Community Experience Distilled

Learning OpenStack

Set up and maintain your own cloud-based Infrastructure as a Service (IaaS) using OpenStack

Alok Shrivastwa Sunil Sarat

[PACKT] PUBLISHING open source* community experience distilled

www.allitebooks.com

Learning OpenStack

Set up and maintain your own cloud-based Infrastructure as a Service (IaaS) using OpenStack

Alok Shrivastwa

Sunil Sarat

BIRMINGHAM - MUMBAI

www.allitebooks.com

Learning OpenStack

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 authors, 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: November 2015

Production reference: 1261115

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

ISBN 978-1-78398-696-5

www.packtpub.com

www.allitebooks.com

Credits

Authors
Alok Shrivastwa
Sunil Sarat

Reviewers
Dr. Ketan Maheshwari
Ben Silverman

Commissioning Editor
Akram Hussain

Acquisition Editors
Harsha Bharwani
Rahul Nair

Content Development Editor
Amey Varangaonkar

Technical Editor
Deepti Tuscano

Copy Editor
Kausambhi Majumdar

Project Coordinator
Judie Jose

Proofreader
Safis Editing

Indexer
Tejal Soni

Graphics
Abhinash Sahu

Production Coordinator
Melwyn Dsa

Cover Work
Melwyn Dsa

www.allitebooks.com

About the Authors

Alok Shrivastwa is a technologist from India, currently working as a director of Cloud Services for Microland Limited in their Center of Innovation. He has a keen interest in all things physical and metaphysical and is an innovator at heart. He has worked with multiple large- and medium-sized enterprises, designing and implementing their network security solutions, automation solutions, VoIP environments, datacenter designs, public and private clouds, and integrations.

He has also created several tools and intellectual properties in the field of operationalization of emerging technologies. He has also authored several white papers and blogs on technology and metaphysical topics, in addition to writing poems in Hindi. Also, he has been a guest speaker for undergraduate engineering students in Chennai.

You can connect with him at `https://in.linkedin.com/in/alokas` or follow him at `@alok_as`.

One of the best kept secrets about any form of art, including writing, is that it is through you and not by you! I would firstly like to thank God for giving me the knowledge, then the zeal, and finally, the opportunity to share it with everyone; secondly, my mother, Seema, without whose support and efforts I wouldn't have even dared to think about writing this book; thirdly, my father, Arun; and finally, my sisters, Kawshiki and Abhabya, without whom I would be so lost. I thank Microland for giving me the opportunities to work with every technology that I could have wished for.

I would like to thank the reviewers for their patient reviews and the excellent insights through all the drafts. which have not only helped me grow as an author, but also helped to shape the book. Thank you all! All the editors who have worked on the book have really worked their magic with it and no amount of thanks would suffice. Finally, I would like to thank Packt Publishers for providing the platform to connect with all enthusiastic minds, in the form of this book. I truly hope you enjoy it.

---

www.allitebooks.com

Sunil Sarat is the vice president of Cloud and Mobility Services at Microland Ltd, an India-based global hybrid IT infrastructure services provider.

He played a key role in setting up and running emerging technology practices dealing with areas such as public/private cloud (AWS and Azure, VMware vCloud Suite, Microsoft, and OpenStack), hybrid IT (VMware vRealize Automation/Orchestration, Chef, and Puppet), enterprise mobility (Citrix Xenmobile and VMware Airwatch), VDI /app virtualization (VMware Horizon Suite, Citrix XenDesktop/XenApp, Microsoft RDS, and AppV), and associated transformation services.

He is a technologist and a business leader with an expertise in creating new practices/ service portfolios and in building and managing high performance teams, strategy definition, technology roadmaps, and 24/7 global remote infrastructure operations. He has varied experiences in handling diverse functions such as innovation/ technology, service delivery, transition, presales/solutions, and automation.

He has authored white papers, blogs, and articles on various technology- and service-related areas. Also, he is a speaker at cloud-related events and reviews technical books. He has reviewed the books *Learning Airwatch and Mastering VMware Horizon 6, Packt Publishing*.

He holds various industry certifications in the areas of compute, storage, and security and also has an MBA degree in marketing.

Besides technology and business, he is passionate about filmmaking and is a part-time filmmaker as well.

For more information, you can visit his Linkedin profile at `https://www.linkedin.com/in/sunilsarat` or follow him at `@sunilsarat`.

Firstly, I would like to thank the existence for enabling me to write this book. I would like to thank my family—my mother, Ratna, my wife, Abhaya, and my twins, Advika and Agnika, for supporting me throughout. Thanks to my friends KK, Syed, Samina, and Mayank for their encouragement.

I would like to thank Microland for all the exposure and support provided, which was instrumental for me to write this book. I am grateful to my coauthor, Alok Shrivastwa, and last but not least Packt Publishers for the opportunity and also for the guidance.

---

www.allitebooks.com

About the Reviewers

Dr. Ketan Maheshwari is a postdoctoral researcher in the Mathematics and Computer Science Division at Argonne National Laboratory. He earned his doctoral degree in 2011 from the University of Nice with the highest honors. His research interests include distributed and parallel computing with an emphasis on workflow programming of high-level science and engineering applications.

Dr. Maheshwari's recent paper on clouds, coauthored with colleagues and collaborators, has won the best paper award at the 2014 HPDC workshops. He has coauthored several papers in the area of HPC and distributed and cloud computing in international journals and conferences. He currently serves as an editor in Future Generation Computer Systems.

Ben Silverman has worn many hats in his career—Unix administrator, Engineer, Programmer, Consultant, IT manager, and Enterprise OpenStack Architect. Currently, Ben is currently a Cloud Architect at Mirantis where he works with product managers, project managers, engineering teams, customers, and sales teams to develop custom cloud solutions for customers based on the OpenStack platform and third-party partners.

Previously, Ben was the Lead OpenStack Architect and Engineer at American Express where he led the challenge of building and running American Express' first OpenStack private cloud. Previously, Ben was the Lead Technical Architect of a four-year, $1 billion data center migration project. During this engagement, Ben faced the challenge of designing strategies to move American Express' production datacenter from the West Coast to the East Coast.

Ben also holds a Master of Science in Information Management degree from Arizona State University's W.P. Carey School of Business and enjoys playing the banjo in his very limited free time.

I'd like to thank my wife, Jennifer, and my sons, Jason and Brayden, for all their support for my late nights and long days.

---

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

www.allitebooks.com

Table of Contents

www.allitebooks.com

Preface

The cloud is the new IT paradigm, and has moved beyond being probable to being inevitable. No one can ignore it. Organizations are embracing the cloud for various reasons such as agility, scalability, capex reduction, and a faster time to market their products and services. There are choices available in terms of the following:

- Ownership and control, with the options of public, private, or hybrid cloud
- Delivery model, with the options of SaaS (Software as a Service), PaaS (Platform as a Service), and IaaS (Infrastructure as a Service).

If the focus of an organization (or a cloud service provider) is on Infrastructure as a Service, then one needs to look at ways to build a cloud and deliver IaaS to their users. The cloud operating system, or cloud control layer or cloud software system or simply put cloud orchestrator, is at the heart of building a cloud delivering IaaS. While there are many choices available as far as the cloud orchestrator goes, OpenStack is a popular choice in the open source segment.

OpenStack is rapidly gaining momentum and is poised to become the leader in this segment. Therefore, it becomes imperative for organizations and IT managers / support teams to have these critical OpenStack skills. The challenge, however, stems from the fact that OpenStack is not a single product, but is a collection of multiple open source projects. Therefore, the challenge really is to have an understanding of these projects independently, along with their interactions with the other projects and how they all are orchestrated together. While there is documentation available from the OpenStack project, it is important to have the necessary knowledge to stitch all of these services/components together and build your own cloud. There are not many books/reading material that are available out there to address this challenge.

This book is an attempt to provide all the information that is just about sufficient to kickstart your learning of OpenStack and build your own cloud. In this book, you will be introduced to all major OpenStack services, the role they play, installation, and the basic configuration and troubleshooting of each of these services. This book takes a more practical-oriented approach to learning, as the knowledge from each chapter will culminate in you being able to build your own private cloud by the time you finish reading this book. We hope you will enjoy reading this book and more importantly find it useful in your journey towards learning and mastering OpenStack.

What this book covers

Chapter 1, *An Introduction to OpenStack*, introduces the concepts of the cloud, IaaS, and its building blocks. It talks about the core component of the cloud, which is the Orchestrator, in a bit more detail, looks at various orchestrators available in the market, and how they compare to OpenStack. This chapter provides a brief history of OpenStack and introduces its services as well.

Chapter 2, *Authentication and Authorization Using Keystone*, introduces the concepts of identity in Keystone. It also deals with the architecture of Keystone and how Keystone provides identity, token, catalog, and policy services. This is followed by step-by-step instructions to install, configure, and troubleshoot Keystone.

Chapter 3, Storing and Retrieving Data and Images using Glance, Cinder, and Swift introduces the concepts of block and object storage in the context of OpenStack. It introduces the architecture of Cinder, Swift, and Glance. This is followed by step-by-step instructions to install, configure, and troubleshoot all these storage services.

Chapter 4, Building your Cloud Fabric Controller Using Nova, introduces the concept of a Cloud Computing Fabric controller and "compute as a service". It also introduces the architecture of Nova and the different types of controllers that it has. This is followed by step-by-step instructions to install, configure, and troubleshoot Nova.

Chapter 5, *Technology-Agnostic Network Abstraction Using Neutron*, introduces the concepts of "Networking as a Service" and "Software Defined Networking". It talks about the networking challenges that are introduced by the cloud and how Neutron handles them. It then introduces the architecture of Neutron. This is followed by step-by-step instructions to install, configure, and troubleshoot Neutron.

Chapter 6, *Building Your Portal in the Cloud*, introduces the need for dashboards in a cloud environment. It also introduces the architecture of Horizon and the terminologies used in the context of Horizon such as panels, tabs, dashboards, workflows, actions, tables, URLs, and views. This is followed by step-by-step instructions to install, configure, and troubleshoot Horizon.

Chapters 7, *Your OpenStack Cloud in Action*, stitches all the pieces together and presents how all of the components come together to provide IaaS to users while highlighting the role each of these components plays. This also introduces aspects such as user and tenant management using GUI and CLI, network management, services request, and template creation.

Chapter 8, *Taking Your Cloud to the Next Level*, introduces two OpenStack optional components, Ceilometer and Heat. It discusses Heat, Heat API, Heat CF API, Heat Engine, and Heat Orchestration Templates. This chapter also discusses data collection, alarms, and meters in Ceilometer, and how can this be used to provide billing and usage reporting.

Chapter 9, *Looking Ahead*, introduces the various distributions of OpenStack and vendor offerings based on OpenStack. It also discusses different use cases where OpenStack is being used and concludes by briefly touching upon the roadmap.

Appendix, *New Releases*, introduces the major differences between the last three releases of OpenStack.

What you need for this book

The following are the software, hardware and OS requirements:

- Software required: Ubuntu OpenStack (The Juno release)
- Hardware required: Four servers, physical or virtual, with 4 GB RAM / 2 core processors each
- OS required: Ubuntu 14.04 LTS

Who this book is for

This book is intended for IT administrators/architects/managers, with a basic knowledge of IaaS and cloud computing. It is assumed that the readers have a firm grasp of the Linux operating system, virtualization, networking, and storage principles. This book will help any reader who is trying to build and enhance their skills with OpenStack. We believe that this is the right kind of opportunity for all those readers who have embarked on a journey to build OpenStack skills and enhance their career in the next generation cloud world.

www.allitebooks.com

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 roles are defined in the `policy.json` file located at `/etc/keystone/policy.json`."

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

```
sudo apt-get install software-properties-common

sudo apt-key adv --recv-keys --keyserver
hkp://keyserver.ubuntu.com:80 0xcbcb082a1bb943db
```

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: "If you need to change the defaults for every project, click on **Update Defaults** and change to the values you need "

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
An Introduction to OpenStack

Enterprises traditionally ran their IT services by running appropriate applications on a set of infrastructures and platforms. These were comprised of physical hardware in terms of compute, storage, and network along with software in terms of hypervisors, operating systems, and platforms. A set of experts from infrastructure, platform, and application teams would then put the pieces together and get a working solution tailored to the needs of the organization.

With the advent of virtualization and later on cloud, things have changed to a certain extent, primarily in the way things are built and delivered. Cloud, which has its foundations in virtualization, delivers a combination of relevant components as a service; be it **Infrastructure as a Service** (**IaaS**), **Platform as a Service** (**PaaS),** or **Software as a Service** (**SaaS**). In this book, we will only discuss how to provide a system with IaaS using an OpenStack-based private cloud. The key aspect of providing a system with IaaS is cross-domain automation. The system that helps us achieve this is called a Cloud Service Orchestrator or Cloud Platform or Cloud Controller. For the purposes of this book, we will refer to OpenStack as the Cloud Service Orchestrator. The Cloud Service Orchestrator or, simply put, the orchestrator is primarily responsible for the following:

- The stitching together of hardware and software to deliver a defined service (in the context of our book, IaaS)
- Automating the workflows that are required to deliver a service

Thus, in a cloud environment, the most important component is the orchestrator. There are several orchestrators; both free and open-source (FOSS) and commercial, which can help turn your virtualized IT infrastructure into a cloud.

Some of the choices in the FOSS segment for the orchestrators are as follows:

- OpenStack
- Apache CloudStack
- Open Nebula

Some choices of commercial orchestrators are as follows:

- VMware vRealize Automation and vRealize Orchestrator
- VMware vCloud Director
- Cisco Intelligent Automation for the cloud (CIAC) and UCS Director
- Microsoft Opalis and Systems Center
- BMC Atrium

In this book, we embark on a journey to understand the concepts, to install and configure the components of OpenStack, and finally, to build your own cloud using OpenStack. At the time of writing this book, OpenStack has been by far the most famous and widely adopted FOSS orchestrator or Cloud Software Platform in the market and the most comprehensive offering that provides IaaS among FOSS alternatives.

In this chapter, we will cover the following:

- The differences between commercial orchestrators and FOSS orchestrators, and where each of these types of orchestrators fit well in today's world
- The basic building blocks of a private cloud and how OpenStack is different from commercial orchestrators in building a private Cloud
- The key differences between commercial orchestrators and OpenStack
- An introduction to OpenStack architecture, services, and service dependencies
- A preparation for OpenStack setup where we discuss the details of a test setup, which will lead us on a journey of building our own private cloud using OpenStack

Choosing an orchestrator

There are some key differences between commercial orchestrators, such as vRealize Automation and CIAC, and FOSS orchestrators, such as OpenStack. While both of them attempt to provide IaaS to users, it is important to understand the difference between both the types of orchestrator in order to appropriately design your Cloud.

Let's begin with commercial orchestrators; these provide a base IaaS to their users. They normally sit on top of a virtualized environment and enable an automated provisioning of compute, storage, and network, even though the extent of automation varies. As a part of the toolset, they also typically have a workflow engine, which in most cases provides us with an extensibility option.

The commercial orchestrators are a better choice when the entire orchestration needs to be plugged in to the current IT processes. They work wonderfully well when extensibility and integration are major tasks of the cloud environment, which is typically seen in large enterprises given the scale of operations, the type of business critical applications, and the maturity of IT processes.

In such large enterprises, in order to take full advantage of the private cloud, the integration and automation of the orchestrator in the IT systems of the company becomes necessary. This kind of orchestration is normally used when minimum changes are anticipated to be made to the applications. A primary use case of this is IaaS, where virtual machines are provisioned on a self-service basis and a very small learning curve is involved.

FOSS orchestrators are less extensible, but more standardized in terms of offerings. They offer standardized services that a user is expected to use as building blocks to offer a larger solution. In order to take full advantage of the FOSS orchestrators, some amount of recoding of applications is required as they need to make use of the newly offered services. The use cases here are both IaaS and PaaS (for example, Database as a Service, Message Queue as a Service, and so on).

For this reason, the APIs that are used among the FOSS orchestrators need to have some common ground. This common ground that we are talking about here is **Amazon Web Services** (**AWS**) API compatibility, as Amazon has emerged as the gold standard as far as the service-oriented cloud architecture is concerned. At the time of writing the book, OpenStack Nova still had AWS EC2 API compatibility, but this may be pushed out to the StackForge project.

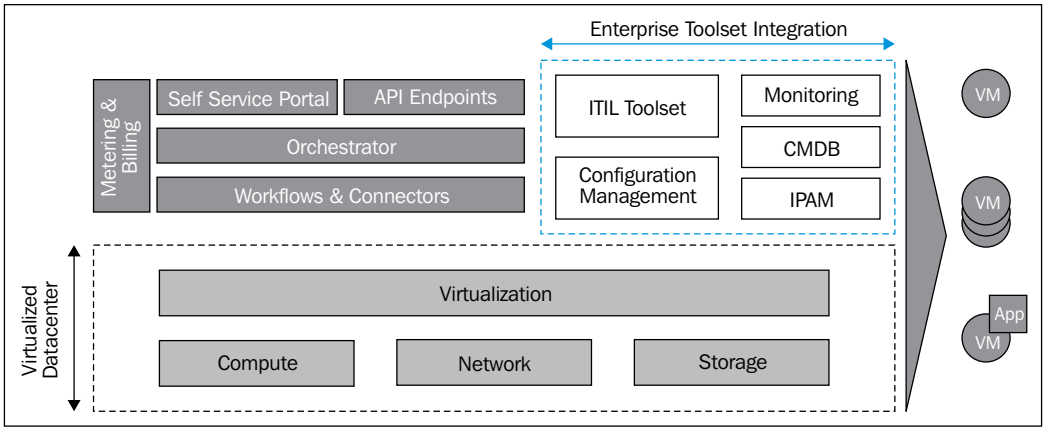
- Most FOSS orchestrators provide us with a way to use Amazon APIs wherever possible. It is for this reason that in the next section, we will compare the services available in OpenStack to the equivalent services offered by AWS.

Building a private cloud

Clouds fall under different categories depending on the perspective. If we look at it from an ownership and control standpoint, they will fall under private, public, hybrid, and community cloud categories. If we take a service perspective, it could be IaaS, PaaS, or SaaS. Let's look at the basic building blocks of a private cloud and understand how commercial orchestrators fit in vis-à-vis OpenStack.

Commercial orchestrators

The following block diagram shows the different building blocks of a cloud that are normally seen in a private implementation with a commercial orchestrator:



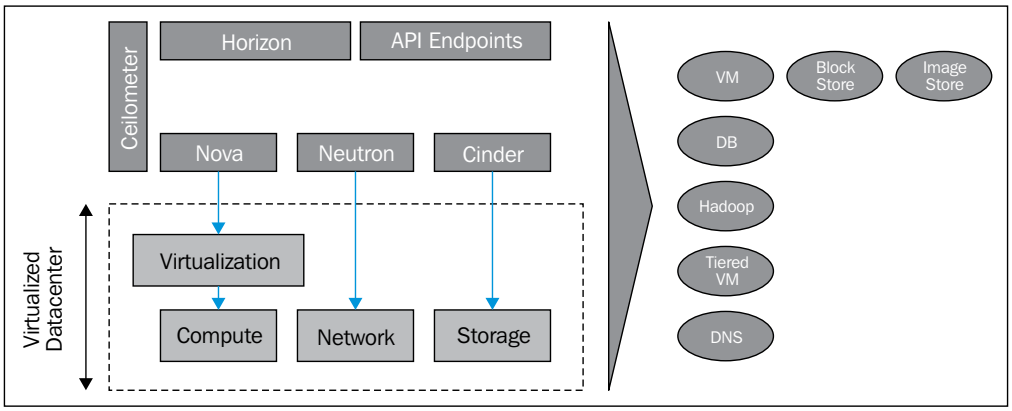
A private cloud with a commercial orchestrator

As we can see, in this private cloud setup, additional blocks such as **Self Service Portal**, **Metering & Billing**, and **Workflows & Connectors** sit on top of an already existing virtualized environment to provision a virtual machine, a stack of virtual machines, or a virtual machine with some application installed and configured over it.

While most of the commercial orchestrators are extensible, some of them have prebuilt plugins or connectors to most commonly used enterprise toolsets.

OpenStack

OpenStack doesn't natively support integration with enterprise toolsets, but in lieu of this, it provides more standardized services. OpenStack feels and behaves more like a public cloud inside an enterprise and provides more flexibility to a user. As you can see in the following diagram, apart from VM provisioning, services such as database, image storage, and so on are also provisioned:



A private cloud with OpenStack

Please note that some of these services, which are provided as a part of the standard offering by OpenStack, can be also be orchestrated using commercial orchestrators. However, this will take some efforts in terms of additional automation and integration.

When to choose OpenStack?

So the big question is: under what circumstances should we choose OpenStack over the commercial orchestrators or vice versa? Let's look at the following table that compares the features that are significantly different.

Please note that the ease of installation and management are not covered in the following table:

Feature	OpenStack	Commercial orchestrator
Identity and access management*	Yes	Yes
Connectivity to enterprise toolsets	Not natively (Possible with ManageIQ)	Yes
Flexibility to the user	Yes	Somewhat

Feature	OpenStack	Commercial orchestrator
Enterprise control	Not natively (Possible with ManageIQ)	Yes
Standardized prebuilt services	Yes	No (Except virtual machines)
EC2-compatible API	Yes	No

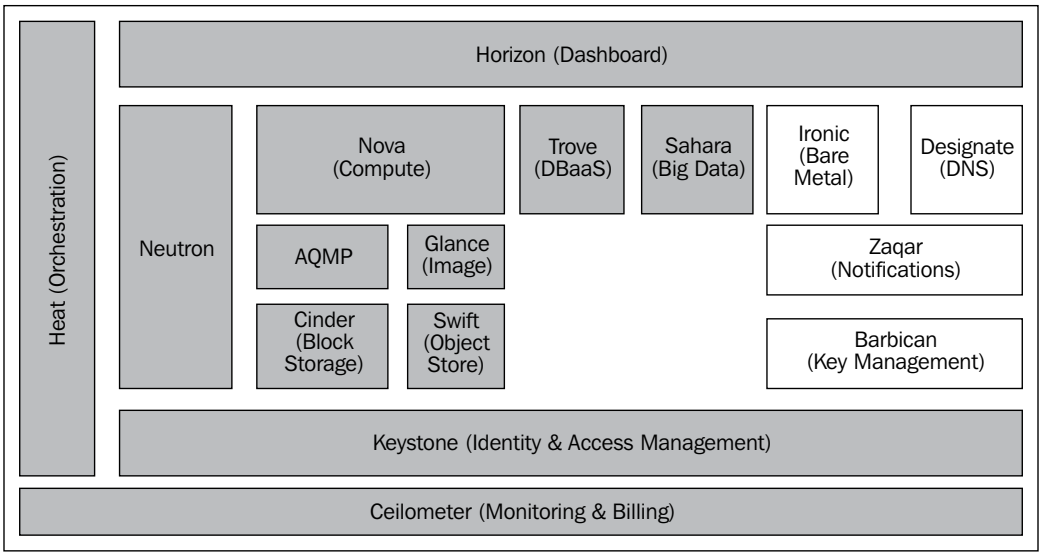
So based on the previous table, OpenStack is an amazing candidate for an enterprise dev-test cloud and for providing public cloud-like services to an enterprise, while reusing existing hardware.

The currently supported stable release of OpenStack is codenamed Liberty. This book will deal with Juno, but the core concepts and procedures will be fairly similar to the other releases of OpenStack. The differences between Juno, Kilo, and Liberty and the subtle differences between the installation procedures of these will be dealt with in the *Appendix* section of the book.

OpenStack has a very modular architecture. OpenStack is a group of different components that deliver specific functions and come together to create a full-fledged orchestrator.

OpenStack architecture

The following architecture diagram explains the architecture of the base components of the OpenStack environment. Each of these blocks and their subcomponents will be dealt with in detail in the subsequent chapters:



An OpenStack block diagram

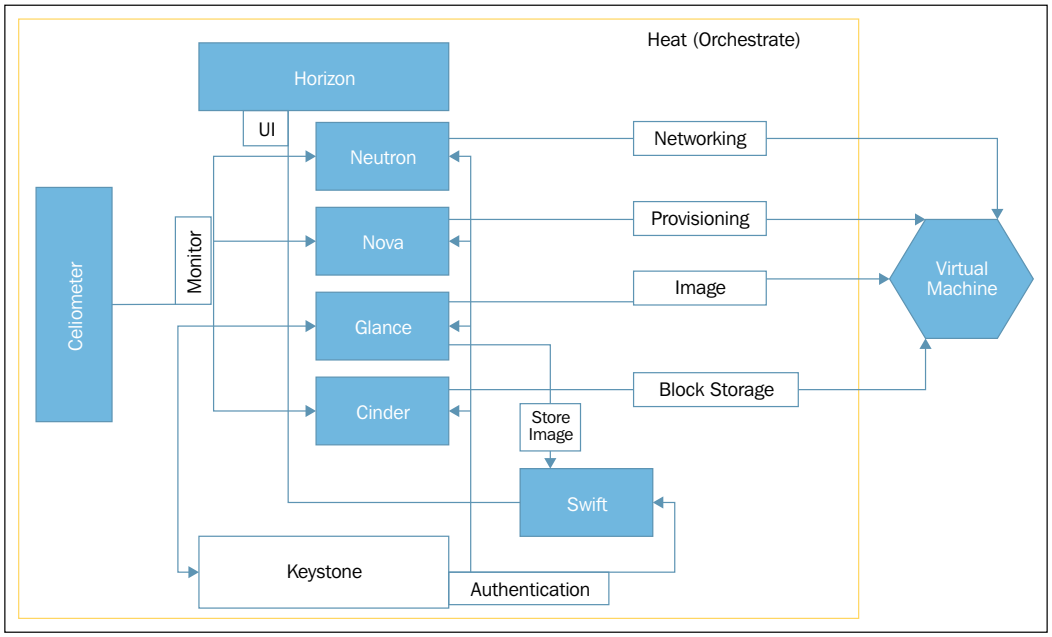
The gray boxes show the core services that OpenStack absolutely needs to run. The other services are optional and are called Big Tent services, without which OpenStack can run, but we may need to use them as required. In this book, we look at the core components and also look at Horizon, Heat, and Ceilometer in the Big Tent services.

Each of the previously mentioned components has their own database. While each of these services can run independently, they form relationships and have dependencies among each other. As an example, **Horizon** and **Keystone** provide their services to the other components of OpenStack and should be the first ones to be deployed.

www.allitebooks.com

Service relationships

The following diagram expands on the preceding block diagram and depicts the different relationships amongst the different services:



Service relationships

The service relationship shows that the services are dependent on each other. It is to be noted that all the services work together in harmony to produce the end product as a **Virtual Machine** (**VM**). So the services can be turned on or off depending on what kind of virtual machine is needed as the output. While the details of the services are mentioned in the next section, if, as an example, the VM or the cloud doesn't require advanced networking, you may completely skip the installation and configuration of the **Neutron** service.

Services and releases history

Not all the services of the OpenStack system were available from the first release. More services were added as the complexity of the orchestrator increased. The following table will help you understand the different services that can be installed, or should you choose to install another release in your environment:

Release name	Components
Austin	Nova, Swift
Bexar	Nova, Glance, Swift
Cactus	Nova, Glance, Swift
Diablo	Nova, Glance, Swift
Essex	Nova, Glance, Swift, Horizon, Keystone
Folsom	Nova, Glance, Swift, Horizon, Keystone, Quantum, Cinder
Grizzly	Nova, Glance, Swift, Horizon, Keystone, Quantum, Cinder
Havana	Nova, Glance, Swift, Horizon, Keystone, Neutron, Cinder, Heat, Ceilometer
Icehouse	Nova, Glance, Swift, Horizon, Keystone, Neutron, Cinder, Heat, Ceilometer, Trove
Juno	Nova, Glance, Swift, Horizon, Keystone, Neutron, Cinder, Heat, Ceilometer, Trove, Sahara
Kilo	Nova, Glance, Swift, Horizon, Keystone, Neutron, Cinder, Heat, Ceilometer, Trove, Sahara, Ironic, Zaqar, Manila, Designate, Barbican
Liberty	Nova, Glance, Swift, Horizon, Keystone, Neutron, Cinder, Heat, Ceilometer, Trove, Sahara, Ironic, Zaqar, Manila, Designate, Barbican, Murano, Magnum, Kolla, Congress

The OpenStack services and releases

At the time of writing, the only fully supported releases were Juno, Kilo, and Liberty. Icehouse is only supported from the security updates standpoint in the OpenStack community. There are, however, some distributions of OpenStack that are still available on older releases such as that of Icehouse. (You can read more about different distributions in the last chapter of the book.).

Service functions

It is important to know about the functions that each of these services performs. We will discuss the different services of OpenStack. In order to understand the functions more clearly, we will also draw parallels with the services from AWS. So if you ever want to compare your private cloud with the most used public cloud, you can.

Please refer to the preceding table in order to see the services that are available in a particular OpenStack release.

Keystone

This service provides identity and access management for all the components of OpenStack. It has internal services such as identity, resource, assignment, token, catalog, and policy, which are exposed as an HTTP frontend.

So if we are logging in to Horizon or making an API call to any component, we have to interact with the service and be able to authenticate ourselves in order to use it. The policy services allow the setting up of granular control over the actions allowed by a user for a particular service. The service supports federation and authentication with an external system such as an LDAP server.

This service is equivalent to the IAM service of the AWS public cloud.

Horizon

Horizon provides us with a dashboard for both self-service and day-to-day administrative activities. It is a highly extensible Django project where you can add your own custom dashboards if you choose to. (The creation of custom dashboards is beyond the scope of this book and is not covered here).

Horizon provides a web-based user interface to OpenStack services including Nova, Swift, Keystone, and so on.

This can be equated to the AWS console, which is used to create and configure the services.

Nova

Nova is the compute component of OpenStack. It's one of the first services available since the inception as it is at the core of IaaS offering.

