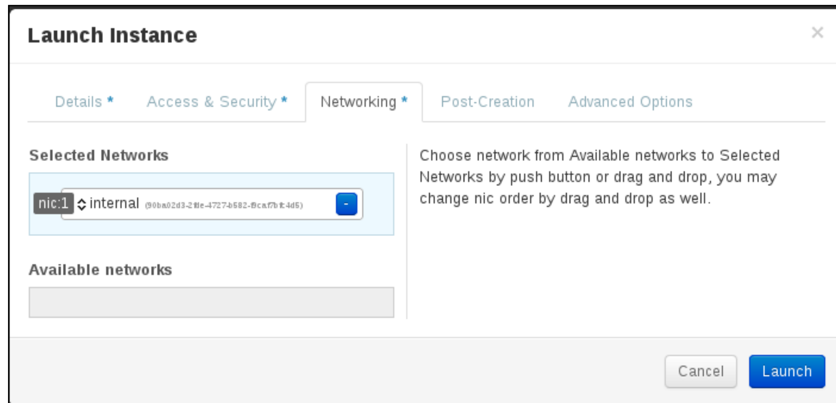
1. Go ahead and log in to the web interface as the non-administrative user you created in *Chapter 3, Identity Management*.
2. Under the **Compute** menu, select **Instances** and then click on the **Launch Instance** button.
3. In the **Launch Instance** dialog, fill in a name for the instance, choose a flavor, and select **Boot from image** as the boot source, as shown in the following screenshot.
4. Select the image you uploaded in *Chapter 4, Image Management*.



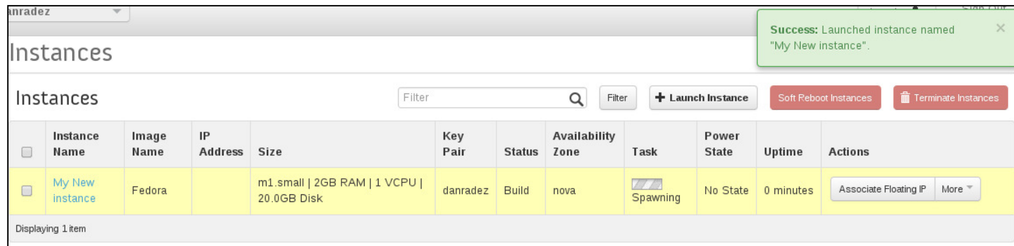
5. Select the next tab, **Access & Security**. This view should already be filled in for you. This is the view in which you can select a different SSH key pair and select a combination of security groups for the instance. The following screenshot is a view of the **Access & Security** tab:



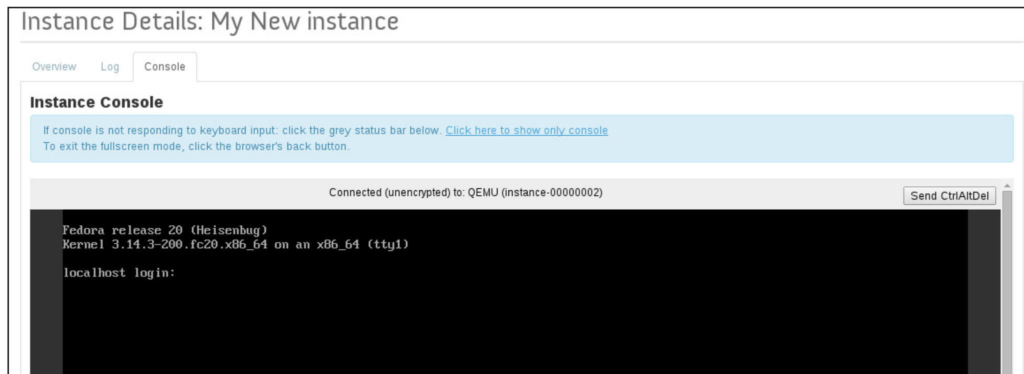
6. Next, select the **Networking** tab. In this view, if you have only one tenant network defined, it will be automatically added to the selected networks. If you have more than one network, you will have to add one to the selected networks. More than one network can be added to the selected networks. Each will be added to your instance as a virtual network interface and each interface will DHCP to a separate DHCP service to get its IP address. The following screenshot is a view of the **Networking** tab:



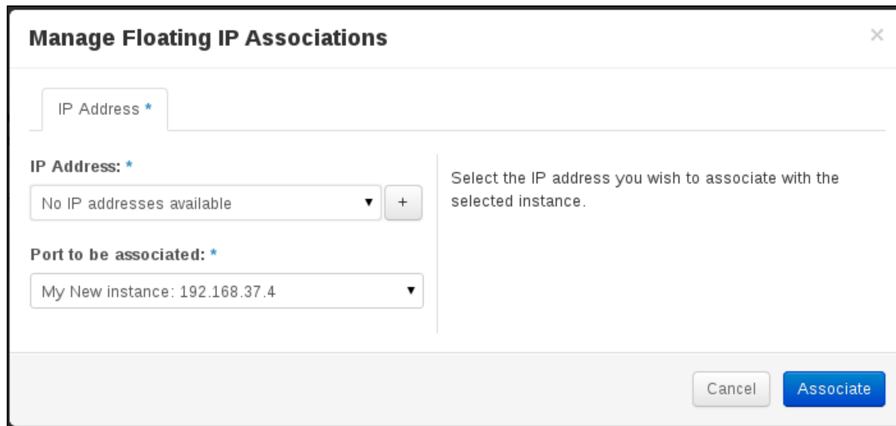
7. Go ahead and launch the instance; the web interface will poll OpenStack to keep you informed of the progress of the launching instance. You can watch the instance's progress as it launches. The polling will update the web interface as the state changes. This screenshot captures a still of the launching instance on the web interface:



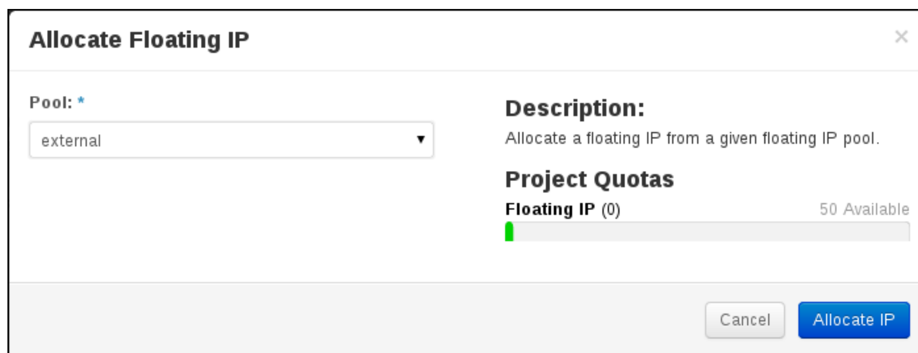
8. Click on the instance's name; there is a **Log** tab and a **Console** tab. The **Console** tab will load the URL that was displayed by the `get-vnc-console` command executed in *Launching an Instance* section. Here's a screenshot of this page:



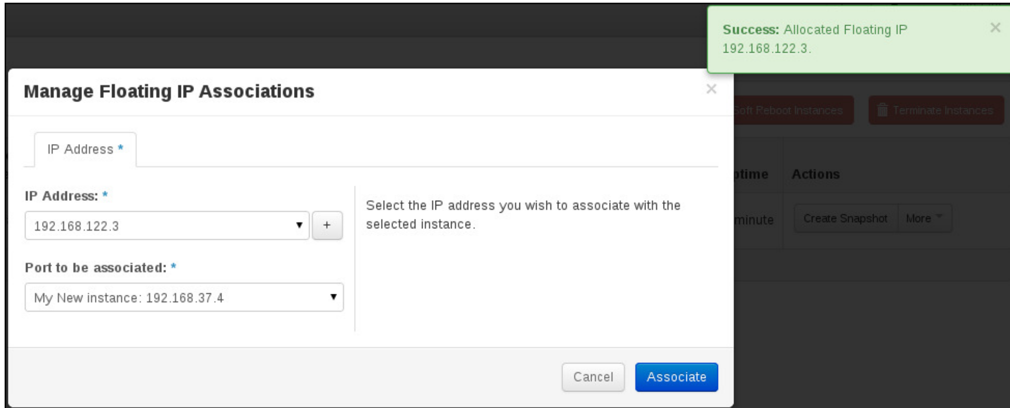
9. The next step is to associate a floating IP address with the instance. You can use the button that is displayed before the instance is activated. Once the instance is activated, you can find the associated floating IP option in the **More** menu on the instance. Select **Associate** to associate the floating IP address with the instance:



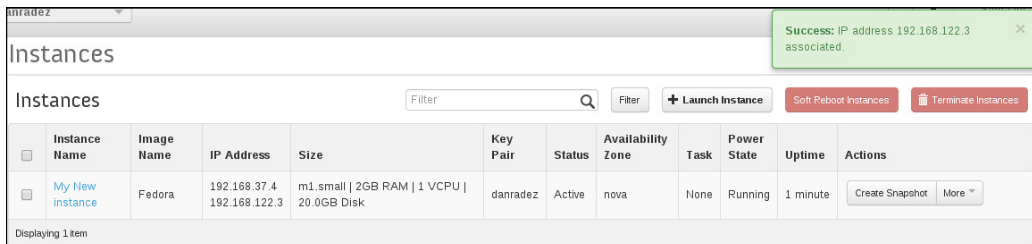
10. Remember that a floating IP first needs to be allocated to a project before it can be associated with an instance. Click on the **+** button next to the IP address selection box. The following screenshot captures the instructions in this step:



11. Click on **Allocate IP** to allocate an IP to your project. Then, complete the association with the running instance, as shown in the following screenshot:



12. Once the floating IP has been associated, it will show up in the IP address box next to the instance. This is not an auto-update piece of information. You will have your browser's refresh button to refresh the page to see the update. The following screenshot captures this step:



13. The final step is to open up the security groups. Click on the **Access & Security** menu, select the **Security Groups** tab, and click on the **Manage Rules** button. In the top-right corner, click on the **Add Rule** button.

14. Fill out the form for ICMP, as shown in the following screenshot:

15. Repeat this for SSH and for any other port you need opened up.

Summary

In this chapter, we looked at managing flavors, key pairs, instances, security groups, and floating IP addresses. Now that we have a running OpenStack instance, let's attach some virtual storage to it. A running instance's storage is ephemeral by design. This means that any data stored in the instance's local disk is lost upon the instance's termination. In the next chapter, we will attach a virtual block storage device to the running instance. This storage will persist after an instance that it is attached to is terminated.

7
Block Storage

Cinder is the block storage component in OpenStack. In the previous chapter, all the necessary resources were collected to launch an instance. Now that this instance is running, let's look at the use case for block storage and the process of attaching block storage to the OpenStack instance. Then, we will take a look at the storage engine used to store these block devices and the other options available for the backing store.

Use case

OpenStack instances run on ephemeral disks—disks that only exist for the life of the instance. When an instance is terminated, the disk is discarded. This is a problem if there is any information that requires persistence. Block storage is one type of storage that can be used to persist data beyond the termination of an OpenStack instance.

Using Cinder, users can create block devices on demand and present them to running instances. The instances see this as a normal, everyday block device—as if an extra hard drive was plugged into the machine. The extra drive can be used as any other block device by creating partitions and file systems on it. Let's look now at how to create and present a block storage device using cinder.

Creating and using block storage

Creating a block device is as simple as specifying the size and an optional name for the block device being created.

Create two volumes, one with a display name and another without:

```
control# cinder create 1
control# cinder create 1 --display_name data_vol
```

These two commands created two virtual block devices that are 1 GB of storage space each. To see the two devices, use Cinder's list command:

```
control# cinder list
```

The two volumes will be listed with information about them. As with the components already covered, the `--all-tenants` option can be passed as an administrative user to see a list of all volumes that are in Cinder:

```
control# cinder list --all-tenants
```

When volumes are created, they cycle through a progression of states that indicate the status of the new block device. When the status reaches *Available*, it is ready to be attached to an instance.

Attaching the block storage to an instance

The virtual storage device we just created is not much good to us unless it is attached to an instance that can make use of it. Luckily for us, we just launched an OpenStack instance and logged in to it. Perform the following steps to attach the block storage to an instance:

1. Start by listing the existing block devices on the instance that was started:

   ```
   instance# ls /dev/vd*
   /dev/vda  /dev/vda1
   ```

   The boot device for this instance is `vda`.

2. Now use Nova to attach the volume you just created to the instance you have running. When you list the devices on the instance again, you will see the Cinder volume show up as `vdb`:

   ```
   control# nova volume-attach instance_name {volume-id}
   instance# ls /dev/vd*
   /dev/vda  /dev/vda1  /dev/vdb
   ```

3. Now that we have a new block device on the instance, we treat it just as we would any other block device. Make a partition, create a file system, mount it, and read and write to it. The output from the following commands will be truncated for brevity:

   ```
   instance# fdisk /dev/vdb
   Command (m for help): n
   ```

```
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-2097151, default 2048):
Last sector, +sectors or +size{K,M,G,T,P} (2048-2097151,
default 2097151):
Created a new partition 1 of type 'Linux' and of size 1023
MiB.
Command (m for help): w
The partition table has been altered.
instance# mkfs -t ext4 /dev/vdb1
Writing superblocks and filesystem accounting information:
done
control# mount /dev/vdb1 /mnt
control# echo "test" > /mnt/test
control# cat /mnt/test
test
```

4. For the sake of an example, let's `unmount` the device and detach it from the running instance:

```
instance# umount /mnt
control# nova volume-detach instance_name {volume-id}
instance# ls /dev/vd*
/dev/vda  /dev/vda1
```

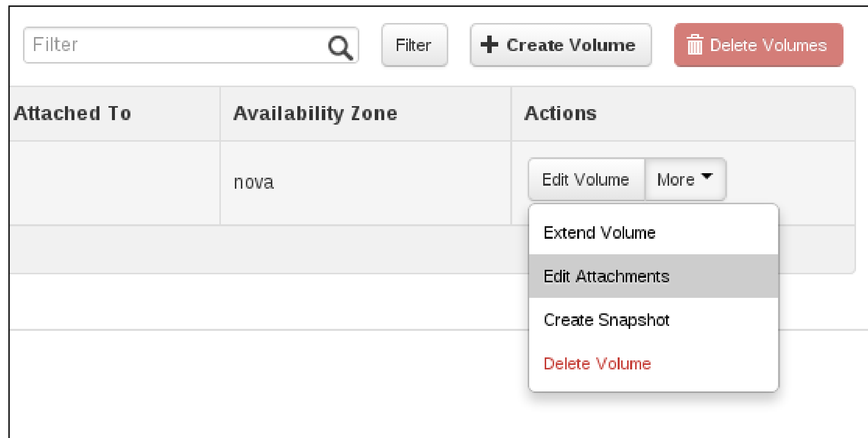
In these steps, we showed that only `vda` exists on the instance. Next, we attached the volume and showed you how the instance sees it as `vdb`. Then, we partitioned, mounted, and wrote to the file system. Finally, the device was unmounted and detached, and it was shown that `vdb` has been removed.

Managing Cinder volumes in the web interface

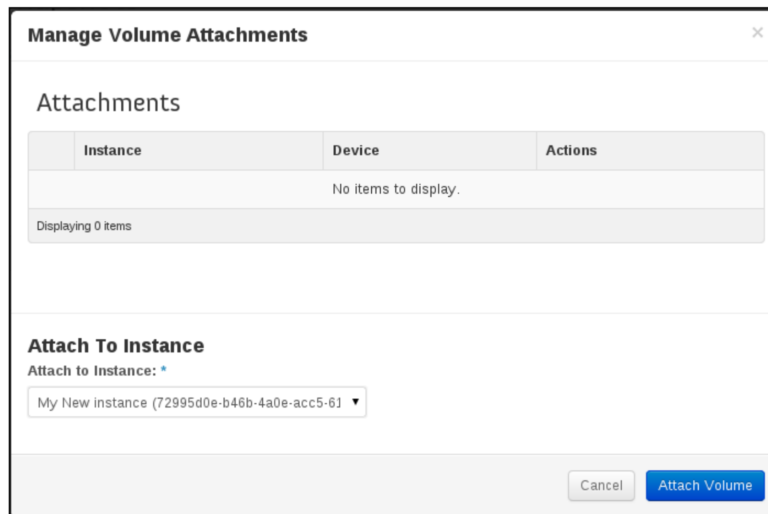
Now that we have used the command line to manage Cinder volumes, let's take a look at using the web interface to accomplish the same thing:

1. Log in to the web interface as your non-administrative user and select the **Volumes** submenu from the **Compute** menu.
2. In the top-right corner, click on the **Create Volume** button.
3. Fill in the name and size and click on **Create Volume** on the form.

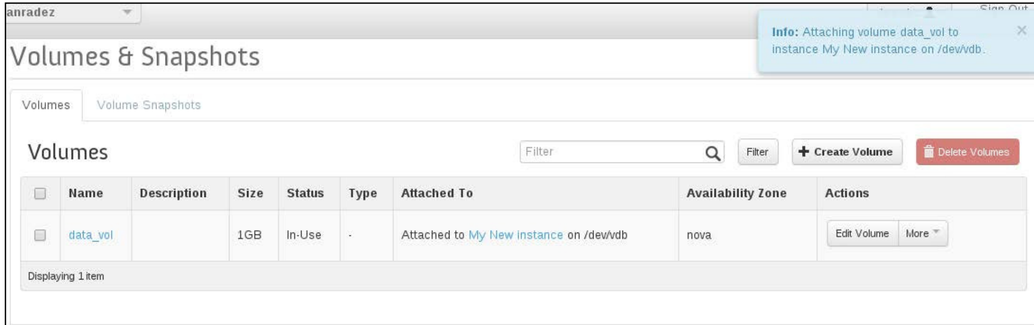
4. The web interface will update itself as the volume status changes. Once it becomes available, click on the **More** menu on the volume page and select **Edit Attachments**. In this dialog, the volume will be connected to the running instance. The following screenshot captures this step:



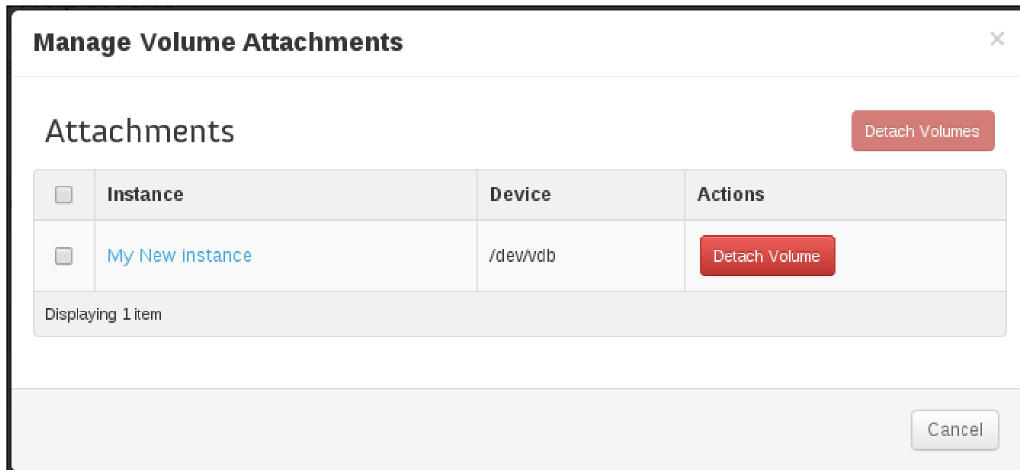
5. In the **Attachments** dialog, select the instance to attach the volume to and click on the **Attach Volume** button, as shown in the following screenshot:



6. Once again, the web interface will get updated as the status of the volume changes. The volume's status will become **In-Use** when it is attached to the instance and ready for the initial partitioning and file system creation. The following screenshot encapsulates this step:



7. To detach the volume, open the **Edit Attachments** dialog in the **More** menu and click on **Detach Volume**. The status of the volume will return to *Available* once the detach process is complete. The following screenshot captures this step aptly:



Backing storage

Now that you have seen how to use Cinder, you may be wondering where that volume that was created was stored. The cloud may be a facade of endless resources, but the reality is that there are actual physical resources that have to back the virtual resources of the cloud. By default, Cinder is configured to use LVM as its backing store. Packstack will look for a volume group named `cinder-volumes` configured on the node running the cinder-volume service. If one does not exist, Packstack will create a virtual disk and mount it as a loopback device to use as the physical volume to create a `cinder-volumes` volume group. The volume you just created was a logical volume in the `cinder-volumes` volume group. This is not an ideal place to store a virtual storage resource for anything more than a demonstration. Using a virtual disk mounted as a loopback device has very poor performance and will quickly become a bottleneck under load.