Nova supports various hypervisors for virtual machines such as XenServer, KVM, and VMware. It also supports **Linux Containers** (**LXC**) if we need to minimize the virtualization overhead. In this book, we will deal with LXC and KVM as our hypervisors of choice to get started.

It has various subcomponents such as compute, scheduler, xvpvncproxy, novncproxy, serialproxy, manage, API, and metadata. It serves an EC2 (AWS)-compatible API. This is useful in case you have a custom system such as ITIL tool integration with EC2 or a self-healing application. Using the EC2 API, this will run with minor modifications on OpenStack Nova.

Nova also provides proxy access to a console of guest virtual machines using the VNC proxy services available on hypervisors, which is very useful in a private cloud environment. This can be considered equivalent to the EC2 service of AWS.

Glance

Glance service allows the storage and retrieval of images and corresponding metadata. In other words, this will allow you to store your OS templates that you want to be made available for your users to deploy. Glance can store your images in a flat file or in an object store (such as Swift).

Swift

Swift is the object storage service of OpenStack. This service is primarily used to store and retrieve **Binary Large Object** (**BLOBs**). It has various subservices such as ring, container server, updater, and auditors, which have a proxy server as their frontend.

The swift service is used to actually store Glance images. As a comparison, the EC2 AMIs are stored in your S3 bucket.

The swift service is equivalent to the S3 storage service of AWS.

Cinder

Cinder provides block storage to the Nova VMs. Its subsystems include a volume manager, a SQL database, an authentication manager, and so on. The client uses AQMP such as Rabbit MQ to provide its services to Nova. It has drivers for various storage systems such as Cloud Byte, Gluster FS, EMC VMAX, Netapp, Dell Storage Centre, and so on.

This service provides similar features to the EBS service of AWS.

Neutron

Previously known as Quantum, Neutron provides networking as a service. There are several functionalities that it provides such as Load Balancer as a Service and Firewall as a Service. This is an optional service and we can choose not to use this, as basic networking is built into Nova. Also, Nova networking is being phased out. Therefore, it is important to deal with Neutron, as 99 percent of OpenStack implementations have implemented Neutron in their network services.

The system, when configured, can be used to create multi-tiered isolated networks. An example of this could be a full three-tiered network stack for an application that needs it.

This is equivalent to multiple services in AWS such as ELB, Elastic IP, and VPC.

Heat

Heat is the core orchestration service of the orchestrator. What this means is that you can script the different components that are being spun up in an order. This is especially helpful if we want to deploy multicomponent stacks. The system integrates with most of the services and makes API calls in order to create and configure different components.

The template used in Heat is called Heat Orchestrator Template (HOT). It is actually a single file in which you can script multiple actions. As an example, we can write a template to create an instance, some floating IPs and security groups, and even create some users in Keystone.

The equivalent of Heat in AWS would be the cloud formation service.

Ceilometer

Ceilometer service is used to collect metering data. There are several subsystems in the Ceilometer such as polling agent, notification agent, collector, and API. This also allows the saving of alarms abstracted by a storage abstraction layer to one of the supported databases such as Mongo DB, Hbase, or SQL server.

Trove

Trove is the Database as a Service component of OpenStack. This service uses Nova to create the compute resource to run DBaaS. It is installed as a bunch of integration scripts that run along with Nova. The service requires the creation of special images that are stored in Glance.

This is equivalent to the RDS service of AWS.

Sahara

Sahara service is the Big Data service of OpenStack; it is used to provision a Hadoop cluster by passing a few parameters. It has several components such as Auth component, Data Access Layer, Provisioning Engine, and Elastic Data Processing.

This is very close to getting the MapReduce AWS service in your very own cloud.

Designate

The Designate service offers DNS services equivalent to Route 53 of the AWS. The service has various subsystems such as API, the Central/Core service, the Mini DNS service, and Pool Manager. It has multiple backend drivers that can be used, examples being PowerDNS, BIND, NSD, and DynECT. We can create our own backend drivers as well.

Ironic

The Ironic service allows bare metal provisioning using technologies such as the PXE boot and the **Intelligent Platform Management Interface** (**IPMI**). This will allow bare metal servers to be provisioned provided we have the requisite drivers for them.

Please remember that the requisite networking elements have to be configured, for example, the DNS, DHCP configuration and so on, which are needed for the PXE boot to work.

Zaqar

Zaqar is the messaging and notification service of OpenStack. This is equivalent to the SNS service from AWS. It provides multitenanted HTTP-based messaging API that can be scaled horizontally as and when the need arises.

Barbican

Barbican is the key management service of OpenStack that is comparable to KMS from AWS. This provides secure storage, retrieval, provisioning and management of various types of secret data such as keys, certificates, and even binary data.

Manila

Manila provides a shared filesystem as a service. At the moment, it has a single subcomponent called the manila-manage. This doesn't have any equivalent in the AWS world yet. This can be used to mount a single filesystem on multiple Nova instances, for instance a web server with shared assets, which will help to keep the static assets in sync without having to run a block-level redundancy such as DRBD or continuous rsyncs.

Murano

Murano is an application catalog, enabling application developers and cloud administrators to publish various cloud-ready applications in a catalog format. This service will use Heat at the backend to deliver this and will only work on the UI and API layer.

Magnum

Magnum introduces Linux Containers such as Dockers and Kubernetes (by Google) to improve migration option. This service is in some ways like Trove, it uses an image with Docker installed on it and orchestrates Magnum with Heat. It is effectively **Container as a Service** (**CaaS**) of OpenStack.

Kolla

Kolla is another project that is focused on containers. While it did make its first appearance in Kilo, it was majorly introduced in the Liberty release. This is aimed at better operationalization by containerizing OpenStack itself. That means, we can now run the OpenStack services in containers, and thereby make governance easier.

At the time of writing, the Kolla project supported services such as Cinder, Swift, Ceph, and Ironic.

Congress

Congress is another project focused on governance. It provides Policy as a Service, which can be used for compliance in a dynamic infrastructure, thereby maintaining the OpenStack components to be compliant to the enterprise policy.

Service dependency maps

The following table shows the dependency of services. The **Dependent on** column shows all the services, which are needed for successful installation and configuration of the service. There might be other interactions with other services, but they are not mentioned here:

Service name	Core service	Dependent on
Keystone	True	None
Horizon	False	Keystone

Service name	Core service	Dependent on
Glance	True	Swift Keystone Horizon
Swift	True	Keystone
Nova	True	Keystone Horizon Glance Cinder (Optional) Neutron (Optional)
Heat	False	Keystone
Cinder	False	Keystone
Neutron	False	Keystone Nova
Ceilometer	False	Keystone
Trove	False	Keystone Nova Glance
Sahara	False	Keystone Nova Glance Swift Keystone
Magnum	False	Heat Nova Glance Swift Keystone
Murano	False	Heat

Service dependency

Preparing for the OpenStack setup

In the remainder of this book, we will be installing and configuring various OpenStack components. Therefore, let's look at the architecture that we will follow in the remainder of the book and what we need to have handy.

While we can set up all the components of the OpenStack on a single server, it will not be close to any real-life scenario, so taking this into consideration, we will do a minimal distributed installation. Since this book is intended to be a beginner's guide, we shall not bore ourselves with cloud architecture questions.

Selecting the services

As we are aware by now that OpenStack is made up of individual components, we need to be careful in selecting the appropriate services. As we have already seen in the dependency maps table, some services are sort of mandatory and the others are optional depending on the scenario. Too many services and you complicate the design, too little and you constrain it; so it is imperative that we strike a good balance. In our case, we will stick to the basic services:

- Keystone
- Horizon
- Nova
- Cinder
- Swift
- Glance

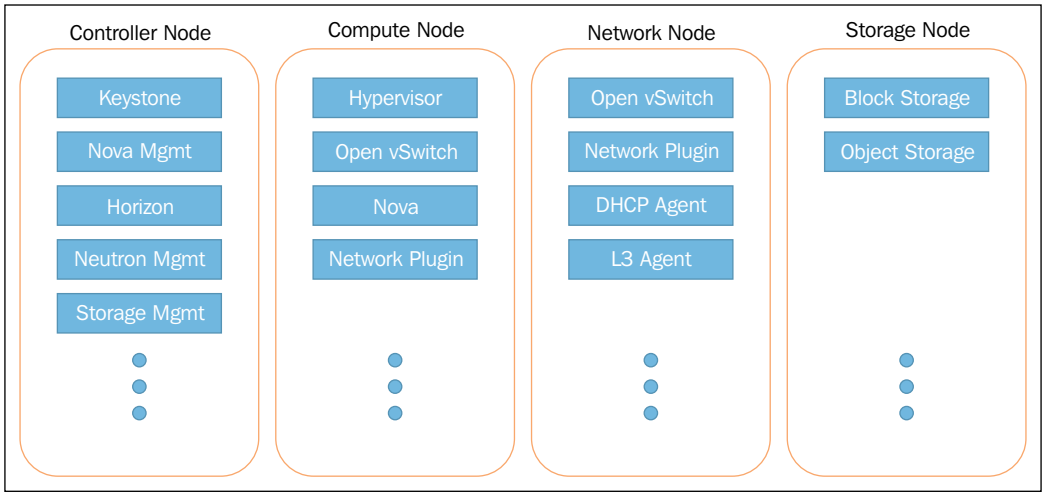
In the optional section, we will choose Neutron. This should help us in getting a pretty robust cloud with the essential features rolled out in no time.

Service layout

We will be installing these components on virtual machines for our learning purposes; we will use four different virtual machines to run our cloud:

- Controller node
- Network node
- Compute node
- Storage node

The following diagram shows the kind of services that will be hosted in each of the different nodes in the rest of the book. We will identify the servers with the previously mentioned names:



The OpenStack service layout

Controller node

The controller node will house the manager services for all the different OpenStack components such as message queue, Keystone, image service, Nova management, and Neutron management.

Network node

The network node server will house Neutron components such as the DHCP Agent, the L3 Agent, and Open vSwitch. This node will provide networking to all the guest VMs that spin up in the OpenStack environment.

Compute node

The compute node will have the hypervisor installed on itself. For the purpose of this setup, we will use LXC or KVM to keep things simple. It also houses network agents.

www.allitebooks.com

Storage node

The storage node will provide block and object storage to the rest of the OpenStack services. This will be the node that needs to be connected to the iSCSI storage in order to create different blocks.

Operating system

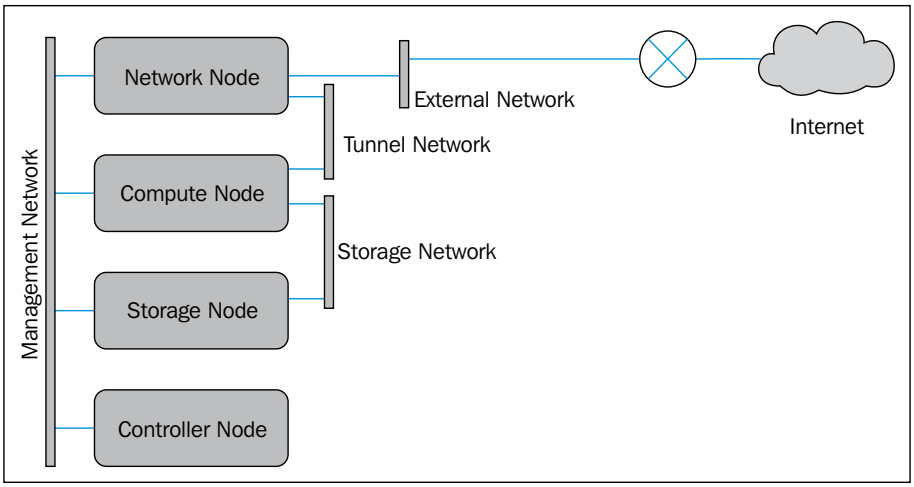
We will use Linux Ubuntu 14.04 as the operating system of choice to install and configure the different components. All the previously mentioned nodes should be running Ubuntu.

Network layout

Since we are going to use Neutron, the following network architecture needs to be followed:

- **Management network**: This network is available on all the OpenStack servers.
- **Tunnel network**: This network is used to tunnel the traffic between the compute nodes and the network node and is available on all the compute and the network nodes. There can be more than one if we are going for a multi-tiered environment.
- **Storage network**: This connects the compute and storage nodes. This is used as a separate network to ensure that there is no network congestion.
- **External network**: This is connected only to the network node and can be accessed using Neutron. The elastic IPs are configured on this network.

The following diagram shows the different connections in our network. The compute node is connected to all the networks except the external network. It is to be noted that the storage and the tunnel network can be completely internal networks. The management network is primarily the one that needs to be accessible from the LAN of the company, as this will be the network that the users will need to reach in order to access the self-service portal:



Network connectivity

For the purpose of learning, let's set up the network ranges that we will use in our installation. The following is the table of the network range:

Network Name	**IP Range**
Management Network	172.22.6.0/24
Tunnel Network	10.0.0.0/24
Storage Network	192.168.10.0/24
External Network	192.168.2.0/24

Network ranges

Since we are using this in the lab network, the external network is assumed and will need to be changed depending on the routing rules.

Summary

In this chapter, we were introduced to orchestrators, both commercial and FOSS. At a very high level, we looked at the differences between these two types of orchestrators and the appropriate use cases for OpenStack. We also looked at the basic building blocks of a private cloud and their correlation in the OpenStack world. We looked at the OpenStack architecture and services. And finally, we covered the lab setup that would be required to learn the deployment of your private cloud using OpenStack.

We start our journey in the next chapter by learning to install and configure the common components that form the basis of most of the OpenStack services. The key topic covered, however, would be installation and configuration of Keystone, which is the core authentication and authorization service of OpenStack

2
Authentication and Authorization Using Keystone

Most of the OpenStack components have a basic in-built authentication mechanism, which is adequate for them to function on their own. However, when they have to come together, Keystone forms the bridge, a common platform for authentication and authorization.

Keystone was launched in the Essex release and has been deemed a core component of the OpenStack deployment ever since. In this chapter, we will understand in some detail the following:

- The Keystone architecture and the subsystems
- Installing the prerequisite common components
- Installing Keystone
- Initial configuration
- Basic troubleshooting

Please be advised that this will be installed and configured on the controller node.

The entire installation and configuration of common components and the core Keystone service takes between 60-90 minutes.

Identity concepts in Keystone

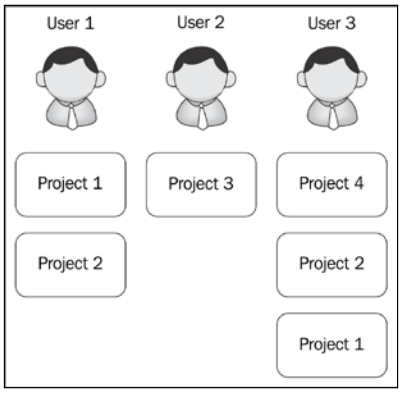
Let's understand identity-related concepts that are used in Keystone.

User

User represents a person or a service with a set of credentials such as a user name, password, or username and an API key. A user needs to be a member of at least one project, but can be a part of multiple projects.

Project (or tenant)

A group of users in OpenStack is called a project or a tenant. Both of these terms are used interchangeably and mean the same thing. Please be advised that tenant is the new terminology, and the term project has seeped in from the initial days when Keystone was not available. The policies and quotas are all applied at the project or the tenant level



As shown in the figure, users can be a part of one or more projects.

Role

The role determines what the user is allowed to do. This is controlled by the policy subsystem of the Keystone service. The roles are defined in the `policy.json` file located at `/etc/keystone/policy.json`.

By default, there are only two roles that come with OpenStack:

- admin
- member

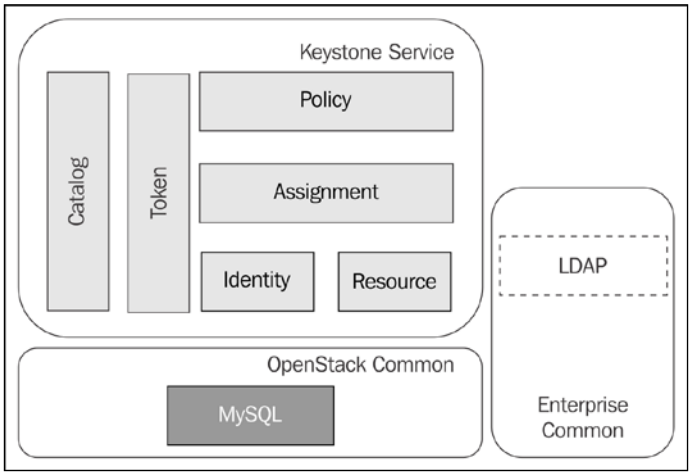
As the name implies, the admin role has all the privileges and a member comes with limited privileges. In a general deployment, these two are enough with most users being members.

Roles are associated with the users, and hence a user associated with an admin role is granted privileges across all the projects that the user is assigned to. Hence, please use the admin role carefully.

In order to assign a user to multiple projects, we will need to create a role and add it to the user project pair. We will see this in the later part of the book.

Architecture and subsystems

Keystone comprises a bunch of services. We will understand them and their functionalities; before this, let's take a quick look at the Keystone architecture:



In the preceding diagram, you will see the different subsystems of the service and the common components that will be shared with the other components of OpenStack. The MySQL server will be used by most of the components of the OpenStack, and hence it is classified as **OpenStack Common**. The **LDAP** service is optional and will be common from an enterprise tool set perspective.

Identity

Identity verifies the credentials and data of the users and user groups. It can store the user data in the local database (MySQL), or it can connect to the LDAP to get this data. If the local database is used, this service is capable of performing the CRUD (Create, Read, Update, and Delete) operations.

Resource

Resource is similar to identity, but it does this for resources, such as projects and domains. The LDAP-versus-local-database concepts that were discussed in the preceding section hold true in this case and for assignment as well.

Assignment

The assignment service categorizes different users and resources by providing information about roles that are assigned to an identity or a resource.

Policy

Policy ties together the role name and what they are authorized to do. This is an authorization engine, with a rule management interface.

For instance, a user called John Doe is trying to access the OpenStack environment and tries to log in; the identity subsystem will authenticate him either locally or using LDAP (as configured). The assignment sub-system will provide the different roles assigned to John, and the policy sub-system will provide the action he is allowed to perform.

Token

Token manages the tokens that are assigned to a user once they authenticate so as to provide a single sign-on experience across all different components of OpenStack.

Catalog

Catalog provides the endpoint registry that is used in order to discover the various end points. This is internally used by different services and also by the API, should we choose to use the API in order to interact with the services.

Installing common components

There are two common components (database and messaging broker) that are used by most of the OpenStack services. We will see how to install and configure them. These are required before we go ahead and install Keystone. Please note that this will be done only once. If deploying in an enterprise production environment, chances are these components may already be present and could be shared with other applications.

Setting up the database

For our purpose, we will set up MariaDB as the database of our choice. MariaDB is a community-driven fork from MySQL. This happened just around the time when Oracle took over MySQL.

We will be using MariaDB, but MySQL can be also used with little to no modification and this is true for rest of the topics in the remainder of the book. If you prefer another database, such as PostgreSQL, this can be used too, but then the appropriate drivers need to be installed and configured.

Installing MariaDB

We will need the following information handy when installing MariaDB on our controller node.

Name	Info
Access to the Internet	Yes
Proxy needed	No
Proxy IP and port	Not applicable
Database root password	`dbr00tpassword`
Node name	`OSControllerNode`
Node IP	172.22.6.95
Node OS version	Ubuntu 14.04.1 LTS

In a production environment, you can have a database cluster in order to eliminate single points of failure.

Please choose the root password in accordance with a password complexity of your choice or the organization's choice.

In the book, we will use `dbr00tpassword` as our MariaDB root password. If you are planning to use a different password, please substitute it in the relevant places.

Using a proxy server

If you are setting this up in an environment where you need to use a proxy server for Internet access, the following steps need to be taken. Set the `http_proxy` and `https_proxy` environment variables as shown here:

- `export http_proxy=http://proxy_ip_address:proxy_port`
- `export https_proxy=http://proxy_ip_address:proxy_port`
- Set the aptitude proxy server by modifying the `/etc/apt/apt.conf` file (if the line doesn't exist, please add it)
- `Acquire::http::Proxy "http://proxy_ip_address:proxy_port";`

Step 1: Setting MariaDB repository

Please log in to the controller node using SSH. Ensure you have permissions to install the software:

```
sudo apt-get install software-properties-common
sudo apt-key adv --recv-keys --keyserver
hkp://keyserver.ubuntu.com:80 0xcbcb082a1bb943db
```

In order to use `apt-key` with a proxy, please use the following format of command, substituting `proxy_ip` and `proxy_port`:

```
sudo apt-key adv --recv-keys --keyserver-options http-
proxy=http://proxy_ip:proxy_port --keyserver
hkp://keyserver.ubuntu.com:80 0xcbcb082a1bb943db
```

```
sudo add-apt-repository 'deb
http://kartolo.sby.datautama.net.id/mariadb/repo/5.5/ubuntu trusty
main'
```

We first install the `software-properties-common` package. Chances are you may already have the package. The next step is optional but it is recommended that you do it, as this will allow the public key to be installed so that there is no error during package signing.

The last line is the most important one. However, all it does is add the line in single quotes to the `/etc/apt/sources.list` file.

Once the preceding code is done we will update aptitude:.

```
sudo apt-get update
```

Step 2: Installing the MariaDB package

Installing the package requires a single command, as follows:

```
sudo apt-get install mariadb-server python-mysqldb
```

This will prompt you to download the files and install them. Once complete, MariaDB is installed and ready for use.

Please ensure that no errors are encountered during this step.

During installation, you will be prompted for the root password, where you will have to enter our database root password. If you have left it blank, we will set this up in the next section.

Configuring the database

We will be configuring only the basic settings that are required for OpenStack to run. The following configurations will be made to the database in order for it to be able to work properly with the different services of OpenStack:

- Allow connections from outside the box

  This is needed so that the components on other physical boxes can communicate with the database

- Set UTF8 character sets

Edit the `/etc/mysql/my.cnf` file. Under the `[mysqld]` header, you will find the bind-address keyword pointing to localhost. Set this to the IP address of the controller node.

www.allitebooks.com

Also, add the following lines shown just below the bind-address in order to enable UTF-8 encoding:

```
[mysqld]
bind-address = 172.22.6.95
default-storage-engine = innodb
collation-server = utf8_general_ci
init-connect = 'SET NAMES utf8'
character-set-server = utf8
```

Once this is done, restart the database service with the following command:

```
sudo service mysql restart
```

The database service is now up and ready for the next step.

Securing the database

This is an optional step. It is recommended that you secure the database in production environments. If there are enterprise-specific standards, please follow those. The following command will secure the database:

```
mysql_secure_installation
```

On executing the preceding command, you will be prompted for the root password. Enter the root password if you have set it up during the installation; otherwise, if there is no root password, press *Enter*.

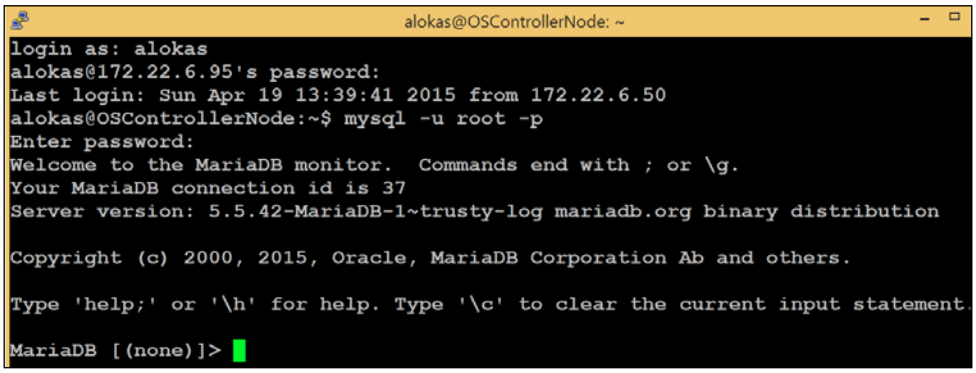
Work through the options; the defaults work well. So leave the defaults in force. You may also choose to change the root password or set one up here.

Testing the installation

If you have followed all the previous steps and no errors were thrown along the way, then you have a working installation. Let us log in to MariaDB using the following command (and entering the password):

```
mysql -u root -p
```

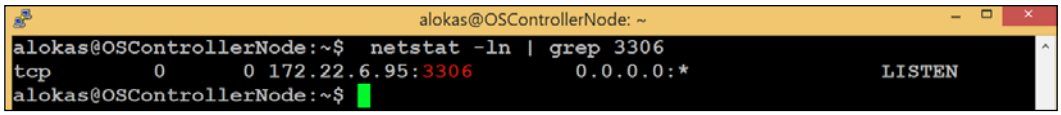
This shows that the database is active and functional:



In order to test that the database is listening on the IP address and not just localhost, execute the following command:

```
netstat -ln | grep 3306
```

This shows the currently listening processes. We `grep` for `3306` as this is our database port. You should be able to see something similar to what is shown here:



This shows that the server is accepting connections on the IP address and hence listening to the network.

Setting up the messaging broker

OpenStack needs a messaging system in order to queue requests and communicate among different services.

There are several options such as RabbitMQ, ZeroMQ, and Qpid. We will use RabbitMQ as the AQMP protocol of our choice. As in the case of the database, this system will also be set up only once and in an enterprise environment; this component can be shared.

Installing RabbitMQ

We will need the following information handy when installing RabbitMQ on our controller node.

Name	Info
Access to the Internet	Yes
Proxy needed	No
Proxy IP and port	Not applicable
Rabbit MQ guest password	`rabb1tmqpass`
Node name	OSControllerNode
Node IP	172.22.6.95
Node OS version	Ubuntu 14.04.1 LTS

Step 1: Setting up the RabbitMQ repository

We will set up the Rabbit MQ repository in the same way as we set one up for MariaDB. Execute the following commands:

```
sudo add-apt-repository 'deb http://www.rabbitmq.com/debian/ testing main'
wget https://www.rabbitmq.com/rabbitmq-signing-key-public.asc
sudo apt-key add rabbitmq-signing-key-public.asc
```

There are three commands as you can see. The first one will add the URL to the `/etc/apt/sources.list` file.

The second one downloads the signing key, and the third command installs the signing key. You can choose to skip the second and the third command, and you will have to just ignore the warnings.

As always, just update the aptitude by using this command:

```
sudo apt-get update
```

Step 2: Installing the RabbitMQ package

Installing the package needs a single command:

```
sudo apt-get install rabbitmq-server
```

Once the packages are downloaded and installed, you should now have a working RabbitMQ service.

Configuring the RabbitMQ server

There are several configurations that are possible such as clustering the RabbitMQ server and setting the queue thresholds. However, we will only perform a few basic configurations:

- Allow the guest account to connect from outside the localhost

  The guest account is created by default when Rabbit MQ is installed, but it is restricted only to localhost. We need to open this up so that the other OpenStack components can use the service.

- Set up a password for the RabbitMQ guest user

  Since the guest user can now access from outside, we need to set up a password that we can configure in various OpenStack service configurations.

Type the following command:

```
echo '[{rabbit, [{loopback_users, []}]}].' >>
/etc/rabbitmq/rabbitmq.config
```

This just adds a line in the `/etc/rabbitmq/rabbitmq.config` file. Please note that, this file is not created by default.

In order to set the guest password, execute the following command:

```
rabbitmqctl change_password guest rabb1tmqpass
```

Please use the same password as you have chosen in the preceding table.

In a production environment, we can use different user accounts for different services of OpenStack in RabbitMQ but, for the purpose of this book, we will use the guest account.

In the case of production environments, we can use the following commands to create a RabbitMQ user and disable the guest user. Please note that, if you do follow this, you will have to change the RabbitMQ username and password in the configuration files wherever they occur:

```
rabbitmqctl delete_user guest
rabbitmqctl add_user openstack rabb1tmqpass
rabbitmqctl set_user_tags openstack administrator
```

This will add the user `openstack` and give them administrator permission. You will then have to change the RabbitMQ section of all the other configuration files.

To restart RabbitMQ server type the following command:

```
sudo service rabbitmq-server restart
```

Please ensure the service starts without any errors.

Testing the installation

In order to test the installation, we check whether RabbitMQ is listening on the network. This can be tested using our good old `netstat` command:

```
netstat -lnp | grep beam
```

You will see a `5672` or `25672` port listening.

In addition to this, we can use the `rabbitmqctl` tool:

```
sudo rabbitmqctl status
```

This will show you the status of the service. The following diagram shows the kind of output requested:

```
root@OSControllerNode:/home/alokas# netstat -lnp | grep beam
tcp        0      0 0.0.0.0:25672           0.0.0.0:*               LISTEN      22256/beam.smp
tcp6       0      0 :::5672                 :::*                    LISTEN      22256/beam.smp
root@OSControllerNode:/home/alokas# rabbitmqctl status
Status of node rabbit@OSControllerNode ...
[{pid,22256},
 {running_applications,[{rabbit,"RabbitMQ","3.5.1"},
                        {os_mon,"CPO  CXC 138 46","2.2.14"},
                        {mnesia,"MNESIA  CXC 138 12","4.11"},
                        {xmerl,"XML parser","1.3.5"},
                        {sasl,"SASL  CXC 138 11","2.3.4"},
                        {stdlib,"ERTS  CXC 138 10","1.19.4"},
                        {kernel,"ERTS  CXC 138 10","2.16.4"}]},
 {os,{unix,linux}},
```

The status output also shows the version of the RabbitMQ components.

This book assumes that we are installing on a test bed with no additional firewalls between the different nodes.

In a production environment, however, you may have additional physical firewalls, IP tables, and so on, that may block access to the ports.

Please ensure that you check with your network and system administrator and allow the ports for the various services such as MySQL and RabbitMQ (the port numbers are mentioned in the tables that precede the installation) for the environment to work.

Installing Keystone

After the common components are installed, we can proceed to install Keystone. We will use the same table to collect all the information that we need before starting the installation.