If there is a `cinder-volumes` group, then Packstack will simply use it. If you use this environment for more than demonstration, make sure that this volume group exists physically on the disk and not as a virtual loopback device. Tearing down and recreating a volume group and its associated physical volume is outside the scope of this book. It should be sufficient to explain that if the `cinder-volumes` volume group disappears, Cinder will throw an error upon trying to interact with it. When it reappears, things will all work properly again. In that case, it is safe to delete all Cinder volumes using Cinder, tear down the `cinder-volumes` volume group and the associated physical volume, and rebuild it with physical disk architecture in lieu of a loopback disk architecture. If you use LVM as the backing store for Cinder, it will look for a volume group named `cinder-volumes`, and the volumes that Cinder creates will live as logical volumes in that volume group.

Cinder types

There are many different types of storage that can be used to back Cinder. LVM is an easy choice although software-defined solutions, such as GlusterFS and Ceph, are also very popular. Another option would be to engage your favorite hardware storage vendor as you ask them about their support for Cinder storage. When multiple backing storage solutions are used in Cinder, they are defined and referred to as types. To demonstrate this, let's keep the LVM storage available and add another backing store option as a second Cinder type to our OpenStack cluster. To do this, we'll add the GlusterFS software-defined storage solution as an alternative to a Cinder type.

GlusterFS setup

Conveniently enough, a simple GlusterFS installation is not extremely complicated to set up. Assume three rpm-based Linux nodes named gluster1, gluster2, and gluster3 with an `sdb` drive attached for use by a GlusterFS storage cluster. The file system XFS is recommended although an `ext4` file system will work fine in some cases. Research the pros and cons of each file system related to GlusterFS before you deploy a production GlusterFS storage cluster. Create a partition and a file system on the `sdb` disk. We'll begin our demonstration for this book with mounting the disk and creating and starting the GlusterFS volumes. The following steps should be performed on each of the GlusterFS nodes:

1. Start by preparing the host and installing GlusterFS:

```
# mkdir -p /export/sdb1 && mount /dev/sdb1 /export/sdb1
# echo "/dev/vdb1 /export/vdb1 ext4 defaults 0 0"  >>
/etc/fstab
# yum install -y glusterfs{,-server,-fuse,-geo-replication}
```

2. The following commands should be run only on one node as they propagate across the GlusterFS storage cluster via the Gluster services:

```
# service glusterd start
# gluster peer probe gluster2
# gluster peer probe gluster3
# gluster volume create openstack-cinder rep 3 transport tcp
gluster1:/export/vdb1 gluster2:/export/vdb1
gluster3:/export/vdb1
# gluster volume start openstack-cinder
# gluster volume status
```

   The last command should show you the Gluster volume you just created and the bricks that are being used to store the GlusterFS volume `openstack-cinder`. What these commands set up is a three-node Gluster installation where each node is a replica. That means that all the data lives on all three nodes. Now that we have GlusterFS storage available, let's configure Cinder to know about it and present it to the end user as a backing storage option for Cinder volumes.

Now that GlusterFS is set up, we need to tell Cinder about setting up the Cinder volume types. Let's configure Cinder to use GlusterFS as a backing store:

1. Start by editing `/etc/cinder/cinder.conf`; make sure that the `enable_backends` option is defined with the following values and that the respective configuration sections are defined in the file:

```
enabled_backends=my_lvm,my_glusterfs
[my_lvm]
volume_group = cinder-volumes
volume_driver = cinder.volume.drivers.lvm.LVMISCSIDriver
volume_backend_name = LVM
[my_glusterfs]
volume_driver = cinder.volume.drivers.glusterfs.GlusterfsDriver
glusterfs_shares_config = /etc/cinder/shares.conf
glusterfs_sparsed_volumes = false
volume_backend_name = GLUSTER
```

The `my_lvm` definition preserves the existing LVM setup that has already been used to create a Cinder volume. The `my_glusterfs` section defines options to attach to the GlusterFS storage we have just configured. You will also need to edit the `/etc/cinder/shares.conf` file. This file defines the connection information to the GlusterFS nodes. Reference the first and second Gluster nodes in the `shares.conf` file. It contains the following line:

```
gluster1:/openstack-cinder -o backupvolfile-
server=gluster2:/openstack-cinder.
```

2. Next, you'll need to restart the Cinder services to read the new configurations added to the `cinder.conf` file:

```
control# service openstack-cinder-scheduler restart
control# service openstack-cinder-volume restart
control# mount | grep cinder
gluster1:openstack-cinder on /var/lib/cinder/... type
fuse.glusterfs
gluster2:openstack on /var/lib/cinder/... type fuse.glusterfs
```

3. The `mount` command shown here just verifies that Cinder has automatically mounted the Cinder volumes defined. If you don't see the Cinder volumes mounted, then something has gone wrong. In that case, check the Cinder logs and the Gluster logs for errors to troubleshoot why Cinder couldn't mount the Gluster volume. At this point, the backing stores have been defined, but there is no end user configuration that has been exposed. To present the end user with this new configuration, Cinder type definitions must be created through the API:

```
control# cinder type-create lvm
control# cinder type-key lvm set volume_backend_name=LVM
control# cinder type-create glusterfs
control# cinder type-key glusterfs set
volume_backend_name=GLUSTER
```

```
control# cinder type-list
```

4. Now there are two types available that can be specified when a new volume is created. Further, when you list volumes that are in Cinder, they will have a volume type corresponding to which backing store is being used for each volume:

```
control# cinder-create --volume-type glusterfs 1
control# cinder list
```

5. The next time you create a new volume in the web interface, the two types will be available for selection on the volume creation dialog. The original `lvm` backing store is available and GlusterFS has been added too. This is shown in the following screenshot:

Summary

In this chapter, we looked at creating Cinder volumes and adding an additional storage type definition. Cinder block storage is just one virtual storage option available. In the next chapter, we will take a look at the Swift object storage system to compare the storage options available to OpenStack instances. Cinder offers block storage that attaches directly to the instances. Swift offers an API-based object storage system. Each storage offering has its advantages and disadvantages and is chosen to meet specific needs in different use cases. It is important to know how each of these works so that you can make an informed decision about which is right for you when the time comes to choose a storage solution.

8
Object Storage

In the previous chapter, we looked at managing block storage with Cinder. Block storage attaches directly to the instances, and the operating system on the instance writes to the file system. Object storage is an alternative storage option. Object storage is a simple form of storage that handles file operations on the instances by way of API calls. Swift is the object storage component in OpenStack. In this chapter, we're going to take a deeper look at what object storage is, how to use it, and some options available to use as the backend storage engine.

Use case

Object storage works by using a client to send and receive files to and from the object store. The files are stored with very little metadata and are treated as a whole entity. The object server does not work in partial pieces of an object the way block storage would work with file blocks. It is a very simple storage method focused on storing and retrieving the contents of the files with minimal overhead to the operating system while interacting with the storage server. The power of the Swift object storage engine is its robust software-defined storage backend. The Swift storage engine has distribution and replication capabilities across its storage nodes. First, let's take a look at the client side of using Swift, and later, we will look at the backend storage engine.

Architecture of a Swift cluster

Swift has a proxy layer and a storage layer. These layers are associated with each other by way of the ring. The ring is a catalog of the objects that are being stored in the cluster and where they are being stored. This information is replicated to every storage node to improve the performance of the cluster. The proxy layer is a service that presents an API interface to end users and communicates with the storage layer on behalf of the end user. The storage layer is not generally communicated with directly.

By default, Swift uses the Swift storage engine for its storage backend. The Swift storage is a storage engine designed specifically for the Swift object storage cluster that is distributed in nature and able to be replicated. The Swift storage engine has a few subcomponents to it: the account server, the object server, and the container server.

Swift can also be backed by storage engines other than the Swift storage engine. There are a few other storage solutions that have integration with Swift.

Each object stored in Swift is associated with a container, and containers are owned by an account. These associations are stored in the ring; there is a separate ring file for each of the accounts, containers, and objects.

Creating and using object storage

The two main concepts when using Swift are containers and objects. Containers are groups of files that contain objects. Objects are simply files and must exist inside of a container. On the Swift command line, you cannot create an empty container. A container is created when the first object is uploaded to it. Make sure that your `keystonerc` file is sourced, and start by uploading a file to a container. Let's use the `packstack.txt` file used for installation as an example file to upload:

```
control# swift upload my_container packstack.txt
control# swift list
control# swift list my_container
```

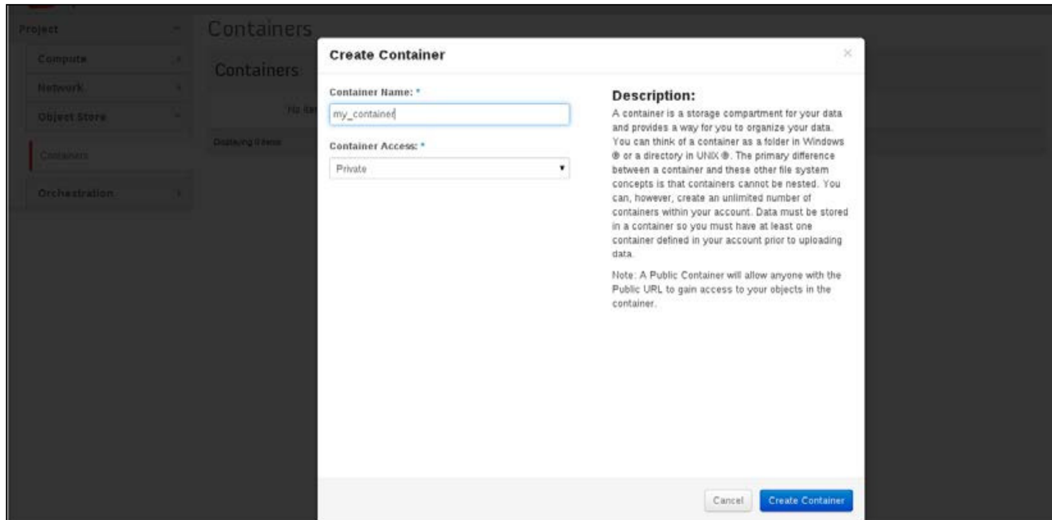
You will notice that the list command is used to list both containers and objects. If you don't pass any arguments to the list command, you will get a list of containers. If you pass a container as an argument, you will get a list of the objects in that container. Next, upload the same file, but use its absolute path to upload it, as follows:

```
control# swift upload my_container /root/packstack.txt
control# swift list my_container
```

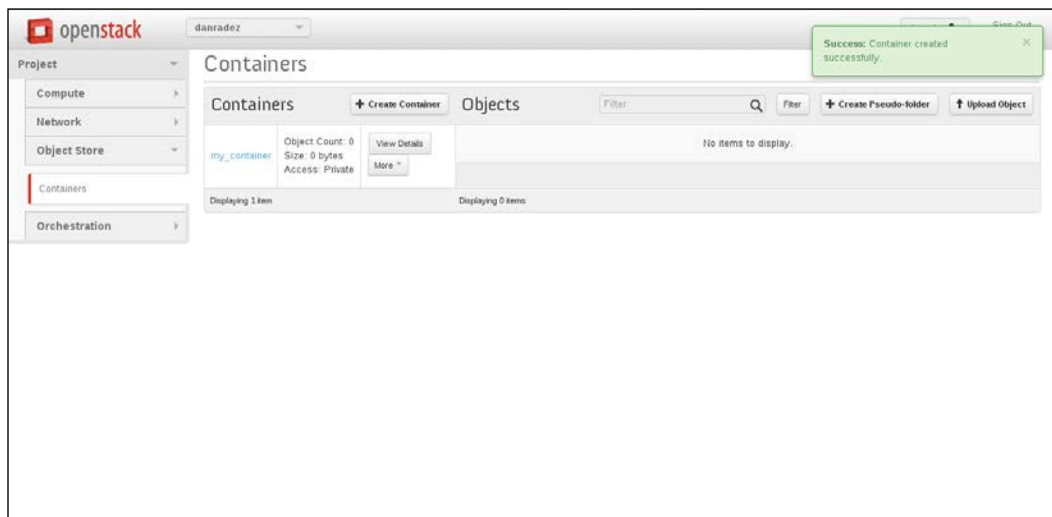
Note how the file gets named when you use the absolute file path. It is important here to understand that the way the object is named is not a directory structure. The filename just has a slash in it. Be aware that if you address a file in a subfolder when you upload it, the path to the file being uploaded will be included in the name of the object that is created to store the uploaded object.

Object file management in the web interface

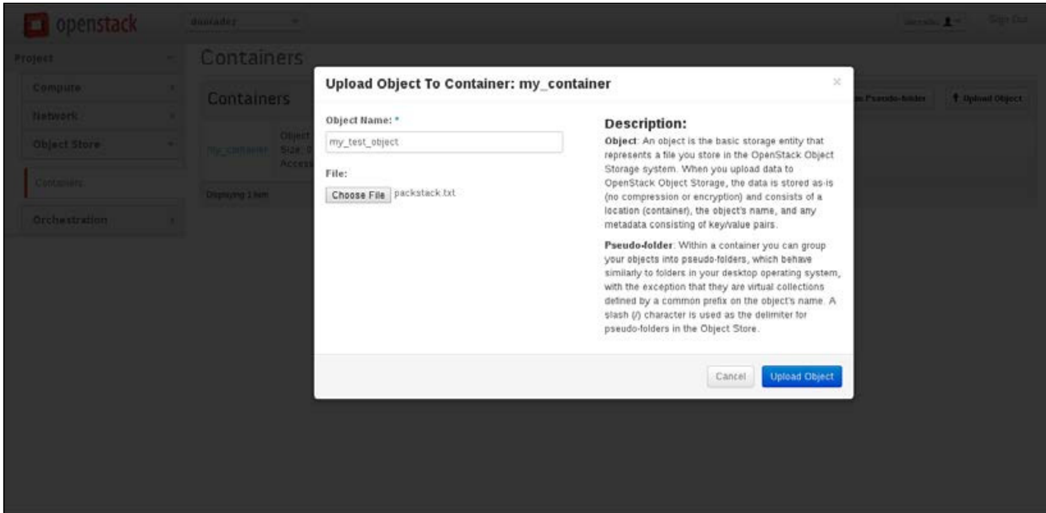
Now, let's take a look at managing containers and objects in the web interface. Once you have logged in, open the **Object Store** menu and select the **Containers** submenu. Click on the **Create Container** button. The following screenshot captures this step:



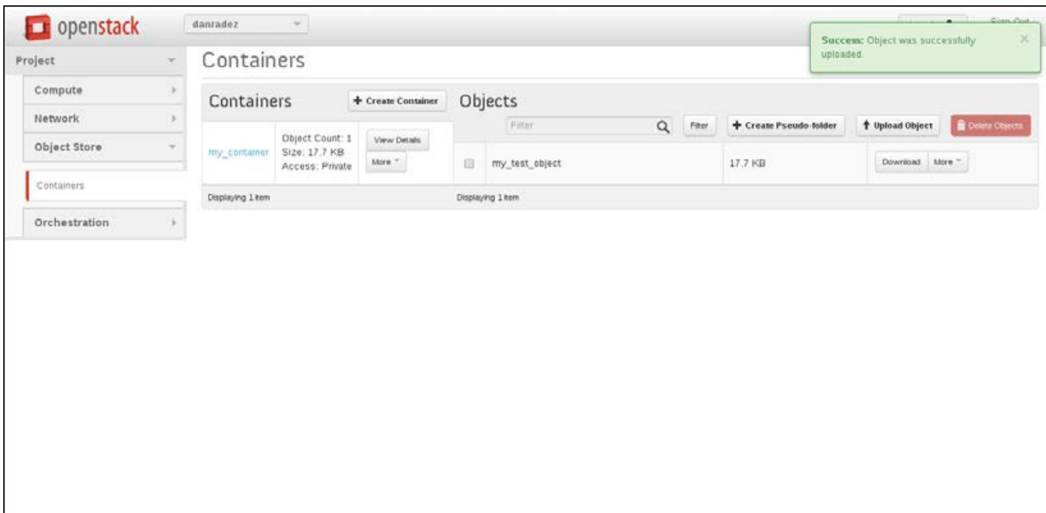
Now, you will be presented with the following screenshot:



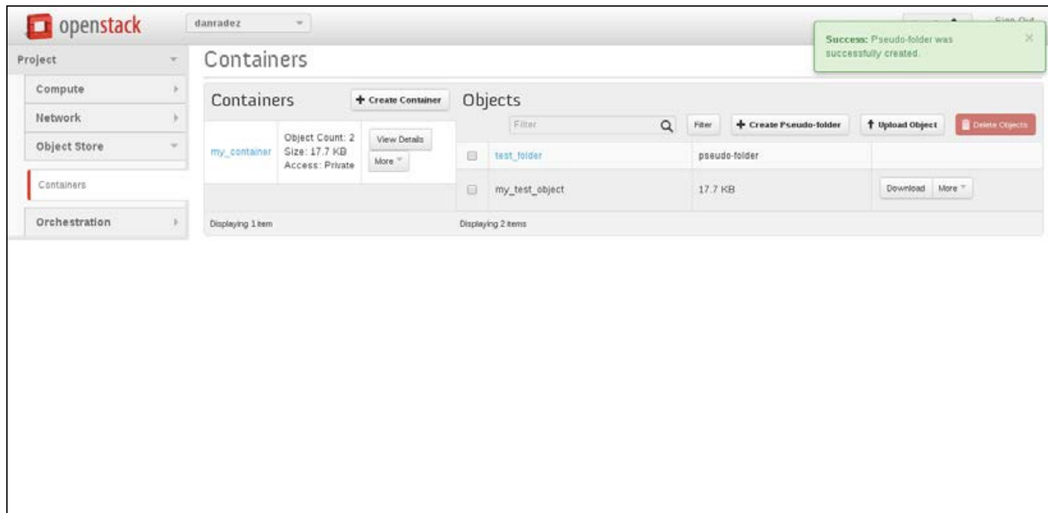
Once you have a container created, upload an object to the container, as shown in the following screenshot:



Both the filenames and object names are filled in when the web interface is used to upload a document. The object does not automatically take the filename as it does on the command line. These actions are amply illustrated in the following screenshot:



Earlier, we looked at using a path that would include a slash in the object name. In the web interface, this is called a pseudo-folder. The following screenshot shows us how this is done:



Once a pseudo-folder is created, it can be opened and any object uploaded would then be named with a prefix of the selected pseudo-folder's name.

Using object storage on an instance

Now that we have seen how to get files in and out of Swift, let's look at installing the necessary client libraries on an instance to be able to interact with object storage on the instance. We will need to have the Swift client installed so that the same Swift commands that were run on the control node's command line can be executed on an instance. In this cluster, Swift is using Keystone for authentication. Swift can also use other authentication. It has a built-in authentication system and can also be used with other common authentication systems. Since Swift is configured to use Keystone for authentication, the Keystone client will also need to be installed. Let's install those clients now:

```
instance# yum install -y openstack-swiftclient openstack-keystoneclient
```

Once those are installed, you will need to create a `keystonerc` file to source. Here's how we go about this:

```
instance# cat > ~/keystonerc_danradez << EOF
export OS_USERNAME=danradez
```

```
export OS_TENANT_NAME=danradez
export OS_PASSWORD=password
export OS_AUTH_URL=http://192.168.122.101:5000/v2.0/
export PS1='[\u@\h \W(keystone_danradez)]\$ '
EOF
```

Now, execute the Swift commands to list containers and objects, and create and upload them using the same commands as was previously done from the control node.

Ring files

Ring files are a kind of catalog of the files that are stored within the Swift storage engine and where within the storage cluster they are stored. As content is written to the Swift object storage cluster, these ring files are updated across the storage cluster and on the proxy server. When alternative storage backends are used in place of the Swift object storage engine, they mock the ring file system that the proxy expects and map this ring file system to its storage engine. Because the Swift proxy service always expects a set of ring files to operate, it is important to know how these are generated and installed. Learning how to set up ring files for the Swift object storage engine will teach you the basics that will translate into object storage backed by alternative storage engines.

There are various ring files that must all be generated and copied to each of the servers that will use the ring files. These include the account, container, and object rings. To generate these rings, each of the devices on each of the storage servers need to be added to the ring files, and the files need to be rebalanced before they are copied. In our example installation in *Chapter 2*, *RDO Installation*, Packstack did the setting up for us. This example is intended to give you an idea of how ring files are generated for future interactions you may have with generating these ring files.

Creating ring files

Let's use an example architecture that will include three storage nodes and one device on each of the nodes. If the IP addresses `192.168.123.11`, `192.168.123.12`, and `192.168.123.13` were the storage nodes and if each of them had a partition on a second drive named `sdb1`, then these three devices across the servers would be added to a new set of ring files using the `swift-ring-builder` command. First, create the ring files; this can be done on any of the storage nodes, and then the ring files will be copied to the other nodes. Here's how we use the `swift-ring-builder` command:

```
storage-node$ swift-ring-builder account.builder create 12 3 24
storage-node$ swift-ring-builder container.builder create 12 3 24
storage-node$ swift-ring-builder object.builder create 12 3 24
```

Here, you see a create command for each of the types of ring files. The three numbers behind the create command are the part power (`12`), replica count (`3`), and minimum part hours (`24`). There are partitions created within the Swift storage engine that help Swift to distribute the storage properly. The number of partitions is equal to 2 raised to the part power (12). In this example, 2^12 means that 4,096 partitions will be created in each ring. The replica count is how many copies of each item will be stored; it is recommended that you use three here. The minimum part hours is the minimum number of hours before a partition can be moved in succession. Swift recommends 24 as a good value for the minimum part hours. For more information on these values, search the Internet for *Swift preparing the rings* and you should find the official Swift documentation that goes into greater detail on how to choose values for these properties. Next, add the devices to the rings using the Swift ring builder command again. Here's how we go about it:

```
storage-node$ swift-ring-builder account.builder add z1-
192.168.123.11:6002/sdb1 100
storage-node$ swift-ring-builder account.builder add z1-
192.168.123.12:6002/sdb1 100
storage-node$ swift-ring-builder account.builder add z1-
192.168.123.13:6002/sdb1 100
storage-node$ swift-ring-builder account.builder rebalance
storage-node$ swift-ring-builder container.builder add z1-
192.168.123.11:6001/sdb1 100
storage-node$ swift-ring-builder container.builder add z1-
192.168.123.12:6001/sdb1 100
storage-node$ swift-ring-builder container.builder add z1-
192.168.123.13:6001/sdb1 100
storage-node$ swift-ring-builder container.builder rebalance
storage-node$ swift-ring-builder object.builder add z1-
192.168.123.11:6000/sdb1 100
storage-node$ swift-ring-builder object.builder add z1-
192.168.123.12:6000/sdb1 100
storage-node$ swift-ring-builder object.builder add
192.168.123.13:6000/sdb1 100
storage-node$ swift-ring-builder object.builder rebalance
```

In this example, there is only one zone used that is referenced by `z1-` as a prefix to each of the IP addresses. It is recommended that a minimum of five zones are used to avoid conflicts within zones on the same IP address. Behind each of the addresses and device names in these commands, there is a weight. This weight helps Swift to determine how many partitions are placed on the device relative to the rest of the devices in the cluster. It is recommended that you start with 100 times the number of terabytes on the drive. When this is complete, you should be able to list the files in your `/etc/swift` directory and see a `.ring.gz` file for each of the ring types. These are the files that need to be copied to each of the storage nodes and the Swift proxy server node before services are started. Also, ensure that these files on each of the nodes are owned by `root:swift`.

Once these files are in place across the servers, the services can be started. Note that Swift's configuration files do not reference the other servers in the storage cluster. The servers reference each other by way of ring files. These ring files can also be updated to help Swift work around hardware failures in the storage cluster. Just make sure that any changes that are made are copied across the cluster so that the ring files match on all the nodes.

Summary

In this chapter, we looked at using the Swift object storage and how to generate the ring files that the Swift storage engine uses to manage its storage. Now that we have covered the storage components in OpenStack, let's take a look at the Telemetry component that OpenStack uses to measure the usage of resources across the cluster.

9
Telemetry

Ceilometer is the telemetry component in OpenStack. While all the other components in OpenStack are busy managing virtual resources, Ceilometer keeps a watchful eye over them and measures the usage of resources, what resources are being used, and how they are being used within the cluster. In this chapter, we are going take a look at what is being measured and how to query the telemetry data. Then, we will use gnuplot to plot some of the data on a graph.

Understanding the data store

Before we start exploring Ceilometer, it is important to know that, by default, Ceilometer uses MongoDB to store all of its telemetry data. This data store can grow very rapidly and can use excess space. It is in your interest to keep Mongo's data store separate from the root partition of your control node so that the telemetry data does not fill up your control node's root disk. OpenStack has a horrible time functioning without a disk to write to. Mongo's data store is `/var/lib/mongodb/` by default. A simple way to be sure that the node's root disk doesn't fill up would be to mount another partition, logical volume, or some other external storage to `/var/lib/mongodb/`. If there isn't any important data in Ceilometer, you can even stop the MongoDB service, delete the contents of the data store directory, mount the new storage, ensure the ownership is correct, and restart the central and collector services of both the MongoDB and Ceilometer APIs. The files that were deleted will be recreated as an empty Mongo database for Ceilometer to start dumping data into again.

Definitions of Ceilometer's configuration terms

As resources are being managed within the OpenStack cluster, there are certain types of things that are being measured by Ceilometer. These types of things are called meters in Ceilometer. Each of these types of measurements gathers samples. Samples are single measurements or data points of a certain meter. The definition of how often to sample a meter is called a pipeline. Once enough samples are collected, they can be aggregated into statistics. Ceilometer statistics show a collection of samples over time for a particular meter. Ceilometer also has the ability to set alarms that will monitor statistics and is able to respond to matching criteria.

Pipelines

Pipelines are something that you shouldn't have to spend time configuring. Ceilometer has a collection of predefined pipelines that should suit most of your needs. If you end up needing a custom pipeline, it would be done in the `/etc/ceilometer/pipeline.yaml` configuration file. Take a look at this file if you would like to familiarize yourself with the pipeline configuration. We are not going to spend any more time beyond mentioning pipelines here.

Meters

Meters are the types of data being measured and the resources being measured. To see which meters have been collected and which resources have metered data, simply list the meters:

```
control# ceilometer meter-list
```

Only meters that have collected data are included in this list. If a meter is absent, then there haven't been any events to generate data for the absent meter. If no meters are listed, then the Ceilometer services are not properly collecting and reporting data. There is also a command that will show you which resources have collected telemetry data:

```
control# ceilometer resource-list
```

After Ceilometer has been collecting data for an extended period of time, a very large meter-list could come back. This list can be filtered using the query argument:

```
control# ceilometer meter-list -q name=vcpus
```

As a non-privileged user, you will only see meters for your project; as the administrator, you will probably need to filter the meters to a specific project. To do this, use the query argument to filter the list of meters:

```
control# ceilometer meter-list -q project=<PROJECT_ID>
```

You can also pass multiple items to filter using a semicolon to delimit the filter items:

```
control# ceilometer meter-list -q project=<PROJECT_ID>;name=vcpus
```

Samples

Now that you can retrieve the meters that are collecting data, you can look at what data has been collected for those meters. Samples are a single measurement of a meter for a resource. To get a list of samples, you will need to provide the meter that you would like to list samples for:

```
control# ceilometer sample-list -m vcpus
```

As with the meter-list command, you can also filter the results with the query argument:

```
control# ceilometer sample-list -m vcpus -q resource_id=<INSTANCE_ID>
```

You can also filter certain fields to get a range of results; for example, samples within a limited time period can be returned by filtering on the timestamp field:

```
control# ceilometer sample-list -m vcpus -q
'resource_id=<INSTANCE_ID>;timestamp>2014-09-
27T07:30:00;timestamp<=2014-09-27T011:00:00'
```

Statistics

By listing the meters, we have looked at what is being measured, and by listing the samples, we have looked at the actual raw data that is being collected for the meters. The final aggregation of this data into something useful is called statistics in Ceilometer. As with samples, the statistics command requires you to provide the meter for which you would like to see statistics. Here's the statistics command:

```
control# ceilometer statistics -m vcpus
```

As with meters and samples, a query argument can be passed to filter the data:

```
control# ceilometer statistics -m vcpus -q 'timestamp>2014-09-
27T07:30:00;timestamp<=2014-09-27T011:00:00'
```

An additional argument that is available with statistics is the period argument. The period is the number of seconds into which samples can be grouped for the statistics generated. If you do not pass the period argument, you will get a single statistic returned to you with all the data for your meter and query. If you pass a period, you will get multiple statistics returned—one for each grouping of samples according to the period specified. For example, to get statistics for each 10-minute period within the timestamp range we have been using, the command would look like this:

```
control# ceilometer statistics -m vcpus -q 'timestamp>2014-09-27T07:30:00;timestamp<=2014-09-27T011:00:00' -p 600
```

Alarms

Alarms are a resource used mainly in conjunction with orchestration. We will look at alarms again when we look at orchestration in the next chapter. Alarms have to be created; they will not appear magically like the meters and samples we just looked at. Let's create an alarm that will watch for high CPU usage on a particular instance:

```
control# ceilometer alarm-threshold-create --name cpu_alarm --description 'cpu usage is high!' --meter-name cpu --threshold 80.0 --comparison-operator gt  --statistic avg --period 600 --evaluation-periods 3 --alarm-action 'log://' --query resource_id=<INSTANCE_ID>
```

This will create an alarm that watches the CPU on a specific instance and logs to the file if the instance's CPU usage is above 80 percent over three checks, 10 minutes apart. Use the list command to see the alarm just created:

```
control# ceilometer alarm-list
```

Get the alarm's ID from the list and check its history:

```
control# ceilometer alarm-history -a <ALARM_ID>
```

There is probably only a creation event; other events will show up in the history as they are triggered, though. Finally, it may be necessary to enable or disable the alarm for some reason. There is an enabled flag on alarms that you can use to turn it on and off:

```
control# ceilometer alarm-update --enabled False -a <ALARM_ID>
```

Updates to the alarm using the alarm-update command are logged as history. After you have updated the alarm, look at the history again, and you will see an event for the update you made. This applies to any alarm property that is updated.

Graphing the data

Up until now, we have just seen data points flowing through our screen that may or may not be very useful to us. Wouldn't it be nice to make something visual to help display this data? There are plenty of options that could be used to plot this data. As an example, let's take a quick look at gnuplot, which is a command-line program that is packaged with most modern Linux distributions. This book has been using Fedora; to install gnuplot, simply `yum install` it:

```
control# yum install -y gnuplot
```

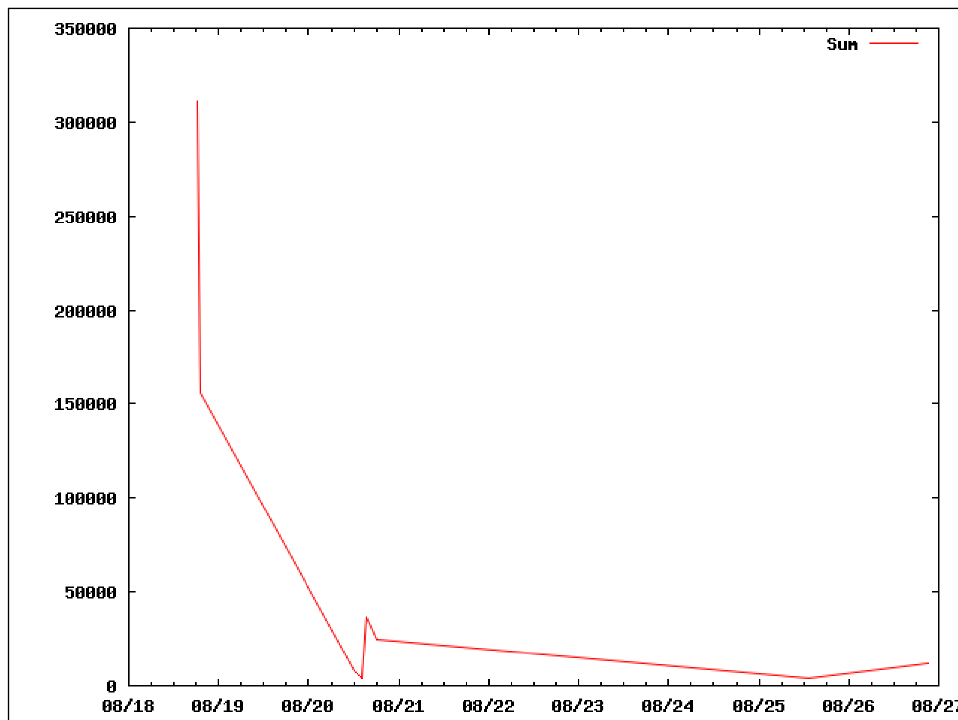
There are options that need to be fed into `gnuplot` to tell it how to render the graph that it creates. Let's use a configuration file that will be passed to gnuplot. Put the following content into a file. I'm going to name mine `memory.cfg` because I will plot the memory usage that's already been aggregated by the Ceilometer statistics command:

```
#memory.conf
set terminal png truecolor
set output "memory.png"
set autoscale
set xdata time
set timefmt '%Y-%m-%dT%H:%M:%S'
set style data lines
plot '<cat' using 2:7 title "Sum"
```

The `set terminal` line tells gnuplot to generate a `.png` image. `Set output` sets the filename to write to. `Autoscale` turns on autoscaling. The `xdata` and `timefmt` lines define the format to read the time from. The `set style` line tells gnuplot to make a line graph. Finally, the plotline `<cat` reads from standard input. `2:7` tells us to use the second column for the *x* axis, the seventh column for the *y* axis, and the title `"Sum"` sets the title for the line that will be drawn. Next, let's execute the string of commands that will clean Ceilometer's output and give it to gnuplot:

```
control# ceilometer statistics -m memory -q project=<PROJECT_ID> -p
3600 | tail -n +4 | head -n -1 | tr -d '|' | tr -s ' ' | gnuplot
memory.cfg
```

The Ceilometer statistics command uses the memory meter for the project of the ID that is passed and groups the memory measurements into hour-long periods. The Ceilometer output is piped to the tail, which strips off the rows that display the column headers. The head command strips off the last line, which is just another line as the one that was included in the header that borders the bottom of the data. The first `tr` command deletes all the pipes that are delimiting the columns, and the second `tr` command squashes all the spaces into single spaces. What we end up with is no pipe delimiting, no column headers, and no special output formatting. This is just the raw data with single-spaced delimiting. There may be a way to make Ceilometer do this automatically for us. Finally, the cleaned-up data is passed to gnuplot, which reads our configuration file and generates `.png`. Here is an image I generated with some sample data:



As a second example, let's plot two lines. This can't be achieved by piping data directly to gnuplot. We will have to dump the data into a data file so that the data can be read twice, once for each line. We will use `vcpus` this time instead of memory and a period of 30 minutes. Also make a copy of the `cfg` file so that it can be modified:

```
control# ceilometer statistics -m vcpus -q project=<PROJECT_ID> -p
1800 | tail -n +4 | head -n -1 | tr -d '|' | tr -s ' ' > vcpus.txt
control# cp memory.cfg vcpus.cfg
```

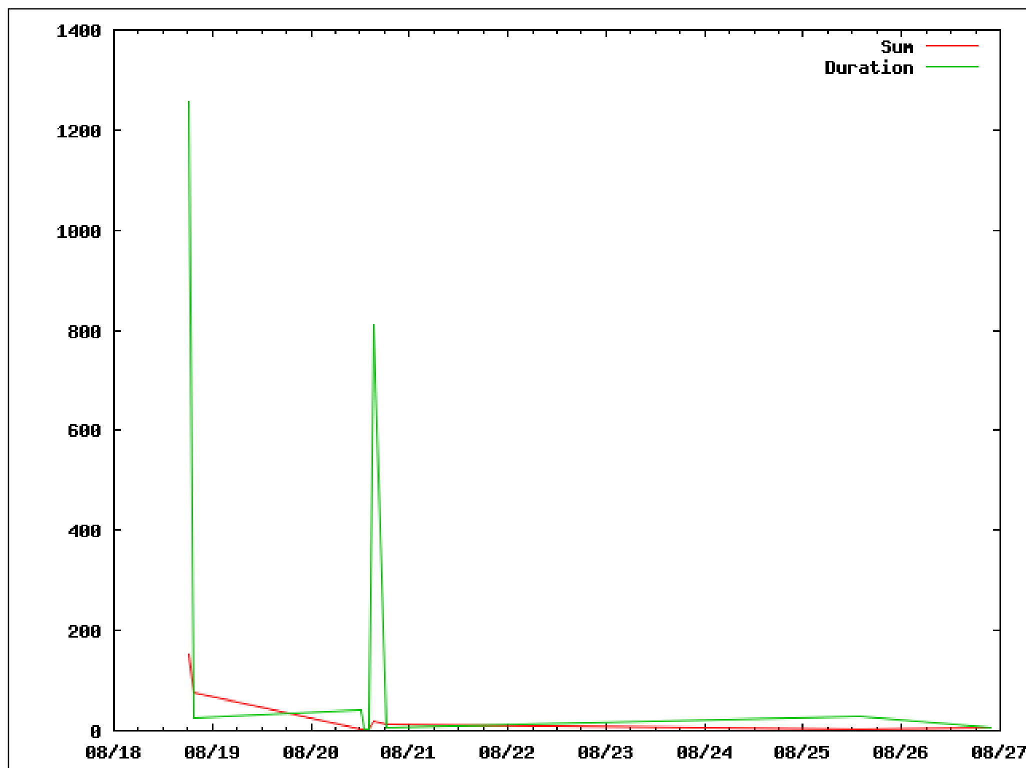
Next, update the `vcpus.cfg` file to use the `vcpus.txt` file and to plot two lines instead of one. To do this, update the output line to a new filename so that you don't overwrite your `memory.png` file and update the plotline. The new file's content will look like this:

```
#vcpus.conf
set terminal png truecolor
set output "vcpus.png"
set autoscale
set xdata time
set timefmt '%Y-%m-%dT%H:%M:%S'
set style data lines
plot 'vcpus.txt' using 2:7 title "Sum", 'vcpus.txt' using 2:9
title "duration"
```

Once you have the new `cfg` file and the data file, run `gnuplot`:

`control# gnuplot vcpus.cfg`

This will generate a `vcps.png` file. Here's one I generated with sample data:



Another example that could be worked with is to dump the memory data into one file and have one line plotted from the memory data and the second line plotted from the `vcpus` data. As illustrated, `gnuplot` can be a powerful tool. These examples show what can be done with the data that Ceilometer produces. They show the only possible tool to consume and plot the data.

Summary

In this chapter, we looked at how to view, aggregate, and plot telemetry data using Ceilometer. This data is useful to monitor the health of a set of instances, a billing client, and so on. As mentioned in this chapter, Ceilometer's alarms are a useful resource for the orchestration tool in OpenStack. Next, we will look at cloud orchestration using the OpenStack component named Heat.

10
Orchestration

In the previous chapter, we looked at Ceilometer and used telemetry in OpenStack. In this chapter, we will take a look at orchestration using OpenStack's Heat component. We will take a look at what orchestration is and how to write a template for Heat. Then, we will use the template to launch a Heat stack.

About orchestration

In *Chapter 6*, *Instance Management*, we used Nova to launch instances in OpenStack. This example walked through launching a single instance or, if the instance count was increased, multiple instances with the same configuration. What if a collection of instances needed to be launched that required each of them to have a different configuration or they all needed to know about each other as part of their post-boot configuration? For example, maybe a different Glance image is needed for each instance or a different flavor is needed for the different roles within this collection of instances. It's even a possible requirement to control the order in which the instances are spawned to make sure that they are available in a specific order for post-boot configuration purposes. Enter orchestration. With OpenStack's orchestration component, Heat, all of these requirements and more can be met with Heat's capabilities.