Name	Info
Access to the Internet	Yes
Proxy needed	No
Proxy IP and port	Not applicable
Node name	`OSControllerNode`
Node IP address	172.22.6.95
Node OS	Ubuntu 14.04.1 LTS
Keystone database password	`k3yst0ne`
One time token	Will be generated during the install step

In choosing the password, please don't use the `@` symbol as it will conflict during configuration with the URL syntax.

Setting up the OpenStack repository

Before we start our installation, we will add the `juno` repository from Ubuntu to the aptitude to install the components. Please remember to do so in all the nodes where we will be installing the OpenStack components. Please remember that this needs to be done only once. Execute the following command to set up the OpenStack repository:

```
apt-get install ubuntu-cloud-keyring

echo "deb http://ubuntu-cloud.archive.canonical.com/ubuntu trusty-
updates/juno main" > /etc/apt/sources.list.d/openstack-juno.list
```

The preceding commands will simply install the cloud key ring component and add the `http://ubuntu-cloud.archive.canonical.com/ubuntu trusty-updates/juno main` line to the `openstack-juno.list` file.

Finish up by updating the aptitude package manager:

```
sudo apt-get update
```

Now we are ready to install various OpenStack Juno services.

Depending on the Linux distribution, the repositories are different. Please note that, in order to get the latest builds, you can compile the services from the source if you choose to do so.

Creating the database

The next step along the way is creating the database for Keystone. This will be created in MariaDB, which we installed earlier. Log in to the MySQL client by using the following command:

```
mysql –u root –p
```

Enter the `dbr00tpassword` password and execute the following command:

```
create database keystone;
```

This command will create a database called Keystone; we will now create the credentials that will be used to access the database.

We will now be granting privileges to the Keystone user, in the cases where the request originates from localhost or a remote host. In most versions of most operating systems, calling out the localhost separately is not needed and is covered by the `%wildchar`. The command for localhost is added simply to cover both possibilities.

```
GRANT ALL PRIVILEGES ON keystone.* TO 'keystone'@'localhost'
IDENTIFIED BY 'k3yst0ne';

GRANT ALL PRIVILEGES ON keystone.* TO 'keystone'@'%' IDENTIFIED BY
'k3yst0ne';
```

Please replace the password if you are using a different one. Type `exit` to get out of the client:

```
MariaDB [(none)]> GRANT ALL PRIVILEGES ON keystone.* TO 'keystone'@'localhost' \
    ->   IDENTIFIED BY 'k3yst0ne';
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]> GRANT ALL PRIVILEGES ON keystone.* TO 'keystone'@'%' \
    ->   IDENTIFIED BY 'k3yst0ne';
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]>
MariaDB [(none)]> quit
Bye
root@OSControllerNode:~# mysql -u keystone -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 46
Server version: 5.5.42-MariaDB-1~trusty-log mariadb.org binary distribution

Copyright (c) 2000, 2015, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> show databases
    -> ;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| keystone           |
+--------------------+
2 rows in set (0.00 sec)
```

In order to verify that the previous commands (of creating new credentials and allowing access to only one database) actually took, we will now login using the Keystone user name and password we created using similar commands that we used for the root user.

Once you login with the Keystone username, you should be able to see only the Keystone database (along with `information_schema`).

Installing the package

We will now install the Keystone package and its client using the aptitude package manager with the following command:

```
sudo apt-get install keystone python-keystoneclient
```

Please verify that the install completes successfully.

The initial configuration

The initial configuration of Keystone needs the following to be done:

- Generate a token
- Modify the Keystone configuration file

The administrator token is generated only for the installation as there are no users in the system at the moment. Once the default admin user is created, this token is no longer necessary.

Generating the admin token

The admin token is a random number. We will use the `openssl` tool in order to create it:

```
openssl rand -hex 10
```

In my case, it is `b7098d7d5eb7bf889842`; please make a note of this and have it ready for the next section.

We will also set the environment variable to be used later in the installation and configuration process, using the following command:

```
export OS_SERVICE_TOKEN=b7098d7d5eb7bf889842
```

This will set the service token.

Modifying the Keystone configuration file

The Keystone configuration file is located at `/etc/keystone/keystone.conf`. Edit this using your favorite editor by making the following changes:

- `[default]` section:
  - Set the `admin_token` value to the string you have generated
  - Set the verbose flag to `true` (we will set it back to `false` once the setup is complete)
- `[database]` section:
  - Set the connection string to `mysql://keystone:k3yst0ne@localhost/keystone` (or the equivalent in your environment). The format of the URL is `mysql://<username>:<password>@<host>:<port>/<databasename>`. (The port is not mentioned as it is the default `3306` port.)

    ```
    connection=mysql://keystone:k3yst0ne@localhost/keystone
    ```

- `[token]` section:
  - Set the provider and driver values as shown in the following, if the following lines already exist, uncomment them:

```
provider = keystone.token.providers.uuid.Provider
driver = keystone.token.persistence.backends.sql.Token
```

- `[revoke]` section:
  - Set the MySQL revocation driver:

```
driver = keystone.contrib.revoke.backends.sql.Revoke
```

Once the configuration file is completed, it will look as shown in the following screenshot:

```
[DEFAULT]
admin_token=b7098d7d5eb7bf889842
verbose=true
log_dir=/var/log/keystone
[assignment]
[auth]
[cache]
[catalog]
[credential]
[database]
connection=mysql://keystone:k3yst0ne@localhost/keystone
[ec2]
[endpoint_filter]
[endpoint_policy]
[federation]
[identity]
[identity_mapping]
[kvs]
[ldap]
[matchmaker_redis]
[matchmaker_ring]
[memcache]
[oauth1]
[os_inherit]
[paste_deploy]
[policy]
[revoke]
driver = keystone.contrib.revoke.backends.sql.Revoke
[saml]
```

Populating the Keystone DB

We have a blank Keystone database at the moment; we will use the `sync` command in order to populate it with base data.

www.allitebooks.com

Please ensure that you are running the following command as root:

```
keystone-manage db_sync keystone
```

Alternatively, if you don't have root access or are performing it with `sudo` rights, then use the following command. (Both commands don't need to run.)

```
su -s /bin/sh -c "keystone-manage db_sync" keystone
```

Once this completes, you will see messages as shown in the following screenshot:

```
root@OSControllerNode:~# su -s /bin/sh -c "keystone-manage db_sync" keystone
2015-04-20 03:05:43.448 23946 INFO migrate.versioning.api [-] 33 -> 34...
2015-04-20 03:05:44.328 23946 INFO migrate.versioning.api [-] done
2015-04-20 03:05:44.329 23946 INFO migrate.versioning.api [-] 34 -> 35...
2015-04-20 03:05:44.363 23946 INFO migrate.versioning.api [-] done
2015-04-20 03:05:44.363 23946 INFO migrate.versioning.api [-] 35 -> 36...
2015-04-20 03:05:44.381 23946 INFO migrate.versioning.api [-] done
2015-04-20 03:05:44.381 23946 INFO migrate.versioning.api [-] 36 -> 37...
2015-04-20 03:05:44.415 23946 INFO migrate.versioning.api [-] done
2015-04-20 03:05:44.415 23946 INFO migrate.versioning.api [-] 37 -> 38...
2015-04-20 03:05:44.471 23946 INFO migrate.versioning.api [-] done
2015-04-20 03:05:44.472 23946 INFO migrate.versioning.api [-] 38 -> 39...
2015-04-20 03:05:44.528 23946 INFO migrate.versioning.api [-] done
2015-04-20 03:05:44.528 23946 INFO migrate.versioning.api [-] 39 -> 40...
```

You can also log in to the MySQL client and execute the `show tables` command to see all the different tables that are created:

```
root@OSControllerNode:~# mysql -u keystone -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 48
Server version: 5.5.42-MariaDB-1~trusty-log mariadb.org binary distribution

Copyright (c) 2000, 2015, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> use keystone;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [keystone]> show tables;
+------------------------+
| Tables_in_keystone     |
+------------------------+
| assignment             |
| credential             |
| domain                 |
| endpoint               |
| group                  |
| id_mapping             |
| migrate_version        |
| policy                 |
| project                |
| region                 |
| revocation_event       |
| role                   |
| service                |
```

Restart the service using the following command:

```
sudo service keystone restart
```

While modifying the configuration, you will have noted that there was already a SQL configuration; I am just reminding you of this because Keystone by default comes with a SQLite database. Since we won't be using this anymore, let's delete it. It is located at `/var/lib/keystone/keystone.db`. We will delete this file using the following command:

```
rm -f /var/lib/keystone/keystone.db
```

Setting up your first tenant

We will need to create an administrative tenant (or project) that will allow us to log in to Horizon (once we install it in the next chapter) and perform other Keystone related functions. The order in which the components are created is as follows:

- Tenant (or project)
- Users
- Roles

You will need the following information in order to perform the actions:

Name	Info
Tenant name	firsttenant
Tenant description	Our First Tenant
User name	`admin`
User e-mail	`admin@test.com`
Password	`h3ll0world`
Role name	admin
Admin token	b7098d7d5eb7bf889842
Controller node name	OSControllerNode

The role names are defined in the `policy.json` file in Keystone, so we will use the `admin` role.

Setting up environment variables

We generated a token in the previous section; we have already exported it in that section. Please verify this using the `env` command and check whether `OS_SERVICE_TOKEN` is set. If the token is not set (you may have logged out and logged in to the shell), then you can set it using the following command:

```
export OS_SERVICE_TOKEN=b7098d7d5eb7bf889842
```

We should also export the service endpoint. The default port for the Keystone administrative function is `35357` (you can verify this by the `netstat -nlp` command; you will see the server listening on this port). To export service endpoint, execute the following command:

```
export OS_SERVICE_ENDPOINT=http://OSControllerNode:35357/v2.0
```

```
root@OSControllerNode:/etc/keystone# netstat -lnp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:35357           0.0.0.0:*               LISTEN      18351/python
```

Please ensure that you are able to ping the name and telnet on the `35357` port.

We can skip this step, but then we have to add the following commands with two additional parameters, `--os-token` and `--os-endpoint`, in all the commands we execute.

Creating the tenant

Once the environment variables are set up, execute the following command to create the tenant:

```
keystone tenant-create --name firsttenant --description "Our First
Tenant"
```

```
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
| description |         Our First Tenant         |
|   enabled   |               True               |
|      id     | c95ff542161f494eb973fc9cc9977233 |
|     name    |           firsttenant            |
+-------------+----------------------------------+
```

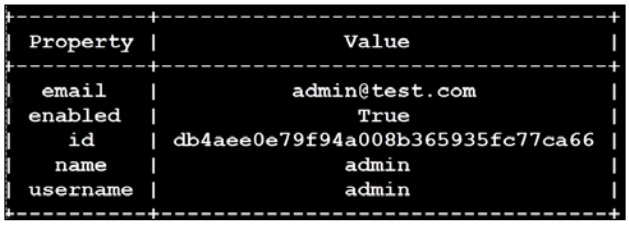
You will see the output confirming that the tenant has been created.

Creating the user

We will now create the user, again using a single command:

```
keystone user-create --name admin --pass h33l0world --email
admin@test.com
```

This will create the user called `admin`:



Creating and mapping the role

In this section, we will create a role called `admin` and associate it with the `admin` user and the `firsttenant` tenant. Execute the following command to create the role:

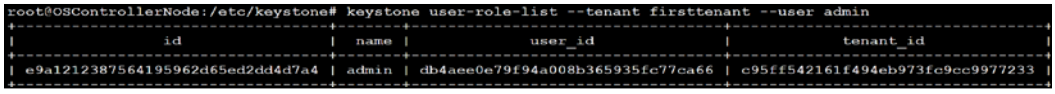
```
keystone role-create --name admin
```

For mapping, execute the following command:

```
keystone user-role-add --user admin --tenant firsttenant --role admin
```

This command will not give us any output. However, we can verify it by the following command:

```
keystone user-role-list --tenant firsttenant --user admin
```



We should be able see the GUID of the tenant and the user that we created earlier.

This task is now complete.

Creating service endpoints

In a distributed tool such as OpenStack, it is a good practice for each component to talk to the others using an API, which means that, even when you are using the Horizon dashboard, in the backend the functions will be performed using an API.

In order for the APIs to work, Keystone provides a catalog sub-system. This provides us with the different services of OpenStack and their URLs, so we will start with the identity service itself.

Keystone has two ports, one is used for administrative functions (TCP 35357) and the other one is used for the services to authenticate against it (TCP 5000). The admin port can also be used to authenticate, but this would be an overkill.

Creating the service

The service needs to be created in the database before its endpoints can be added; this is done by the following command:

```
keystone service-create --name keystone --type identity --description
"OpenStack Identity"
```

```
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
| description |        OpenStack Identity        |
|   enabled   |               True               |
|      id     | c0dc26226c42450a82838fc1c18b11fe |
|     name    |             keystone             |
|     type    |             identity             |
+-------------+----------------------------------+
```

Note down the ID as this is needed for the next step. In my case, it is `c0dc26226c42450a82838fc1c18b11fe`.

Execute the following command:

```
keystone service-list
```

This will show the currently created service, as shown in the following screenshot:

```
root@OSControllerNode:/etc/keystone# keystone service-list
+----------------------------------+----------+----------+--------------------+
|                id                |   name   |   type   |    description     |
+----------------------------------+----------+----------+--------------------+
| c0dc26226c42450a82838fc1c18b11fe | keystone | identity | OpenStack Identity |
+----------------------------------+----------+----------+--------------------+
```

Sometimes the system acts up and you will get a HTTP 5XX error; just restart the service using the service Keystone `restart` command.

Creating the endpoint

We will need the following information handy:

Name	Info
Controller node name	OSControllerNode
Service port	5000
Admin port	35357
Service ID (from previous step)	`c0dc26226c42450a82838fc1c18b11fe`
Public URL	`http://OSControllerNode:5000/v2.0`
Internal URL	`http://OSControllerNode:5000/v2.0`
Admin URL	`http://OSControllerNode:35357/v2.0`
Region name	dataCenterOne

Before we dive in, we should understand the following things that we have filled:

- **Public URL**: This is the one that should be accessible from outside (by other departments); so, if you have a different name or FQDN that you are trying to publish, use that here. Please note that we have used the service port here.
- **Internal URL**: This is same as the public URL but from inside the company.
- **Admin URL**: This is for administrative tasks and allows things such as creating a user (if we are using a local database).
- **Region name**: This is needed simply because we need to keep the API EC2-compliant.
- **V2.0**: We have used the Keystone identity version 2.0, even after the 3.0 version is out. We can just change these URLs to v3.0, and the new identity can be used. However, at the time of writing, the v2.0 version was the most compatible one.

Armed with the preceding information, let's construct the command. This is a single-line command, but is broken into multiple lines for readability purposes; the \ instructs the shell executing the command to wait for the remaining:

```
keystone endpoint-create \
  --service-id c0dc26226c42450a82838fc1c18b11fe \
  --publicurl http://oscontrollernode:5000/v2.0 \
  --internalurl http://oscontrollernode:5000/v2.0 \
  --adminurl http://oscontrollernode:35357/v2.0 \
  --region dataCenterOne
```

```
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
|   adminurl  | http://oscontrollernode:35357/v2.0 |
|      id     |  128f30e8b6554c608a91640e6e749101  |
| internalurl | http://oscontrollernode:5000/v2.0  |
|  publicurl  | http://oscontrollernode:5000/v2.0  |
|    region   |           dataCenterOne            |
|  service_id |  c0dc26226c42450a82838fc1c18b11fe  |
+-------------+----------------------------------+
```

So we have successfully created our endpoint, which the different OpenStack components can use in order to communicate with the Keystone service.

Verifying the installation

We have at various points verified that the Keystone service is working as expected. However, let's execute the commands with the actual username and password rather than using the admin token that we have generated.

Open a new terminal window, or unset the environment variables that we set up:

```
unset OS_SERVICE_TOKEN OS_SERVICE_ENDPOINT
```

You don't have to do this if you have opened a new terminal window.

Using Keystone CLI

We will list all the tenants in the system; we should expect to see the one we created:

```
keystone --os-tenant-name firsttenant --os-username admin --os-
password h3310world --os-auth-url http://oscontrollernode:5000/v2.0
tenant-list
```

Property	Value
adminurl	http://oscontrollernode:35357/v2.0
id	128f30e8b6554c608a91640e6e749101
internalurl	http://oscontrollernode:5000/v2.0
publicurl	http://oscontrollernode:5000/v2.0
region	dataCenterOne
service_id	c0dc26226c42450a82838fc1c18b11fe

As you can see, we are now using the user credentials in order to execute the Keystone `cli` and fetch the information.

We could try the same thing for various other commands such as user-list and role-list.

Using the API

Under the hood, the Python client uses the RESTful API. You would use the API if you were trying to call the services directly without using the client. As an example, consider a developer of a company trying to create hooks in the OpenStack system from the company's custom request portal so that end users don't get to see the actual Horizon database, and even the cloud provisioning engine is abstracted from the end user. We will quickly verify that we are able to use Keystone using the RESTful API.

This step is optional and is provided only for informational purposes.

We will use curl in order to make the RESTful call. The URI for tenant is `/tenants`, so the full URL will be a public URL appended with the API URI, which in our case is `http://oscontrollernode:5000/v2.0/tenants`. This is using a GET verb, so we will need an authentication token in order to execute the following curl command to get the token:

```
curl -vv -d '{"auth":{"passwordCredentials":{"username": "admin",
"password": "h33l0world"}}}' -H "Content-type: application/json"
http://localhost:5000/v2.0/tokens | python -m json.tool
```

This will give us the token that we can use in the header:

```
{
    "access": {
        "metadata": {
            "is_admin": 0,
            "roles": []
        },
        "serviceCatalog": [],
        "token": {
            "audit_ids": [
                "yQl5lzQDSOWx03f-SHhVhw"
            ],
            "expires": "2015-04-21T21:26:27Z",
            "id": "685590d73e81437da6e97a9b6764213b",
            "issued_at": "2015-04-21T20:26:27.682198"
        },
        "user": {
            "id": "db4aee0e79f94a008b365935fc77ca66",
            "name": "admin",
            "roles": [],
            "roles_links": [],
            "username": "admin"
        }
    }
}
```


We extract this token and use it in our next call. The tokens are normally valid for a few minutes to hours, depending on the configuration of Keystone.

Tokens are akin to authentication cookies in a lot of ways. If you have come across any kind of a **single sign-on** (**SSO**) system for the Web, you may already be aware of how this functions.

In a single sign-on scenario, when you try to access a protected resource, you send a request to the resource, the web page in question checks for a presence of a cookie and whether that cookie is valid. The cookie has information about the user and sometimes authorization as well.

If the cookie is not present or is present but not valid, then the user is sent to a common webpage where the SSO system authenticates them and assigns a cookie. The SSO system then redirects them to the resource and the user is now granted access.

The token system is similar but with some differences, as follows:

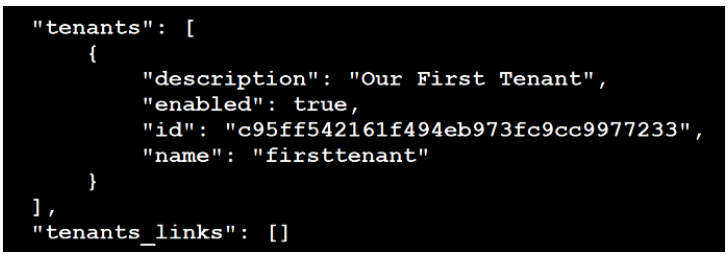
1. User checks whether they have the valid token.
2. If the valid token is not found, a request is made to Keystone.
3. Keystone authenticates and assigns a token. It also stores the token in its localdatabase table called "tokens".

4. User makes the call to the service, such as Horizon, Nova, or any other OpenStack service that supports Keystone, with the token.
5. The service at the backend checks with Keystone whether the token is valid and then allows access to the resource.

This way, the services themselves never get access to the user credentials but only the tokens that are time-bound to expire. This serves two purposes, that of security and of a single sign-on feel.

```
curl -s \
  -H "X-Auth-Token: 685590d73e81437da6e97a9b6764213b" \
  http://localhost:5000/v2.0/tenants | python -m json.tool
```

And we then get the tenants configured in the system, as shown in the following screenshot:



We can now use the token until its expiry for the particular user.

There are several toolkits and SDK's available for OpenStack in order to enable faster code development.

Please visit `https://wiki.openstack.org/wiki/SDKs` for the different SDKs that are available.

The languages include a varied range of choices such as libraries for C, C++, Java, Node.Js , Perl, and PHP.

Troubleshooting the installation and configuration

By following the steps given in this chapter, you shouldn't encounter any issues as such with the installation. However, let's look at troubleshooting some common issues, which may help you with a successful installation.

www.allitebooks.com

If your installation was successful, you may choose to skip this section.

DB sync errors

When attempting the database sync in the populate the Keystone DB section, you may encounter errors. Please follow the following steps to try and fix the errors.

System language settings

The language setting of your computer can cause issues while conducting the initial database sync. In order to prevent such issues, export the language variables using the following command:

```
export LC_ALL=en_US.UTF-8
export LANG=en_US.UTF-8
```

Configuration errors

Please ensure that the configuration file has the correct SQL URL. The format is `mysql://username:password@host:port/databaseName`. If your password has an `@` symbol, the database sync will fail.

Failing Keystone commands

If your Keystone commands are failing, you should check for the following.

Service non-responsive

The service may hang, due to which you will see 5XX errors. Restart the service using the service Keystone `restart` command.

DNS issues

Please ensure that you are able to resolve the names that are mentioned in several of the configuration URLs. If you cannot ping them, check the DNS server configuration in `/etc/resolv.conf` (if it is an enterprise environment) or simply ensure that the `/etc/hosts` file has the appropriate entries.

Network issues

Please ensure that there are no IP table services, that, if present, they are disabled, or that the proper rules are in place to allow different ports and communications. Please note, Keystone uses the `5000` and `35357` ports.

Please ensure that the ports are listening to all IP addresses or to an external IP address rather than to the 127.0.0.1 or localhost. You can verify this by `netstat -ln`.

Please ensure that you have proper routing rules and a default gateway in place. The routing table can be checked using the `netstat -rn` command.

Summary

In this chapter, we discussed the basic need for Keystone and were introduced to the identity concepts in Keystone. We then discussed the architecture of Keystone and its various sub-systems. We covered the installation of components that are common to most of the OpenStack services such as the database and messaging broker—in this case, MariaDB and RabbitMQ. We then discussed the installation of the Keystone service. We verified our installation and also covered some basic troubleshooting, just in case you face any issues during installation.

Having looked at Keystone, in the next chapter we will look at the storage services of OpenStack: Cinder, Swift, and Glance.

3
Storing and Retrieving Data and Images using Glance, Cinder, and Swift

This chapter introduces the storage services of OpenStack. There are three major services whose names we have already seen in the title of this chapter. Now, you must be wondering why we would need three different services.

We will cover the following topics in this chapter:

- Image storage – Glance
- Block storage – Cinder
- Object storage – Swift
- Troubleshooting steps specific to Swift

Introducing storage services

Object storage or **binary large object** (**BLOB**) storage is in some form like a file server. This kind of storage is needed when the requirement is to place files to be retrieved later. Each of these files is considered as an object, and you really don't care about the disk or the filesystem in which it is stored at the backend. A good example of object storage would be Dropbox, Google Drive, or a public cloud AWS, the S3.

Block storage, on the other hand, can be considered as an independent disk drive. We can choose what filesystem goes into it, and we can access this by volume rather than by file name.

Image storage is used more for special cases. It is a service that uses BLOB storage or sometimes even its own filesystem to store and retrieve virtual machine images such as AMI in the case of AWS or template in the case of VMware virtualization.

Another popular open source storage solution is Ceph. It provides three major services to various cloud platforms, namely the Ceph Object Store, the Ceph Block Device, and the filesystem. It competes with the OpenStack services that will be discussed in this chapter, and you may even choose to use Ceph with OpenStack infusing the KVM/QEMU hypervisors.

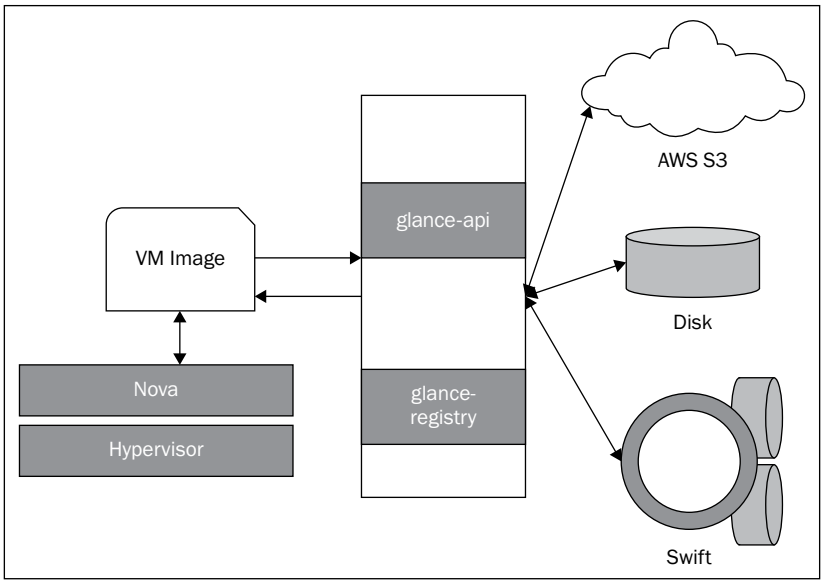
There are some advantages of using Ceph Block Storage (RBD – RADOS Block Device) over Cinder, as it is based on **Reliable Autonomic Distributed Object Store** (**RADOS**) and stripes the data across the entire storage cluster providing us with better IOPS and reliability. However, the integration with Ceph and its architecture are beyond the journey that we about to embark on, so let's continue with the three OpenStack services.

The only mandatory service among the three is Glance, which can serve images from its local filesystem and few other options, such as AWS S3 and Swift. Even though Glance is the only mandatory storage service, if we don't have Cinder and Swift services, we will not be able to provide block storage to our VMs and BLOB storage to our users. In order to keep it simple, we will have Glance configured for local storage and switch to swift or others as and when required.

Let's start with Glance. We will be using both the controller node and the storage node for this chapter. So, ensure that you have the storage node installed with two network cards, as shown in *Chapter 1*, *An Introduction to OpenStack*. Also ensure that you have added the Juno repository.

Working with Glance

As mentioned earlier, Glance is a mandatory service, and without this service, Nova (compute service) will not know where to pick its images from. We will install Glance on the controller node itself. The following diagram explains its architecture:



Glance has two components: **glance-api** and **glance-registry**. As the name suggests, **glance-api** provides the API calls required to retrieve and store the images, while **glance-registry** handles more of the backend functions regarding where the images are stored.

Let's start with the installation of Glance. Its installation follows similar steps to those used for Keystone in the previous chapter. We will use the following checklist to have all the information ready:

Name	**Info**
Access to the Internet	Yes
Proxy needed	No
Proxy IP and Port	Not Applicable
Node name	`OSControllerNode`
Node IP address	172.22.6.95
Node OS	Ubuntu 14.04.1 LTS
Glance password	`g1anc3pwd`
Glance repository	Local disk
Disk partition info	`/glance/images/`
Glance Keystone password	`g1anc3keypwd`
Glance port	9292

Creating the database

Log in to the MySQL instance (MariaDB) installed in the previous chapter using the root password that we created:

```
mysql -u root -p
Enter the password:dbr00tpassword
```

Once in the database, execute the following command:

```
create database glance;
```

This will create an empty database called Glance. Let's now set up the Glance database user credentials:

```
GRANT ALL PRIVILEGES ON glance.* TO 'glance'@'localhost' IDENTIFIED
BY 'g1anc3pwd';
GRANT ALL PRIVILEGES ON glance.* TO 'glance'@'%' IDENTIFIED BY
'g1anc3pwd';
```

Access to the glance database is allowed from all the hosts denoted by `%`. However, as previously mentioned, we need to provide explicit permissions for localhost, hence the two lines.

You can quickly verify that the database has been created by typing the following command:

```
show databases
```

You should see the following output:

```
MariaDB [(none)]> create database glance;
Query OK, 1 row affected (0.00 sec)

MariaDB [(none)]> GRANT ALL PRIVILEGES ON glance.* TO 'glance'@'localhost' IDENT
IFIED BY 'g1anc3pwd';
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]> GRANT ALL PRIVILEGES ON glance.* TO 'glance'@'%' IDENTIFIED BY
 'g1anc3pwd';
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| glance             |
| keystone           |
| mysql              |
| performance_schema |
+--------------------+
5 rows in set (0.00 sec)
```

You will notice that you can see the Keystone and Glance databases along with the system databases; this is because you are logged into the root account. If you use the Glance account, you will not be able to see the Keystone database.

In order to verify that, after exiting the client, execute the following command:

```
mysql -u glance -p
```

When prompted for the password, enter `g1anc3pwd`. Execute the show databases; command and you will see only the schema and the Glance database.

Installing the packages

As a final step in the installation, let's use the package manager to install the components of Glance:

```
sudo apt-get install glance python-glanceclient
```

This will install the two components used by Glance. By default, the service comes with its SQL lite database like Keystone; hence, after installing and configuring it, we will delete the database.

Initial configuration of Glance

While performing the initial configuration of the Glance service, we will do the following:

- Create a Glance user in Keystone
- Create a Glance service in Keystone
- Create a Glance endpoint in Keystone
- Modify the configuration file
- Populate the Glance database

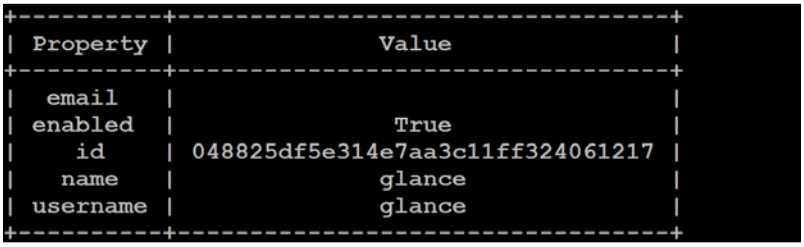
We will need to use the admin user that we created in the previous chapter in order to perform Keystone functions.

Creating a user in Keystone

We will create a user with the name `glance` and the password `g1anc3keypwd`:

```
keystone --os-tenant-name firsttenant --os-username admin --os-
password h33l0world --os-auth-url http://oscontrollernode:5000/v2.0
user-create --name glance --pass g1anc3keypwd
```

This will create the Glance user in the Keystone in the first `tenant`, as seen in the following screenshot:



Property	Value
email	
enabled	True
id	048825df5e314e7aa3c11ff324061217
name	glance
username	glance

As we can see, the command is too long and hence confusing. Therefore, we will export the common variables so that the Keystone command line becomes easier. Execute the following commands to export the variables:

```
export OS_TENANT_NAME=firsttenant
export OS_USERNAME=admin
export OS_PASSWORD=h33l0world
export OS_AUTH_URL=http://OSControllerNode:5000/v2.0
```

After executing the commands, the Keystone commands will be simpler, as seen in the subsequent configurations.

Next, we will create a service `tenant`, which will be used to interact with different services. This is a one-time process. We could ignore and proceed; however, it is a good practice, so let's follow it:

```
keystone tenant-create --name service --description "Service Tenant"
```

We will map the Glance user we just created to this tenant as an administrative role, by using the following command:

```
keystone user-role-add --user glance --tenant service --role admin
```

Since we will be exporting these, we can just save the preceding export commands in a file of our choice and use the source command to export the variables.

Therefore, in our case, we will save the export statements in the file called `~alokas/os.txt` as follows:

```
source ~alokas/os.txt
```

Creating a Glance service in Keystone

We will follow the exact same steps as we followed in the previous chapter, and since we have exported the environment variables, we can use the command without passing the various parameters:

```
keystone service-create --name glance --type image --description
"OpenStack Image Service"
```

This will create the service in the Keystone, as can be seen in the following screenshot:

```
root@OSControllerNode:/etc/keystone# keystone service-create --name glance --type image --d
escription "OpenStack Image Service"
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
| description |     OpenStack Image Service      |
|   enabled   |               True               |
|      id     | 8d85877e169b40aa82aefdf23df74012 |
|     name    |              glance              |
|     type    |              image               |
+-------------+----------------------------------+
root@OSControllerNode:/etc/keystone#
```

Creating a Glance endpoint

The endpoint helps other services discover Glance. We will need the ID of the Glance service we created in the previous step. Execute the following command:

```
keystone service-list
```

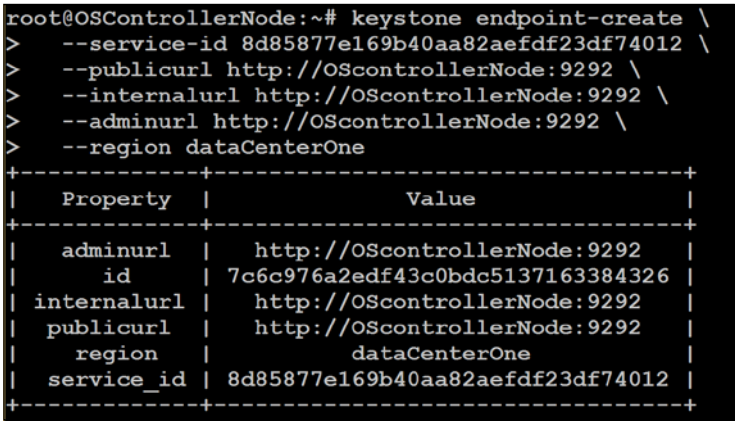
Copy the UID of the glance service, in our case `8d85877e169b40aa82aefdf23df74012`. We also need to remember that the default port for Glance is `9292`.

```
keystone endpoint-create \
  --service-id 8d85877e169b40aa82aefdf23df74012 \
  --publicurl http://OSControllerNode:9292 \
  --internalurl http://OSControllerNode:9292 \
```


www.allitebooks.com

```
--adminurl http://OSControllerNode:9292 \
--region dataCenterOne
```

You should see something like the following screenshot:



Modifying Glance configuration

Once we have the Keystone commands out of the way, we will have to modify the Glance configuration file located in `/etc/glance/glance-api.conf`.

We will need to modify five sections in the file, as follows:

- Under the `[default]` section, we will set up the verbose logging for testing and we will set up the notification driver to **NO Operation** (**NOOP**). We will set up the RabbitMQ configuration based on our common settings:

```
notification_driver = noop
verbose = True
rabbit_host = OSControllerNode
rabbit_port = 5672
rabbit_use_ssl = false
rabbit_userid = guest
rabbit_password = rabb1tmqpass
default_store = file.
```

- Under the `[database]` section, we change the connection variable as follows:

  ```
  connection = mysql://glance:g1anc3pwd@172.22.6.95/glance
  ```

  This change will ensure that Glance now points to the MySQL we set up instead of pointing at SQLite database. There will be another configuration statement currently pointing to the SQLite database in the configuration file; comment that line.

- Under the `[keystone_authtoken]` section:

  ```
  auth_uri = http://OScontrollerNode:5000/v2.0
  identity_uri = http://OScontrollerNode:35357
  admin_tenant_name = service
  admin_user = glance
  admin_password = g1anc3keypwd
  ```

  This section provides URLs to retrieve the tokens from Keystone.

- Under the `[paste_deploy]` section, we will set the flavor to Keystone:

  ```
  flavor = keystone
  ```

  This sets up glance to use Keystone for authentication.

- Under the `[glance_store]` section, we specify where we will store the images (`:1` mentions the order of directories if multiple of them are used):

  ```
  filesystem_store_datadir = /glance/images/:1
  ```

We have to make the same changes in the Glance registry configuration, with the exception of the `[glance_store]` section. The configuration file is located at `/etc/glance/glance-registry.conf`.

Verify the contents of the file by executing the following command:

```
cat /etc/glance/glance-api.conf | grep -v "#" | egrep -v '^[[:space:]]*$'
```

This shows only non-commented and non-empty lines from the configuration.

Let us now create the directory where we store the images:

```
sudo mkdir -p /glance/images/
sudo chown -R glance:glance /glance/images/
```

We could mount the directory on a separate hard disk in a development or production environment if we choose to use file storage in Glance.

Please ensure that you have made the changes to both the files (API and registry configuration).

Populating the Glance database

To populate the Glance database, as root, execute the following command:

```
/bin/sh -c "glance-manage db_sync" glance
```

This should sync the database. If you see any errors, you will need to check the configuration files.

Finalizing the installation

We will now remove the database that came with Glance:

```
rm -f /var/lib/glance/glance.sqlite
```

Then, we will finally restart the Glance components (both API and registry):

```
sudo service glance-api restart
sudo service glance-registry restart
```

Validating the installation

There is no better way to validate an installation than to put it in action, so we will go ahead, download a KVM image from the Internet, and upload it to the Glance service.

Let us first check that there are no images in the current Glance, by executing the following command:

```
glance image-list
```

Remember to export the variables mentioned earlier in the chapter for username, password, and so on.

You should see the following output:

```
root@OSControllerNode:~# glance image-list
+----+------+-------------+------------------+------+--------+
| ID | Name | Disk Format | Container Format | Size | Status |
+----+------+-------------+------------------+------+--------+
+----+------+-------------+------------------+------+--------+
```

As you can see there are no images in Glance. We can download any `qcow2` format image, which can be used for KVM or qemu and then upload it. Let us download a CirrOS image. CirrOS is a minimal distribution which was created for the sole purpose of testing a cloud environment; hence we will use the image.

Let us navigate to a folder, say, our home folder, using the `cd ~` command. We will download the image by using the `wget` command:

```
wget http://download.cirros-cloud.net/0.3.3/cirros-0.3.3-x86_64-
disk.img
```

This will download a 13 MB file, which can be used in Nova once we get there.

The credentials for this image are (username/password): `cirros/cubswin`.

Once the download is complete, we can upload it to the glance registry using the following command:

```
glance image-create --name "CirrosTest" --file /var/cirros-0.3.3-
x86_64-disk.img --disk-format qcow2 --container-format bare --is-
public True –progress
```

You should see the following result:

```
root@OSControllerNode:~# glance image-create --name "CirrosTest" --file /var/cirros-0.3.3-x
86_64-disk.img --disk-format qcow2 --container-format bare --is-public True --progress
[=============================>] 100%
+------------------+--------------------------------------+
| Property         | Value                                |
+------------------+--------------------------------------+
| checksum         | 133eae9fb1c98f45894a4e60d8736619     |
| container_format | bare                                 |
| created_at       | 2015-05-09T13:23:31                  |
| deleted          | False                                |
| deleted_at       | None                                 |
| disk_format      | qcow2                                |
| id               | 34b205dc-63aa-4b66-a347-2ab98451252d |
| is_public        | True                                 |
| min_disk         | 0                                    |
| min_ram          | 0                                    |
| name             | CirrosTest                           |
| owner            | c95ff542161f494eb973fc9cc9977233     |
| protected        | False                                |
| size             | 13200896                             |
| status           | active                               |
| updated_at       | 2015-05-09T13:23:32                  |
| virtual_size     | None                                 |
+------------------+--------------------------------------+
```

Once the image is uploaded, we can verify it by using the `glance image-list` command, as seen in the following screenshot:

```
root@OSControllerNode:~# glance image-list
+--------------------------------------+------------+-------------+------------------+
-----+--------+
| ID                                   | Name       | Disk Format | Container Format |
     | Status |
+--------------------------------------+------------+-------------+------------------+
-----+--------+
| 34b205dc-63aa-4b66-a347-2ab98451252d | CirrosTest | qcow2       | bare             |
0896 | active |
+--------------------------------------+------------+-------------+------------------+
-----+--------+
root@OSControllerNode:~#
```

This validates that we are able to push the image to the Glance service. This service will provide the images to Nova, so you may consider putting in different images in the `qcow2` format for the KVM.

Glance is responsible for providing images to the different hypervisors, so it will store images in any format that the hypervisor will accept. In the case of multihypervisor deployments (which is beyond the scope of this book), Glance will hold different images for different hypervisors. KVM also supports raw images, so those can also be used.

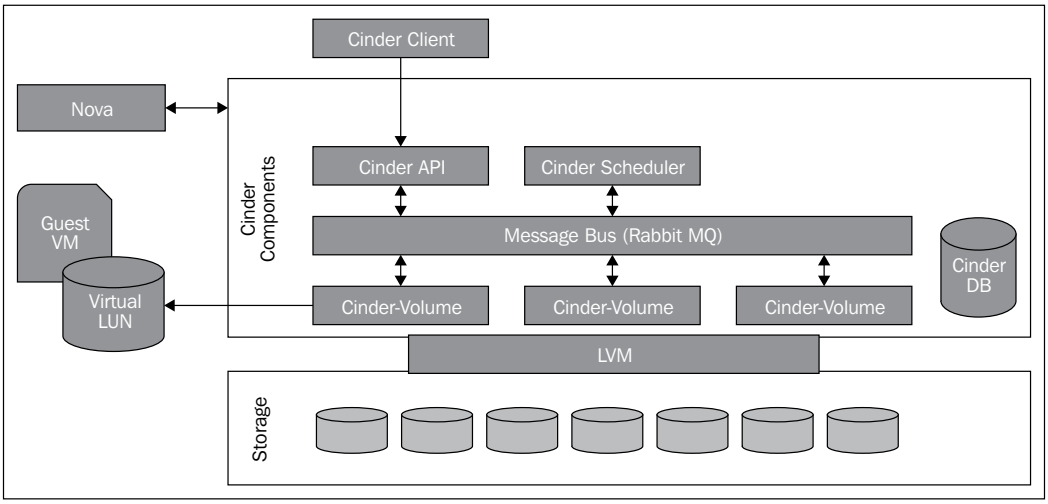
Working with Cinder

Cinder provides block store or the equivalent of an LUN as it were to a virtual machine, which can then format it at its will. We can create logical volumes with these disks or any other thing that we may want to do with the block storage.

Cinder service has a few components that will run in the management node and a few that will run on the storage node. The following components are part of Cinder service:

- API
- Scheduler
- Volume

Out of these, the first two are installed in the controller node and the volume service is installed on the compute node. The following diagram shows the architecture and the communication pattern of the Cinder components:



The API receives the request from the client (either the Cinder client or an external API call) and passes the request on to the scheduler, which then passes the request to one of the Cinder volumes.

The Cinder volumes use one of the physical/virtual storage presented to them as an LVM and provision a part of it as a block storage device, which can be attached to the Guest VM on Nova using iSCSI.

iSCSI is just one of the methods that can be used. We could use Fiber Channel or even NFS. However, we do need to bear in mind that the underlying storage connection dictates the speed and performance of the Cinder volumes themselves.

Controller node

We will need the following information to complete the install on the controller node, so let us fill the following checklist:

Name	Info
Access to the Internet	Yes
Proxy needed	No
Proxy IP and port	Not applicable
Node name	OSControllerNode
Node IP address	172.22.6.95
Node OS	Ubuntu 14.04.1 LTS
Cinder DB password	`c1nd3rpwd`
Cinder Keystone password	`c1nd3rkeypwd`

Creating the database

We will create a blank database after logging in to the MySQL server:

```
create database cinder;
```

This will create an empty database called Glance. Let us now set up the Glance database user credentials:

```
GRANT ALL PRIVILEGES ON cinder.* TO 'cinder'@'localhost' IDENTIFIED
BY 'c1nd3rpwd';
GRANT ALL PRIVILEGES ON cinder.* TO 'cinder'@'%' IDENTIFIED BY
'c1nd3rpwd';
```

This allows the username called `cinder` using our password to access the database called Cinder.

Installing packages

We will install three packages on the controller node: the scheduler, the API, and the client package.

We will use the aptitude package manager to do this by running the following command:

```
sudo apt-get install cinder-scheduler python-cinderclient cinder-api
```

Once the installation is complete, we can move on to the next step.

Initial configuration

Let us proceed to the configuration tasks. We will need to export the variables that we did in the beginning of the chapter. You could use the tip to save them in a file and source them.

Creating a user in Keystone

We will create the user using the `keystone` command as shown in the following screenshot. The output will show the UID of the user created. Note that the UID generated will be a different one for you:

```
keystone user-create --name cinder --pass c1nd3rkeypwd
```

```
root@OSControllerNode:~# source ~alokas/os.txt
root@OSControllerNode:~# keystone user-create --name cinder --pass c1nd3rpwd
+----------+----------------------------------+
| Property |              Value               |
+----------+----------------------------------+
|  email   |                                  |
| enabled  |               True               |
|    id    | d00e91b0f3d64e20adb746fed1a54d0a |
|   name   |              cinder              |
| username |              cinder              |
+----------+----------------------------------+
```

As the next step, we will add the user to the admin role in the service tenant that we created while installing Glance:

```
keystone user-role-add --user cinder --tenant service --role admin
```

This command doesn't provide any output, so if you see nothing, it is indeed good news!

Creating Cinder service in Keystone

We will create the service in Keystone so that Keystone can publish it to all the services that need it. However, in Cinder, we need to create two services, one each for version 1 and version 2.

```
keystone service-create --name cinder --type volume --description
"OpenStack Block Storage"
keystone service-create --name cinderv2 --type volumev2 --description
"OpenStack Block Storage"
```

You should see the following result:

```
root@OSControllerNode:~# keystone service-create --name cinder --type volume --description "OpenStack Block Storag
e"
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
| description |     OpenStack Block Storage      |
|   enabled   |               True               |
|      id     | a3e71c643105452cb4c8239d98b85245 |
|     name    |              cinder              |
|     type    |              volume              |
+-------------+----------------------------------+
root@OSControllerNode:~# keystone service-create --name cinderv2 --type volumev2 --description "OpenStack Block St
orage"
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
| description |     OpenStack Block Storage      |
|   enabled   |               True               |
|      id     | d9f4e0983eda4e8ab7a540441d3f5f87 |
|     name    |             cinderv2             |
|     type    |             volumev2             |
+-------------+----------------------------------+
```

We will also have to note down the IDs of both the services that we have used to create the endpoints, so note it down.

Creating Cinder endpoints

We will need to create two endpoints as well, one per service. At this point, you must know that Cinder uses `8776` as the default port.

```
keystone endpoint-create \
  --service-id a3e71c643105452cb4c8239d98b85245\
  --publicurl http://OScontrollerNode:8776/v1/%\(tenant_id\)s \
  --internalurl http://OScontrollerNode:8776/v1/%\(tenant_id\)s \
  --adminurl http://OScontrollerNode:8776/v1/%\(tenant_id\)s \
  --region dataCenterOne


keystone endpoint-create \
  --service-id d9f4e0983eda4e8ab7a540441d3f5f87 \
  --publicurl http://OScontrollerNode:8776/v2/%\(tenant_id\)s \
```

```
--internalurl http://OScontrollerNode:8776/v2/%\(tenant_id\)s \
--adminurl http://OScontrollerNode:8776/v2/%\(tenant_id\)s \
--region dataCenterOne
```

You will notice that the URLs for this are different from the other endpoints so far. This is because it has the tenant ID as a variable, and the URL will be modified during runtime by the client using the endpoint.

Modifying the configuration files

We modify the configuration file located at `/etc/cinder/cinder.conf`. We will modify three sections:

- `[default]` section
- `[database]` section
- `[keystone_authtoken]` section

Note that some sections themselves don't exist in the default configuration shipped with the distro. You should create them and add the options as follows:

- In the `[database]` section, add the MySQL user credentials:

  ```
  connection = mysql://cinder:c1nd3rpwd@OSControllerNode/cinder
  ```

- In the `[default]` section, modify the RabbitMQ user credentials and the `my_ip`. The `my_ip` is used to allow Cinder to listen on the IP address of the controller node:

  ```
  rpc_backend = rabbit
  rabbit_host = OSControllerNode
  rabbit_password = rabb1tmqpass
  auth_strategy = keystone
  my_ip = 172.22.6.95
  verbose = true
  ```

- The `[keystone_authtoken]` section will have the following information:

  ```
  auth_uri = http://OScontrollerNode:5000/v2.0
  identity_uri = http://OScontrollerNode:35357
  admin_tenant_name = service
  admin_user = cinder
  admin_password = c1nd3rkeypwd
  ```


www.allitebooks.com

Populating the Cinder database

The Cinder database will be populated using the following command:

```
/bin/sh -c "cinder-manage db sync" cinder
```

This command should be run as a root.

Finalizing the installation

The Cinder installation on the controller node is now complete. We will delete the SQLite database that is installed with the Ubuntu packages:

```
rm -rf /var/lib/cinder/cinder.sqlite
```

Let us restart the API and scheduler service to complete this part of the install:

```
service cinder-scheduler restart
service cinder-api restart
```

This concludes the first part of Cinder installation.

Storage node

In the second part of the installation, we will install the cinder-volume component on the storage node.

Understanding the prerequisites

As a prerequisite, we will need to ensure the following:

- Storage node IP address is added in the host file of the controller node (or) DNS server
- Storage node has the second hard disk attached to it

As the first step, we will install the LVM tools to ensure we create the volume groups:

```
apt-get install lvm2
```

The next step in the process is to create a volume group on the second hard disk. If we are doing the install in a production environment, the second hard drive or the other drives will be a part of disk array and will be connected using FC or iSCSI. If we are using a VM for the production deployment, the storage node can be connected to the disks using virtual disks or even raw device mappings.

We should check the presence of the disks and that they can be detected by the storage node. This can be done executing the following command:

```
fdisk -l
```

In our case, you can see that we have `sdb` and `sdc`. We are going to use `sdb` for the Cinder service. We will create a partition on the `sdb` drive:

```
fdisk /dev/sdb
```

Create a new partition by using the menu and entering `n` for new partition and `p` for primary partition. Choose the defaults for other options. Finally, enter `w` to write to the disks and exit `fdisk`.

The following screenshot demonstrates the result:

```
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):
Using default value 1
First sector (2048-83886079, default 2048):
Using default value 2048
Last sector, +sectors or +size{K,M,G} (2048-83886079, default 83886079):
Using default value 83886079

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
```

As the next step, we will create an LVM physical volume:

```
pvcreate /dev/sdb1
```

On that, we will create a volume group called `cinder-volume`:

```
vgcreate cinder-volumes /dev/sdb1
```

As a best practice, we will add `sda` and `sdb` to the filter configuration in the `/etc/lvm/lvm.conf` file.

The `sdb` is the LVM physical volume for the volumes used by Cinder and `sda` is the operating system partition, which is also LVM in our case. However, if you don't have LVM in the operating system volume, then only `sdb` needs to be added:

```
filter = [ "a/sda/","a/sdb/","r/.*/" ]
```

Therefore, the preceding filter means that it will accept LVMs on the `sda` and `sdb` and reject everything else.

Next, we will add host entries to the `/etc/hosts` file so that `OSControllerNode` and `OSStorageNode` knows the IP address of each other.

- On the storage node:

```
echo "172.22.6.95    OSControllerNode" >> /etc/hosts
```

- On the controller node:

```
echo "172.22.6.96    OSStorageNode" >> /etc/hosts
```

After this, verify that both the servers are able to ping each other by name.

Now, let us look at the checklist for the configuration, presented as follows:

Name	Info
Access to the Internet	Yes
Proxy needed	No
Proxy IP and port	Not applicable
Node name	`OSStorageNode`
Node IP address	172.22.6.96
Node OS	Ubuntu 14.04.1 LTS
Cinder DB password	`c1nd3rpwd`
Cinder Keystone password	`c1nd3rkeypwd`
Rabbit MQ password	`rabb1tmqpass`

Installing the packages

Before working on this node, ensure that you have added the Juno repository shown in *Chapter 2, Authentication and Authorization Using Keystone*.

We have already created the Cinder database when we were performing actions on the controller node, so we can directly start with installation of the packages.

We will install the `cinder-volume` package; we will install the Python client and drivers for MySQL, as this node will need to access the database on the controller node:

```
apt-get install cinder-volume python-mysqldb
```

Ensure that the installation is complete.

If you are unable to find the packages, follow the process to add the Juno repository from the second chapter, where it was done on the controller node before installing Keystone.

Modifying the configuration files

We will need to modify the `/etc/cinder/cinder.conf` file as we had done it on the controller node. We will modify the following sections:

- `[database]` section
- `[default]` section
- `[keystone_authtoken]` section

It is going to be exactly same as we did in the controller node, except for the `my_ip` directive, which will in this case, have the IP address of the storage node rather than the controller node. The configuration modification will look like this:

```
[default]
rpc_backend = rabbit
rabbit_host = OSControllerNode
rabbit_password = rabb1tmqpass
auth_strategy = keystone
my_ip = 172.22.6.96
verbose = true

[database]
connection = mysql://cinder:c1nd3rpwd@OSControllerNode/cinder

[keystone_authtoken]
auth_uri = http://OScontrollerNode:5000/v2.0
```

```
identity_uri = http://OScontrollerNode:35357
admin_tenant_name = service
admin_user = cinder
admin_password = c1nd3rkeypwd
```

If you have changed the volume group name when creating the LVM, the name needs to be changed in the `/etc/cinder/cinder.conf` under the default section.

Finalizing the installation

To finalize the installation, we will remove the SQLite db that came with the Ubuntu packages, as we will not use it:

```
rm -f /var/lib/cinder/cinder.sqlite
```

We will then restart the Linux target framework that controls iSCSI connections and then finally the cinder-volume itself:

```
service tgt restart
service cinder-volume restart
```

We now have completed the installation of Cinder.

Validating the installation

We will perform a simple validation by executing the following commands on the controller node:

- cinder service-list
- cinder list

You should see something like the following:

```
root@OSControllerNode:~# cinder service-list
+------------------+------------------+------+---------+-------+----------------------------+-----------------+
|      Binary      |       Host       | Zone |  Status | State |         Updated_at         | Disabled Reason |
+------------------+------------------+------+---------+-------+----------------------------+-----------------+
| cinder-scheduler | OSControllerNode | nova | enabled |   up  | 2015-05-10T06:19:15.000000 |       None      |
|  cinder-volume   |  OSStorageNode   | nova | enabled |   up  | 2015-05-10T06:19:15.000000 |       None      |
+------------------+------------------+------+---------+-------+----------------------------+-----------------+
root@OSControllerNode:~# cinder list
+----+--------+--------------+------+-------------+----------+-------------+
| ID | Status | Display Name | Size | Volume Type | Bootable | Attached to |
+----+--------+--------------+------+-------------+----------+-------------+
+----+--------+--------------+------+-------------+----------+-------------+
```

The `service-list` will show all the nodes that take part in the Cinder service. As you can see, our two nodes show up, which is as expected. The `cinder-list` command shows us the virtual volumes that are created (to be connected to the Nova instances), but since we don't have any, we don't expect to see any output there.

Working with Swift

Swift, as we already know, is the object store service that stores BLOBs and their metadata. The important part is that multiple copies of the object are stored for redundancy and resiliency. In most organizations, three copies of data are considered good from the redundancy standpoint, but we can choose to have more or fewer copies.

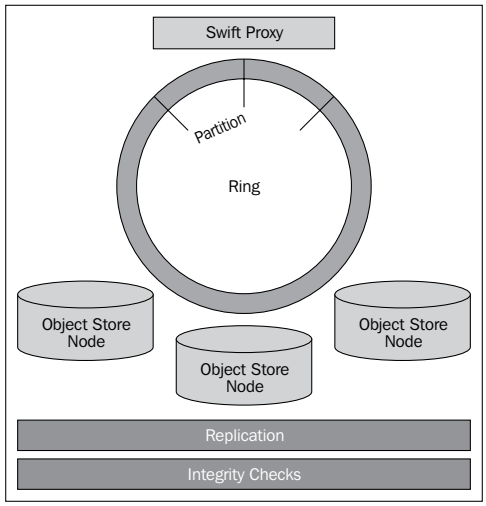
Naturally, more than one node should be used to store the objects. For the purpose of this book, we will just use one storage node—that we have also used for Cinder in the previous section—but this will have a third drive added to it, and use that for Swift.

Swift is a distributed system. In order to store and find data and ensure its integrity and redundancy, the architecture revolves around the concept of rings. Rings essentially are configuration-cum-database files that help in placing the object on the nodes and searching through them.

The rings essentially map the data to the physical device where the data is being stored. There are three rings:

- **Account ring**: This maps accounts to containers
- **Container ring**: This maps containers to objects
- **Object ring**: This maps objects to storage partitions

So in effect, if we are looking for all the objects stored for a particular account, all the three rings are checked.



The preceding diagram shows the different components of the Swift service, where ring is the key component. Rings are actually modified versions of the consistent hashing rings.

Let us take an example of the object ring. Say we have three nodes. While creating, we partition the ring in to multiple parts. Since we are using the modified version of the consistent hashing ring, we will get an equal partition size. The partitions are essentially blocks from different nodes where the objects are stored depending on the name of the file.

We create the partition size at the beginning of the ring creation, and it cannot be changed once created. The larger the partition size, the bigger the ring will be and the smoother the object distribution amongst its nodes. Once the partition size is created, `2^Part_Number` is the number of partitions created in the ring.

Rule of thumb states that we need 100 partitions per physical disk that is being used for Swift, and hence, the part power is calculated by the following formula:

*log2(number of disks * 100)*

Always choose the number on the larger side, as these days the memory (a few megabytes of it) is not a big deal.

Note that the ring partition is not to be confused with the disk partition.

The Swift proxy is the frontend between the object store nodes and the users. In our installation, we will install the proxy on the `OSControllerNode` and the `ObjectStore` on the `OSStorageNode`.

Controller node

We will install the Swift proxy and some helper packages on the controller node. Let us prepare the checklist so we have all the information handy:

Name	Info
Access to the Internet	Yes
Proxy Needed	No
Proxy IP and port	Not Applicable
Node name	`OSControllerNode`
Node IP address	172.22.6.95
Node OS	Ubuntu 14.04.1 LTS
Swift Keystone password	`sw1ftkeypwd`
Swift port	`8080`

Unlike other services, Swift doesn't need access to the database. Hence, we won't create one. This is because all the data Swift needs is kept in the rings.

Installing packages

We will install the Swift proxy packages and fall back on our reliable aptitude package manager for this:

```
apt-get install swift swift-proxy python-swiftclient python-
keystonemiddleware memcached
```

Ensure these packages are installed. Now, we see some additional packages in this list. Let us talk about them for a moment. The `memcached` package caches the objects to serve them fast to the end users, Swift client to configure the system, and the Python modules for the Keystone and Keystone middleware.

We also need to understand the difference in the terminologies when it comes to authentication between Swift and Keystone. These have been explained as follows:

Swift	Keystone
Account	Tenant / Project
User	User
Group	Role

In order to map the two preceding authentication definitions, the Keystone middleware is used. Another point to be noted is that the Swift user doesn't have any rights by default, but there is a user called the swift operator, which can modify the ACLs on the files. This mapping is also done by the Keystone middleware.

Once the packages are installed, we can move on to the configuration.

Initial configuration

The configuration steps are similar to the other services:

- Create a Keystone user and map the roles
- Create a service in Keystone
- Create an endpoint
- Modify configuration files

Creating a user in Keystone

We start by exporting the credentials. Since we have saved it in the file, we will just source the file by typing source `~alokas/os.txt`:

```
keystone user-create --name swift --pass sw1ftkeypwd
```

Once the user is created, we will then make it an admin user:

```
keystone user-role-add --user swift --tenant service --role admin
```

Creating a Swift service in Keystone

We create the service by using the following command:

```
keystone service-create --name swift --type object-store   --
description "OpenStack Object Storage"
```

Note down the ID that we will use in the next step of creating the endpoint (in this case, it is `febc806b960b496bb3e000fefe992e2b`).