Writing templates

The two core concepts to get started with Heat are stacks and templates. A stack is a collection of resources related to one another and launched by way of a template. A template is a text document definition of a stack. To launch a Heat stack, a Heat template is launched. Let's look at both of these in more depth, starting with templates.

Before we can launch a stack, we need a template that will define the stack. There are two formats of template that you can use to launch a stack in Heat. One is the **AWS CloudFormation** template format. If you have ever used CloudFormation in AWS, then you would be familiar with this template format. Heat templates are very similar to those used within **Amazon Web Services** (**AWS**), and add additional capabilities within OpenStack. The second format is HOT, which stands for Heat Orchestration Template. HOT is a native Heat template format that is written in the YAML Ain't Markup Language syntax. For more examples of both, visit the Heat-templates github repository and browse through the collection of example scripts. The examples used in this chapter were pulled from `https://github.com/openstack/heat-templates`.

The AWS CloudFormation format

Let's pull an example from the Heat-templates repository to gain some familiarity with the AWS CloudFormation format. This is a large document, so the entire content will not be provided here. I'll reference the document from top to bottom. The document in its entirety is available at the following link and is available in the code resources provided with this book:

`https://github.com/openstack/heat-templates/blob/master/cfn/F19/WordPress_NoKey.yaml`

Let's take a look at the configuration options used in this template:

- **HeatTemplateFormatVersion** is just for versioning so that Heat knows which syntax version is being used.
- **Description** is a description of what the template will launch. This template indicates it will launch a single-instance WordPress install.
- **Parameters** is a section that defines what information is needed for this template to be launched. You can see that each of the parameters is defined by its name first and then a set of parameters to help the end user enter the correct information in it. For the template we are looking at, there are InstanceType, which references the flavors in Nova, and the DBName, DBUsername, DBPassword, and DBRootPassword properties that will be used to configure the database that will be created.
- **Mappings** in this template only map flavors with images so that when you launch a specific flavor, you will get the associated image.

- The **resources** section defines the resources that will be created in OpenStack. In this template, security group rules and an instance are created. You can see the security group rules to allow ICMP, port 80 (HTTP), and port 22 (SSH) traffic. You can also see the configuration options that will be passed to the instance when it is booted. These include packages to be installed, services to be started, the image to be used to launch the instance, which references the mappings we just looked at, the security group for the instance to reside in, and the user data that cloud-init should execute.
- Finally, there is an **outputs** section. This is data passed back to Heat from the stack once it has been launched.

Next, let's take a look at the HOT format before we use a template to launch a stack.

The Heat Orchestration Template (HOT) format

Take a look at the hello world template in the HOT directory in the same github repository:

`https://github.com/openstack/heat-templates/blob/master/hot/hello_world.yaml`

The HOT format uses most of the same keywords that the AWS CloudFormation format uses. In the hello world template, you can see that almost all the same sections exist: heat_template_version, parameters, resources, and outputs. The configuration options for each of these sections look very similar; the main difference is that the HOT format is pure YAML and the AWS CloudFormation is a kind of YAML/JSON hybrid. The next step for us is to take these templates and launch a stack using them. We will use the HOT format to launch a stack next, so you will gain familiarity with this format in our practical application.

Launching a stack

Let's use the HOT hello world. The template can be passed to the Heat `stack-create` command as a local file, a URL to pull it from the network somewhere, or even as a Swift object if it was stored in Swift. I had to pull down a copy of the HOT we just looked at to remove a few lines from it to get it to work. Pull down a copy of the template to your local file system. A template can be validated with the `template-validate` command. The template as it is in github didn't validate for me:

```
control# heat template-validate -f hello_world.yaml
ERROR: Unknown Property admin_pass
```

I edited the file to remove the references to `admin_pass` from the server resource and from the parameter definitions in the template. The `admin_pass` property is not really needed since a key pair is being passed into the instance when it's launched, so we will be fine if we remove it. Here's what the template looked like after editing it:

```
#
# This is a hello world HOT template just defining a single
# compute server.
#
heat_template_version: 2013-05-23

description: >
  Hello world HOT template that just defines a single server.
  Contains just base features to verify base HOT support.

parameters:
  key_name:
    type: string
    description: Name of an existing key pair to use for the server
    constraints:
      - custom_constraint: nova.keypair
  flavor:
    type: string
    description: Flavor for the server to be created
    default: m1.small
    constraints:
      - custom_constraint: nova.flavor
  image:
    type: string
    description: Image ID or image name to use for the server
    constraints:
      - custom_constraint: glance.image
  db_port:
    type: number
    description: Database port number
    default: 50000
    constraints:
      - range: { min: 40000, max: 60000 }
        description: Port number must be between 40000 and 60000

resources:
  server:
    type: OS::Nova::Server
    properties:
```

```
        key_name: { get_param: key_name }
        image: { get_param: image }
        flavor: { get_param: flavor }
        user_data:
          str_replace:
            template: |
              #!/bin/bash
              echo db_port
            params:
              db_port: { get_param: db_port }

  outputs:
    server_networks:
      description: The networks of the deployed server
      value: { get_attr: [server, networks] }
```

After you pull out the references to `admin_pass`, it will validate properly, and the command will return a JSON structure with all the template's data in it:

```
control# heat template-validate -f hello_world.yaml
```

Next, pass into the `stack-create` command a stack name and all the parameters that the template requires to launch:

```
control# heat stack-create -f hello_world.yaml -P key_name=danradez -
P image=Fedora -P admin_pass=Abadpass ""My First Stack""
```

This command will launch a stack named My First Stack from the template that you just downloaded and edited. Once a stack has been launched, you can keep track of the stack's progress and details using Heat's `stack-list` command and the `stack-show` command. Further, you can list the resources associated with the stack with the `resource-list` command, and you can see the individual resources through the other OpenStack components using the appropriate command associated with those resources. In this example, the only resource created was an instance through Nova, so use the `nova list` command to see the instance that the stack created. A stack also has a set of events. Those events can be listed with the `event-list` command. The details of resources and events can be seen with their respective show commands:

```
control# heat stack-list
control# heat stack-show {STACK_ID}
control# heat resource-list {STACK_ID}
control# heat resource-show {RESOURCE_ID}
control# nova list
control# heat event-list {STACK_ID}
control# heat event-show {EVENT_ID}
```

Note here that the resources that are created through Heat can be managed independently of Heat. The instance that was created by way of the hello world stack could be deleted directly through Nova. Deleting the instance will not delete the stack, but deleting the stack will delete all the resources that are associated with the stack.

Autoscaling instances with Heat

In *Chapter 9*, *Telemetry*, Ceilometer alarms were introduced. These were monitoring objects that were able to trigger external actions based on a certain criterion being met for a predetermined set of iterations. Heat's autoscaling is the primary use case for this functionality. Using Heat's autoscaling, it is possible to monitor a set of instances and add or subtract instances to meet load demands. In the same GitHub repository that the previous examples were taken from, there is an autoscaling example. For the autoscaling example to work, you will need to grab two templates:

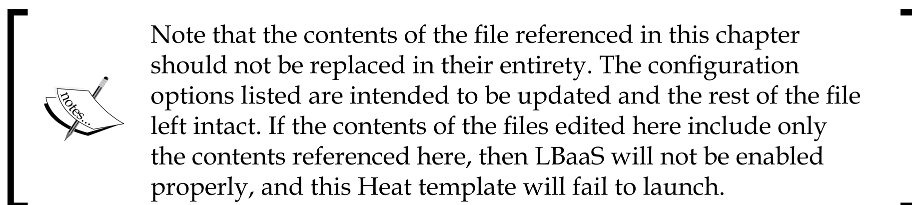
- `https://github.com/openstack/heat-templates/blob/master/hot/autoscaling.yaml`
- `https://github.com/openstack/heat-templates/blob/master/hot/lb_server.yaml`

These templates are written to launch a single database instance and to add and remove web server instances respective to the load put on the WordPress stack running on the web servers.

LBaaS setup

Before we get to walking through these templates, we need to enable the **Load Balancer as a Service** (**LBaaS**) functionality of Neutron. Packstack does not configure it when it installs Neutron. There are a couple of configuration files to be updated and a couple of services to restart. First off, ensure that HAProxy is installed on the network node:

```
network# yum install -y haproxy
```



Note that the contents of the file referenced in this chapter should not be replaced in their entirety. The configuration options listed are intended to be updated and the rest of the file left intact. If the contents of the files edited here include only the contents referenced here, then LBaaS will not be enabled properly, and this Heat template will fail to launch.

Next, edit `/etc/neutron/neutron.conf` on the network nodes and add the value `lbaas` to the `service_plugins` configuration option. If there are already values, leave them there and add lbaas to the comma-delimited list. Mine was commented out with out a value, so I just added lbaas as the only value to this configuration:

```
service_plugins = lbaasLastly edit /etc/neutron/lbaas_agent.ini on
the network node and make sure that the device_driver options is
set to HAProxy, the interface_driver is set to OVS and that the
[haproxy] user_group is set to nobody.
[DEFAULT]
device_driver =
neutron.services.loadbalancer.drivers.haproxy.namespace_driver.Hap
roxyNSDriver
interface_driver =
neutron.agent.linux.interface.OVSInterfaceDriver
[haproxy]
user_group = nobody
```

Finally, restart the Neutron server and lbaas services on the network node:

```
network# service neutron-server restart
network# service neutron-lbaas-agent restart
```

Now that the lbaas service is enabled, let's take a look at the `autoscaling.yaml` file. Here are the contents; there is more explanation after the contents of the file:

```
heat_template_version: 2013-05-23
description: AutoScaling Wordpress
parameters:
  image:
    type: string
    description: Image used for servers
  key:
    type: string
    description: SSH key to connect to the servers
  flavor:
    type: string
    description: flavor used by the web servers
  database_flavor:
    type: string
    description: flavor used by the db server
  subnet_id:
    type: string
    description: subnet on which the load balancer will be located
  database_name:
    type: string
```

```
      description: Name of the wordpress DB
      default: wordpress
    database_user:
      type: string
      description: Name of the wordpress user
      default: wordpress
    external_network_id:
      type: string
      description: UUID of a Neutron external network
  resources:
    database_password:
      type: OS::Heat::RandomString
    database_root_password:
      type: OS::Heat::RandomString
    db:
      type: OS::Nova::Server
      properties:
        flavor: {get_param: database_flavor}
        image: {get_param: image}
        key_name: {get_param: key}
        user_data_format: RAW
        user_data:
          str_replace:
            template: |
              #!/bin/bash -v
              yum -y install mariadb mariadb-server
              systemctl enable mariadb.service
              systemctl start mariadb.service
              mysqladmin -u root password $db_rootpassword
              cat << EOF | mysql -u root --password=$db_rootpassword
              CREATE DATABASE $db_name;
              GRANT ALL PRIVILEGES ON $db_name.* TO ""$db_user""@""%""
              IDENTIFIED BY ""$db_password"";
              FLUSH PRIVILEGES;
              EXIT
              EOF
            params:
              $db_rootpassword: {get_attr: [database_root_password,
  value]}
              $db_name: {get_param: database_name}
              $db_user: {get_param: database_user}
              $db_password: {get_attr: [database_password, value]}
    web_server_group:
      type: OS::Heat::AutoScalingGroup
```

```
    properties:
      min_size: 1
      max_size: 3
      resource:
        type: lb_server.yaml
        properties:
          flavor: {get_param: flavor}
          image: {get_param: image}
          key_name: {get_param: key}
          pool_id: {get_resource: pool}
          metadata: {""metering.stack"": {get_param:
""OS::stack_id""}}
          user_data:
            str_replace:
              template: |
                #!/bin/bash -v
                yum -y install httpd wordpress
                systemctl enable httpd.service
                systemctl start httpd.service
                setsebool -P httpd_can_network_connect_db=1

                sed -i ""/Deny from All/d""
/etc/httpd/conf.d/wordpress.conf
                sed -i ""s/Require local/Require all granted/""
/etc/httpd/conf.d/wordpress.conf
                sed -i s/database_name_here/$db_name/
/etc/wordpress/wp-config.php
                sed -i s/username_here/$db_user/
/etc/wordpress/wp-config.php
                sed -i s/password_here/$db_password/
/etc/wordpress/wp-config.php
                sed -i s/localhost/$db_host/ /etc/wordpress/wp-
config.php

                systemctl restart httpd.service
              params:
                $db_name: {get_param: database_name}
                $db_user: {get_param: database_user}
                $db_password: {get_attr: [database_password,
value]}
                $db_host: {get_attr: [db, first_address]}
  web_server_scaleup_policy:
    type: OS::Heat::ScalingPolicy
    properties:
      adjustment_type: change_in_capacity
```

```
        auto_scaling_group_id: {get_resource: web_server_group}
        cooldown: 60
        scaling_adjustment: 1
    web_server_scaledown_policy:
      type: OS::Heat::ScalingPolicy
      properties:
        adjustment_type: change_in_capacity
        auto_scaling_group_id: {get_resource: web_server_group}
        cooldown: 60
        scaling_adjustment: -1
    cpu_alarm_high:
      type: OS::Ceilometer::Alarm
      properties:
        description: Scale-up if the average CPU > 50% for 1 minute
        meter_name: cpu_util
        statistic: avg
        period: 60
        evaluation_periods: 1
        threshold: 50
        alarm_actions:
          - {get_attr: [web_server_scaleup_policy, alarm_url]}
        matching_metadata: {''metadata.user_metadata.stack'':
  {get_param: ""OS::stack_id""}}
        comparison_operator: gt
    cpu_alarm_low:
      type: OS::Ceilometer::Alarm
      properties:
        description: Scale-down if the average CPU < 15% for 10
  minutes
        meter_name: cpu_util
        statistic: avg
        period: 600
        evaluation_periods: 1
        threshold: 15
        alarm_actions:
          - {get_attr: [web_server_scaledown_policy, alarm_url]}
        matching_metadata: {''metadata.user_metadata.stack'':
  {get_param: ""OS::stack_id""}}
        comparison_operator: lt
    monitor:
      type: OS::Neutron::HealthMonitor
      properties:
        type: TCP
        delay: 5
```

```
      max_retries: 5
      timeout: 5
  pool:
    type: OS::Neutron::Pool
    properties:
      protocol: HTTP
      monitors: [{get_resource: monitor}]
      subnet_id: {get_param: subnet_id}
      lb_method: ROUND_ROBIN
      vip:
        protocol_port: 80
  lb:
    type: OS::Neutron::LoadBalancer
    properties:
      protocol_port: 80
      pool_id: {get_resource: pool}

  # assign a floating ip address to the load balancer
  # pool.
  lb_floating:
    type: ""OS::Neutron::FloatingIP""
    properties:
      floating_network_id: {get_param: external_network_id}
      port_id: {get_attr: [pool, vip, port_id]}

outputs:
  scale_up_url:
    description: >
      This URL is the webhook to scale up the autoscaling group.
You can invoke the scale-up operation by doing an HTTP POST to
this URL; no body nor extra headers are needed.
    value: {get_attr: [web_server_scaleup_policy, alarm_url]}
  scale_dn_url:
    description: >
      This URL is the webhook to scale down the autoscaling group.
You can invoke the scale-down operation by doing an HTTP POST to
this URL; no body nor extra headers are needed.
    value: {get_attr: [web_server_scaledown_policy, alarm_url]}
  pool_ip_address:
    value: {get_attr: [pool, vip, address]}
    description: The IP address of the load balancing pool
  website_url:
    value:
      str_replace:
        template: http://host/wordpress/
```

```
        params:
          host: { get_attr: [lb_floating, floating_ip_address] }
    description: >
      This URL is the ""external"" URL that can be used to access
the Wordpress site.
  ceilometer_query:
    value:
      str_replace:
        template: >
          ceilometer statistics -m cpu_util
          -q metadata.user_metadata.stack=stackval -p 600 -a avg
        params:
          stackval: { get_param: ""OS::stack_id"" }
    description: >
```

This is a Ceilometer query for statistics on the `cpu_util` meter samples about `OS::Nova::Server` instances in this stack. The `-q` parameter selects samples according to the subject's metadata. When a VM's metadata includes an item of the form `metering.X=Y`, the corresponding Ceilometer resource has a metadata item of the form `user_metadata.X=Y` and samples about resources so tagged can be queried with a Ceilometer query term of the form `metadata.user_metadata.X=Y`. In this case the nested stacks give their VMs metadata that is passed as a nested stack parameter, and this stack passes a metadata of the form `metering.stack=Y`, where Y is this stack's ID.

You will see that the parameters collect the information necessary to dynamically launch the instances, attach them to networks, and create a database name and user to set up the database. The first three resources in the resource definitions include the database server itself and randomly generated passwords for the database users. The next resource is an auto-scaling group. The group is of the AutoScalingGroup type, and the resource defined in this group is of the `lb_server.yaml` type. This refers to the other yaml file. Let's quickly look at this template:

```
heat_template_version: 2013-05-23
description: A load-balancer server
parameters:
  image:
    type: string
    description: Image used for servers
  key_name:
    type: string
    description: SSH key to connect to the servers
  flavor:
    type: string
    description: flavor used by the servers
  pool_id:
```

```
      type: string
      description: Pool to contact
    user_data:
      type: string
      description: Server user_data
    metadata:
      type: json
  resources:
    server:
      type: OS::Nova::Server
      properties:
        flavor: {get_param: flavor}
        image: {get_param: image}
        key_name: {get_param: key_name}
        metadata: {get_param: metadata}
        user_data: {get_param: user_data}
        user_data_format: RAW
    member:
      type: OS::Neutron::PoolMember
      properties:
        pool_id: {get_param: pool_id}
        address: {get_attr: [server, first_address]}
        protocol_port: 80
```

The `lb_server.yaml` template is a fairly basic server definition to launch a single instance using Heat. The extra definitions to note are the `pool_id` parameter and the `Neutron PoolMember` resource. These associate the servers that are launched with this template with the LBaaS pool resource created in the `autoscaling.yaml` template. This also shows an example of how Heat templates can reference each other. Let's jump back to the `autoscaling.yaml` template now.

The next two resources defined after the AutoScalingGroup resource are the Heat policies that are used to define what to do when scaling up or scaling down. The next two resources are the Ceilometer alarms that trigger the Heat policies to scale up or down accordingly when the CPU usage is too high or too low for the number of instances that are currently running. The last four resources define a load balancer, an IP address for the load balancer, a monitor for the load balancer, and a pool to add servers to for the load balancer to balance the load.

Lastly, the `autoscale.yaml` template defines a set of outputs to get URLs and the pool IP address or that the heat stack can be used.

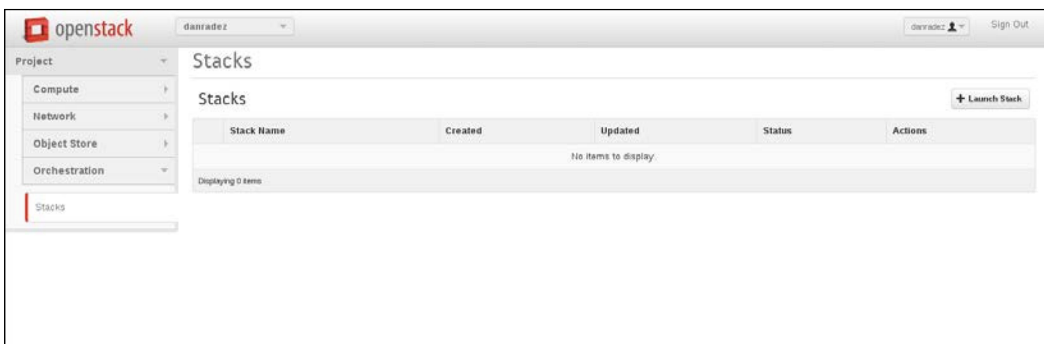
Now that we've walked through these templates, let's launch the autoscale template. You will need to pass in a glance image ID to launch all the instances off, the ID of your internal subnet and your external network, a key pair's name, and Nova flavor names for the database and the web server instances. The `stack-create` command should be executed as the admin user. The policies in Ceilometer require admin access. They could be created ahead of time and provided to end users if it was necessary for non-administrative users to launch auto-scaling stacks. For our demonstrations here, just use the admin user. The command will look something like this.

```
heat stack-create -f autoscaling.yaml -P database_flavor=m1.small -P
subnet_id={INTERNAL_SUBNET_ID} -P external_network_id={EXT_NET_ID} -P
image={GLANCE_IMAGE_ID} -P key=danradez -P flavor=m1.small autoscale_me
```

Once the stack launches, you can use the stack, resource, and event commands to list and show information about the stack, monitor its progress, and troubleshoot any errors that might be encountered. This stack is now ready to scale automatically using the resources Heat has put into place to monitor the set of resources created through this stack. If you were to put a load on the web service instance enough to trigger the scale-up alarm, another instance would spawn. You can also accomplish this via POST to the scale-up URL listed in the outputs of the auto-scaling template. Similarly, reducing the load to trigger the scale-down alarm or a POST to the scale-down URL in the outputs section of the template would reduce the number of instances in the web server pool.