Creating a Swift endpoint

We will create the Swift endpoint with the following command:

```
keystone endpoint-create \
  --service-id febc806b960b496bb3e000fefe992e2b \
  --publicurl 'http://oscontrollernode:8080/v1/AUTH_%(tenant_id)s' \
```

```
--internalurl 'http://oscontrollernode:8080/v1/AUTH_%(tenant_id)s' \
--adminurl http://oscontrollernode:8080 \
--region dataCenterOne
```

Modifying the configuration files

The Swift packages don't come with the configuration files. So, we will need to download some sample configuration files from GitHub (`https://raw.githubusercontent.com`) and then modify them:

```
mkdir /etc/swift
chown -R swift:swift /etc/swift
```

This creates a directory for Swift configuration files.

We will first download the `swift.conf` file from the GitHub repository. The following command downloads the sample configuration in the directory that we just created:

```
curl -o /etc/swift/swift.conf
https://raw.githubusercontent.com/openstack/swift/stable/juno/etc/swi
ft.conf-sample
```

If you are using a proxy server to download the file, add the `-x proxyip:port` at the end of the `curl` command to download the file.

In this file, we will have to choose a unique suffix and prefix for our environment. Remember that once chosen, the prefix cannot be changed. In this case, we will use `packtpub` for our prefix and suffix. This prefix and suffix are used in the hashing algorithm.

Edit the `/etc/swift/swift.conf` file as follows:

- In the `[swift-hash]` section:

  ```
  swift_hash_path_suffix = packtpubsuffix
  swift_hash_path_prefix = packtpub
  ```

There will already be default be a storage policy `0` in the file, verify the presence of the following:

```
[storage-policy:0]
name = Policy-0
default = yes
```

As a next step, we will download the `proxy-server` configuration file and modify its configuration:

```
curl -o /etc/swift/proxy-server.conf
https://raw.githubusercontent.com/openstack/swift/stable/juno/etc/pro
xy-server.conf-sample
```

We will make the following changes to the file:

- Under the `[DEFAULT]` section of the configuration, we will mention the user account it would use, the configuration directory, and the port on which it would bind—we have chosen `8080`:

  ```
  bind_port = 8080
  swift_dir = /etc/swift
  user = swift
  ```

- In the `[pipeline:main]` section we will enable the modules:

  ```
  pipeline = authtoken cache healthcheck keystoneauth proxy-logging
  proxy-server
  ```

  This allows for the logging and Keystone authentication.

- In the `[app:proxy-server]` section, we will enable account management:

  ```
  allow_account_management = true
  account_autocreate = true
  ```

- In the `[filter:authtoken]` section, we will configure the Keystone details. The `delay_auth_decision` value is set to `true` so that Swift waits until the Keystone middleware and Keystone check the user token and respond to Swift:

  ```
  paste.filter_factory = keystonemiddleware.auth_token:filter_
  factory
  auth_uri = http://OSControllerNode:5000/v2.0
  identity_uri = http://OSControllerNode:35357
  admin_tenant_name = service
  admin_user = swift
  admin_password = sw1ftkeypwd
  delay_auth_decision = true
  ```

- In the `[filter:keystoneauth]` section, we configure the operator role, which is effectively a mapping that mentions which `role` of Keystone will be considered an operator in Swift. These should exist in the configuration; we can just uncomment the lines rather than having to retype them:

  ```
  use = egg:swift#keystoneauth

  operator_roles = admin,_member_
  ```

- Finally, in the `[filter:cache]` section, we configure the memcached location, which is the current node in our case:

  ```
  memcache_servers = 127.0.0.1:11211
  ```

The file should appear as seen in the following screenshot:

```
root@OSControllerNode:~# cat /etc/swift/proxy-server.conf | grep -v -e "^#" | egrep -v '^[[:space:]]*$'
[DEFAULT]
bind_port = 8080
swift_dir = /etc/swift
user = swift
[pipeline:main]
pipeline = authtoken cache healthcheck keystoneauth proxy-logging proxy-server
[app:proxy-server]
use = egg:swift#proxy
allow_account_management = true
account_autocreate = true
[filter:authtoken]
paste.filter_factory = keystonemiddleware.auth_token:filter_factory
auth_uri = http://OScontrollerNode:5000/v2.0
identity_uri = http://OScontrollerNode:35357
admin_tenant_name = service
admin_user = swift
admin_password = sw1ftkeypwd
delay_auth_decision = true
[filter:keystoneauth]
use = egg:swift#keystoneauth
operator_roles = admin,_member_
[filter:healthcheck]
use = egg:swift#healthcheck
[filter:cache]
use = egg:swift#memcache
memcache_servers = 127.0.0.1:11211
[filter:ratelimit]
```

Note that this is not the full configuration. It merely shows the relevant sections to give you an idea about how it should look once done.

This concludes the installation of the Swift configuration on the controller node. Please note that in our case, the controller is also the Swift proxy server. In a production environment, we will have to perform the steps on all the different nodes acting as the proxy as the proxy server.

The storage node

Since we are using the same storage node that we used for Cinder, we already have the `DNS/Hosts` file figured out. If we choose to have more than one storage node, the same principles apply. We will quickly create a single node.

Understanding the prerequisites

The storage nodes use `rsync` in order to keep multiple copies of data in sync. Also, the XFS filesystem works very well for the BLOB storage, so we will install both of those packages:

```
apt-get install xfsprogs rsync
```

We will use the `fdisk` and create two partitions in it using the third drive that we have mounted (`/dev/sdc`). Alternatively, we could choose to partition from an already existing drive. Let us make it into an XFS filesystem and then mount it to a directory. The reason we have created two partitions is to distribute data uniformly. In the production environment, there will be several nodes and several drives per node, and we can choose to create just one partition per drive.

Check that the drive for Swift is visible to the system by looking for the `/dev/sdc` in the output of `fdisk -l`. We will format and partition the disk using the following command:

```
fdisk /dev/sdc
```

We will choose the option `n` to create a new partition, then `p` for primary, and then we will choose the partition number `1`. We will leave the initial sector as default, and for the final sector, we will set the partition size as 50 percent of the disk size. In my case, since the disk is 100 GB, I will create the first partition as 50 GB and hence use +50GB for the last sector.

We will repeat the process and leave everything to default for the second partition, and it will use the remaining space, which is the remaining 50 GB. We will then write to the partition table and come out of the `fdisk` utility, and at the end of this, we will end up with two partitions: `/dev/sdc1 and /dev/sdc2`.

You can see the screenshot for the `fdisk` utility earlier in this chapter, when we created the partition for the Cinder volumes.

As a next step, we will create filesystems on these partitions. XFS is especially suited for an object store; hence, we will use it:

```
mkfs.xfs /dev/sdc1
mkfs.xfs /dev/sdc2
```

We will now create folders and mount them:

```
mkdir -p /srv/node/sdc1
mkdir -p /srv/node/sdc2
```

Then, we add the new mount points in the `/etc/fstab` file:

```
echo "/dev/sdc1 /srv/node/sdc1 xfs
noatime,nodiratime,nobarrier,logbufs=8 0 2 " >> /etc/fstab
echo "/dev/sdc2 /srv/node/sdc2 xfs
noatime,nodiratime,nobarrier,logbufs=8 0 2 " >> /etc/fstab
```

We will mount the drives using mount `-a` command and `df -k` command to verify.

Since we only have one node, we don't actually need to configure `rsync`. However, it's a good practice to do so because it makes it easier to add nodes in the future.

Add the following to `/etc/rsyncd.conf` (after replacing the IP address of the storage node):

```
uid = swift
gid = swift
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid
address = 172.22.6.96

[account]
max connections = 2
path = /srv/node/
read only = false
lock file = /var/lock/account.lock

[container]
max connections = 2
path = /srv/node/
read only = false
lock file = /var/lock/container.lock

[object]
max connections = 2
path = /srv/node/
```

```
read only = false
lock file = /var/lock/object.lock
```

In the `/etc/default/rsync` file, set `RSYNC_ENABLE` to `true` and start the service using service `rsync` start.

Installing the packages

We will install the account, container, and object components:

```
apt-get install swift swift-account swift-container swift-object
chown swift:swift /etc/swift
```

Ensure that the installation is successful.

We will also change the permissions of the `/srv/node` folder:

```
chown -R swift:swift /srv/node
```

Modifying the configuration files

There are three configuration files that we need to modify; we will download a sample copy from the repository:

```
curl -o /etc/swift/account-server.conf \
https://raw.githubusercontent.com/openstack/swift/stable/juno/etc/account-server.conf-sample
curl -o /etc/swift/container-server.conf \
https://raw.githubusercontent.com/openstack/swift/stable/juno/etc/container-server.conf-sample
curl -o /etc/swift/object-server.conf \
https://raw.githubusercontent.com/openstack/swift/stable/juno/etc/object-server.conf-sample
```

Once the files are downloaded, make the following changes in the three different configuration files.

Account server configuration

In the `/etc/swift/account-server.conf` file, we need to ensure that the following settings are present, and are needed:

- Under the `[Default]` section:

  ```
  bind_ip = 172.22.6.96
  bind_port = 6002
  ```

```
user = swift
swift_dir = /etc/swift
devices = /srv/node
```

- Under the `[pipeline:main]` section, we will enable the account server:
  ```
  pipeline = healthcheck recon account-server
  ```
- Under the `[filter:recon]` section, just set up the metrics path:
  ```
  recon_cache_path = /var/cache/swift
  ```

Container server configuration

In the `/etc/swift/container-server.conf` file, make the same changes as in case of the account server, but replace the `bind_port` to `6001`, and under the `pipeline:main` section, replace `account-server` with `container-server`, if it already doesn't exist.

Object server configuration

In the file `/etc/swift/object-server.conf`, make the same changes as in case of the account server, but replace the `bind_port` to `6000`, and under the `pipeline:main` section, replace `account-server` with `object-server`, if it doesn't already exist.

We will now create the recon directory and ensure its proper ownership:

```
mkdir -p /var/cache/swift
chown -R swift:swift /var/cache/swift
```

We will also copy the `/etc/swift/swift.conf` file from the controller node to here:

```
scp root@OSControllerNode:/etc/swift/swift.conf
/etc/swift/swift.conf
```

Creating the rings

The most important part of the configuration is creating the rings. We will create three rings: one for account, one container, and one object ring. In order to create this, we need to select some values, which we will put in the checklist presented in this section.

We remember from the initial information of Swift that the Swift partition on the ring is technically just directories, and we discussed the rule of thumb formula to choose the partition size.

Since we have two disk partitions (`/dev/sdc1` and `sdc2`) that are being used for object storage, we will substitute in the formula and get *log 2 (2 *100)*. If you are using more than one node, the total numbers of disks need to be taken from all the nodes.

So, our log base 2 calculation yields 7.64, and so rounding it off to the next whole number, we set the partition size to 8. Since our data is not very important, we can live with two copies of it. In a production environment, we will use at least three copies. As discussed in the following table:

Name	**Info**
Part size	8
No of replicas needed	2
Minimum time between moving a partition	1 hour
No. of regions	1
No. of zones	1

If you are familiar with AWS, the regions and zones shown in the table are similar to the concept of regions and zones in AWS.

Account ring

All the rings are created by a utility called `swift-ring-builder`.

We create the `account.builder` file with three arguments, namely, partition size, number of replicas and minimum time:

```
cd /etc/swift
swift-ring-builder account.builder create 8 2 1
```

Once we have created the ring, we will have to add the nodes in there using the command format:

```
swift-ring-builder account.builder  add r1z1-
management_ip_of_storage_node:6002/device_name weight
```

We choose port `6002` for the account, `6001` for container, and `6000` for object in our configuration files above, and we have disk partitions sdc1 and sdc2. The weightage is a relative number when compared to other nodes; it is recommended that we keep it directly proportional to the amount of storage on the drive.

```
swift-ring-builder account.builder  add r1z1-172.22.6.96:6002/sdc1
100
swift-ring-builder account.builder  add r1z1-172.22.6.96:6002/sdc2
200
```

The preceding commands create the ring; we will follow the exact same steps for the container and object rings.

Container ring

We create the ring with the same parameters as in the preceding section:

```
swift-ring-builder container.builder create 8 2 1
```

We then add the drives:

```
swift-ring-builder container.builder  add r1z1-172.22.6.96:6001/sdc1
100
swift-ring-builder container.builder  add r1z1-172.22.6.96:6001/sdc2
200
```

Note the change in the port number.

Object ring

We create the ring with the same parameters:

```
swift-ring-builder object.builder create 8 2 1
```

We then add the drives:

```
swift-ring-builder object.builder  add r1z1-172.22.6.96:6000/sdc1 100
swift-ring-builder object.builder  add r1z1-172.22.6.96:6000/sdc2 200
```

This creates the object rings. Once all the rings are created, we rebalance them:

```
swift-ring-builder container.builder rebalance
swift-ring-builder object.builder rebalance
swift-ring-builder account.builder rebalance
```

Please note that this may take some time depending on your storage size.

Distributing the ring

Now we have three files: `account.ring.gz`, `container.ring.gz`, and `object.ring.gz` in the `/etc/swift` directory. We need to copy these files to all the other servers running the Swift proxy or the Swift storage components. Since we have created it on the storage node, we will copy it over to the `OScontrollerNode`:

```
scp object.ring.gz root@OSControllerNode:/etc/swift
scp container.ring.gz root@OSControllerNode:/etc/swift
scp account.ring.gz root@OSControllerNode:/etc/swift
```

Finalizing and validating the install

As a final step, we will restart the services on the nodes. On the controller node (and where ever else the proxy is installed):

```
sudo service memcached restart
sudo service swift-proxy restart
```

On all the object store nodes:

```
swift-init all start
```

This should start all the storage components. If there are errors in the configuration files, the services will show them here.

In order to validate, just execute the `swift stat` command, and you will get the output stating that Swift is configured:

```
root@OSControllerNode:~# swift stat
        Account: AUTH_c95ff542161f494eb973fc9cc9977233
     Containers: 0
        Objects: 0
          Bytes: 0
   Content-Type: text/plain; charset=utf-8
    X-Timestamp: 1431420852.83986
     X-Trans-Id: txd07047f6a52c4f3ab87b2-005551bfb4
X-Put-Timestamp: 1431420852.83986
```

We should now be able to create objects and upload them. But we will park the thought for now and revisit this when we test our cloud.

Troubleshooting steps

In the previous chapter, we have seen some troubleshooting steps, which are also applicable here, such as checking the configuration files for extra spaces in the URI configuration and restarting the services and the location of the log files.

There are some additional troubleshooting steps specific to Swift, which have been discussed in this section.

Swift authentication error

If the Swift service doesn't start or gives an authentication error when upi try to execute the `swift stat` command, follow these steps:

- Check the Swift endpoint:
  - Execute the command Keystone endpoint-list and check that Swift's endpoint has the `AUTH_ variable` in the URI of the public and the private endpoint address.
  - If there is an error in the endpoint, just delete it with the Keystone `endpoint-delete` command and recreate it as shown in this chapter.
- Check the Swift configuration files:
  - Check that the `tempauth` is not enabled in the pipeline. The `tempauth` is the temporary authentication module of Swift. Since we are offloading the authentication to Keystone, it needs to be enabled.
- Check that the `/etc/swift` directory has all the required configuration files and permissions:
  - Ensure that the `/etc/swift` directory is owned by Swift, that it has the configuration file `swift.conf`, and depending on the node type, it has policy, account, and object configuration files.
- Check that the ring files have been redistributed:
  - Ensure that the `.gz` ring files for all the three rings are available on all the nodes irrespective of it being a proxy or a storage node.
  - If not available, copy it to all the nodes using the `scp` example.

Ring files don't get created

If the ring files don't get created, ensure that you have typed in the `rebalance` command for all the three. The `rebalance` command is the one that creates the `ring.gz` file, and it needs to be run in the case of all the three rings individually.

Summary

In this chapter, we covered three types of storage services in OpenStack: BLOB storage Swift, block storage Cinder, and image storage Glance. We discussed the two components of Glance, API and registry, and installation of the service. Followed by this, we discussed the three components of Cinder, API, scheduler, and volumes and installation of the service. Finally, we discussed the two components of Swift, Swift Proxy and rings. We also discussed the three types of rings: account, container, and object rings. In addition, we verified our installation and covered some basic troubleshooting just in case you face any issues during installation.

Now that we have seen how to build a storage base to store images and other stuff, in the next chapter, we shall look at building a cloud fabric controller using Nova, which is considered the pivot around which most of the OpenStack components function.

4
Building Your Cloud Fabric Controller Using Nova

In the previous chapter, we looked at the different types of storage options available in OpenStack and the associated services.

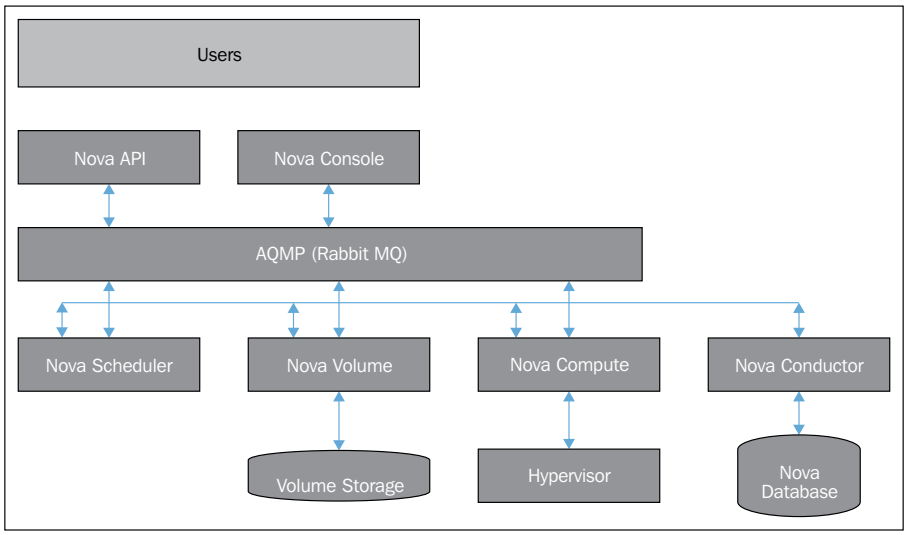
In this chapter, we will set up and configure what is considered to be the heart of the OpenStack system, the cloud computing fabric controller. It defines drivers that work with a variety of virtualization technologies such as Xen and KVM. It is the single, most important component of the system. All that we have set up, and will set up in future chapters, in some form or another will revolve around this central component. In this chapter, we will need a third server, OSComputeNode. The node is set up with Ubuntu 14.04.01 similar to the other nodes and has three network connections (management, storage, and tunnel network). We will use KVM as the hypervisor of our choice, as we already have the test cirros image in the **Qemu Copy On Write** (**QCOW2**) format when we created the Glance service.

In this chapter, we will cover the following topics:

- Working with Nova
- Installing Nova
- Exploring how the console subsystem works
- Designing the Nova environment

Working with Nova

Nova has various components that we will install. A representation of the communication flow among the components is shown in the following block diagram:



Nova utilizes other services such as Glance to provide images and Cinder for block volumes, which is also not shown in the block diagram. Nova supports multiple hypervisors such as KVM, Qemu, XenServer, and VMware.

The **Nova Compute** service is the one that takes care of communication with different hypervisors using virtualization drivers. Each and every supported **Hypervisor** has a driver associated with it. You can see the drivers and the code associated with these in the `nova/nova/virt` directory of the source code. Also, please note that the optional components such as nova-xvpnvncproxy, nova-spicehtml5proxy, or euca tools are not shown in the block diagram.

The block diagram shows the major functional components of Nova, and they can be installed on a single node or multiple nodes as they all communicate using the message bus. However, there are certain restrictions on the high availability aspect of the design, which is beyond the scope of this book.

Nova, like all OpenStack components, exposes all of its functionalities using APIs. Any custom code implementing the API, Horizon, or the command-line utilities can be used to request and modify compute resources. However, a console-based access mechanism is also given to the user; they can use this to login to the guest virtual machines that have been spun up.

The **Nova API** is the subsystem that is responsible for accepting the API calls from different sources; with Nova, we can also offer users access to their virtual machine consoles, a functionality which is not exposed in public clouds, such as that of AWS.

The console is not a mandatory component, especially if we are just trying to match the features of the public cloud. The console functionality is dependant on different services such as the VNC proxy and console authentication. Depending on the hypervisor, we may need more than these services to enable the console feature.

So, once Nova API gets a request, what does it do next? It leaves the request on the **Advanced Message Queueing Protocol** (**AMQP**) message queue for the **Nova Scheduler** to pick up. The **Nova scheduler** is responsible for finding a physical server with the available resources to fulfill the request. It finds the server, and again leaves the request in the message queue.

Nova Compute is the service that runs on the hypervisor and interacts with it. It picks up the message from the queue and creates the compute resource. The ephemeral storage for this compute is processed by the **Nova Volume**. If we need permanent storage, then we have already set up Cinder in the previous chapters; this can expose the block storage to the compute resource that we create.

Of course, the record for all actions is kept in the database, but in the case of Nova, the **Nova Conductor** is responsible for writing and reading from the database in order to keep the data coherent and avoid locking issues.

We will install all the components with the exception of **Nova Compute** on the Controller node. **Nova compute** will be installed on the compute node.

Installing Nova components

We will install most of the components on the controller node and the core components on the compute node. The communication is controlled by the AMQP message bus and all the subcomponents talk to each other using RabbitMQ, which we set up initially.

Installing on the controller node

The controller node follows the same installation procedure that we have seen in the previous services. On the controller node, we will perform the following:

- Creating a database
- Installing the services

And after this, we will configure the system by following these steps:

- Creating a Keystone user, service, and endpoints
- Modifying the configuration files

As usual, let's create our checklist to ensure we have all the pre-requisites before hand:

Name	**Info**
Access to the Internet	Yes
Proxy needed	No
Proxy IP and port	Not applicable
Node name	`OSControllerNode`
Node IP address	172.22.6.95
Node OS	Ubuntu 14.04.1 LTS
Nova DB password	`n0vapwd`
Nova Keystone password	`n0vakeypwd`
Nova port	8774

Creating the database

We create a blank database after logging in to the MySQL server by typing the following command:

```
mysql -u root -p
```

Enter the `dbr00tpassword`. password.

Once in the database, execute the following command:

```
create database nova;
```

This will create an empty database called Nova. Let's now set up the `nova` database user credentials as discussed before:

```
GRANT ALL PRIVILEGES ON nova.* TO 'nova'@'localhost' IDENTIFIED BY
'n0vapwd';
GRANT ALL PRIVILEGES ON nova.* TO 'nova'@'%' IDENTIFIED BY 'n0vapwd';
```

All this does is allow the username called `nova` using our password to be able to access the database called `nova`.

Installing components

The Nova control components are installed using the aptitude package manager with the following command:

```
apt-get install nova-api nova-cert nova-consoleauth nova-novncproxy
nova-scheduler python-novaclient nova-conductor
```

Let's talk about these components:

- `nova-api`: The Nova API accepts the API calls made to the Nova service; we can have multiple instances of this load balanced in a larger deployment.
- `nova-cert`: This is one of the support services of Nova. This is used to manage the x509 certificates that are generated and managed to secure the communications between the components.
- `nova-consoleauth`: This is a part of the Nova console subsystem; it provides authentication for users who want to use the Nova console.
- `nova-novncproxy`: Being a part of the Nova Console subsystem, this provides access to the compute using browser-based `novnc` clients.
- `nova-scheduler`: This component makes the decision regarding the compute node on which the virtual machine will reside. This service balances the load between multiple compute nodes. However, since in our setup we have a single compute node, all our VM's will reside here.
- `nova-conductor`: This proxies the connection to the Nova database.
- `python-novaclient`: This provides the Nova command-line tools. We can install the other proxy components such as the XVPN VNC proxy or the Spice HTML 5 proxy; since they are not needed for now, however, we will ignore them for the moment. If you do have to install them, then the controller node will be the ideal place to install these components.

Once the command is executed, let's wait for the packages to be downloaded and installed, which might take some time depending on the Internet connection speed.

Initial configuration

Let's now look at the initial configuration steps you need to keep in mind.

Creating the Nova user in Keystone

We will create the user in Keystone using the following command; by now, you are familiar with exporting credentials in order to use the different OpenStack command-line utilities:

```
keystone user-create --name nova --pass n0vakeypwd
```

You should be able to see the following output:

```
root@OSControllerNode:~# source ~alokas/os.txt
root@OSControllerNode:~# keystone user-create --name nova --pass n0vakeypwd
+----------+----------------------------------+
| Property |              Value               |
+----------+----------------------------------+
|  email   |                                  |
| enabled  |               True               |
|    id    | 71e19e9028244bcb9e6af1a9eebb060d |
|   name   |               nova               |
| username |               nova               |
+----------+----------------------------------+
```

We then add the user to the admin role by the following command:

```
keystone user-role-add --user nova --tenant service --role admin
```

Creating the Nova service in Keystone

The Nova service is created using the following command:

```
keystone service-create --name nova --type compute --description
"OpenStack Compute"
```

The following screenshot demonstrates the output:

```
root@OSControllerNode:~# keystone service-create --name nova --type compute
>   --description "OpenStack Compute"
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
| description |        OpenStack Compute         |
|   enabled   |               True               |
|      id     | 885fb274a5084daf92b09d378bab56a7 |
|     name    |               nova               |
|     type    |             compute              |
+-------------+----------------------------------+
```

We will have to note the ID of the service, which we will use in the next section. In our case, this is `885fb274a5084daf92b09d378bab56a7`.

Creating the Nova endpoint in Keystone

The endpoint is created using the following command; you have to replace the ID with the ID you got during your service creation:

```
keystone endpoint-create \
  --service-id 885fb274a5084daf92b09d378bab56a7\
  --publicurl http://OSControllerNode:8774/v2/%\(tenant_id\)s \
```

```
--internalurl http://OSControllerNode:8774/v2/%\(tenant_id\)s \
--adminurl http://OSControllerNode:8774/v2/%\(tenant_id\)s \
--region dataCenterOne
```

This will create the Nova endpoint in Keystone.

Modifying the configuration file

We will now be modifying a single configuration file, `/etc/nova/nova.conf`.
The following changes are done to this file:

- In the `[database]` section, do the following:
  - We will set the database connection string in the file, as this:

    ```
    connection = mysql://nova:n0vapwd@OSControllerNode/nova
    ```

- In the `[default]` section, these changes are done:
  - Configure the Rabbit MQ access, as shown here:

    ```
    rpc_backend = rabbit
    rabbit_host = OSControllerNode
    rabbit_password = rabb1tmqpass
    ```

  - Set the authentication strategy to Keystone, as follows:

    ```
    auth_strategy = keystone
    ```

  - Set the management IP address:

    ```
    my_ip = 172.22.6.95
    ```

  - Set the VNC configuration:

    ```
    vncserver_listen = 172.22.6.95
    vncserver_proxyclient_address = 172.22.6.95
    ```

- In the `[keystone_authtoken]` section, do the following:
  - Set the Keystone configuration as:

    ```
    auth_uri = http://OSControllerNode:5000/v2.0
    identity_uri = http://OSControllerNode:35357
    admin_tenant_name = service
    admin_user = nova
    admin_password = n0vakeypwd
    ```

- In the `[glance]` section, do as follows:
    - Set the Glance Host:

      ```
      host = OSControllerNode
      ```

The file will look like the following screenshot:

```
[DEFAULT]
dhcpbridge_flagfile=/etc/nova/nova.conf
dhcpbridge=/usr/bin/nova-dhcpbridge
logdir=/var/log/nova
state_path=/var/lib/nova
lock_path=/var/lock/nova
force_dhcp_release=True
libvirt_use_virtio_for_bridges=True
verbose=True
ec2_private_dns_show_ip=True
api_paste_config=/etc/nova/api-paste.ini
enabled_apis=ec2,osapi_compute,metadata
rpc_backend = rabbit
rabbit_host = OSControllerNode
rabbit_password = rabb1tmqpass
auth_strategy=keystone
my_ip=172.22.6.95
vncserver_listen = 172.22.6.95
vncserver_proxyclient_address = 172.22.6.95
verbose=true
[database]
connection = mysql://nova:n0vapwd@OSControllerNode/nova
[keystone_authtoken]
auth_uri = http://OSControllerNode:5000/v2.0
identity_uri = http://OSControllerNode:35357
admin_tenant_name = service
admin_user = nova
admin_password = n0vakeypwd
[glance]
host = OSControllerNode
```

Populating the database

We can populate the database using the following command (under `root`):

```
/bin/sh -c "nova-manage db sync" nova
```

Ensure that the database is created and we don't have any errors.

If there are any errors, please check the connection string that is mentioned in the `nova.conf` file and ensure that the MySQL instance is up.

Finalizing the installation

We will delete the SQLite database file and restart all the services by the following command:

```
rm -rf /var/lib/nova/nova.sqlite
service nova-api restart
service nova-cert restart
service nova-consoleauth restart
service nova-scheduler restart
service nova-conductor restart
service nova-novncproxy restart
```

If you have installed any other services such as `html5 proxy`, please restart them as well. This concludes the installation on the controller node.

Installing on the compute node

The compute virtualization is so prevalent that all the hardware components, especially the CPUs, are virtualization-enabled, which in essence is the CPU being virtually split and supporting additional command sets that let the virtual machine manager control the CPUs more effectively and offer a better performance.

These flags are masked when we install a hypervisor on an already virtualized machine, hence hardware support is not present. This doesn't actually mean that you cannot do this, but just means that now the second-level guests will be a little slower in their performance, as they have to do this virtualization in software rather than in hardware.

It should also be noted that some virtualization platforms don't work when they are nested (as an example, KVM), and hence Qemu needs to be used. However, modern hypervisors such as VMware also allow these CPU flags to be passed down to the guest OS as well so that the guest can run a hypervisor if it so chooses. Since virtualization is beyond the purview of this book, we will park this topic for now.

On the compute node, we will install the hypervisor, which in our case is KVM. If you are also using, for example, VMware to run this node in a virtualized environment, you will have to expose the AT-V flags and hardware acceleration so that you can run KVM.