Web interface

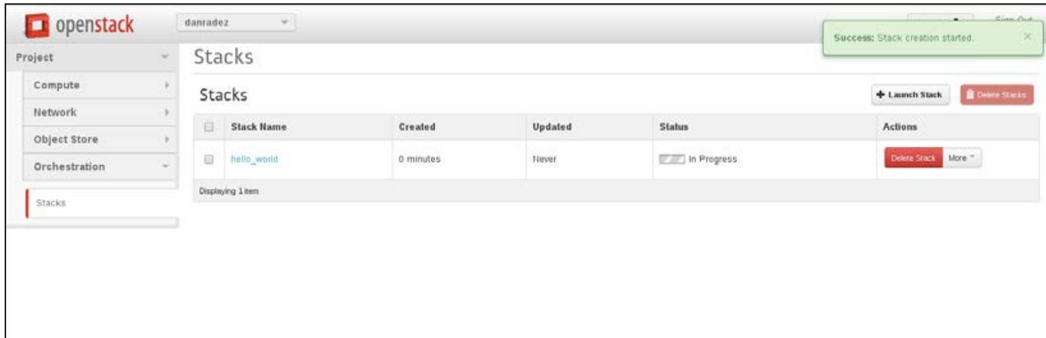
Now that we've looked at Heat on the command line and explored some of its functionality, let's take a look at the dashboard and the support available for Heat in the dashboard web interface. Log in to the dashboard, find the **Orchestration** menu, and select the **Stacks** menu option. The following screenshot captures the dashboard:



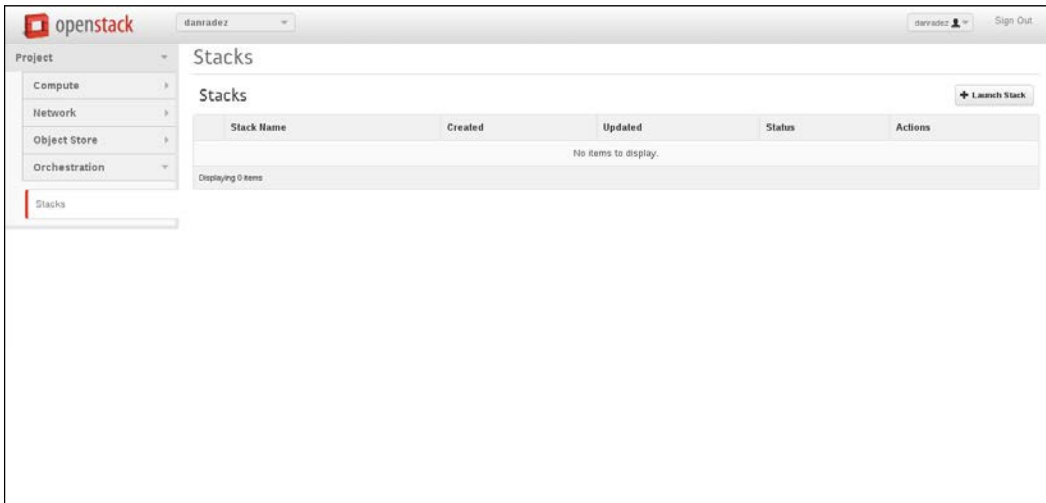
To launch a new stack, click on the **Launch Stack** button in the top-right corner:

To launch your stack, you have the same options to pull your template from. I've chosen the same `hello_world.yaml` file used earlier.

Fill out the form to provide the parameters required to launch the stack. Then, click on the **Launch** button at the bottom of the form. You may have to scroll down to get to it. If the form validation fails, you will be notified as to what needs to be updated:



The web interface will auto-update itself until the stack reaches a complete or failed state. Click on the **Stack** to drill down and find out more information about the stack. There are tabs with identification details, a **Resources** list, and an **Event** list. One other tab available is the topology tab. This tab is more interesting with more resources; the hello world stack only has one item on it. Here's what the topology looked like after I launched the autoscaling template:



Each of the circles can be moused over to show what each of them are and what their status is. Their status also shows failure reasons, so if something fails, this is sometimes a convenient place to get a visual representation of what failed and what the error message from the failure is. The graph is also interactive in a drag-around kind of way, which makes for great eye candy.

Summary

In this chapter, we looked at the different kinds of Heat templates and how to launch Heat stacks from these templates. Heat stacks offer a new level of opportunity to launch instances and tie them all together into a useful and functional set of instances and complementary resources. This completes the set of OpenStack components that we are going to review in this book. In the upcoming chapters, we are going to look at how to architect an OpenStack cluster, how to monitor the cluster, and how to troubleshoot OpenStack infrastructure. In *Chapter 11*, *Scaling Horizontally*, we will start with the architecture of an OpenStack cluster and look at how to scale it.

11
Scaling Horizontally

One of the foundations of OpenStack is that it was built to run on generic commodity hardware and is intended to scale out horizontally very easily. Scaling horizontally means adding more servers to get the job done. Scaling vertically means getting larger, more specialized servers. Whether the servers you run have a handful of processors and a few gigabytes of RAM, or double digits of processors and RAM approaching or exceeding triple digits, OpenStack will run on your servers. Further, whatever assortment of servers of varying horsepower you have collected, they can all be joined into an OpenStack cluster to run the API services, service agents, and hypervisors within the cluster. The only hard requirement is that your processors have virtualization extensions built into them, which is pretty much a standard feature in most modern-day processors. In this chapter, we will look at the process of scaling an OpenStack cluster horizontally on the control, network, and compute layers. Then, we will discuss the concepts around making the cluster highly available.

Scaling compute nodes

The first and easiest way to scale an OpenStack cluster is to add compute power. One control node and one network node can support more than one compute node. Remember installing RDO in *Chapter 2*, *RDO Installation*? We have come a long way since then! In that example, only one compute node was installed. One control and one network node can support a large collection of compute nodes. The exact number that can be handled depends on the demand put on the cluster by its end users. It is probably safe to say that the capacity provided by one compute node probably isn't going to meet most use cases, so let's take a look at how to add additional compute nodes to our OpenStack installation.

Technically, there are only two OpenStack services plus the supporting networking infrastructure that need to be running for a new compute node to be joined into an OpenStack cluster and start sharing the computing workload. These two services are the Nova compute service and the Neutron Open vSwitch agent. In our example installation, the supporting networking infrastructure is Open vSwitch, so Open vSwitch is required. The Ceilometer compute agent should also be installed if the telemetry data is expected to be collected. As soon as the Nova and Neutron agents communicate over the message bus with the control tier, the node will be available for new instances to be scheduled to it as long as everything is properly configured.

Enter the great configuration complexity of OpenStack. A configuration management engine will make this process much simpler. There are a handful of configuration management engines out there that have a vibrant community with and active investment in a set of maintained modules to install and configure OpenStack. In *Chapter 2*, *RDO Installation*, Packstack was used to install OpenStack. Under the hood, Packstack uses puppet to do the heavy lifting of installation and configuration. Packstack offers the facility to make additions and modifications to the original configuration that was used to add additional compute nodes by editing and reusing the original answer file. Before we use Packstack to add another node, let's see what compute nodes are already associated with our OpenStack cluster and what their status is using Nova's `service-list` command.

```
control# nova service-list
```

This command will output a list of services that are currently checking in with Nova. For now, the compute node service is the interesting one. If this compute node is in regular communication with the control node, its status will show a happy face and if the compute node loses contact with the control node, the status will be XXX. This status shows whether the compute node is attached to the message bus and checking in with the Nova control tier. Next, let's go ahead and add a new compute node to the cluster.

If you remember the installation from *Chapter 2*, *RDO Installation*, an answer file was used to invoke Packstack. This answer file was populated with all the passwords, IP addresses, and configuration options necessary to configure an OpenStack installation. In particular, there was a line in the answer file that defined the IP addresses of the compute nodes that looked like this:

```
CONFIG_NOVA_COMPUTE_HOSTS=192.168.123.103
```

This parameter specifies one compute node. To add a second compute node is as simple as adding another freshly installed host, one that has just the operating system on it, to the comma-delimited list of compute hosts, and rerunning Packstack:

```
CONFIG_NOVA_COMPUTE_HOSTS=192.168.123.103,192.168.123.104
```

Rerun Packstack passing in the answer file with the updated value. Remember that Packstack will change any configuration that has been manually changed back to the value in the answer file. If you haven't made any modifications to the configuration on the control, network, or compute node file systems, this shouldn't be an issue for you:

```
control# packstack --answer-file myanswers.txt
```

When this finishes, run the Nova service list command, and the second compute node should be listed:

```
control# nova service-list
```

This process can be repeated for more compute nodes. As more nodes are added and show that they are checking in with the Nova control services, the Nova scheduler will spawn instances across all the nodes according to its scheduling algorithm.

Installing more control and networking

To scale the control and networking nodes, you will need to install and configure control and networking services on additional nodes. In our installation example in *Chapter 2*, *RDO Installation*, Packstack was used to do the installation. In a production environment, a more complex configuration management setup should be used; Packstack is intended for demonstration and proof-of-concept installs and is incapable of maintaining an OpenStack installation long term. Foreman, Staypuft, and Triple-O are all more robust options that can handle a longer-term installation. There are also other open source projects and commercial products that have these long-term management capabilities.

Packstack does not include direct functionality to duplicate control and networking services. Although requiring some manual intervention, it can help get plenty of the heavy lifting done for us. Let's take a quick, high-level overview of what it will take to shoehorn Packstack into helping us out. Note that we won't be able to take an exhaustive look at this. The majority of OpenStack components are not addressed here. For the topics not addressed in this chapter, make sure to check the resources for your configuration management tool and test your changes.

The most important part of configuring additional nodes in an OpenStack cluster is to make sure that they have the same users and passwords configured to talk to each other. Fortunately, the answer file you used to install the initial set of nodes contains all the usernames and passwords that will be needed to replicate a node. In light of this, the first thing to do is make a copy of the original answer file so that modifications can be made to the copy to configure the new nodes. Because all the users, passwords, and components that should be installed are the same, search and replace the original IP addresses with the new addresses. I am going to use the same subnets, but I will use the 200s for the fourth octet of the IP addresses instead of the 100s:

```
control# cp myanswer.txt myanswers2.txt
control# sed -i ''s/192.168.123.101/192.168.123.201/g''
myanswers2.txt
control# sed -i ''s/192.168.123.102/192.168.123.202/g''
myanswers2.txt
control# sed -i ''s/192.168.122.101/192.168.122.201/g''
myanswers2.txt
```

Notice that `192.168.123.103` is not updated. Scaling compute nodes was just covered, so it is unnecessary to include it in this Packstack run. However, whichever hosts are defined in the compute hosts directive will still be checked for the proper compute services, so additional hosts could be added for this Packstack run. Leaving aside the already installed and configured hosts, there won't be changing anything about them.

A detailed walkthrough of scaling the database and message bus is beyond the scope of this chapter. They will be touched upon briefly in a conceptual perspective later in this chapter. Since they will not be addressed in detail, set the database and **Advanced Message Queuing Protocol** (**AMQP**) hosts back to the first compute node so that a common database and message bus are used for the cluster:

```
CONFIG_MYSQL_HOST=192.168.123.101
CONFIG_AMQP_HOST=192.168.123.101
```

When Packstack runs for the second set of hosts, it will see that the database and message bus services are already configured on the IP address specified and will not change their configuration. Now that the IP addresses of the new hosts have been put into the new answer file, invoke Packstack with the new file to configure the new nodes:

```
control# packstack --answer-file myanswers2.txt
```

Once the Packstack run is complete, you have a new set of configured nodes, but there are further steps for traffic to be sent to them. This is also the time to note that if you would like more of any of the nodes than you already have, you should go ahead and install them now using additional Packstack answer files. Configuration changes will be made to the cluster that will reverted to by any additional Packstack runs from here on.

Scaling control and network services

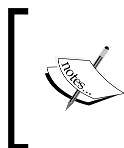
When more compute services are added to the cluster, OpenStack's scheduler distributes the new instances appropriately. When new control or network services are added, traffic has to be deliberately sent to them. There isn't anything in OpenStack that handles traffic being distributed across the API services. There is a load-balancing service called **HAProxy** that can do this for us. HAProxy can be run anywhere it can access the endpoints that you will be balancing. It could go on its own node or it could be put on a node that already has a bit of OpenStack installed on it. Let's put it on the first control node in our example. Start by installing it:

```
control# yum install -y haproxy
```

HAProxy has a concept of frontends and backends. The frontends are where HAProxy listens for incoming traffic, and the backends define where the incoming traffic will be sent to and balanced across. Some of the services will need additional configuration beyond HAProxy to be able to work properly. Let's look at the first example, Keystone.

Load-balancing keystone

Keystone is a single-threaded service that can benefit from some tuning and scaling when done properly. To run HAProxy on the same node as the API services, the API services will have to be told not to listen on all IP addresses on the node. Edit your `/etc/keystone/keystone.cfg` file and change the `public_bind_host` and `admin_bind_host` options from `0.0.0.0` to the internal IP address of the node. If these are commented out, just uncomment them and set them to the internal IP address of your control node.



> Configuration changes from here on made to OpenStack components in configuration files, or in the database, are subject to be overwritten by Packstack with original answer file values if you run Packstack again.

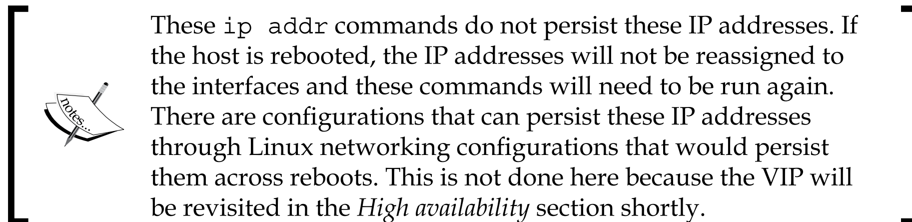
```
public_bind_host=192.168.123.101
admin_bind_host=192.168.123.101
```

Next, restart Keystone so that it sees the change:

```
control# service openstack-keystone restart
```

Now that Keystone is bound to a specific IP address, another IP address can be added to HAProxy to listen in on. Add an IP address for both internal and external traffic. These IP addresses are sometimes called VIPs (Virtual IP addresses). We will talk more about these VIPs when we talk about high availability. For now, know that when a VIP is referenced, it is referring to these IP addresses:

```
control# ip addr add 192.168.123.111/24 dev eth0
control# ip addr add 192.168.122.111/24 dev eth1
```



These `ip addr` commands do not persist these IP addresses. If the host is rebooted, the IP addresses will not be reassigned to the interfaces and these commands will need to be run again. There are configurations that can persist these IP addresses through Linux networking configurations that would persist them across reboots. This is not done here because the VIP will be revisited in the *High availability* section shortly.

Instead of persisting them through Linux networking, we will use a pacemaker to have them dynamically managed. Next, add a frontend and a backend for Keystone to your `/etc/haproxy/haproxy.cfg` file. Note that there is a frontend and backend for both of the ports that keystone listens in on, but the admin port is only bound to the internal IP address. The admin port is not intended for public use. These stanzas could look something like this:

```
frontend keystone-admin-frontend
 bind 192.168.123.111:35357
 mode http
 default_backend keystone-admin-backend
frontend keystone-frontend
 bind 192.168.122.111:5000
 bind 192.168.123.111:5000
 mode http
 default_backend keystone-backend
backend keystone-admin-backend
 balance roundrobin
 mode http
 server control 192.168.123.101:35357 check inter 10s
 server control-deux 192.168.123.201:35357 check inter 10s
backend keystone-backend
 balance roundrobin
 mode http
 server control 192.168.123.101:5000 check inter 10s
 server control-deux 192.168.123.201:5000 check inter 10s
```

Another helpful configuration to add to the `haproxy.cfg` file is an admin listener to get stats from:

```
listen admin
 bind *:8081
 mode http
 stats enable
```

Once HAProxy is listening in on port 8081, the stats page can be accessed at the `/haproxy?stats` URI on one of the IP addresses on the host it is running on. Now, start HAProxy so that it starts to listen on the ports just defined:

```
control# service haproxy start
```

What has been done here? Keystone is now only listening on the internal IP address of the control node on port 5000 and port 35357. HAProxy is listening in on both the internal and public VIPs on both port 5000 and port 35357. When traffic is sent to either of these VIPs on either of the listening ports, HAProxy is going to balance the traffic in a round robin fashion across the active nodes specified in the Keystone backend's stanza in HAProxy's configuration.

There is one missing piece to fully balance the Keystone traffic. HAProxy will only balance the traffic sent to it. If you recall from *Chapter 3*, *Identity Management*, there are endpoints defined for each OpenStack component. These endpoints are stored in Keystone and need to be updated for each component that is going to be load balanced. As it stands now, any incoming traffic from end users that uses the VIPs will be balanced. As soon as a call is made to Keystone to get the endpoint for Keystone to send subsequent requests, the original control node's IP address will be returned and the rest of the calls to Keystone will be sent to the IP address in the Keystone endpoint list. Currently, this IP address is not the load balancer's IP address. This may be a bit confusing because we are currently working on load balancing Keystone. To be clear, a service that is added to HAProxy to be load balanced that has an endpoint registered with keystone must have its endpoint definitions updated so that all requests to that service are load balanced. To do this, let's first list the services to get Keystone's service ID and list Keystone's current endpoints:

```
control# keystone service-list | grep keystone
control# keystone endpoint-list | grep {SERVICE_ID}
```

Unfortunately, a Keystone endpoint update requires us to first delete the existing endpoint and re-add it with the new values. There isn't an endpoint-update command:

```
control# keystone endpoint-delete {ENDPOINT_ID}
```

```
control# keystone endpoint-create \
 --service-id=the_service_id_above \
 --publicurl=http://192.168.122.111:5000/v2.0 \
 --internalurl=http://192.168.123.111:5000/v2.0 \
 --adminurl=http://192.168.123.111:35357/v2.0
```

Now that the endpoint for the Keystone service uses the VIP address, all traffic will be load balanced correctly across all nodes that are configured in HAProxy.

Additional Keystone tuning

By default, Keystone keeps stored tokens in the database. The database can become bogged down by moderate use when trying to keep up with all the token lookups that happen in OpenStack. It is recommended that you reduce the token expiration period, enable caching and that memcached be configured to replace the database as a storage engine for Keystone tokens. These three items will dramatically increase Keystone's performance regardless of whether you have one Keystone instance or multiple.