In order to test whether we can run KVM, execute the following command:

```
egrep -c '(vmx|svm)' /proc/cpuinfo
```

If this returns `1`, which means that it can see the VT flags in the CPU and it will be able to use the CPU virtualization instruction set, in this case we are okay to proceed with KVM; otherwise, we have to use Qemu. Another way to test is to install the `cpu-checker` package and execute the `kvm-ok` command):

```
sudo apt-get install cpu-checker
kvm-ok
```

If the output is `KVM acceleration can be used`, we can go ahead and use KVM; otherwise, we have to use Qemu.

Installing KVM

Let's install the packages that are needed for KVM using aptitude:

```
sudo apt-get install qemu-kvm libvirt-bin ubuntu-vm-builder bridge-
utils
```

Optionally, if we intend to install a GUI to manage the KVM instances, install the Ubuntu GUI (or the stripped down version of it):

```
sudo apt-get install --no-install-recommends ubuntu-desktop
```

Once this is installed, we can install the virtualization manager:

```
sudo apt-get install virt-manager
```

The optional steps can be ignored if we don't want the GUI. Most people working with KVM will not be interested in the GUI; in our case, we will not be installing it.

Installing Nova compute components

Nova compute components are installed by executing the following command:

```
sudo apt-get install nova-compute sysfsutils
```

Let's talk about the following components that we are installing:

- `nova-compute`: This component is responsible for communicating with the hypervisor locally in order to create and manage the guest operating system
- `sysfsutils`: We also install the system's filesystem utilities that are used to see the different storages available; this will be used by the Nova volume, (installed as a part of `nova-compute`) in order to create the ephemeral storage

We have to ensure that the preceding components are installed without any errors.

Modifying the host files

On the controller nodes and the storage nodes, we need to modify the host entries to include the compute node and vice versa on the compute node so that they can resolve each other.

Modifying the configuration file

We will now modify the configuration file at `/etc/nova/nova.conf`; there are very subtle differences between what we changed in the controller node; for example, we don't need the database section and there are some changes in the VNC configuration. The variables are taken from the checklist that we prepared previously:

- In the `[default]` section, do this:
  - Configure the RabbitMQ access:

    ```
    rpc_backend = rabbit
    rabbit_host = OSControllerNode
    rabbit_password = rabb1tmqpass
    ```

  - Set the authentication strategy to Keystone:

    ```
    auth_strategy = keystone
    ```

  - Set the management IP address:

    ```
    my_ip = 172.22.6.97
    ```

  - Set the VNC configuration:

    ```
    vncserver_listen = 0.0.0.0
    vnc_enabled = True
    vncserver_proxyclient_address = 172.22.6.97
    novncproxy_base_url =
    http://OSControllerNode:6080/vnc_auto.html
    ```

- In the `[keystone_authtoken]` section, do the following:
  - Set the Keystone configuration:

    ```
    auth_uri = http://OSControllerNode:5000/v2.0
    identity_uri = http://OSControllerNode:35357
    admin_tenant_name = service
    admin_user = nova
    admin_password = n0vakeypwd
    ```

- In the `[glance]` section, do as follows:
    - Set the Glance host:

      ```
      host = OSControllerNode
      ```

If we don't have KVM installed, modify `/etc/nova/nova-compute.conf` and set `virt_type = qemu` (instead of `kvm`).

Finalizing the installation

We will restart the `nova-compute` service and remove the SQLite database by the following command:

```
rm -rf /var/lib/nova/nova.sqlite
service nova-compute restart
```

This concludes the installation on the compute node.

Verifying the installation

The installation can be verified on the controller node. Once on the controller node, export the credentials and list the nova services using this command:

```
nova service-list
```

Please verify that you can see both the controller services and the compute services, as shown in the following screenshot:

```
root@OSControllerNode:~# nova service-list
+----+------------------+------------------+----------+---------+-------+
--------------------+-----------------+
| Id | Binary           | Host             | Zone     | Status  | State |
d_at                 | Disabled Reason |
+----+------------------+------------------+----------+---------+-------+
--------------------+-----------------+
| 1  | nova-cert        | OSControllerNode | internal | enabled | up    |
5-26T16:10:23.000000 | -               |
| 2  | nova-consoleauth | OSControllerNode | internal | enabled | up    |
5-26T16:10:24.000000 | -               |
| 3  | nova-scheduler   | OSControllerNode | internal | enabled | up    |
5-26T16:10:21.000000 | -               |
| 4  | nova-conductor   | OSControllerNode | internal | enabled | up    |
5-26T16:10:25.000000 | -               |
| 5  | nova-compute     | OSComputeNode    | nova     | enabled | up    |
5-26T16:10:23.000000 | -               |
+----+------------------+------------------+----------+---------+-------+
```

Execute the following command:

```
nova image-list
glance image-list
```

Both of them should show the cirros image that we uploaded during the Glance installation. This also shows that Nova can see the images stored by Glance.

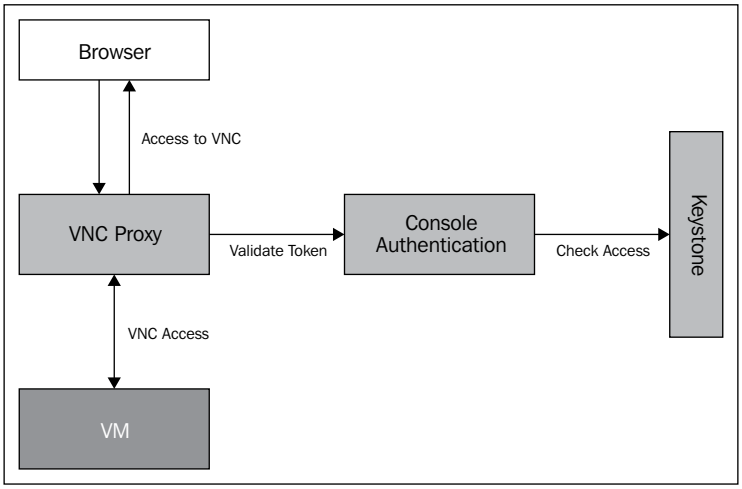
Console access

Since one of the key differentiators when compared to the public cloud is the console, and as, in this section, we will also install the console components, let's take a look at how the console subsystem works.

Later in the book, when we install the dashboard component, we will then be able generate the console URL right from the dashboard. However, we can also generate the URL using the nova CLI command:

```
nova get-vnc-console
```

This will generate the VNC URL that points to the proxy. The URL will be of the following format: `http://ProxyIP:Port/?token=axyasaas`. The proxy IP address is set in the `nova.conf` files in the previous configuration, as shown in the following figure:



The following happens when the console URL is accessed:

1. **Browser/Client** connects to VNC Proxy.
2. **VNC Proxy** talks to `nova-consoleauth` to authorize the user's token.
3. **VNC Proxy** maps the token to the private host and the port of an instance's VNC server.

   The compute host specifies the address that the proxy should use to connect in the `nova.conf` file.

4. **VNC Proxy** initiates a connection to the VNC server and continues proxying until the session times out or a user closes the session.

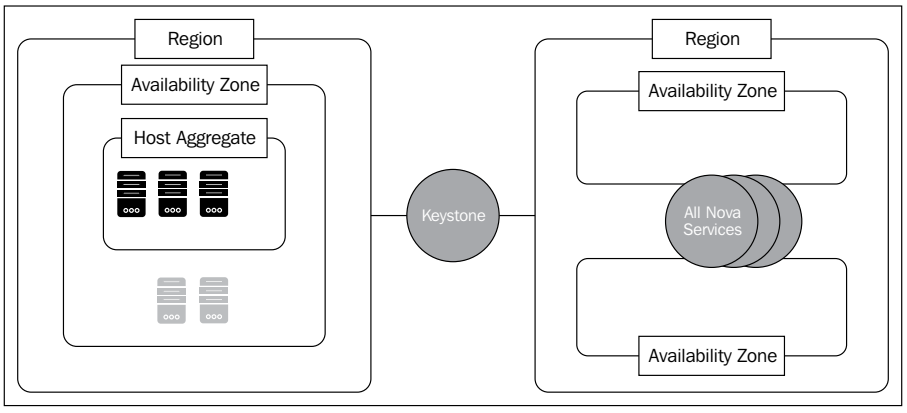
Designing your Nova environment

In a production environment, we could be running several of the tens to hundreds of Nova compute nodes. There will be possibly one component that will be used more than any other component.

Designed properly, Nova can be used to support multiple hypervisors on a cloud and provision instances in different regions. While we are not going to supply a guide on how to perform the advanced configuration of Nova, we will definitely take a look at this theoretically.

Logical constructs

The following diagram shows the different logical constructs of Nova. When architecting a production environment, we will need to think about which of these we would use depending on requirements and the scale of the cloud that we are deploying:



In this diagram, we have not shown a Nova cell as this was still an experimental feature during the time of writing the book, however, it's fairly similar to that of region, with the exception that the `nova-api` service is also shared.

Region

This is the top level construct and has all the components of Nova installed. If there are two regions, then we will have two full blown Nova installations (of all the Nova services), and the only component that is common between the two regions is that both of them use the same Keystone component for authentication.

Availability zone

An **availability zone** (**AZ**) doesn't really need any new services, but it uses the same set of the nova services that are installed for the region. This is merely a configuration change and can be quickly modified.

The reason you may want to use availability zones is to show the users the presence of fault-tolerant hardware—for example, hardware with different power supplies, and so on. A user can normally request a nova instance to be booted on the AZ. An AZ cannot exist without the host aggregates, as they in some ways are simply a way to expose the host aggregates to the end users.

The host aggregates

To overly simplify this, you can consider host aggregates as tags that are used to group compute servers. These could be servers with certain common traits, such as Hypervisors, or performance, such as servers with SSD drives or flash storage. The order of configuration, however, happens to be that the host aggregate is first created and then they can be placed in an AZ, if needed.

It should also be noted that a single compute node can be placed in multiple aggregates, just like adding multiple tags. The Nova API doesn't actually allow the choosing of host aggregates, so we normally expose this as an AZ for users to choose from.

However, the host aggregates can be chosen by the use of metadata, which can be added on both the aggregate and, say, the nova flavor, to push the servers to a certain host aggregate.

Both AZ and host aggregates can be used together. They can even be used separately. A good use case for using them together a multihypervisor cloud.

Virtual machine placement logic

The following logic is used in a virtual machine placement:

1. The client chooses the region.
2. The client queries Keystone for the Nova endpoint of that region.
3. The client submits the request to the Nova API of that region.
4. The Nova system lists the compute nodes in the region.
5. The Nova system then filters it with the AZ metadata and the second level filters the host aggregate (in the request) metadata.
6. The Nova system finds the compute nodes that are suitable for handling the VM, depending upon the size of the VM.
7. One of the compute nodes is then chosen in a round robin fashion, and the VM is spun up on this node.

In our case, we have not created any availability zones or host aggregates for the purpose of simplicity, but the logic remains the same regardless.

Sample cloud design

So, in order to understand what we have seen so far, let's take a fictitious example where your company will need the private cloud with the following qualities:

- Two major datacenters—that is, London and Boston
- Support required for two hypervisors: VMware (for production workloads) and KVM (for development workloads)
- Each DC will be fed by two separate power grids
- There will be servers with SSD as internal storages that need to be used for caching static content

Now with this in mind, let's take a look at how we should proceed from here on.

Since we have two datacenters, we can create this as two different regions. This way, each region will be independent of the other. We could also look at creating Nova cells, if we do not want the clients to first query Keystone for the endpoint URL or just use a single endpoint URL. Since they are an experimental feature, we will stick with two regions.

In each region, we will create two AZs. The servers belonging to the AZs will be connected to different power grids. This way, we can tell our users to create application servers on the different AZs for high availability.

Now, it comes down to the host aggregates. For this one, we need to think a little more. So we have this question: will the SSD servers be available for both the dev test environment and production environment?

Please remember, this is not for the OpenStack environment, but for the applications that will be running on the virtual machines that will be spun up and managed by OpenStack.

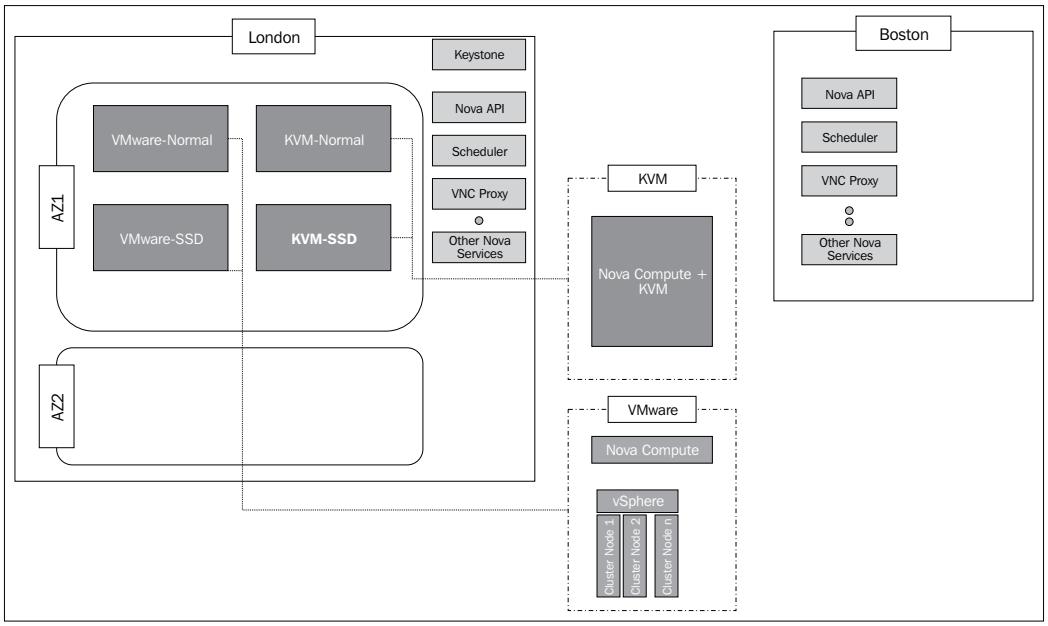
We can have up to four host aggregates in the preceding scenario:

- VMware-normal
- VMware-SSD
- KVM-normal
- KVM-SSD

Now depending on the actual use case that is supplied, we can have all four host aggregates or just three of them. We may even have more host aggregates created based on other classifications as well.

So, this is how we can design our Nova deployment. Please do note that in this case, we have also created the multihypervisor cloud, and the compute nodes that are working with VMware will talk to vSphere and provision the virtual machines. So, they can be a little under-sized, as all they do is just make API calls. The compute nodes, which also have the KVM hypervisor installed, will be bigger as they are hosting the virtual machines.

The following figure shows our sample cloud design:



So, as you can see that we have only installed Keystone in the London region, and the Boston region is also using the same server.

The diagram also takes the liberty to show the Nova compute instance sizes and functions, in case of the different hypervisors.

Troubleshooting installation

Troubleshooting an installation of Nova is similar to the ones described in the previous chapters. While we have still not spun up an instance, let's take a look at some failure scenarios when we do spin an instance up and it refuses to come up.

Please remember that the Nova log files are situated in the `/var/log/nova/` directory of the nodes where any of the Nova services are installed.

Here are some of the preliminary checks:

1. Check the Keystone log to see an authentication or token failures.
2. Check whether RabbitMQ is functional and reachable from all the different compute nodes.
3. Check whether the installation and configuration have been done properly.

If the authentication was not a problem, we follow the following procedure:

1. Check `nova-api.log` on the controller node to check whether the request was properly received. If the request was not properly handled, you will see the reason here.
2. Execute the `nova list` command and note down the instance ID of the system.
3. Execute the `nova console-log <uuid of the VM>` command to see the logs.

These logs should point you to the right direction. If the system is stuck at building or scheduling, and the VM has not begun to be built yet, then there can be two possible reasons:

- Not enough resources
- Issues with the hypervisor

The first log file to be checked will be of the nova scheduler; then check `nova-compute.log` on the compute node. These are general guidelines and further troubleshooting may be needed on the hypervisor side.

Summary

In this chapter, we were introduced to Nova, the key service of OpenStack that provisions and manages compute resources, which is also called the Cloud Fabric Controller. We then looked at the various components of Nova such as Nova scheduler, Nova compute, Nova API, Nova volume, Nova conductor, and Nova console, and their interactions among themselves using a messaging queue. Finally, we looked at the installation and configuration of these components on the compute node and the controller node.

After storage and compute services, the next step is to set up a networking service. We will set up Neutron for networking in the next chapter and take it forward.

5
Technology-Agnostic Network Abstraction Using Neutron

In the previous chapters, we started with basic Ubuntu Linux boxes and progressed to having compute as a defined service. Now, we move on to network connectivity. In this chapter, we will set up Neutron and perform some basic software-defined networking, which will allow us to provide our Nova instances with the networking they need.

Falling as it does in the learning series intended for beginners, this book will allow us to explore some straightforward examples. Once a basic understanding about networking and Neutron itself is gained, Neutron can be used to configure networking in some fancy ways.

In this chapter, we will cover the following topics:

- Understanding the software-defined network paradigm
- Getting started with Neutron
- Installing Neutron
- Troubleshooting Neutron

The software-defined network paradigm

Before we jump straight onto the architecture, installation, and basic configuration of the Neutron service, let's spend some time looking at some of the problem statements that **Software-Defined Networking** (**SDN**) tries to solve and the different approaches that it uses in order to solve them. You may choose to skip directly to the Neutron section; if you so choose, you will still be able to follow the OpenStack installation and configuration part of the book.

The biggest challenge that the cloud and multitenanted environments have posed since their inception is in the realm of network and security. We have dealt with multitenanted networks in the past, in the context of hosted datacenter service providers, and the answer to requirement for separation was **Virtual Local Area Networks** (**VLANs**), which worked well as they provided isolated segments, and the inter-VLAN routing was almost always through a firewall providing security as well.

We have come a long way since the hosted datacenter model to this new cloud world. While the basic idea of sharing resources by use of virtualization is no different from the old paradigm, the way we implement it has changed drastically.

As an example, say, if you wanted to use a hosted datacenter service in the pre-cloud world; you would put in a request, and then someone in the hosting provider's network team would create a new VLAN for you, among other things.

In the cloud world, however, services are auto-provisioned, so we could potentially have many tenants requesting resources in a day. Also, they may request more than one VLAN to create, say, three-tier architecture, not to mention the implications for security.

So now to the main question; does VLAN still not cut it? The answer is subjective; however, let's take a look at the shortcomings of the VLAN technology:

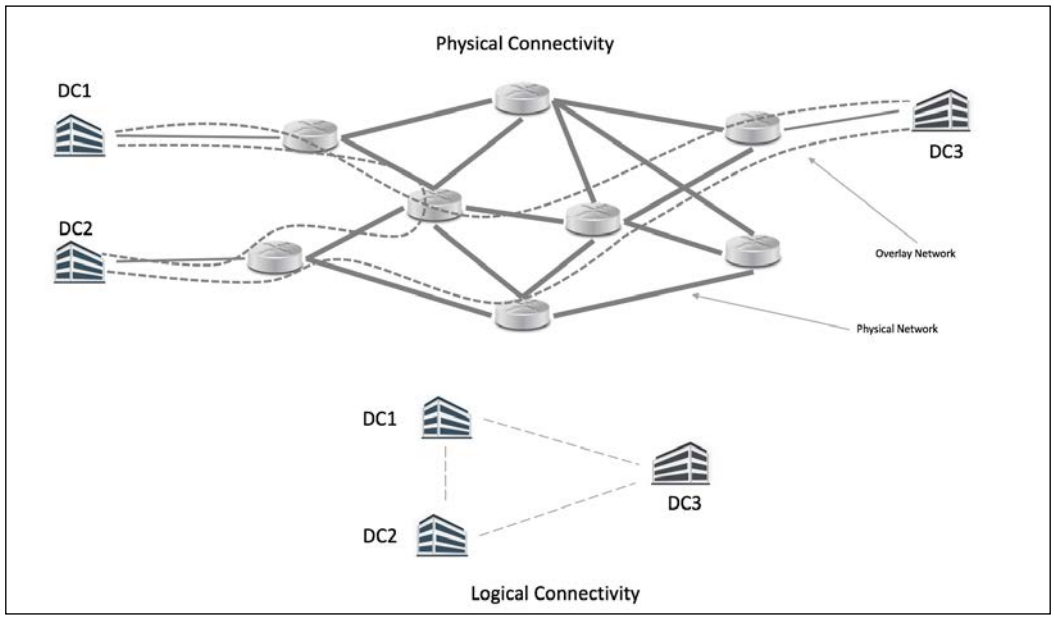
- We can have a maximum of 4,094 VLANs—remove some administrative and pre-assigned ones, and we are left with just over 4,000 VLAN's. Now this becomes a problem if we have say 500 customers in our cloud and each of them is using about 10 of them. We can very quickly run out of VLANs.
- VLANs need to be configured on all the devices in the Layer 2 (Switching) domain for them to work. If this is not done, the connectivity will not work.
- When we use VLANs, we will need to use **Spanning Tree Protocol** (**STP**) for loop protection, and thereby we lose a lot of multipathing ability (as most multi-path abilities are L3 upwards and not so much on the Layer 2 network).
- VLANs are site-specific, and they are not generally extended between two datacenters. In the cloud world, where we don't care where our computing resources stay, we would like to have access to the same networks, say for a **disaster recovery** (**DR**) kind of scenario.

One of the methods that can alleviate some of the aforementioned problems is the use of an overlay network. We will then also take a look at an upcoming technology that is also used, called **Open Flow**.

What is an overlay network?

An overlay network is a network running on top of another network. The different components or nodes in this kind of network are connected using virtual links rather than physical ones.

The following diagram shows the concept of overlay networking between three datacenters connected by an ISP:



An overlay network is created on top of a physical network, and the physical network determines the multipath and failover conditions, as the overlay is only as good as the network below it.

An overlay network generally works by encapsulating the data in a format that the underlay network understands and optionally may also encrypt it.

Components of overlay networks

An overlay network normally contains the following components:

- Physical connectivity and an underlay routable/switched network
- Overlay endpoints

Overlay endpoint devices that need the connectivity. It should be noted that the overlay endpoint must be able to route/switch on its own, as the underlying network normally has no idea of the protocols that are being transported by the overlay.

Overlay technologies

There are several overlay technologies that help in creating overlay networks, some more complicated than others but ultimately solving similar problem statements.

Let's take a brief look at a few of them and understand some of the working concepts. We will not be covering how these are configured on the network layer or any advanced concepts, as these are beyond the scope of this book.

GRE

Generic Routing Encapsulation (**GRE**) is possibly one of the first overlay technologies that existed. It was created in order to start creating standardization in the network Layer-3 protocols.

If you remember, before TCP/IP became the de facto standard for networking, there were several Layer 3 protocols, such as SPX/IPX, Banyan, and Apple Talk. The service providers could definitely not run all of these, as it would become a networking nightmare. Hence, GRE tunneling was used.

GRE encapsulates every layer-3 payload you throw at it inside an IP packet, by setting the destination as the address of the remote tunnel endpoint, and then sends it down the wire; it performs the opposite function on the IP packet at the other end. This way, the underlay network sees the packet as a general IP packet and routes it accordingly. Thus, the endpoints can happily now talk another language if they choose to.

A new form of GRE called **Multipoint GRE** (**MGRE**) evolved in the later years to help with the multipoint tunnel requirements.

VXLAN

Virtual Extensible LAN (**VXLAN**) is an advancement of the VLAN technology itself. It actually brings in two major enhancements, described as follows:

- **Number of VXLANs possible**: Theoretically, this has been beefed up to 16 million VXLANs in a network, thereby giving ample room for growth. However, the devices may support between 16,000–32,000 VXLANs.

- **Virtual tunnel endpoint** (**VTEP**): VXLAN also supports VTEP, which can be used to create a Layer-2 overlay network atop the Layer 3 endpoints.

Underlying network considerations

Overlay technologies are supported in most devices, and if they are not, then they may just be a software update away. The requirements, as we have already seen, are that it needs a routable underlay network and the impact is determined by the properties of the underlying network itself. However, let's take some pointers from one of the most used networks (TCP IP over Ethernet networks).

The **Maximum Transmission Unit** (**MTU**) is the largest **Protocol Data Unit** (**PDU**) that is normally transmitted in any layer. We just need to bear in mind that the usable MTU decreases a little due to the encapsulation that is needed. In case of Ethernet, the MTU is 1,500 bytes.

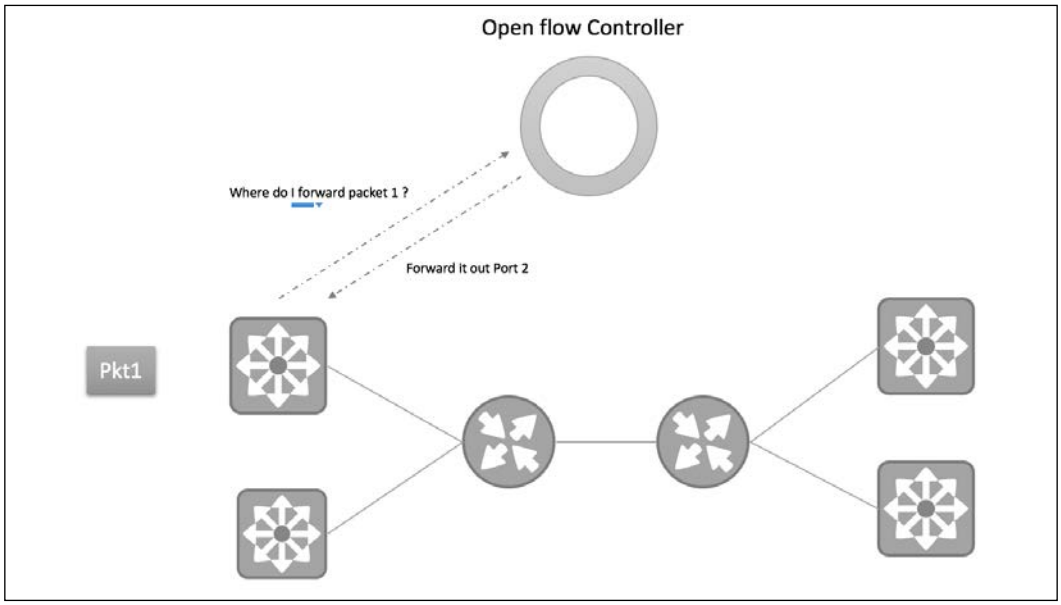
When using the overlay, the actual MTU, and thereby the maximum segment size (MSS), are reduced. In order to alleviate the problem, you may choose to use jumbo frames (up to a 9,000-byte MTU). If you are unable to use this, then the data transfer speeds will reduce because the amount of data being sent in each packet is reduced (if the MTU was adjusted) or due to fragmentation (if the MTU was not adjusted).

Open flow

Open flow is a new technology that works on the principle of the separation of the control and data planes in the network technology. In traditional networks, these planes reside in the device, but with open flow, the data plane would still reside in the switch, and the control plane is moved to a separate controller.

Open flow is considered the beginning of SDN itself, as we can now define the path that the packets need to follow entirely in the software without having to change any configuration on the devices themselves. Since open flow, there have been several topics added to the concept of SDN, principally to make it backward compatible.

The following diagram presents how the forwarding is done in case of an open flow system in place:



The devices talk to the controller and forward the packet details. The controller then tells the device the data path to follow, after which the device caches this information and uses it to forward future packets.

There are several examples of open flow controllers that are available in the market, but some popular ones are as follows:

- Open daylight
- Flood light
- NEC open flow controller
- Arista controller
- Brocade vyatta controller

Underlying network consideration

The underlying network devices should be able to understand open flow itself. So while in most cases this may be a software upgrade, it could potentially also mean upgrading the network hardware if the devices don't already decouple the data and control planes.

Equipped with this information, let us look at the Neutron component of OpenStack.

Neutron

Neutron replaced an older version of the network service called Quantum, which was introduced as a part of the Folsom release of OpenStack. Before Quantum came into the picture, the networking of the Nova components was controlled by Nova networking, a subcomponent of Nova. The name of the networking component was changed from Quantum to Neutron due to a trademark conflict (Quantum was a trademark of a tape-based backup system).

While Neutron is the way to go, if you need only simple networking in your cloud, you can still choose to use the Nova network feature and ignore the Neutron service completely. But if you do go the Neutron route, you can easily offer several services, such as load balancing as a service (using HA proxy) and VPN as a service (openswan). Neutron has a component on the controller node called the neutron server, along with a bunch of agents and plugins that communicate with each other using a messaging queue. Depending on the type of deployment, you can choose the different agents that you want to use.

Some plugins that are available today with Neutron include but are not limited to the following:

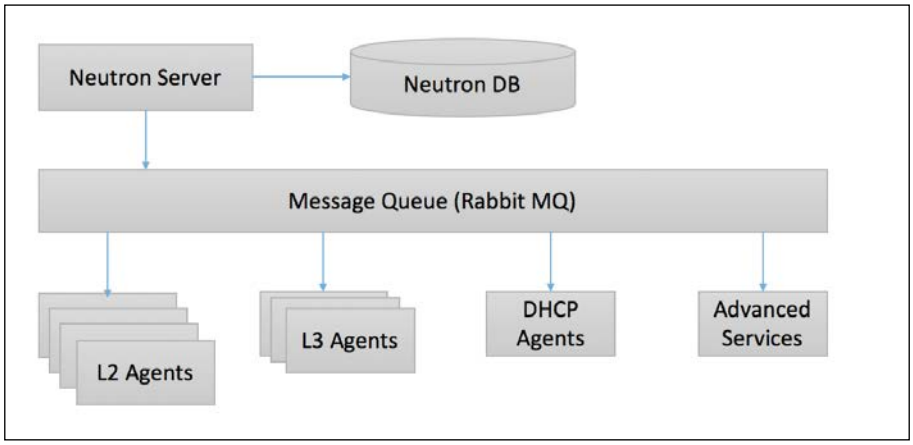
- NEC openflow
- Open vSwitch
- Cisco switches (NX-OS)
- Linux bridging
- VMware NSX
- Juniper opencontrail
- Ryu network OS
- PLUMgrid Director plugin
- Midokura Midonet plugin
- OpenDaylight plugin

You can choose to write more of these, and the support is expanding every day. So, by the time you get on to implementing it, maybe your favorite device vendor will also have a Neutron plugin that you can use.

In order to view the updated list for plugins and drivers, refer to the OpenStack wiki page at `https://wiki.openstack.org/wiki/Neutron_Plugins_and_Drivers`.

Architecture of Neutron

The architecture of Neutron is simple, but it is with the agents and plugins where the real magic happens! Neutron architecture has been presented in the following diagram:



Let's discuss the role of the different components in a little detail.

The Neutron server

The function of this component is to be the face of the entire Neutron environment to the outside world. It essentially is made up of three modules:

- **REST service**: The REST service accepts API requests from the other components and exposes all the internal working details in terms of networks, subnets, ports, and so on. It is a WSGI application written in Python and uses port 9696 for communication.
- **RPC service**: The RPC service communicates with the messaging bus and its function is to enable a bidirectional agent communication.
- **Plugin**: A plugin is best described as a collection of Python modules that implement a standard interface, which accepts and receives some standard API calls and then connects with devices downstream. They can be simple plugins or can implement drivers for multiple classes of devices.

The plugins are further divided into core plugins, which implement the core Neutron API, which is Layer 2 networking (switching) and IP address management. If any plugin provides additional network services, we call it the service plugin—for example, Load Balancing as a Service (LBaaS), Firewall as a Service (FWaaS), and so on.

As an example, **Modular Layer 2** (**ML2**) is a plugin framework that implements drivers and can perform the same function across ML2 networking technologies commonly used in datacenters. We will use ML2 in our installation to work with **Open vSwitch** (**OVS**).

L2 agent

The L2 agent runs on the hypervisor (compute nodes), and its function is simply to wire new devices, which means it provides connections to new servers in appropriate network segments and also provides notifications when a device is attached or removed. In our install, we will use the OVS agent.

L3 agent

The L3 agents run on the network node and are responsible for static routing, IP forwarding, and other L3 features, such as DHCP.

Understanding the basic Neutron process

Let's take a quick look at what happens when a new VM is booted with Neutron. This shows all the steps that take place during the Layer 2 stage:

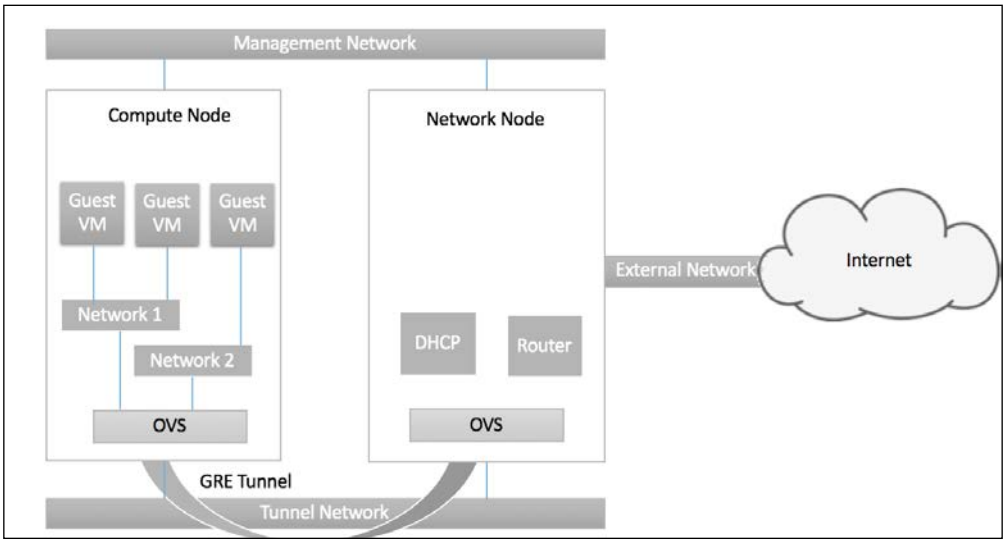
1. Boot VM start.
2. Create a port and notify the DHCP of the new port.
3. Create a new device (virtualization library – libvirt).
4. Wire port (connect the VM to the new port).
5. Complete boot.

Networking concepts in Neutron

It is also a good idea to get familiar with some of the concepts that we will come across while dealing with Neutron, so let's take a look at some of them. The networking provides multiple levels of abstraction:

- **Network**: A network is an isolated L2 segment, analogous to a VLAN in the physical networking world.
- **Subnet**: This is a block of IP addresses and the associated configuration state. Multiple subnets can be associated with a single network (similar to secondary IP addresses on switched virtual interfaces of a switch).
- **Port**: A port is a connection point to attach a single device, such as the NIC of a virtual server, to a virtual network. We have seen physical ports to which we plug our laptops or servers into them; the virtual ones are quite analogous to those, with the difference that these ports belong to a virtual switch and we connect it using a virtual wire to our servers.
- **Router**: A router is a device that can route traffic between different subnets and networks. Any subnets on the same router can talk to each other without a routing table if the security groups allow the connection.

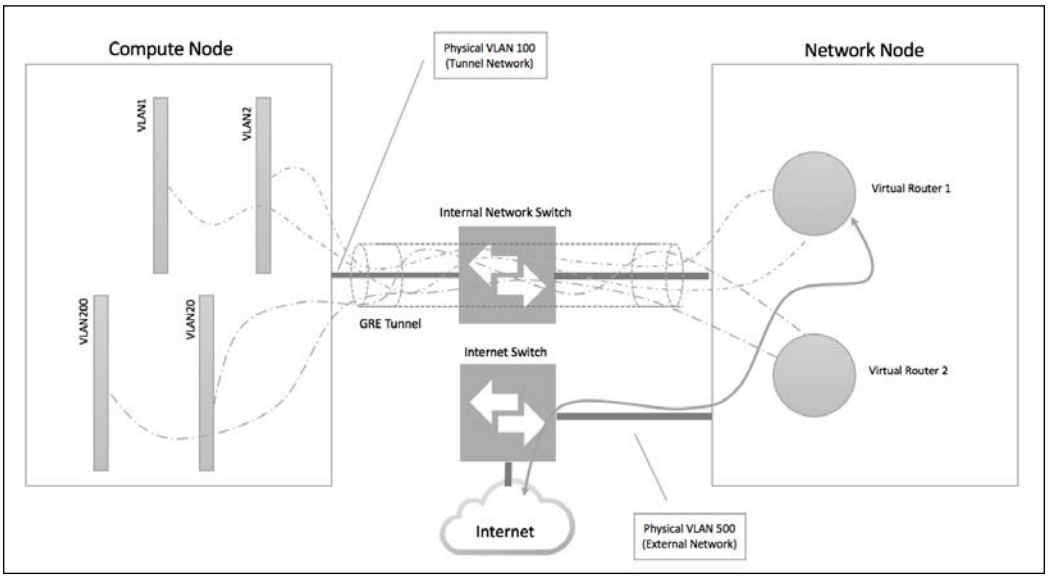
In order to express this better, let's take a look at the following diagram, which shows the connectivity between the Compute node and Network nodes—the management network is used by administrators to configure the nodes and other management activities (these networks and their purposes have been described in *Chapter 1*, *An Introduction to OpenStack*):



The **Tunnel Network** exists between the compute nodes and the network node and serves to build a **GRE Tunnel** between the two. This **GRE Tunnel**, as we know, encapsulates different networks created on the compute nodes, so that the physical fabric doesn't see any of it. In our configurations, we will set up the VLAN ranges for this purpose.

The network node primarily performs the Layer 3 functionality, be it routing between the different networks or routing the networks to an external network using its external interface and the OVS bridge created on the **Network Node**. In addition, it also performs other L3-related functions such as firewalling and load balancing. It also terminates the elastic IPs at this level.

So let's take a look at how this would look in the real world:



As you can see from the diagram, the physical network has absolutely no idea of the existence of the different VLANs, which will now be assigned to different tenants or the same tenant for different purposes. We can create as many VLANs as needed, and the underlying physical network will not be affected.

The virtual router in the network node will be responsible for routing between the different VLANs, and the router may also provide access to the Internet using another network interface in the network node (we call it the external network).

To summarize, ports are the configurations on the networks where the guest VM can be connected. The network is equivalent to the virtual switch VLAN in which there may be one or more subnets. It is on a layer 2 network domain. Different networks are connected to the router on the **Network Node** using a **GRE Tunnel**, and each network is encapsulated with a single VLAN ID for identification.

Installing Neutron

The installation of this service takes place over three nodes: the controller node, the network node, and the compute node.

- The controller node will have the Neutron server component
- The compute nodes will have the L2 agents installed
- The Network node will have the Layer 3 agents installed

Installing on the controller node

The same installation procedure that we have seen in the previous services will be followed on the controller node. We will carry out the following steps:

1. Create a database.
2. Install the services.

After this, we will initially configure the system, and perform the following steps:

1. Create a Keystone user, service, and endpoints.
2. Modify configuration files.

As usual, let's create our checklist to ensure we have all the prerequisite data beforehand:

Name	**Info**
Access to the Internet	Yes
Proxy Needed	No
Proxy IP and port	Not applicable
Node name	`OSControllerNode`
Node IP address	172.22.6.95
Node OS	Ubuntu 14.04.1 LTS

Name	Info
Neutron DB password	`n3utron`
Neutron Keystone password	`n3utronpwd`
Neutron port	9696
Nova Keystone password	`n0vakeypwd` (From the previous chapter)
Rabbit MQ password	`rabb1tmqpass`
Metadata password	`m3tadatapwd`

Creating the database

We create a blank database after logging in to the MySQL server:

```
mysql -u root -p
```

Enter the password: `dbr00tpassword`

Once in the database, execute the following command:

```
create database neutron;
```

This will create an empty database called `neutron`. Let us now set up the Nova database user credentials as we did earlier:

```
GRANT ALL PRIVILEGES ON nova.* TO 'neutron'@'localhost' IDENTIFIED BY
'n3utron';
GRANT ALL PRIVILEGES ON nova.* TO 'neutron'@'%' IDENTIFIED BY
'n3utron';
```

This allows the username `neutron`, using our password, to access the database called `neutron`.

Installing Neutron control components

The Neutron control components are installed using the aptitude package manager by using the following command:

```
sudo apt-get install neutron-server neutron-plugin-ml2 python-
neutronclient
```

Ensure that these are installed successfully.

Initial configuration

Now, let's look at some of the initial configuration tasks on the controller node.

Creating the Neutron user in Keystone

Create the user in Keystone; by now, you will be familiar with exporting the credentials in order to use the different OpenStack command line utilities:

```
keystone user-create --name neutron --pass n3utronpwd
```

You should see something like the following screenshot:

```
root@OSControllerNode:~# source ~alokas/os.txt
root@OSControllerNode:~# keystone user-create --name neutron --pass n3utronpwd
+----------+----------------------------------+
| Property |              Value               |
+----------+----------------------------------+
|  email   |                                  |
| enabled  |               True               |
|    id    | 39a73cffd82140bea62b95c47037492a |
|   name   |             neutron              |
| username |             neutron              |
+----------+----------------------------------+
```

Then assign the `admin` role to the user by running the following command:

```
keystone user-role-add --user neutron --tenant service --role admin
```

Creating the Neutron service in Keystone

The Neutron service is created using the following command:

```
keystone service-create --name neutron --type network --description
"OpenStack Networking"
```

The service will look as follows:

```
root@OSControllerNode:~# keystone service-create --name neutron --type network
-description "OpenStack Networking"
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
| description |       OpenStack Networking       |
|   enabled   |               True               |
|      id     | 73376c096f154179a293a83a22cce643 |
|     name    |             neutron              |
|     type    |             network              |
+-------------+----------------------------------+
```

We will have to note the id of the service, which we will use in the next section. In our case the id is `73376c096f154179a293a83a22cce643`.

Creating the Neutron endpoint in Keystone

The endpoint is created using the following command, where you replace the ID with the ID you received during service creation:

```
keystone endpoint-create \
  --service-id 73376c096f154179a293a83a22cce643\
  --publicurl http://OSControllerNode:9696 \
  --internalurl http://OSControllerNode:9696 \
  --adminurl http://OSControllerNode:9696 \
  --region dataCenterOne
```

The following is the output:

```
root@OSControllerNode:~# keystone endpoint-create \
>   --service-id 73376c096f154179a293a83a22cce643\
>   --publicurl http://OSControllerNode:9696 \
>   --internalurl http://OSControllerNode:9696 \
>   --adminurl http://OSControllerNode:9696 \
>   --region dataCenterOne

+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
|   adminurl  |   http://OSControllerNode:9696   |
|      id     | 79d980b4bda9423f9a7e7d48fd9a9649 |
| internalurl |   http://OSControllerNode:9696   |
|  publicurl  |   http://OSControllerNode:9696   |
|    region   |          dataCenterOne           |
|  service_id | 73376c096f154179a293a83a22cce643 |
+-------------+----------------------------------+
```

Modifying the configuration files

On the controller node, we have a few files to modify:

- `/etc/neutron/neutron.conf`: This file is used to configure Neutron
- `/etc/neutron/plugins/ml2/ml2_conf.ini`: This file helps configure the ML2 plugin
- `/etc/nova/nova.conf`: This allows Nova to use Neutron rather than the default Nova networking

Let's view one file at a time. However before we proceed, we will need the ID of the service tenant that we created. We can obtain it by using the Keystone command `keystone tenant-list` and picking the ID of the service tenant.

We will have to export the variables (or source the file, as we have done in the past) as shown in the following screenshot:

```
root@OSControllerNode:~# source ~alokas/os.txt
root@OSControllerNode:~#
root@OSControllerNode:~# keystone tenant-list
+----------------------------------+-------------+---------+
|                id                |     name    | enabled |
+----------------------------------+-------------+---------+
| c95ff542161f494eb973fc9cc9977233 | firsttenant |   True  |
| 8067841bed8547b0a21459ff4c8d58f7 |   service   |   True  |
+----------------------------------+-------------+---------+
```

So, the service tenant ID is `8067841bed8547b0a21459ff4c8d58f7`. This will be different for you, so substitute this in the following configuration.

In the `etc/neutron/neutron.conf` file, make the following changes:

- Under the `[database]` section:

```
connection = mysql://neutron:n3utron@OSControllerNode/neutron
```

- Under the `[default]` section:

```
rpc_backend = rabbit
rabbit_host = OSControllerNode
rabbit_password = rabb1tmqpass
auth_strategy = keystone
core_plugin = ml2
service_plugins = router
allow_overlapping_ips = True
notify_nova_on_port_status_changes = True
notify_nova_on_port_data_changes = True
nova_url = http://OSControllerNode:8774/v2
nova_admin_auth_url = http://OSControllerNode:35357/v2.0
nova_region_name = dataCenterOne
nova_admin_username = nova
nova_admin_tenant_id = 8067841bed8547b0a21459ff4c8d58f7
nova_admin_password = n0vakeypwd
verbose = True
```

- Under the `[keystone_authtoken]` section:

```
auth_uri = http://OSControllerNode:5000/v2.0
identity_uri = http://OSControllerNode:35357
admin_tenant_name = service
admin_user = neutron
admin_password = n3utronpwd
```

These changes need to be done in the Neutron configuration.

Now, in the `/etc/neutron/plugins/ml2/ml2_conf.ini` file, make the following changes:

- In the `[ml2]` section, enable the GRE and flat networking as follows:

```
type_drivers = flat,gre
tenant_network_types = gre
mechanism_drivers = openvswitch
```

- In the `[ml2_type_gre]` section, enable the tunnel ID ranges—these don't need to be in the physical network, as they will only be between the compute and network node:

```
tunnel_id_ranges = 1:1000
```

- In the `[securitygroup]` section, make the following changes:

```
enable_security_group = True
enable_ipset = True
firewall_driver=
neutron.agent.linux.iptables_firewall.
OVSHybridIptablesFirewallDriver
```

The next set of changes is to be made in the `/etc/nova/nova.conf` file.

- In the `[default]` section, set the following:

```
network_api_class = nova.network.neutronv2.api.API
security_group_api = neutron
linuxnet_interface_driver = nova.network.linux_net.
LinuxOVSInterfaceDriver
firewall_driver =
nova.virt.firewall.NoopFirewallDriver
```

- In the `[neutron]` section, enable the metadata-related configuration, which will be enabled on the network node:

```
url = http://OSControllerNode:9696
auth_strategy = keystone
admin_auth_url = http://OSControllerNode:35357/v2.0
admin_tenant_name = service
admin_username = neutron
admin_password = n3utronpwd
service_metadata_proxy = True
metadata_proxy_shared_secret = m3tadatapwd
```

This will enable Neutron for Nova in line with the configuration.

Setting up the database

The Neutron database can now be populated by running the following command as root user:

```
/bin/sh -c "neutron-db-manage --config-file /etc/neutron/neutron.conf
--config-file /etc/neutron/plugins/ml2/ml2_conf.ini upgrade juno"
neutron
```

Ensure that the command does not result in an error. If it does result in an error, check if the configuration settings have been changed according to the preceding section.

Finalizing the installation

To finalize the installation, delete the SQLite database that comes along with the Ubuntu packages:

```
rm -rf /var/lib/neutron/neutron.sqlite
```

Restart all the services, including Nova and Neutron, by running the following commands:

```
service nova-api restart
service nova-scheduler restart
service nova-conductor restart
service neutron-server restart
```

At this point, the installation is complete on the controller node.

Validating the installation

In order to validate the installation, let's execute some Neutron commands:

```
neutron ext-list
neutron agent-list
```

Make sure that the commands do not throw up any error.

Installing on the network node

The network node is the fourth and final node that we will be using for our setup. This node needs to have at least three network cards—you may recall this from the architecture design that we saw in *Chapter 1*, *An Introduction to OpenStack*:

- Management network
- External network
- Tunnel network

The roles of these networks are fairly straightforward: the management network installs and manages the nodes; the external network provides access to the network; and the tunnel network is used to tunnel traffic between the compute nodes and the network nodes.

Before we begin working, let's prepare our checklist so that we have all the information about the system handy, as follows:

Name	**Info**
Access to the Internet	Yes
Proxy needed	No
Proxy IP and port	Not applicable
Node name	`OSNetworkNode`
Node IP address	172.22.6.98 – Mgmt network 10.0.0.1 – Tunnel interface
Node OS	Ubuntu 14.04.1 LTS
Neutron DB password	`n3utron`
Neutron Keystone password	`n3utronpwd`
Neutron port	9696
Nova Keystone password	`n0vakeypwd` (From the previous chapter)

Name	Info
Rabbit MQ password	`rabb1tmqpass`
Metadata password	`m3tadatapwd`
External interface	`eth2`

Setting up the prerequisites

On the network node, we will need to make some changes, which will help us set up network forwarding before we go into the actual installation.

We start by editing the `/etc/sysctl.conf` file to ensure the following parameters are set:

- `net.ipv4.ip_forward=1`
- `net.ipv4.conf.all.rp_filter=0`
- `net.ipv4.conf.default.rp_filter=0`

These lines are needed on the network node in order for it to be able to forward packets from one interface to the other, essentially behaving like a router. We also disable the **Reverse Path** (**rp**) filter so that the kernel doesn't do the source validation; this is required so that we can use the elastic IPs. In order to ensure that the changes take effect, reload the system control as follows:

```
sudo sysctl -p
```

This command reloads the `sysctl.conf` file that we modified.

Installing Neutron packages

We will install the Neutron packages for the ML2 plugin and the OVS agents along with Layer 3 and DHCP agents:

```
sudo apt-get install neutron-plugin-ml2 neutron-plugin-openvswitch-
agent neutron-l3-agent neutron-dhcp-agent
```

Depending on your Internet speed, this might take some time.

Once the packages are installed, we will configure them. The Neutron configuration is similar to that on the controller node; the ML2 Plugin, DHCP agent, and the L3 agents are additional. We will also configure the metadata agent, which will be used to push the metadata on all the compute nodes when they come up.

Initial configuration on the network node

Now, let us look at some of the initial configuration tasks on the network node.

Neutron configuration

In the `/etc/neutron/neutron.conf` file, make the following changes:

- Under the `[database]` section:
  - Remove any connection string that is present, as the database access is not needed directly by the network node
- Under the `[default]` section:

```
rabbit_host = OSControllerNode
rabbit_password = rabb1tmqpass
auth_strategy = keystone
core_plugin = ml2
service_plugins = router
allow_overlapping_ips = True
verbose = True
```

- Under the `[keystone_authtoken]` section:

```
auth_uri = http://OSControllerNode:5000/v2.0
identity_uri = http://OSControllerNode:35357
admin_tenant_name = service
admin_user = neutron
admin_password = n3utronpwd
```

ML2 plugin

In the `/etc/neutron/plugins/ml2/ml2_conf.ini` file, make the following changes.

- In the `[ml2]` section, enable GRE and flat networking as follows:

```
type_drivers = flat,gre
tenant_network_types = gre
mechanism_drivers = openvswitch
```

- In the `[ml2_type_gre]` section, enable the tunnel ID ranges; these don't need to be in the physical network, as they will only be between the compute and network node:
    - `tunnel_id_ranges = 1:1000`
- In the `[securitygroup]` section, make the following changes:

```
enable_security_group = True
enable_ipset = True
firewall_driver=
neutron.agent.linux.iptables_firewall.OVSHybridIpt
ablesFirewallDriver
```

- In the `[ml2_type_flat]` section, make this change:

```
flat_networks = external
```

- Under the `[ovs]` section for OVS (set the interface address for GRE), make these changes:

```
local_ip = 10.0.0.1
enable_tunneling = True
bridge_mappings = external:br-ex
```

- Under the `[agent]` section, enable the GRE tunnels:

```
tunnel_types = gre
```

Configuring agents

We will configure three agents for Neutron: the Layer 3 agent for routing, the DHCP agent to provide DHCP services to the compute nodes, and the metadata agent, which will push the metadata on to the compute nodes.

Layer 3 agent

The Layer 3 agent will provide the routing services; we only have to provide the external bridge name. Open and edit the `/etc/neutron/l3_agent.ini` file as follows:

- In the `[default]` section, edit the following:

```
interface_driver =
neutron.agent.linux.interface.OVSInterfaceDriver
use_namespaces = True
external_network_bridge = br-ex
router_delete_namespaces = True
verbose = True
```

Layer 3 agent

This is configured by the `/etc/neutron/dhcp_agent.ini` file:

- Under the `[default]` section, make these changes:

```
interface_driver =
neutron.agent.linux.interface.OVSInterfaceDriver
dhcp_driver = neutron.agent.linux.dhcp.Dnsmasq
use_namespaces = True
dhcp_delete_namespaces = True
verbose = True
```

We can also configure other settings using the masquerading file. We can set any DHCP option; for example, it is recommended that we lower the MTU of the interface by 46 bytes so there will be no fragmentation. If we have jumbo frame support, then this step may not be required.

```
dnsmasq_config_file = /etc/neutron/dnsmasq-neutron.conf
```

We just add the preceding line in the `dhcp_agent.ini` file and then create a new file called `/etc/neutron/dnsmasq-neutron.conf` and the line:

```
dhcp-option-force = 26,1454
```

This sets the DHCP option 26 (MTU setting) to 1454. Please note that, if the operating system doesn't honor this setting, it will not have any impact.

Configuring the metadata agent

We finally configure the metadata agent using the `etc/neutron/metadata_agent.ini` file as follows:

- Under the `[default]` section:

```
auth_url = http://OSControllerNode:5000/v2.0
auth_region = dataCenterOne
admin_tenant_name = service
admin_user = neutron
admin_password = n3utronpwd
nova_metadata_ip = OSControllerNode
metadata_proxy_shared_secret = m3tadatapwd
verbose = True
```

Please remember that the metadata shared password should be the same on both the controller node configuration that we did and on the network node.

Setting up OVS

The entire Neutron configuration is based on the OVS. We need to create a bridge (`br-ex`) that will point to the external network.

```
service openvswitch-switch restart
ovs-vsctl add-br br-ex
```

We now need to add the interface to the external bridge. This interface should be the one pointing to the external physical network world. In my case, it is `eth2`. Modify it to reflect the right interface name in your environment:

```
ovs-vsctl add-port br-ex eth2
```

This adds the external network to the bridge that OpenStack knows as `br-ex`.

Verify that the interface is added by executing the following command:

```
ovs-vsctl show
```

You will receive confirmation as shown in the following screenshot:

```
root@OSNetworkNode:~# ovs-vsctl show
bd237ae4-1dc0-4379-afb4-e175e3ddce9f
    Bridge br-ex
        Port "eth2"
            Interface "eth2"
```

Finalizing the installation

We now need to restart all the services we have installed by running the following commands:

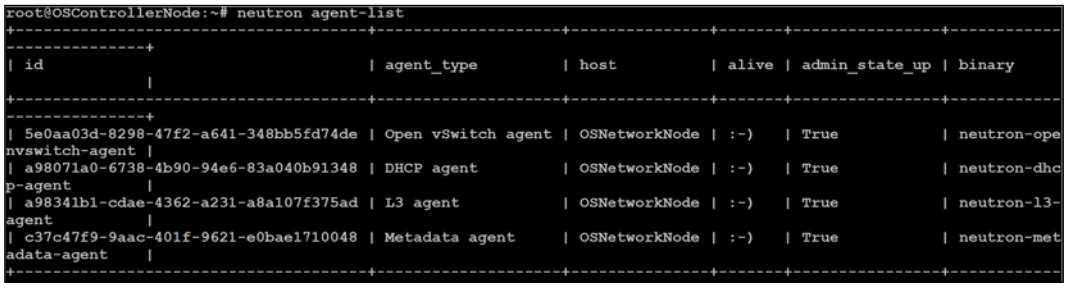
```
sudo service neutron-plugin-openvswitch-agent restart
sudo service neutron-l3-agent restart
sudo service neutron-dhcp-agent restart
sudo service neutron-metadata-agent restart
```

Validating the installation

On the controller node, after exploring the environment variables for authentication, execute the following command:

```
neutron agent-list
```

You will see the four agents we configured on the network node, as in the following screenshot:



Installing on the compute node

We need to install and configure the Neutron plugins, and these need to be done on every compute node so that the system can communicate with the network node. We now go back to our familiar checklist:

Name	Info
Access to the Internet	Yes
Proxy needed	No
Proxy IP and port	Not applicable
Node name	`OSComputeNode`
Node IP address	172.22.6.97 – Mgmt network 10.0.0.5– Tunnel interface
Node OS	Ubuntu 14.04.1 LTS
Neutron DB password	`n3utron`
Neutron Keystone password	`n3utronpwd`
Neutron port	9696
Nova Keystone password	`n0vakeypwd` (from the previous chapter)
Rabbit MQ password	`rabb1tmqpass`

Setting up the prerequisites

We start by editing the `/etc/sysctl.conf` file to ensure the following parameters are set:

```
net.ipv4.conf.all.rp_filter=0
net.ipv4.conf.default.rp_filter=0
```

This disables the Reverse Path (rp) filter so that the kernel doesn't perform the source validation. The kernel will start dropping the packets if we don't set this, as sometimes the packets may actually be destined to another IP address. In order to ensure that the changes take effect, reload the system control:

```
sysctl -p
```

This command reloads the `sysctl.conf` file that we modified.

Installing packages

Install the ML2 plugin and OVS agent by running the following command:

```
sudo apt-get install neutron-plugin-ml2 neutron-plugin-openvswitch-
agent
```

Ensure that the components are installed successfully.

After this, start OVS by running the following command:

```
sudo service openvswitch-switch restart
```

Initial configuration

Now, let us look at some of the initial configuration tasks on the compute node.

Neutron configuration

In the `/etc/neutron/neutron.conf` file, make the following changes:

- Under the `[database]` section:
    - Remove any connection string that is present, as the database access is not needed directly by the network node

- Under the `[default]` section:

```
rabbit_host = OSControllerNode
rabbit_password = rabb1tmqpass
auth_strategy = keystone
core_plugin = ml2
service_plugins = router
allow_overlapping_ips = True
verbose = True
```

- Under the `[keystone_authtoken]` section:

```
auth_uri = http://OSControllerNode:5000/v2.0
identity_uri = http://OSControllerNode:35357
admin_tenant_name = service
admin_user = neutron
admin_password = n3utronpwd
```

ML2 plugin

In the `/etc/neutron/plugins/ml2/ml2_conf.ini` file, make the following changes:

- In the `[ml2]` sections, enable GRE and flat networking:

```
type_drivers = flat,gre
tenant_network_types = gre
mechanism_drivers = openvswitch
```

- In the `[ml2_type_gre]` section, enable the tunnel ID ranges; these don't need to be in the physical network as they will only be between the compute and network nodes:

```
tunnel_id_ranges = 1:1000
```

- In the `[securitygroup]` section, change the following:

```
enable_security_group = True
enable_ipset = True
firewall_driver=
neutron.agent.linux.iptables_firewall.
OVSHybridIptablesFirewallDriver
```

- Under the `[ovs]` section for OVS (set the interface address for GRE), set the following parameters:

```
local_ip = 10.0.0.5
enable_tunneling = True
```

- Under the `[agent]` section, enable the GRE tunnels:

```
tunnel_types = gre
```

Nova configuration

Edit the `/etc/nova/nova.conf` file and make the following changes:

- In the `[default]` section, set the different drivers:

```
network_api_class = nova.network.neutronv2.api.API
security_group_api = neutron
linuxnet_interface_driver= nova.network.linux_net.
LinuxOVSInterfaceDriver
firewall_driver =
nova.virt.firewall.NoopFirewallDriver
```

- In the `[neutron]` section, make these changes:

```
url = http://OSControllerNode:9696
auth_strategy = keystone
admin_auth_url =
http://OSControllerNode:35357/v2.0
admin_tenant_name = service
admin_username = neutron
admin_password = n3utronpwd
```

Finalizing the installation

Finally, restart all the components that we installed on the compute node:

```
sudo service nova-compute restart
sudo service neutron-plugin-openvswitch-agent restart
```

This concludes the installation of Neutron in our OpenStack environment.