Glance load balancing

Let's scale one more service to make sure there isn't confusion with updating the endpoints. The second service we looked at earlier in *Chapter 4*, *Image Management* was Glance. Start by updating the `glance-api.conf` and `glance-registry.conf` files to set `bind_host` to the primary IP address of the host that the Glance services are running on:

```
bind_host=192.168.123.101
```

Make sure you do this for both the `glance-api.conf` and `glance-registry.conf` files. Next, add the glance frontends and backends to the `/etc/haproxy/haproxy.conf` file:

```
frontend glance-frontend
 bind 192.168.122.111:9292
 bind 192.168.123.111:9292
 mode http
 default_backend glance-backend
frontend glance-registry-frontend
 bind 192.168.122.111:9191
 bind 192.168.123.111:9191
 mode http
 default_backend glance-registry-backend
```

```
backend glance-backend
 balance roundrobin
 mode http
 server control 192.168.123.101:9292 check inter 10s
 server control-deux 192.168.123.201:9292 check inter 10s
backend glance-registry-backend
 balance roundrobin
 mode http
 server control 192.168.123.101:9191 check inter 10s
 server control-deux 192.168.123.201:9191 check inter 10sAs
```

With keystone, we first are making Glance listen to a single IP address, then binding HAProxy to the VIP on the same port as the Glance services. Now restart the services so that they see the updates that have been added.

```
control# service openstack-glance-api restart
control# service openstack-glance-registry restart
control# service haproxy restart
```

With these configurations in place and the services restarted, the Glance traffic is now being load balanced across the two control nodes. It is very important for each OpenStack service to be evaluated to be sure that the way it functions is proper when more than one instance of the service is running. Glance needs some extra configuration. In the current configuration, there are two Glance registries; therefore, there are two disk stores for the images that are being stored. This is a problem because when you have a common database, it means that all images are going to be listed as available for us to launch. If a request is made to one node to store the image, and a request is made to the other node to get the image to spawn an instance on a compute node, then the request to retrieve the image will fail because the physical media would be stored on the opposite node.

There is a simple solution this problem. There are also multiple solutions. A quick and easy solution would be to mount a shared storage device into Glance's storage path. NFS, GlusterFS, and most other shared storage solutions are great options for this mount, just mount the same shared storage volume to `/var/lib/glance/images/` on each glance node. If there's data already in the directory on any of the nodes, just copy the images to the storage before you mount it in place.

Another option to store the images is to use Swift as the backend storage for Glance. There is support built into Glance and Swift to use Swift as the storage backend for Glance. Let's do that now for each of the nodes. For each node, edit the `/etc/glance/glance-api.conf` configuration file and update the following configuration options:

```
default_store = swift
swift_store_auth_address = http://192.168.123.111:5000/v2.0/
```

```
swift_store_user = services:glance
swift_store_key = {GLANCE PASSWORD}
swift_store_create_container_on_put = True
```

The value that you need to enter for the `swift_store_key` option is the same value that is in `admin_password` in the same file. Once these values are updated on each server, restart the Glance services on each server:

```
control# service openstack-glance-api restart
control# service openstack-glance-registry restart
```

Now that both nodes are configured to use Glance as the backend file store, any images that were already in the Glance registry are inaccessible for use because they are not in Swift. They need to be reimported into Glance to be available. To make them re-accessible, list the images in Glance and then do an `image-create` on each of the images. Use the images ID in the list for each image and reference the file that needs to be imported by using the path `/var/lib/glance/images/{IMAGE_ID}`. Once the images are recreated and stored in Swift, the old record can be deleted with the `image-delete` command:

```
control# glance image-list
control# glance image-create --name {IMAGE_NAME} --is-public true --
disk-format qcow2 --container-format bare --file
var/lib/glance/images/{IMAGE_ID}
control# glance image-delete {IMAGE_ID}
```

Scaling other services

Unfortunately, we will have to conclude the section on scaling here. Moving forward, the process to scale each service in OpenStack becomes fairly repetitive. Make the same IP address modifications to each of the services and update HAProxy for the ports that the services listen in on. Then, make sure to search for information on any extra configuration that needs to be done for each of the services to be able to coexist with multiple instances of themselves within the cluster.

High availability

Until now, the architecture discussed has added additional instances of services and balanced traffic across them. While HAProxy has monitors built into it to check the health of a host, this is only to know whether or not to send traffic to the host. It doesn't include any capability of recovering from failure.

To make the control and network tiers highly available, Pacemaker can be added to the cluster to monitor services, file systems, networking resources, and other resources that need to be made highly available. Pacemaker is capable of moving services from node to node in a pacemaker cluster and monitoring the nodes to know whether action needs to be taken to recover a particular resource or even one of the entire nodes.

The installation and configuration of Pacemaker is a whole book in itself; here, we will just touch upon some of the major, important items to consider when configuring Pacemaker and the concept behind how it will handle some of our resources and what resources need to be managed by Pacemaker. There will not be any command-line examples of how to use Pacemaker.

There are two major infrastructure considerations to be mindful of when considering setting up Pacemaker. These points are related to the installation of Pacemaker and preparing it to start managing resources that you would like to be highly available. First, at least three nodes are needed to properly configure Pacemaker and establish quorum. With only two nodes, if communication is lost between them, they can enter a state called split brain. This is when the nodes both think that they should be the primary node because they can't reach the other node. In this case, resources that should only reside on one server can be started in two places and cause conflict, for example, the VIPs. We will discuss the VIPs a little more in just a moment. When there are more than two nodes, there will be a vote cast from the nodes before an action takes place. For example, if one node loses communication with the other, two or more nodes will have to vote for that node to be fenced. When the majority agrees on the action, then the power to the node is cut to reboot it.

Second, fencing must be configured for a proper Pacemaker installation. Fencing is the capability of nodes within the cluster to control the power of each other. In the case that one node loses communication with the others, the other nodes must be able to do something about it. Without the ability to communicate with other nodes, it is fenced, that is the power to it is cut to reboot it in the hope that a fresh boot will restore its communication with the cluster.

Once a Pacemaker cluster is set up and running, it will be ready to have resources configured within it to be highly available. An example set of resources that should be made highly available are HAProxy and the VIP that it is listening in on. HAProxy is a single point of failure for all of the API traffic being passed through it. By adding it as a resource to Pacemaker, it will be monitored to ensure that the IP address is always reachable on one of Pacemaker's nodes and that HAProxy is listening in on that IP address to receive incoming traffic. Earlier, it was suggested that the VIP wasn't persisted across boots on the control node. That is because when it is added to Pacemaker as a resource, Pacemaker will handle the configuration and health of that IP address for you.

Almost all of the other OpenStack services should be made highly available. Most of them can be added to Pacemaker in what is called a cloned configuration. That means that Pacemaker expects to run them on more than one node but will monitor their health and restart them if they go down. This is the configuration you would probably want to use for the services that are being load balanced by HAProxy.

Highly available database and message bus

The database and the message bus are not necessarily OpenStack services, but they are services that OpenStack depends on and that you want to be sure are highly available too. One option to make the database highly available is to add it to Pacemaker with a shared storage device. If the database were to fail on a node, then Pacemaker would move the shared storage to another node and start the database on a different node. There are also active/passive and active/active replication scenarios that can be configured for the database. Active/passive means that there is more than one instance of the database running, but only one of them is used as the active writable instance. The other instance(s) are there as passive backups and only become active if the current active instance needs to failover for some reason. Active/active means that there is more than one active writable instance of the database. These are running on different nodes and each can be read from and written to as equal members of the database cluster. Replication is important in both of these scenarios because the database can't read and write to the same datastore at the same time. To overcome this, databases know how to replicate their data so that each instance has its own datastore and so that transactions in the database are duplicated to each of the database instances to preserve the integrity of the data in the database.

The message bus is in a similar situation, the main difference being that it doesn't have a persistent data store such as a database. It can be configured in a pure failover mode where it would only run on one node, an active/passive configuration, or an active/active configuration. Each of these configurations has its positives and negatives and should be researched more in depth before one is chosen for implementation.

Summary

In this chapter, we looked at the concepts involved in scaling and load balancing OpenStack services. We have also touched upon the concepts involved in making OpenStack highly available. Now that an OpenStack cluster is up and running and has been scaled to meet demand, we're going to take a look at monitoring the cluster to keep track of its health and help diagnose trouble when it arises.

12
Monitoring

As an OpenStack cluster is scaled out, the number of moving parts that can get jammed increases. As you have seen, each server added to the cluster will run more than one service. Each of those services interacts and communicates with each other across the cluster using different communication methods and unique endpoints for each service. This presents a complicated web of interdependence that can be very complicated to debug when something goes wrong. Monitoring all the moving parts can save a large amount of time and hassle in trying to figure out what has gone wrong when things stop working.

In this chapter, we will look at setting up monitoring for the cluster to help you have a detailed view of the general health of a running OpenStack cluster.

Monitoring defined

There are two classifications of monitoring, performance monitoring and availability monitoring. Performance monitoring shows the performance of what is being monitored over time. Availability monitoring show the status of what is being monitored at a point in time. Often, the same things are monitored, but the purposes of the two types of monitoring are different. As an example, if a server's CPU utilization was being monitored, availability monitoring checks the CPU utilization, and if it breaches a certain threshold, the monitoring alerts an operator that the utilization is high or may have remained high over the most recent checks. Performance monitoring keeps track of the CPU utilization in the longer term and most likely creates a graph to show the trend of CPU utilization on a server across days or weeks or longer.

In this chapter, we will focus on availability monitoring to be able to determine the current health of an OpenStack cluster based on the current status of the checks being run on the servers in the cluster.

Installing Nagios

When Packstack installed OpenStack in *Chapter 2*, *RDO Installation*, one of the options that's available is to install and configure **Nagios**. If you remember, this was one of the options that was set to yes in our installation:

```
CONFIG_NAGIOS_INSTALL=y
```

Since Packstack did the base Nagios installation for us, we are not going to cover the details of a fresh Nagios installation. There are plenty of resources available on the Internet that you can search for if you want more information on doing a fresh Nagios installation outside of Packstack.

In the following sections, we'll add configurations to Nagios configuration files. To apply configuration changes, the service will need to be restarted for Nagios to read the updates and start checks based on the new configurations. For example purposes, we will refer to a set of monolithic configuration files to configure Nagios in this chapter. Each file that is referred to in this chapter is referred to in the top level Nagios configuration file `/etc/nagios/nagios.cfg`. There are options to break up the files referenced in this chapter. You will have to search for documentation on this if you choose to use it.

Adding Nagios host checks

Start by adding host checks. The first example's configuration file will hold all the configuration stanzas for the hosts in the cluster that we are going to monitor, Let's use the file `/etc/nagios/nagios_host.cfg`. This establishes a check to ensure that each host is up and responding to network communication. If you have additional compute nodes, make sure to add them as well. Here's the code that I am talking about:

```
define host {
address 192.168.123.101
host_name control
use linux-server
}
define host {
address 192.168.123.102
host_name network
use linux-server
}
define host {
address 192.168.123.103
host_name compute
use linux-server
}
```

After adding these configurations, validate the Nagios configuration and restart the Nagios service:

```
$ service nagios configcheck
Running configuration check... OK.
$ service nagios restart
```

Often a configuration gets a fat finger error in it and the configuration validation will fail. When that happens Nagios will fail to start. To find out where the syntax error is, run Nagios by hand by referencing the top-level configuration file:

```
$ nagios -v /etc/nagios/nagios.cfg
```

This will give you the line that the syntax error is on. If Nagios restarts successfully, you should be able to connect to Nagios on port 80, select the host list, and after some time passes and the checks fire, do a health check on your hosts that have been added to the hosts configuration file. Now that Nagios is aware of the hosts that we will be monitoring, let's define an example command that could be used to monitor one of the services on the hosts.

Nagios commands

Before service checks can be executed to start checking a service on a host, there must be a command defined that will be referenced by the service check. Let's put these commands in the `/etc/nagios/nagios_command.cfg` file. We are not going to cover all the commands needed to monitor your OpenStack cloud here. Instead, we will cover the concept of a defined command. Each command has a name that will be referenced later and a path to an executable. The executable runs and returns a zero through three return codes. Zero means the check succeeded, one means the check is warning, two means the check failed, and three or another return code indicates the status is unknown. An example of command definitions in the `/etc/nagios/nagios_command.cfg` file looks like this:

```
define command {
command_line /usr/lib64/nagios/plugins/check_nrpe -H $HOSTADDRESS$
-c $ARG1$
command_name check_nrpe
}
define command {
command_line /usr/lib64/nagios/plugins/example_command
command_name example_command
}
```

Note that the commands are executed live in `/usr/lib64/nagios/plugins/`. If you add executable scripts that Nagios will use to check services, it is a good practice to add the executable scripts to this directory. If the intent of `example_command` was to verify that the host's hostname was set properly, its content may look like this:

```
#!/bin/bash
HOSTNAME=`hostname`
if [ -z $HOSTNAME ] || [ -z $1 ]; then
echo "Host name or argument was blank"
exit 3
fi
if [ $HOSTNAME == $1 ]; then
echo "Hostname is $HOSTNAME"
exit 0
fi
if [[ $HOSTNAME == *$1* ]]; then
echo "Hostname is $HOSTNAME and contains $1"
exit 1
else
echo "Hostname is not $1"
exit 2
fi
```

Note that there is a case for all four of the possible return values. It is not required that return codes three and one be returned. Unfortunately, this command could be terribly useless. If it were associated to a host, it would never be accurate because it would always execute on the host that Nagios is running on and would never return success for any host other than the host that Nagios is running on. This creates the need for a command to be executed remotely on a host that is being monitored.

The `check_nrpe` command shown is important as it allows exactly that—remote execution of commands on the hosts being monitored. **Nagios Remote Plugin Executor** (**NRPE**) checks are issued to the hosts via this command definition. Make sure that the NRPE command definition is in the `nagios_command.cfg` file. On each of the hosts that will have NRPE checks run on them, the NRPE service must be running and TCP port 5666 must be open for the Nagios host to connect to. Make sure this is a private connection. If this is unsecured traffic, it can open a security risk.

The configuration for these checks requires that a host and a command name be passed and that all the details about the command that is run beyond its name will be defined on the remote host that the command is being executed against. These details live in `/etc/nagios/nrpe.cfg` on each host. At the very bottom of this file, there is an `include_dir` directive:

```
include_dir=/etc/nrpe.d/
```

Though for this example, we will put the commands right in the `nrpe.cfg` file underneath the `include_dir` directive. By configuring the NRPE commands, Nagios is able to connect to the nodes and execute the commands to carry out the monitoring. Let's use the `example_command` script as an example NRPE command and make it a useful definition. On the control node, put this line in the `nrpe.cfg` file:

```
command[check_hostname]=/usr/lib64/nagios/plugins/example_command
control
```

If this was added to each of the networks and compute nodes with the respective hostnames, then it could be used to verify that a hostname was properly set on each of the OpenStack nodes.

There is a large collection of commands and NRPE commands that need to be defined on the Nagios host and the hosts that Nagios is monitoring. Look at the example code included with this book for the executable scripts, commands, and NRPE definitions needed to execute the service checks that will be referenced in the rest of this chapter.

Now that a basic overview of adding hosts, commands, and service definitions of Nagios has been covered, let's take a look at the kinds of checks that are useful to monitor the health of an OpenStack cluster.

Monitoring methods

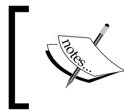
As you begin to design availability monitoring for your cloud, there are at least three schools of thought on the kinds of checks that should be executed. These should be mixed and matched as you deem appropriate to establish the coverage you need to monitor the services in your OpenStack cluster. You may also come across other methods of designing health checks that can be mixed with what is discussed in this chapter.

The first type of check is the service status check. This type of check runs a simple Linux service status check on each of the services. If the service status script returns successfully that the service is running, the health check is successful. The problem with relying on these is that many OpenStack services have the ability to automatically heal from a loss of communication with each other. You can run a service check on an OpenStack service that is up and running but is actively attempting to reconnect to the database or to the message bus. OpenStack is intelligent enough to know when these kinds of connections have been severed and will attempt to re-establish the connections. In this case, the service status check will return positive but users will not be able to use the cluster because things are not functioning properly.

Then come the API checks. This type of check will call the APIs, making sure that a simple resource list returns successfully. This type of check makes service checks a bit redundant if you only have one instance of each service. If the service check fails, the API check will fail too, and there is no need to have two checks telling you that something is not working. The API check can do the job just fine and provides a more thorough check.

API checks become insufficient once you have multiple instances of a service running. In this case, a combination of service checks and the API checks is necessary. If a service is being load balanced, the API check is important to make sure that the instances are being load balanced properly. However, if one of the services gets hung for some reason, the API check will start to flap or change from a successful state to a failed state and back to a successful state over and over as the load balancer still sees both services but one is not healthy. To better monitor this situation, adding extra checks that monitor each instance of the service is necessary. You will have to use your best judgment to decide whether the right way to monitor each individually is to use service checks or API checks.

The third and final check type we will discuss is the resource creation check. These checks use the APIs to actually create resources and then verify that they were successfully created in the cloud as expected. We will not get a chance to look at these. An example of this would be a check that creates an instance and adds it to a network to ensure that it can be connected to. This kind of health check is a little bit more complex to design but is more comprehensive in its coverage.



A word of caution when using this type of check: there are rows that are created in a database for each of these resources and their associated counterparts that are created.

In some cases, when resources are deleted, the rows are not deleted from the database, the resource is just labeled as having been deleted, and the database row remains. A very obvious example of this is a Nova instance. All the instances that are ever launched have a row in the database that can be used to constructs historical record of instances that have existed. Be careful not to bloat your database with health checks and degrade the service with excessive database records unrelated to your end users. There are certain scripts included with OpenStack that are intended to archive some of these records from resources that have been created and deleted. As of now, I've not had them function as expected. There is also discussion in the community to add more archival tools to help manage this kind of archival. Archival generally will just move the records from the tables that active resources are using into an identically structured table with a different name in the same database; they are not completely deleted.

Now that we have taken a look at some of the concepts used to help in defining configurations in Nagios and the kinds of checks that are useful to monitor your OpenStack cluster, let's start to add some checks to start to establish health status beyond the hosts being up or not.

Non-OpenStack service checks

We aren't going to cover generic non-OpenStack service checks in depth here. There is plenty of information you can search for on the Internet that can guide you on generic service checks. We will put these and the OpenStack service checks into `/etc/nagios/nagios_service.cfg`. For OpenStack, it is important to at least add a host load and a disk usage check for each host. OpenStack can consume an excessive amount of disk space and processor load, and the whole cluster can become cranky very quickly if either are used beyond one of the hosts' capacity. There are many other generic checks that can and maybe should be added to your OpenStack hosts though you will have to research others and choose the checks that you deem advantageous. Here are examples of the configurations for checking the load and disk space on `/var`:

```
define service {
check_command check_nrpe!load5
host_name control
normal_check_interval 5
service_description 5 minute load average
use generic-service
}
define service {
check_command check_nrpe!df_var
host_name control
service_description Percent disk space used on /var
use generic-service
}
```

These checks should be set up for all your hosts. You can see that the disk space check uses the default check interval and the load check is set up with a custom check interval. Also, it is important to note that both of these checks are done over NRPE. This means that the Nagios host connects to the NRPE service on the specified hosts, and the command is executed local to the host being monitored. To see the commands executed for these checks, look at the Nagios commands that are included with the code with this book. Now let's get into some OpenStack-specific availability checks.

Monitoring control services

The control tier of an OpenStack cloud has the most moving parts that will need to be monitored. There are a few services that need at least a basic service connection validation. They include, but are not limited to, MySQL, RabbitMQ, and MongoDB. More monitoring can certainly be added beyond simple connection checks to monitor connections, queue sizes, and other statistics of the services. For now, we'll just add a connection check to make sure that these services are running:

```
define service {
check_command check_mysql!nagios! nagios_password
host_name control
service_description MySQL Health check
use generic-service
}
define service {
check_command check_nrpe!check_rabbitmq_aliveness
host_name control
service_description RabbitMQ service check
use generic-service
}
define service {
check_command check_nrpe!check_mongod_connect
host_name control
service_description MongoDB service check
use generic-service
}
```

You can get the scripts for Rabbit and Mongo from `https://github.com/mzupan/nagios-plugin-mongodb` and `https://github.com/jamesc/nagios-plugins-rabbitmq`

Next, we get into checking OpenStack services. We are going to add API checks to make sure that the service is running and that it is not in an error state. Packstack includes a few scripts to cover most of the API services. A few are additional to Packstack. Let's add the service stanzas for Nagios for the API calls:

```
define service {
check_command keystone-user-list
host_name control
normal_check_interval 5
service_description number of keystone users
use generic-service
}
define service {
check_command neutron-net-list
host_name network
service_description Neutron Server service check
```

```
use generic-service
}
define service {
check_command nova-list
host_name control
normal_check_interval 5
service_description number of nova instances
use generic-service
}
define service {
check_command glance-index
host_name control
normal_check_interval 5
service_description number of glance images
use generic-service
}
define service {
check_command cinder-list
host_name control
normal_check_interval 5
service_description number of cinder volumes
use generic-service
}
define service {
check_command heat-stack-list
host_name control
normal_check_interval 5
service_description number of heat stacks for admin
use generic-service
}
define service {
check_command ceilometer-resource-list
host_name control
normal_check_interval 5
service_description number of ceilometer resources
use generic-service
}
define service {
check_command swift-list
host_name control
normal_check_interval 5
service_description number of swift containers for admin
use generic-service
}
```

With these basic checks in place, a set of successful checks in Nagios will show that services are up and running and the API services are healthy enough to list the resources that are being managed. There is a collection of services on the control node that are not API services. It is usually enough to do a service status check on them to make sure they are running. Let's add a service status check for the rest of the services that are not API endpoint services. You will want to add configuration stanzas that look like this for each service:

```
define service {
check_command check_nrpe!check_service_name
host_name 10.100.0.4
service_description Service Name service check
use generic-service
}
```

Do that for each of the following services, replacing `service_name` and `Service Name` with the actual service names:

```
openstack-ceilometer-alarm-evaluator
openstack-ceilometer-alarm-notifier
openstack-ceilometer-central
openstack-ceilometer-collector
openstack-ceilometer-notification
openstack-cinder-backup
openstack-cinder-scheduler
openstack-cinder-volume
openstack-glance-registry
openstack-heat-api-cfn
openstack-heat-engine
openstack-nova-cert
openstack-nova-conductor
openstack-nova-consoleauth
openstack-nova-novncproxy
openstack-nova-scheduler
```

Remember that each of these services points to a corresponding NRPE command, so the hosts that these services run on will have to have the corresponding NRPE command defined on them.

Monitoring network services

Next, let's take a look at monitoring networking services. Networking services in general usually stay running, and things that go wrong are happening inside the running service. We will go ahead and put a service status check on each of them and add additional checks to make sure things are working across the board. Start with giving each of the network services a service status check—the same checks that the control services got:

```
neutron-dhcp-agent
neutron-l3-agent
neutron-lbaas-agent
neutron-metadata-agent
neutron-metering-agent
neutron-openvswitch-agent
neutron-ovs-cleanup
openvswitch
```

Now, let's look at what can be monitored to make sure that when these services say that they are running, the network service is actually running. The configuration we have used in this book uses VXLAN tunnels to build overlay networks for OpenStack tenants. What this means is that each compute node is connected to the network node and to each other with VXLAN tunnels that encapsulate the traffic so that the network that actually connects the nodes doesn't directly handle the network traffic within Open vSwitch. When a packet is put on the network by an instance, it is tagged with a local VLAN to the compute node. When the packet moves from the compute node to the tunnel between that compute node and either another compute node or the network node, it gets retagged and the encapsulation header is added to the packet. This header somewhat hides the VLAN tag given to the packet to move across the tunnel from the actual network that connects the nodes that the packet is moving between. After the packet reaches the destination node it was sent to, the header is removed, and the packet is then retagged again to move around locally within this next node. These tunnels are handled by a running OVS process, and the ports and interfaces that are added and removed are handed by the running neutron OVS agent running on each node. Here is where just because an agent is running, it does not mean that traffic is flowing from node to node without issue. To monitor that traffic is actually flowing, we can build our networking resources in OpenStack that mock the process followed when an instance is attached. Instead of attaching an instance to it, we will expose an interface to the node that we want to make sure is properly networking and send a ping across it. If the ping succeeds across the interface exposed to the node, then we know that the entire encapsulation just described is working properly.

The first step to set up the networking is to create a network specifically to do the tunnel monitoring on. Make sure you have sourced your admin's `keystonerc` file, and create the network. Refer to *Chapter 3*, *Identity Management*, if you need to revisit the `keystonerc` file. Here's the command that is being discussed:

```
control# neutron net-create tun-mon

control# neutron subnet-create tun-mon 10.0.0.0/24
```

Take note of the network ID from the `net-create` command; you will need that at the end of this process. Next, manually create a neutron port for each node that you want to monitor tunnel connectivity on. This is most likely each of your compute nodes. You don't need to do this for the network node or the control nodes. The control node has nothing to do with your networking in OpenStack, and the network node is the node that you will be pinging to verify tunnel connectivity. Here's the command that is being discussed:

```
control# neutron port-create --name moncompute --
binding:host_id=compute

$NETWORK_ID
```

`NETWORK_ID` is the ID of the network that was just created. You should be able to use `grep` to port out of a neutron port list now:

```
control# neutron port-list | grep moncompute
```

This will include a port ID, a MAC address and an IP address. You will need those for the final step. Finally add a port in the OVS on the target machine and give it an IP address:

```
compute# ovs-vsctl -- --may-exist add-port br-int moncompute \

-- set Interface moncompute type=internal \

-- set Interface moncompute external-ids:iface-status=active \

-- set Interface moncompute external-ids:attached-mac=${PORT_MAC}
\

-- set Interface moncompute external-ids:iface-id=${PORT_ID}

compute$ ip link set dev moncompute address ${PORT_MAC}

compute$ ip a add ${PORT_IP}/24 dev moncompute
```

Now on the compute node, you can verify that all is in place. First, look at the interface:

```
compute# ip a s moncompute
```

You should see an interface that has the IP address and MAC address that corresponds to the Neutron port you created for the node. Next, look at the routing table:

```
compute# ip r
```

You should see a routing entry for the subnet that you gave to the tun-mon network. In the example command earlier, it was given as `10.0.0.0/24`, so the routing entry should look like the following line of code:

```
compute# 10.0.0.0/24 dev moncompute proto kernel scope link src
${PORT_IP}
```

Finally, on the compute node, you can look at the port in the OVS:

```
compute# ovs-vsctl show
```

Look for a port named `moncompute`. It will have an interface named `moncompute`, which is the interface you just looked at, and it will have a VLAN tag number. The last thing to do is get the DHCP address from the network you created and ping it. To get the DHCP agent's IP address, show the interfaces in the network namespace for your network on the network node:

```
network# ip netns exec qdhcp-${NETWORK_ID} ip a
```

You will see `127.0.0.1` and another address, probably `10.0.0.2` or `10.0.0.3` if you used the same subnet as the example. This address is the DHCP agent for the network you created. Now, try and ping that address from the compute node:

```
compute# ping 10.0.0.3 -c 3
```

If you get a reply ping when you do this, your tunnel is working. The way this traffic is funneled over the wire ensures that the VXLAN tunnels in your OpenStack cluster are working properly. These resources should stay in place unless you delete them but the OVS interface on the compute node will have to be recreated if the node is rebooted. You will have to get creative about how to persist or reestablish the interface if the node is rebooted. The ping can be added to Nagios so that you get your tunnel status with the rest of your checks. Let's move on to compute services and take care of the ping there.

Monitoring compute services

The final set of services to monitor are those on the compute node. Here, you can make sure a couple of services are running and add the ping from the section you just finished. Start with the generic service status check for these services:

```
neutron-openvswitch-agent
openvswitch
neutron-ovs-cleanup
openstack-ceilometer-compute
openstack-nova-compute
```

Then add a service configuration to Nagios that will run the ping command to check your tunnel connectivity.

```
define service {
check_command check_nrpe!check_ovs_tunnel
host_name compute
service_description OVS tunnel connectivity
use generic-service
}
```

As you can see, this is just an NRPE check command that will execute a ping from the compute node to the network node.

Summary

As a final word of caution, remember that successful health checks across a cluster do not equate to a positive end user experience. Make sure to be in communication with end users about their experience, and use the cluster for your own purposes to ensure you are familiar with the experience the end user is receiving.

In this chapter, we have gone through a list of items that should be checked to monitor the health of an OpenStack cluster; this list is not exhaustive though. The best practice is to keep an eye out for possible points of failure and add checks that make sure that something that could potentially degrade services is monitored for its health.

The last topic for us to cover is troubleshooting. When these health checks start to alert, how should you go about diagnosing the problem and resolving the issue? In the last chapter, we will take a look at how to troubleshoot each of OpenStack's components.

13
Troubleshooting

With the number of moving parts that make up an OpenStack installation, it is inevitable that as the cluster is brought up for the first time, a few things will not work. Further, as the cluster operates, there will also be service failures that should be addressed. It is very important to be able to troubleshoot a running OpenStack installation. Let's take a look at some of the details of how things work under the hood and how to figure out what is going wrong when things are not working properly. We will look at general troubleshooting and then take a look in detail at a few components to help troubleshoot each of them.

The debug command line option

Most of the command-line clients support passing `--debug` before a subcommand. For example, with Keystone or Nova, it could look like this:

```
$ keystone --debug service-list
$ nova --debug list
```

Note that `--debug` is put before the subcommand being executed. Using the debug option like this is helpful because it will show curl commands for each of the API calls that are being made from the command-line client to the API endpoints. Hosts and ports are included in this, so if your command-line client has trouble connecting to the endpoint, you can use the debug option to get more detail. If you need to see what information is being sent from or returned to the command line, the debug option will show those details.

Tail the server logs

There is an extensive collection of logs across an OpenStack cluster, and they are your best friend. Often a good place to start is when an API call succeeds, but the end result is, not as you expect, to tail the log files of a component that you're having trouble with. You can do this as, or right after, you execute the command that you are seeing failure with. For example, if you are having trouble connecting to Keystone, it might not be running properly or might be throwing errors for some reason. Start a tail on `/var/log/keystone/keystone.log` and rerun the command that is failing. This is shown in the following command:

```
$ tail -fn0 /var/log/keystone/keystone.log
```

In this command, `-f` indicates that we follow the log or show new entries as they are added. The `-n0` means *show the most recent zero lines*; in other words, any previous content in the file is suppressed so that you only see new entries when you run the command. All of the OpenStack components are going to have logs in `/var/log/{component_name}/` except Swift-proxy, which will be in `/var/log/messages`. Horizon will have extra logs in Apache's log files at `/var/log/httpd/*log`.

As another example, if Nova is not launching an instance properly, there could be a problem with the API collecting enough information or a problem with the scheduler finding a place to put the running instance. Sometimes, it is helpful to tail all the logs in the log directory instead of just one. Tailing more than one log will get you output from all the services that are related to a component. This is shown in the following command:

```
$ tail -fn0 /var/log/nova/*.log
```

Notice here that `*.log` is indicated and not an asterisk. This is because if logrotate is rotating logs, there could be `.gz` files that you do not want to tail because they are binary. The tail's initial output will indicate what file a new entry is coming from, which will help you narrow down the service that needs a little help.

Often, the case is that one component is showing an error in its log, but the error is being generated by another component, and the error that you are trying to debug was a result of a call being made from one component to another. To debug this kind of behavior, it is helpful to know how components interact with each other. In the following sections, let's take a look at the major components in OpenStack for you to get an better idea of how they interact with each other so that you can effectively debug them.

Troubleshooting Keystone and authentication

Nothing is more frustrating than not being able to log in to your cluster to see what is going on. Thankfully, OpenStack offers an authentication override to bypass authentication and allow you to make Keystone calls to see services, endpoints, and other Keystone resources. This is called using the Keystone admin service token. In *Chapter 3*, *Identity Management*, we looked at creating a `keystonerc` file. To use this service token to override authentication, you need to use a similar methodology. Start by getting the current service token value from the `keystone.conf` file:

```
$ grep admin_token /etc/keystone/keystone.conf
```

The value that keystone's `admin_token` is set to can be passed with a service endpoint URL to Keystone and authentication will be overridden. Get the `OS_AUTH_URL` environment variable from the `keystonerc_admin` file you created, and create a new file with the following content. To keep it separate from your original `keystonerc` file, give it a name like `keystonerc_service_token`. The following command shows this:

```
export OS_SERVICE_TOKEN={value of keystone.conf admin_token }
export OS_SERVICE_ENDPOINT=http://192.168.123.101:35357/v2.0/
```

It is important to note here that `OS_SERVICE_ENDPOINT` points to your Keystone administrative endpoint on port 35357 and not the public or internal endpoint on port 5000. Port 5000 is for authenticated traffic, and port 35357 is for non-public administrative traffic, such as service token calls to override authentication. It is not recommended that port 35357 be publicly accessible. Next, source this file so that the environment includes these variables:

```
$ source keystonerc_service_token
```

Now, run a Keystone command such as `service-list` or `endpoint-list`, and you will see a message that indicates that authentication has been bypassed:

```
WARNING: Bypassing authentication using a token & endpoint
(authentication credentials are being ignored).
```

The first thing you want to do here is reset the admin user's password and then stop using these service tokens. It is very bad practice to operate on Keystone using the service token. So, first, update the admin user's password. Then, unset the service token environment variables. The following command shows this:

```
$ keystone user-password-update admin
$ unset OS_SERVICE_TOKEN
$ unset OS_SERVICE_ENDPOINT
```

Make sure you unset both SERVICE environment variables. If Keystone sees `OS_SERVICE_TOKEN` and not `OS_SERVICE_ENDPOINT`, it will complain. If it sees `OS_SERVICE_ENDPOINT` and not `OS_SERVICE_TOKEN`, weird things happen.

Once you have unset the service token environment variables, make sure that you update the password in your `keystonerc_admin` file and re-source it. If you do not source it, then the new password will not be used. This is shown in the following command:

```
$ source ~/keystonerc_admin
```

If you want to see the value that is being used of any of the variables that you have sourced, then just use the `echo` command; for example, to see your password that is being used, execute the following command:

```
$ echo $OS_PASSWORD
```

If you have other problems to troubleshoot, start with the Keystone log file at `/var/log/keystone.conf`. From there, you'll need to move on to verifying the endpoints by listing them with endpoint-list and making sure that they are correct and that they point to running services. If you are having authentication issues from the services, you need to make sure that the password in the configuration file for the services matches what is in Keystone's database. You can simply use the `user-update-password` command for the service users to force Keystone's password for the service users to match what is in their configuration file. The user name to use should be indicated in the service's configuration file right next to the password it is using.

Troubleshooting Glance image management

It's not often that Glance needs troubleshooting. There are two common ways that you will have things fail related to Glance:

- If Glance cannot access the file system that it will be writing to when it is saving an image into the registry
- If Nova cannot get an image that has been assigned to launch an instance with

In the event that you are not able to save an image to the registry, you will just have to read the logs in `/var/log/glance/*`. Depending on the backing store that you have chosen, or has been configured for you, there will be different errors. In most cases, when you resolve these errors, you will have a working Glance service.

When a new instance is launched on a compute node, one of the things that the nova-compute service does is to check whether it has a cached copy of the image that the instance is being launched from on the compute node. If it does, it will use the local cache; if it does not, it will connect to Glance and download a copy of the image to its cache and then continue launching the image. This, again, is a case where you will have to watch the logs and read the errors that are being thrown. Follow the logs and verify connectivity from the compute node to the Glance endpoint. This should help if `nova-compute` cannot get images from Glance for the instances it is launching.

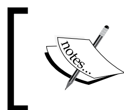
Troubleshooting Neutron networking

Neutron is a bit of a special case among the OpenStack components because it relies on and manages a fairly intricate collection of transport resources. These may be created as a result of Neutron resources being defined by end users. There is not always a straightforward correlation at first sight. Let's walk through the traffic flow for an instance to make sure that you know which agent is doing what within the Neutron infrastructure.

The first thing that an end user will do before launching an instance is create a network specific to their tenant for their instances to attach to. At the system level, this translates into a network namespace being created on the node that is running the Neutron DHCP agent. Network namespaces are virtual network spaces that are isolated from the host-level networking. This is how Neutron is able to do isolated networks per tenant. They all get their own network namespace. You can list the network namespaces on any Linux host that has network namespaces enabled, using the `ip utils` command:

```
$ ip netns
```

When you run this command, if you have some networks already defined, you see namespaces named `qdhcp-{network-id}`.



> The letter Q is a legacy naming convention from the original name Neutron had, which started with the letter Q. The old name had a legal conflict, and it had to be changed.

So a qdhcp network namespace is a namespace created to house the DHCP instance for a private network in Neutron. The namespaces can be interacted with and managed by the same tools as the host's networking by just indicating the namespace that you want to execute commands in. For example, let's list the interfaces and routes on the host and then in a network namespace. Start with the host you're on. The following command shows this:

```
$ ip a
$ ip r
```

The interfaces listed should be familiar, and the routes should match the networks you are communicating with. These are the interfaces and routes that the host that OpenStack is installed on is using. Next, get the ID of the network you would like to debug, and list the interfaces and routes in the namespace using the `netns exec` command. This is shown by the following command:

```
$ ip netns exec qdhcp-{network-id} ip a
$ ip netns exec qdhcp-{network-id} ip r
```

The same commands are executed inside the namespace and the results are different. You should see the loopback device, the DHCP agent's interface, and the routes that match the subnet you created for your network. Any other command can also be executed in just the same manner. Get the IP address of an instance of the tenant network and ping it:

```
$ ip netns exec qdhcp-{network-id} ping {host-ip-address}
```

There is even an independent `iptables` rule space in this namespace:

```
$ ip netns exec qdhcp-{network-id} iptables -nL
```

The ping is important because by pinging the instance that is running on the compute node from the `qdhcp` network namespace, you are passing traffic over the OVS tunnel from the network node to the compute node. OpenStack can appear to be completely functional—instances launch and get assigned IP addresses—but then, the tunnels that carry the tenant traffic aren't operating correctly, and the instances are unreachable by way of their floating IP addresses.

To debug an unreachable host, you have to traverse more than one namespace. The `qdhcp` namespace we just looked at is one of the namespaces that an instance needs to communicate with the outside world; the other is the qrouter. The OpenStack router that the instance is connected to is represented by a namespace, and the namespace is named `qrouter-{router-id}`. If you look at the interfaces in the qrouter, you will see an interface with the IP address that was assigned to the router when the tenant network was added to the router. You will also see the floating IP added to an interface in the qrouter.

What we are working towards is tracing the traffic from the Internet through the OpenStack infrastructure to the instance. By knowing this path, you can ping and **tcpdump** to figure out what in the infrastructure is not wired correctly. Before we trace this, let's look at one more command:

```
$ ovs-vsctl show
```

This command will list the bridges and ports that Open vSwitch has configured in it. There are a couple of them that are important for you to know about. Think of a bridge in OVS as somewhat analogous to a physical switch, and a port in OVS is just a network port on a switch or a physical port on a physical switch. So `br-int`, `br-tun`, `br-ex`, and any others that are listed are virtual switches and each of them have ports. Looking at `br-int` first, we can figure out that this is the name of the bridge that all local traffic will be connected to. Next, `br-tun` is the bridge that will have the tunnel ports on. Not all hosts will need to use `br-ex`; this is the bridge that should be connected to a physical interface on the host to allow external traffic to reach OVS. Finally, only in the case of a VLAN setup is there a custom bridges setup. We are not going to look at them in this book, but you should know that the three we are discussing are not the exclusive list of OVS bridges that OpenStack uses. The last thing to note here is that some of these bridges have ports to each other. This is just like connecting two physical switches to each other.

Now, let's map out the path that a packet will take to get from the Internet to a running instance. The packet will be sent to the floating IP from somewhere on the internet. It should go through the physical interface, which should be a port on `br-ex`. The floating IP will have an interface on `br-ex`. The virtual router will have an interface on `br-ex` and `br-int` and will have iptables rules to forward traffic from the floating IP address to the private IP address of the instance. For the packet to get from the router to the instance, it will travel over `br-int` to `br-tun`, which are patched to each other, over the VXLAN tunnel to br-tun on the compute node, which is patched with br-int on the compute node that has the instance's virtual interface attached to it.

With this many different hoops to jump through, there are quite a few places for traffic to get lost. The main entry points for debugging start with namespaces. Start by trying to ping the instance or the DHCP server for the tenant in the namespaces and move to tcpdump if you need to. Pings and tcpdumps can be a quick diagnosis of where traffic is not flowing. Try these tests to track down where things are failing.

- Make sure that ICMP is allowed for all IP addresses in the security groups
- Ping the instance from the qdhcp namespace
- Ping the instance from the qrouter namespace
- Tcpdump ICMP traffic on physical interfaces, `br-int`, `br-tun`, and `br-ex` as needed

These pings establish first that traffic is flowing over the VXLAN tunnels and that the instance has successfully used DHCP. Secondly, they establish that the router's namespace was correctly attached to the qdhcp namespace. If the ping from qdhcp doesn't succeed, use `ovs-vsctl show` to verify that the tunnels have been created and check the Open vSwitch and Open vSwitch agent logs on the network and compute node if they haven't. If the tunnels are there and you can't ping the instance, then you need to troubleshoot DHCP. Check `/var/log/messages` for DHCP messages from the instance on the network node, and boot an instance you can log in from the console to try to **dhclient** from the instance if you don't see messages from the instance in the logs.

If the initial pings don't succeed, you can use tcpdump on the Open vSwitch bridges to dig a bit deeper. You can specify which interface you want to listen on using the `-i` switch on tcpdump, and you can drill down to where traffic is failing to flow by attaching to the Open vSwitch bridges one at a time to watch your traffic flow. Search the Internet for tcpdump and ping if you are not familiar with using tcpdump:

```
$ tcpdump -i br-int
```

Start with `br-int` on the compute node, and make sure you see the DHCP traffic coming out of the instance onto `br-int`. If you do not see that traffic, then the instance might not even be trying to use DHCP. Log in to the instance through the console and verify that it has an interface and that dhclient is attempting to contact the DHCP server. Next, use tcpdump on `br-int` on the network node. If you do not see the DHCP traffic on the network's br-int, then you may need to attach to `br-tun` on the compute and network nodes to see whether your traffic is making it to and across the VXLAN tunnels. This should help you make sure that the instance can use DHCP, and if it can, then traffic should be flowing properly into the instance from the network node.

Next, you may need to troubleshoot traffic getting from the outside world to the network node. Use tcpdump on `br-ex` to make sure you see the pings coming from the Internet into the network node. If you see the traffic on `br-ex`, then you will want to check the `iptables` rules in the `qrouter` namespace to make sure that there are forwarding rules that associate the floating IP with the instance. The following command shows this:

```
$ ip netns exec qrouter-{router-id} iptables -t nat -L
```

In this list of `iptables` rules, look for the floating IP and the instance's IP address. If you need to, you can use tcpdump in the namespace as well, though it is uncommon to do this. Look up the interface the floating IP is on, and attach to it to listen to the traffic on it:

```
$ ip netns exec qrouter-{router-id} ip a
$ ip netns exec qrouter-{router-id} tcpdump -i {fip-interface}
```

Using this collection of tests, you should be able to identify where the trouble is. From there, check logs for Open vSwitch and the Neutron Open vSwitch agent for tunneling, the Neutron Open vSwitch agent and Neutron DHCP agent for DHCP issues, and the Neutron L3 agent for floating IP issues.

Troubleshooting Nova launching instances

Nova has good logging and will most likely have a pretty good error message to indicate what is going wrong if you have trouble getting instances to launch. You would want to check logs for nova-api, nova-conductor, and nova-scheduler on the control tier, and on each compute node where there is a nova-compute service running. Let's look at what each of these agents does so that you know where to look when something is going wrong.

When a Nova command is executed, it talks to the nova-api service. If an error is received directly when a Nova command is executed, this is generally where you can find more detail beyond the immediate error message returned.

Once a command to act on an instance is accepted into the Nova infrastructure, there are two services that handle passing actions to the compute nodes—nova-scheduler and nova-conductor. Nova-scheduler is just what its name suggests. It handles the decision making as to where to schedule resources across the collection of compute nodes that are available. Start with the scheduler's log file if instances are falling into an error state quickly. There is a chance that there are criteria mismatches for the instance to launch. If the scheduler suggests there are no available compute nodes, then use the nova service-list command to see a list of available compute nodes:

```
$ nova service-list
```

This will show all the Nova services and their statuses. The services of the nova-compute type are the nodes that the scheduler will need to have an *up* status to be able to schedule resources properly.

Once the scheduler has made a decision about where to launch an instance, it will pass on the information to nova-conductor. The main purpose of nova-conductor is to remove the need for the compute nodes to access the database. There is communication between nova-conductor and the nova-compute service on each of the compute nodes via the message bus and they access the database on their behalf. If your instances are having trouble spawning, then they have been scheduled, and there is most likely an issue with nova-conductor or its communication with the nova-compute nodes. Check the logs in `/var/log/nova` for nova-conductor and the assigned compute node's nova-compute service.

Troubleshooting post-boot metadata

Images are built and added to Glance as generic reusable images. This means that there isn't any data included to launch the image that is built into it. To provide the image with configurations to allow login and customization, the images should include a service called cloud-init. Cloud-init calls back into OpenStack to get SSH pub keys and post-boot configuration commands. There is a predetermined URL that cloud-init calls into: `http://169.254.169.254`. If you are getting an access-denied error when you try and SSH to your floating IP address, it is probably because cloud-init is failing to get the SSH pub key for your authorized keys file, you are using the wrong username, or you are using a prepackaged image that you have downloaded with a username other than root.

To troubleshoot the metadata service, make sure that you have an image that you can connect to the console. CirrOS is a good option for debugging things like this; just remember not to use CirrOS for anything other than testing and debugging. CirrOS is an insecure distribution of Linux and is intended only for testing and debugging. Once you have logged into the console of an instance, use curl to mock the call that cloud-init will make:

```
$ curl http://169.254.169.254/latest/meta-data/
```

You will have to memorize the IP address that is used or search the Internet, but you can make a call directly to the IP, and it will give you a list of paths that can be used. Try making a call to the metadata service in this order:

```
$ curl http://169.254.169.254/
$ curl http://169.254.169.254/latest/
$ curl http://169.254.169.254/latest/meta-data/
```

You can see that in the second call, the latest is listed, and in the third call, metadata is listed. In the third call, there is a further collection of paths that can be called to get different information. If you have gotten this far, then the metadata service is working, and maybe cloud-init is not installed on the image that you are having trouble accessing. When you call these URLs and you get errors, you will have to check logs in two places to figure out what the issue is. The first one is the Neutron metadata proxy service. Look in `/var/log/neutron` and you will see one log named `metadata-agent.log` and other logs named `neutron-ns-metadata-proxy-{network-id}.log`. To troubleshoot the metadata service that is not working, you just need to look at the `metadata-agent.log` file. Make sure there aren't any errors in it. There are only a few configuration options in this file. If you get any connection error in the logs, check the URL and port for the Nova metadata proxy service. If you get an authentication error, check that the shared secret matches the shared secret value in the `nova.conf` file on the control tier.

If your investigation takes you up to Nova, you will need to look at the API service. The Nova metadata service is a subprocess of nova-api. Check the logs of nova-api and check the errors that are there. Configuration options related to nova-api are in `nova.conf`.

When troubleshooting the metadata service calls, the issues are usually closer to the instance itself, such as cloud-init not being installed, the instance not using DHCP properly, or in early cases of setting up the cluster, the neutron-metadata-agent not being configured properly to proxy calls.

Troubleshooting console access

The URL for console access is generated from a property that is set on each compute node. Look at the `nova.conf` file on one of your compute nodes for the `novncproxy_base_url` property:

```
novncproxy_base_url=http://control.example.com:6080/vnc_auto.html
```

This base URL is the address of the `nova-novncproxy` service that will be able to create a console connection to the instance. In the web interface, this URL is retrieved for you when you select the console tab for an instance. At the command line, you can use Nova's `get-vnc-console` and pass the `novnc` type to get the console URL:

```
$ nova get-vnc-console {instance-id} novnc
```

This will return a URL that will include `base_url` from the compute node the instance is running on and an authentication token to access the console with. If you paste this URL into a web browser, there are a few steps that happen under the hood to connect you to the instance's console.

First, you need to be able to connect to `nova-novncproxy`. This runs by default on port 6080 on the control tier. Once the request to connect to the instance's console is established, the token that is passed in the URL is passed to `nova-consoleauth` and validated in `nova-cert`. If all that succeeds, then the connection to the instance on the compute is established and the console is provided through the web browser.

If you are having trouble with console connections, make sure that the base URL is pointing to the `nova-novncproxy` service, and that you can connect to the `nova-novncproxy` service. If you see a console window, but the console is not being displayed, then you need to check that `nova-consoleauth` and `nova-cert` are running and check the logs for errors in validating the token. If there aren't any errors in validating the token, check the `nova-novncproxy` log to ensure that a connection to the instance can be made.

Troubleshooting Cinder block storage

Troubleshooting Cinder is similar to troubleshooting Glance. The issues that arise are dependent on the backing storage that you use. The best course of action to troubleshoot it is to watch the logs in `/var/log/cinder/*` and correct errors that show up in there. These logs are where you will find errors related to creating Cinder volumes and connecting them to the instances.

Troubleshooting Swift object storage

The trick to troubleshooting Swift is to remember that it has a proxy tier and a storage backend tier. The proxy is essentially the API layer to Swift, and it is the place to start looking for errors in Swift. As mentioned earlier, Swift-proxy does not have its own log file. Its logs will show up in `/var/log/messages`. If things look good in the proxy's logs, then take a look at the storage backend, and see whether there are any errors in the Swift storage logs.

Troubleshooting Ceilometer Telemetry

Ceilometer has a large number of dependencies and agents that run across the cluster to collect data on the resources being used. Once again, to troubleshoot this component, the best course of action to take is to watch the logs. Any errors that you see should be resolved to ensure that Telemetry data is being collected.

Troubleshooting Heat orchestration

Heat is one of the widest-reaching components within the OpenStack infrastructure to troubleshoot. This is because each template that is built is different and has different dependencies among resources created in the stack that is launched. Further, it is able to depend on or create almost any resource within OpenStack. As with the other components, the best starting place is with the Heat logs in `/var/log/heat`. This will give you a good indication of where things might not be going correctly.

If a stack is launched and does not successfully complete, but you do not see any errors in the Heat logs, then the issue may be in the instances. Heat has callbacks from the running instances that must work for the orchestration of data and the ordering of instance creation within a stack.

If you review the section in this chapter on the metadata server, you will notice that the post-boot configuration that is run on an instance is delivered by the metadata service. When cloud-init fires and receives post-boot configuration, it will log what it is doing in `/var/log/cloud-init.log`. There are callbacks into Heat that are included in your cloud-init scripts that must alert Heat that it is time for the next instance in a stack to be created if it is dependent on a previous instance in the stack completing its provisioning. The service that is called back into is the heat-cfn service. If you see failures calling back into the heat-cfn service, you will need to get this URL and verify and troubleshoot this connectivity.

Getting more help

It is not possible to foresee every error or troubleshooting scenario that could happen in an OpenStack installation. If you need further help in debugging and installation, start with `http://ask.openstack.org`. Each of the projects that we have covered in this book also has a community of developers that actively work on it. You can connect with them through mailing lists, forums, IRC channels, and bug trackers. To find contact information for each of these communication channels, start with `http://www.openstack.org`. If you do not find what you are looking for there, then search the Internet to get further information for each of the projects.

Summary

In this chapter, we have taken a look at some of the architecture and workflow to a few of the components of OpenStack. Knowing how the components operate will help you to troubleshoot issues when they arise in your OpenStack cluster. With the number of modules that are present in OpenStack, you have to follow the order of operations that are being run, validate their success, and check for errors as you go along.

Having looked at troubleshooting OpenStack, we have come to the end of this book. The information in this book is core to OpenStack and defines many of the basic concepts and methodologies that are baked into the OpenStack project. The project moves quickly, and new features are added very rapidly, but because the information here is central to a base installation of OpenStack, you should be able to reference the majority of this book for many releases to come from the OpenStack project.

Index

A

Advanced Message Queuing Protocol (AMQP) 118
alarms 92
Amazon Web Services (AWS) 98
answer file
  generating 12-16
appliance-creator 4
Application Programming Interfaces (API) 1
authentication
  troubleshooting 143, 144
availability monitoring 127
AWS CloudFormation format
  about 98
  configuration options 98, 99
  reference link 98

B

backing storage
  about 75
  Cinder, types 75
block storage
  attaching, to instance 70, 71
  creating 69, 70
  use case 69
  using 69, 70

C

Ceilometer
  about 7, 89, 90
  alarms 92
  meters 90
  pipeline 90
  samples 91
  statistics 91
  Telemetry, troubleshooting 153
Ceph 6, 75
CIDR calculator
  URL 41
Cinder
  about 6, 69
  block storage, troubleshooting 152
  types 75
  using 69
  volumes, managing in web interface 72-74
CirrOS
  about 30, 151
  URL 30
Classless Inter-Domain Routing (CIDR) 41
cloud-init 4, 150
commands, Nagios
  defining 129-131
components, OpenStack
  compute 1
  control 1
  network 1
compute nodes
  scaling 115-117
compute services
  monitoring 139
configuration, Open vSwitch (OVS)
  Generic Routing Encapsulation (GRE) tunnels 39
  Virtual Local Area Network (VLAN) 39
  VXLAN tunnels 40

configuration options, AWS CloudFormation format
description 98
HeatTemplateFormatVersion 98
mappings 98
outputs 99
parameters 98
resources 99
console access
troubleshooting 152
control
installing 117, 118
scaling 119
control services
monitoring 134-136

D

dashboard
about 2
endpoints 26
Keystone, interacting with 24-26
data
graphing 93-95
data store 89
debug command line option 141
dhclient 148
Domain Name System (DNS) 41

E

endpoints 19, 20, 26
external network
accessing 46
creating 50
network, preparing 46-49
setting up, for web interface 51-54

F

Fedora qcow cloud image
URL, for downloading 30
flavors
about 55
managing 55
floating IP addresses
managing 59

G

Generic Routing Encapsulation (GRE) tunnels
about 39
jumbo frames, enabling 40
MTU instances, lowering 39
Glance
about 4
image, downloading 30, 31
image management, troubleshooting 145
image, registering 30, 31
load balancing 122, 123
URL, for documentation 31
using 29
GlusterFS
about 6, 75
setting up 76, 77

H

Heat
about 7
instance, autoscaling 102
orchestration, troubleshooting 153
web interface support 110-113
Heat Orchestration Template (HOT) format
about 99
reference link 99
high availability 124-126
highly available database 126
Horizon 2
host checks
adding, to Nagios 128, 129

I

image
building 34, 35
downloading 30, 31
registering 30, 31
instance
autoscaling, with Heat 102
block storage, attaching 70, 71
communicating with 61
launching 58

launching, web interface used 62-66
object storage, using 85, 86
URL, for autoscaling 102
Internet Control Message Protocol (ICMP) 6

K

key pairs
about 56
managing 56, 57
Keystone
about 2
endpoints 19, 20
interacting with, in dashboard 24-26
load balancing 119-121
services 19, 20
troubleshooting 143, 144
tuning 122

L

Load Balancer as a Service (LBaaS)
setting up 102-110
load balancing
Glance 122, 123
Keystone 119-121
logging
with new user 23
Logical Volume Manager (LVM) 6

M

Maximum Transmission Unit (MTU) 39
message bus 126
meters 90
monitoring
about 127
availability monitoring 127
methods 131, 132
performance monitoring 127

N

Nagios
commands, defining 129-131
host checks, adding 128, 129
installing 128
reference link, for Mongo scripts 134
reference link, for Rabbit scripts 134
Nagios Remote Plugin Executor (NRPE) 130
Network Address Translation (NAT) 46
network fabric 38
networking
about 37, 38
installing 117, 118
Neutron 37, 38
Networking as a Service (NaaS) 5
network services
monitoring 137-139
scaling 119
Neutron
about 5, 37, 38
network fabric 38
network, troubleshooting 145-149
used, for creating network 40-42
nodes
preparing, for RDO installation 11
non-OpenStack service checks 133
Nova
about 5, 6
launching instances,
troubleshooting 149, 150

O

object file management
in web interface 83-85
object storage
about 81
creating 82
use case 81
using 82
using, on instance 85, 86
OpenStack
architecture 1, 2
components 1
other services, scaling 124
URL 2, 154
Open vSwitch (OVS)
about 5, 38
configuration 38
orchestration 97
Oz 4

P

Pacemaker
using 125
Packstack
about 9
installing 12-16
used, for installing RDO 10, 11
performance monitoring 127
pipeline 90
post-boot metadata
troubleshooting 150, 151
pseudo-folder 85
Public Key Infrastructure (PKI) 3, 23

Q

qcow2 30

R

RDO
about 9
installing, Packstack used 10, 11
nodes, preparing 11
Packstack, installing 12-16
URL 10
ring files
about 86
creating 86-88
role
about 20, 21
granting 22

S

samples 90, 91
Secure Shell (SSH) 4
security groups
about 60
managing 60
services 19, 20
Software Defined Networking (SDN) 5
stack
about 97
launching 99-102
statistics 91
staypuft 9
Swift
about 7
architecture 81, 82
object storage, troubleshooting 153

T

tail, server logs 142
tcpdump 147
template
about 97
AWS CloudFormation format 98, 99
Heat Orchestration Template
(HOT) format 99
URL 98
writing 97, 98
tenant
about 20, 21
creating 22
triple-o 9
troubleshooting
authentication 143, 144
Ceilometer Telemetry 153
Cinder block storage 152
console access 152
Glance image management 145
Heat orchestration 153
Keystone 143, 144
Neutron networking 145-149
Nova launching instances 149, 150
OpenStack installation 154
post-boot metadata 150, 151
Swift object storage 153

U

user
about 20, 21
creating 21
logging in, with new user 23
User Datagram Protocol (UDP) 40

V

virt-install 4
Virtual Extensible LAN (VXLAN) 13, 40
Virtual Local Area Network (VLAN) 39

W

web interface
Cinder volumes, managing 72-74
external network, setting up 51-54
network management 42-45
object file management 83-85
used, for launching instance 62-66
using 32, 33
with Heat 110-113

Thank you for buying
OpenStack Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

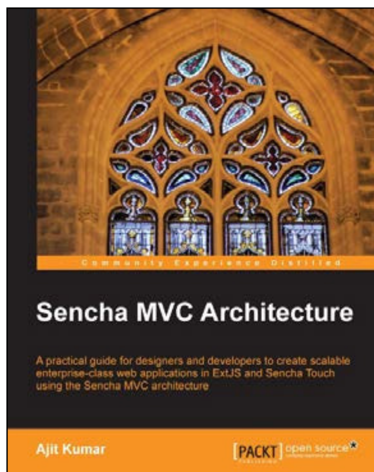
[PACKT] open source
community experience distilled
PUBLISHING

Creating Mobile Apps with Sencha Touch 2

ISBN: 978-1-84951-890-1 Paperback: 348 pages

Learn to use the Sencha Touch programming language and expand your skills by building 10 unique applications

1. Learn the Sencha Touch programming language by building real, working applications.
2. Each chapter focuses on different features and programming approaches; you can decide which is right for you.
3. Full of well-explained example code and rich with screenshots.



Sencha MVC Architecture

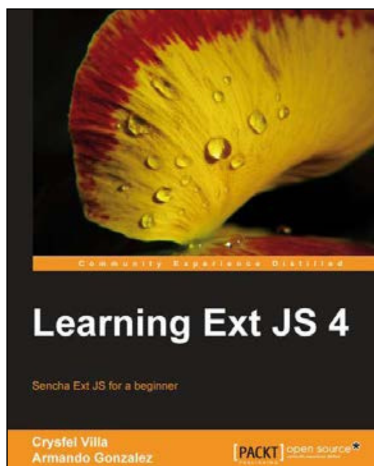
ISBN: 978-1-84951-888-8 Paperback: 126 pages

A practical guide for designers and developers to create scalable enterprise-class web applications in ExtJS and Sencha Touch using the Sencha MVC architecture

1. Map general MVC architecture concept to the classes in ExtJS 4.x and Sencha Touch.
2. Create a practical application in ExtJS as well as Sencha Touch using various Sencha MVC Architecture concepts and classes.
3. Dive deep into the building blocks of the Sencha MVC Architecture including the class system, loader, controller, and application.

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING



Learning Ext JS 4

ISBN: 978-1-84951-684-6 Paperback: 434 pages

Sencha Ext JS for a beginner

1. Learn the basics and create your first classes.
2. Handle data and understand the way it works, create powerful widgets and new components.
3. Dig into the new architecture defined by Sencha and work on real world projects.

Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-84951-686-0 Paperback: 488 pages

Over 110 easy-to-follow recipes backed up with real-life examples, walking you through basic Ext JS features to advanced application design using Sencha's Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style.
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application.

Please check **www.PacktPub.com** for information on our titles