Validating the installation

Validate the installation in the same way as we did for the Network node.
On the controller, after exporting the credentials, execute the following command:

```
neutron agent-list
```

We already had four entries from our installation on the network node, but now you can see an entry has been created for the compute node as well:

```
root@OSControllerNode:~# neutron agent-list
+--------------------------------------+--------------------+--------------+-------+----------------+------------
---------------+
| id                                   | agent_type         | host         | alive | admin_state_up | binary
               |
+--------------------------------------+--------------------+--------------+-------+----------------+------------
---------------+
| 5e0aa03d-8298-47f2-a641-348bb5fd74de | Open vSwitch agent | OSNetworkNode | :-)   | True           | neutron-ope
nvswitch-agent |
| 685f90cb-1af7-4d8e-a1c6-b59a8134973f | Open vSwitch agent | OSComputeNode | :-)   | True           | neutron-ope
```

Troubleshooting Neutron

The troubleshooting of Neutron has a very similar procedure when it comes to OpenStack.

The logs are located in the `/var/logs/neutron` directory. Depending on the services that are installed on the node, the following files are found:

- `dhcp-agent.log` for the `neutron-dhcp-agent` service
- `l3-agent.log` for the `neutron-l3-agent`
- `metadata-agent.log` for the `neutron-metadata-agent`
- `openvswitch-agent.log` for the `neutron-openvswitch-agent` service
- `server.log` for the `neutron-server`

These log files contain information related to the services and can help in determining the case of the problem.

However, before we rummage through the logs, we perform the following steps:

1. We will ping the network interfaces connecting the compute node and the network node to ensure that the underlying network connectivity is ruled out. If the problem is between the underlying network, we may have to follow the troubleshooting steps for the underpinning system.
2. As a next step, we will check the status of the network interfaces and bridges by the `ip a | grep` state, and ensure all the bridges are up. Once this is confirmed and the configurations are checked, the system will work.
3. In order to troubleshoot further, we will need to follow how Neutron connects the different components. The virtual machine running on QEMU or KVM normally connects to the tap interface, which is connected to the internal bridge and then connected to the external bridge interface.

4. We can find which tap a virtual machine is using by looking at the XML file of the instance located at `/etc/libvirt/qemu/`. The file will be of the format `instance-<instance id>.xml`.
5. We can set up mirroring to a dummy interface in order to troubleshoot this further. We can use tools similar to tcpdump, iptables, and the openvswitch control tools to troubleshoot this pretty much as we did for the underpinning network system.

Summary

In this chapter, you were introduced to Neutron, which is a networking service, and how is it different from the Nova network service. You then looked at Neutron's architecture and learnt about its components, such as Neutron server, L2 agent, and L3 agent. You also briefly learnt about networking concepts before you proceeded to install and configure Neutron.

Neutron also hopes to integrate with a pluggable **IP Address Management System** (**IPAM**) in order for enterprises to be able to easily assign IP addresses. In the coming years, Neutron will see new plugins and drivers and the overall system being made extremely modular. The ability to work with any kind of networking will soon be a reality rather than just a dream.

In the next chapter, we will look at Horizon, a portal to OpenStack.

6
Building Your Portal in the Cloud

Horizon is the web component of OpenStack. It is an extensible component, which means new views can be added without losing consistency in its look and feel, as it provides the necessary core classes and templates for developers to use. In this chapter, we will look at the architecture of Horizon and install it in our environment. After completing this, we will have our cloud ready and take it for a spin in the next chapter.

In this chapter, we will cover the following topics:

- Working with Horizon
- Installing and configuring Horizon
- Troubleshooting Horizon

Working with Horizon

Horizon is a component of OpenStack that is slightly different from the others. This is a Django web application, and its sole purpose is to provide a user interface for other components of OpenStack services such as Nova and Swift.

One more thing about Horizon is that it does not actually use a database like other services, as all the information that it needs is pulled out from the other components using their respective APIs. In order to use Horizon, the use of Keystone is a must, which means we cannot use the individual authentication modules that the services themselves provide.

If you have followed the book to this point, all our services have been already authenticated using Keystone, so we can proceed. However, if we have anything that is locally authenticated, then that needs to be changed.

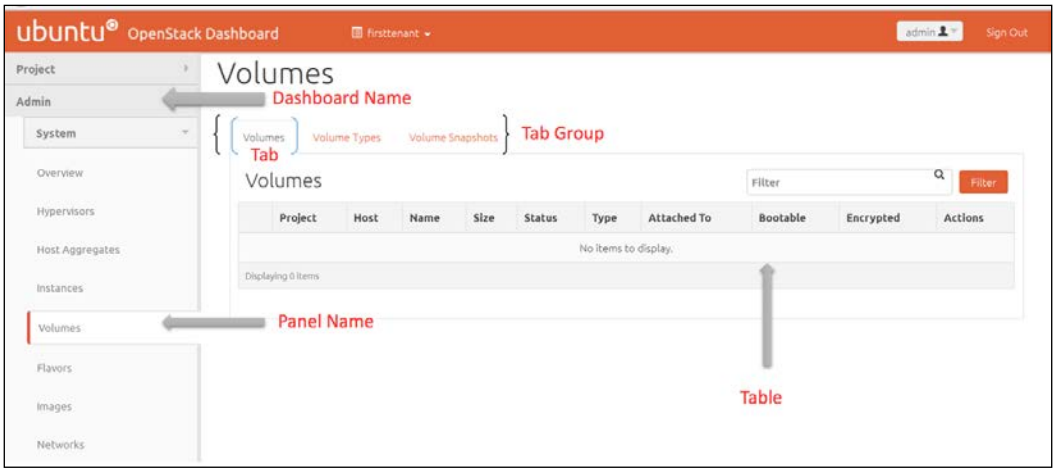
Some basic terminologies

Let's discuss some terminologies used in the context of Horizon; this will help us in the remaining part of the chapter and also when we are either creating or modifying Horizon dashboards:

- **Panel**: This is the main UI component. Every panel has its own directory and a standardized directory structure. The panels are configured in the `<Dashboard Name>/<Panel Name>/panel.py` file.
- **Panel groups**: These are used to organize similar panels together and provide a top-level drop-down. These are configured using the `dashboard.py` file.
- **Dashboard**: This is a top-level UI component. A dashboard contains panel groups and panels. They are configured using the `dashboard.py` file. Some of the dashboards installed by default are `admin`, `identity`, `project`, and so on.
- **Tab groups**: A tab group contains one or more tabs. A tab group can be configured per panel using the `tabs.py` file.
- **Tabs**: Tabs are units within a tab group. They represent a view of the data.
- **Workflows**: A workflow is a series of steps that allow the collection of user inputs. Workflows are created under `<Dashboard Name>/<Panel Name>/workflows/workflow.py`.
- **Workflow steps**: A workflow consists of one or more steps. A step is a wrapper around an action that understands its context within a workflow. Using workflows and steps, we can build multiple input forms that guide a user through a complex configuration process or a multistep input in order to accomplish something.
- **Actions**: An action allows us to spawn a workflow step. Actions are typically called from within a data table. Two of the most common actions are `DeleteAction` and `LinkAction`.
- **Tables**: Horizon includes a componentized API to dynamically create tables in the UI. Data tables are used to display information to the user. Tables are configured per panel in the `tables.py` file.

- **URLs**: URLs are needed to track context. URLs are configured in the `urls.py` file.
- **Views**: A view displays a data table and encompasses the main panel frame. Views are configured per panel in the `views.py` file.

Examining some of these components on the UI, as shown in the following screenshot, will give us a better understanding:



System requirements to install Horizon

Some system requirements need to be kept in mind while installing the Horizon dashboard:

- Since Horizon is going to use the memory cache, the amount of RAM in the server will have to be planned based on the number of concurrent users that are planned.
- The system needs Python 2.6 or 2.7 with Django installed. Please check the versions using the following commands:

```
python -c "import sys; print(sys.version)"
python -c "import django; print(django.get_version())"
```

  You should be able to see the versions of Python and Django as shown here:

```
root@OSControllerNode:~# python -c "import sys; print(sys.version)"
2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2]
root@OSControllerNode:~# python -c "import django; print(django.get_version())"
1.6.1
```

As you can see in our case, Python version 2.7.6 and Django version 1.6.1 are being used.

- A routable IP is required because this system needs to be accessible to all the users.
- If this machine is set behind a load balancer, some form of persistence needs to be enabled. The persistence methods in the order of preference are cookie persistence and source hash persistence. If the load balancer doesn't support either, then we should enable a source address persistence on the load balancer as the last resort.
- This machine should be able to reach all the endpoints that have been configured in Keystone.
- You can use the same features as any web application including proxy pass, reverse proxying, and so on.

Installing Horizon

The installation is very simple; we will install the dashboard and memory cache. Let's fill in our familiar check list:

Name	**Info**
Access to Internet	Yes
Proxy needed	No
Proxy IP and port	Not applicable
Node name	`OSControllerNode`
Node IP address	172.22.6.95
Node OS	Ubuntu 14.04.1 LTS
Keystone installed on	172.22.6.95
Admin username	`Admin`
Admin password	`h33l0world`

Now run the following command:

```
sudo apt-get install openstack-dashboard apache2 libapache2-mod-wsgi
memcached python-memcache
```

Ensure that the command is successful and the packages are installed successfully. This will install Horizon on our controller node.

The initial configuration of Horizon

The initial configuration of Horizon is simple as well, we need to modify a single `/etc/openstack-dashboard/local_settings.py` file and configure three aspects:

- The location of the Keystone server
- The hosts that are allowed to access Horizon (the source IP addresses)
- The location of Memcache Daemon

Edit the previously mentioned file and set the values as:

- `OPENSTACK_HOST = "172.22.6.95"`: This will set `OSControllerNode` for the Keystone services
- `ALLOWED_HOSTS = ['*']`: We are allowing all the hosts to access the Horizon dashboard
- `CACHES`: This sets the location to 127.0.0.1 (or to localhost)

You should see something like the following screenshot:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
OPENSTACK_HOST = "172.22.6.95"
OPENSTACK_KEYSTONE_URL = "http://%s:5000/v2.0" % OPENSTACK_HOST
```

Finalizing the installation

In order to finalize the installation, we restart the apache and memcached services using these commands:

```
sudo service apache2 restart
sudo service memcached restart
```

This concludes the installation of the Horizon dashboard.

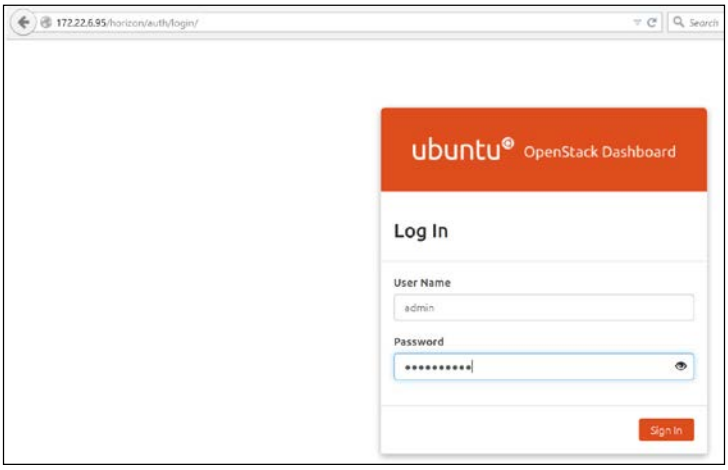
Validating the installation

Once the installation has been completed, open a browser (which is HTML5-compatible and has JavaScript and cookies turned on). You can use Firefox, Google Chrome, or any up-to-date browser.

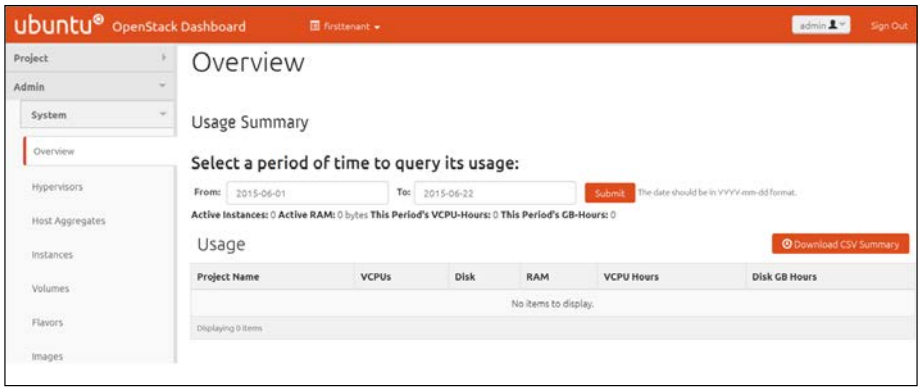
The dashboard will be available at (replace the IP with your controller node's IP address)

```
http://172.22.6.95/horizon/
```

You will need to authenticate with the admin credentials, and you should be able to log in to the dashboard:



After you've logged in, the page will look as follows:



The structure of the Horizon dashboard

While it is not necessary to know this, it is good to know about the structure. This helps us to make connections with the terminologies that we discussed previously, as an example. Let's see the structure of the dashboard.

You can view this yourself after you have installed Horizon, with the following instructions:

1. Install a program called tree:

   ```
   sudo apt-get install tree
   ```

2. Navigate to the directory where the dashboards are installed:

   ```
   cd /usr/share/openstack-
   dashboard/openstack_dashboard/dashboards
   ```

3. In order to see the structure of any dashboard, just type `tree <dashboard name>`, so take a look at the settings of the dashboard:

   ```
   tree settings
   ```

4. You will see the directory structure of the dashboard, as shown in the following screenshot:

```
root@OSControllerNode:/usr/share/openstack-dashboard/openstack_dashboard/dashboards# tree settings
settings
|-- __init__.py
|-- __init__.pyc
|-- dashboard.py
|-- dashboard.pyc
|-- models.py
|-- models.pyc
|-- password
|   |-- __init__.py
|   |-- __init__.pyc
|   |-- forms.py
|   |-- forms.pyc
|   |-- panel.py
|   |-- panel.pyc
|   |-- templates
|   |   `-- password
|   |       |-- _change.html
|   |       `-- change.html
|   |-- tests.py
|   |-- tests.pyc
|   |-- urls.py
|   |-- urls.pyc
|   |-- views.py
|   `-- views.pyc
`-- user
    |-- __init__.py
    |-- __init__.pyc
    |-- forms.py
    |-- forms.pyc
    |-- panel.py
    |-- panel.pyc
    |-- templates
```

In the preceding screenshot, you can see that we have dashboard settings and multiple panels such as user and password, and that each of them has URLs, views, and so on.

Troubleshooting Horizon

Horizon being a Django application follows the standard Django logging model. The configuration of logging is set in the `/etc/openstack_dashboard/local_settings.py` file; if we have to modify the logging levels, we need to look at the `logger_root` and `handler_file` section of `file/etc/keystone/logging.conf`.

Let's take a look at the different logging levels for both standard OpenStack services and Horizon in escalating levels of severity:

Logging levels	OpenStack standard	Horizon
`DEBUG`	Yes	Yes
`INFO`	Yes	Yes
`AUDIT`	Yes	No
`WARNING`	Yes	Yes
`ERROR`	Yes	Yes
`CRITICAL`	Yes	Yes
`TRACE`	Yes	No

When we set the logging levels to `DEBUG`, all the different messages are logged; when we set the logging level to anything else, the logs for that level and the ones on a higher level are logged. This is true for both Horizon and OpenStack services.

The `TRACE` logs are shown wherever there is a stack trace to be shown. In Horizon, the `TRACE` logs are shown almost at all times, as the underlying Python call will generate the stack trace.

You can configure Horizon to send different logs to different files and apply filters in exactly the same way you would do for any other Django application, but this is beyond the scope of the book and so is not covered here.

Coming to the errors that you can face in Horizon, one is the installation error; this error is normally caused by a wrong version of Python and related libraries. The second most common error is the HTTP500 Internal server error, which normally ensues from one of the underlying systems—for instance, Apache not functioning properly or an error in configuration.

Understanding the Horizon log

The Horizon log normally has two parts: the actual log and the Python stack trace. Let's take look at the following log, as an example:

```
2015-08-22 01:08:51 17409 CRITICAL cinder [-] Bad or unexpected
response from the storage volume backend API: volume group

 cinder-volumes doesn't exist

2015-08-22 01:08:51 17409 TRACE cinder Traceback (most recent call
last):

2015-08-22 01:08:51 17409 TRACE cinder File "/usr/bin/cinder-volume",
line 48, in

 in check_for_setup_error
```

Some more of the stack trace follows:

```
2015-08-22 01:08:51 17409 TRACE cinder raise
exception.VolumeBackendAPIException(data=exception_message)

2015-08-22 01:08:51 17409 TRACE cinder VolumeBackendAPIException: Bad
or unexpected response from the storage volume

 backend API: volume group cinder-volumes doesn't exist

2015-08-22 01:08:51 17409 TRACE cinder
```

As you can see, the first line shows that the problem is with the Cinder service and the other lines are the stack trace of the service. So, in order to troubleshoot this further, we will have to go to the Cinder logs and continue the troubleshooting from here.

Horizon can also throw errors if you have installed custom dashboards that interfere with the normal functioning of Horizon's other dashboards. Hence, if you have created a new dashboard, panel, or anything else, you may want to run the tests on another instance of Horizon using the `run_tests.sh` script. Remember to use a different port rather than the standard port on which the Horizon production is running.

Summary

So far, we have learned about the various components of OpenStack such as Keystone, Nova, Swift, Cinder, Glance, and Neutron. In this chapter, we have looked at Horizon, which is fundamentally the web interface to all these components.

In the next chapter, we will see how all the components we have installed and configured so far will work together to provide a cloud-based Infrastructure as a Service. This is where we will see Horizon put to use, to configure services, request virtual machines, storage, and so on.

7
Your OpenStack Cloud in Action

Finally, we are at the chapter towards which we were inching gradually in the previous chapters. In this chapter, we will put together all that we have set up and see a real life scenario of us being a cloud provider to, say, our internal development team. We will stitch everything together and see all the services we have set up in action, so without any further ado let's dive in.

To be specific, we will cover the following topics in this chapter:

- Gathering service requirements
- Managing your network
- Requesting services
- Creating VM templates

Gathering service requirements

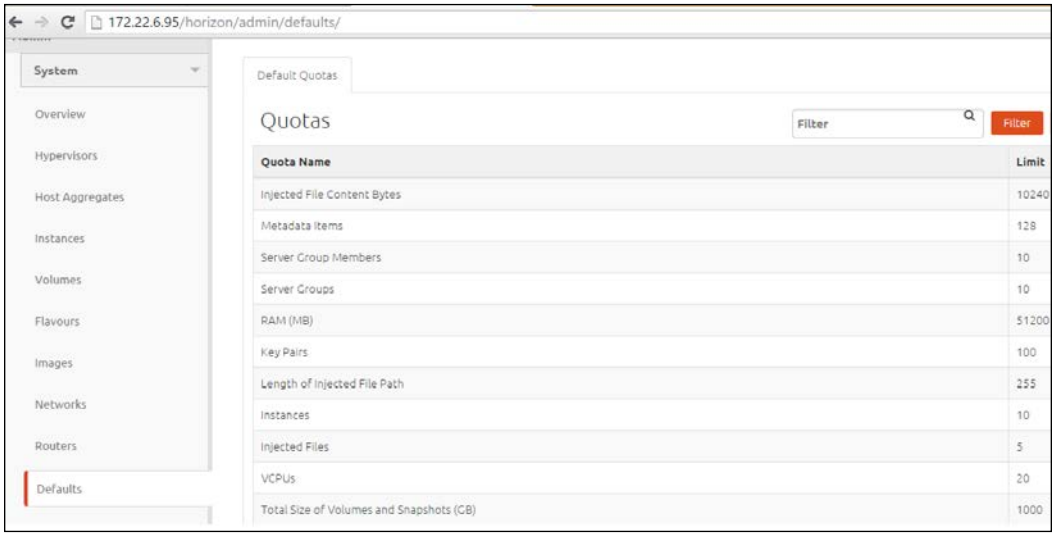
We hope that you are ready to start pointing, clicking and typing commands, but before we do just that, let's create a scenario for which we will be providing services.

So, we are a part of the IT wing of a company whose core business is software development. The company wants to make its processes agile and speed up the time to market the products it has been developing, and so we have been tasked with providing a private cloud environment to the various development and testing teams.

We have successfully done that by setting up our OpenStack environment and we have our first customer at our door, who is a project manager for a new product that has been conceptualized. They need an environment where the developers can freely spin instances up and down.

So, we have to ask the following questions:

- What is the name of the project?
- Project members – We will create individual accounts and will send the project to them as this is not yet Active Directory Integrated
- Operating systems required – Please note that this can be predefined by us as a part of corporate standards
- Networking configuration needed – We may choose to give the project members access to create their own networks, but in our case, we will set up everything for them for the first time
- Quotas (The maximum number of instances, RAM, CPU, and so on) – This is required to set a quota, by default, and a tenant/project will be allowed a certain amount of resource that you can check out on the Horizon dashboard as shown in the following:



If you need to change the defaults for every project, click on **Update Defaults** and change to the values you need.

Of course, the actual capacity also needs to be available in order to perform these actions. The ones shown in the preceding screenshot are merely limits that the system will enforce and not allow the users to request over these limits.

So our fictitious project manager gives us the following information:

- **Project/Tenant Name**: `TestingCloud`
- **Users**: John Doe (`john.doe@test.com`), Jane Doe (`jane.doe@test.com`)
- **Networking requirement**: Single Layer 2 network with external access, DHCP IP addressing
- **Quotas**: Defaults are fine

So, armed with the preceding information, we go on to take our cloud out for a spin.

Tenant and user management

The first task will be to create the new tenant/project in Keystone and add the users, Jane and John Doe, as users and members. We will also be adding our admin account as an admin on the new tenant, so we can support them in the future.

We can perform these actions in two ways, the command line and the GUI; you can choose either of the methods depending on your preference (please remember that only one of them needs to be followed).

GUI

Log in to Horizon, by going to `http://<controllernodeIP>/horizon`, which in our case is `http://172.22.6.95/horizon`. Authenticate using the admin credentials that we have been using, `admin/h33l0world`

Creating the project

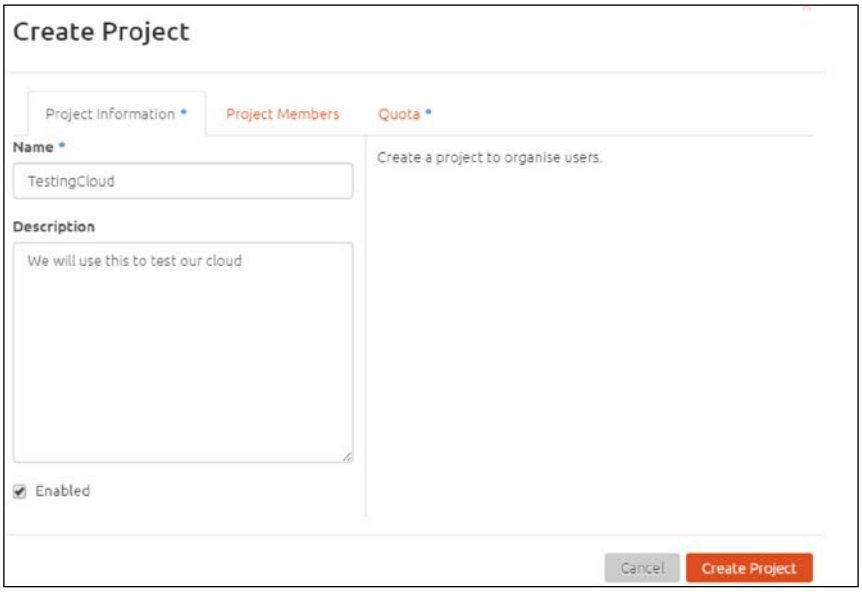
To create the project, follow these steps:

1. Click on **Identity** | **Projects** (On the left pane)

   You would already see the existing tenants, admin and service tenant that we created in the beginning.

2. Now click on **Create Project**, as seen in the following screenshot:

Please see the chapter on Keystone; projects and tenants are interchangeably used.



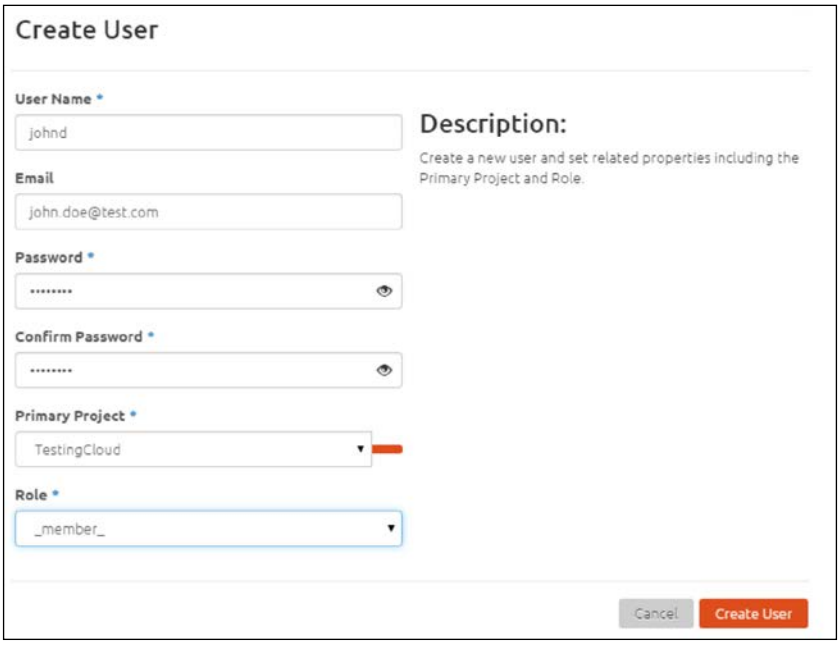
3. We can modify the quota for this project by going to the **Quota** tab, but since the project manager was fine with the defaults, we don't update the values. Click on **Create Project**.

   The project is now created, as shown here:

Adding users

To add users, follow these steps:

1. Navigate to **Identity** | **Users**.
2. You should be able to see the currently created admin and the service users; let's click on **Create User**, and then we can create the accounts for Jane Doe and John Doe and give them the password, `Pa55word`. We will also associate them with a new tenant and give them `member` permissions.



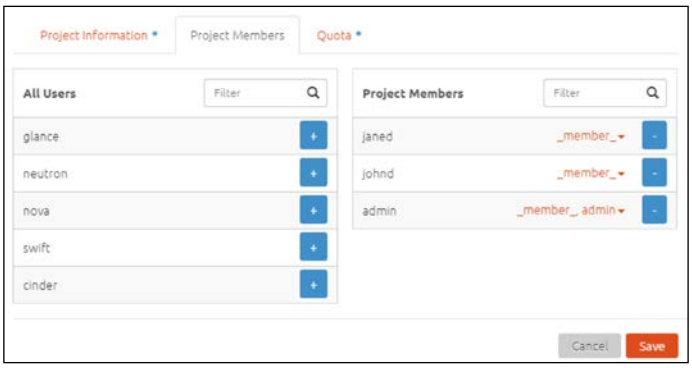
Repeat the preceding steps for the second user, and now both the users have been created in the system.

Associating users to the project

The users that we have created are already associated with the project as members, but we need to add our admin user as an admin to the project, and potentially other users that were created before could be added by this method, or we can change the role mapping of the user using this method. To do this, follow these steps:

1. Click on **Identity** | **Projects** and click on **Modify Users**.

2. Click on **+** next to the admin user, and it will be added to the right-hand side. You can then choose the roles. We choose both the admin and member roles for this user.



We can choose to give more roles to Jane and John from this screen, or remove them from the project.

So, the first part of setting up the project is done. You can test this by logging out of the portal and logging in as Jane or John, as shown in the following screenshot:

If a user is assigned to multiple projects, they can change the current project from the top of the screen as shown in the previous screenshot.

CLI

In order to do the same things we did using the GUI while using the CLI, all it takes is a bunch of commands:

If you have already created the users and project using GUI, then this is just for your information and need not be followed.

We log in to the controller node, and as with all the CLI commands, we export the authentication parameters in the command line using the `source` command or individually using the `export` command. We have stored it in the `os.txt` file in our home folder.

```
source ~alokas/os.txt
```

Creating the project

To create a project, we execute the `create tenant` command:

```
keystone tenant-create --name TestingCloud --description ""
```

Remember that the tenant and the project are interchangeable.

This should create the project.

Creating the users

We create the users in the project using the `user-create` command:

```
keystone user-create --name johnd --pass Pa55word  --email
john.doe@test.com

keystone user-create --name janed --pass Pa55word  --email jane.doe@test.
com
```

Associating users to the roles

We now associate the users to the tenant and map them to the `member` roles, using the following commands:

```
keystone user-role-add --user johnd --tenant TestingCloud --role member
keystone user-role-add --user janed --tenant TestingCloud --role member
keystone user-role-add --user admin --tenant TestingCloud --role admin
```

Network management

After Keystone changes, we need to set up the networks that the tenant will use. Since the tenant needs access to the Internet, we will need two networks; one is the tenant network, which is created per tenant, and the other is an external network, which is shared by all the tenants. Hence we need to create this only once. Since this is our first tenant, we will be creating both the networks. For the next tenant, we will not create the external network, but just reuse it, should Internet access be needed.

We will use the command line to create the external network and the provider network, as the functionality provided by the GUI is limited when it comes to network management.

Before we go into actually creating the networks, we will deal with some terminologies that we will use, such as the network type, physical network, and virtual network.

Network types

In order for us to be able to create and manage networks, let's understand some terminologies that we will use in the remainder of this section.

Physical network

This is the network connecting the OpenStack servers with each other and the physical world.

Virtual network

A Layer 2 network whose ports can be connected to the Nova compute and different agents. With the OVS, there are several implementations of the virtual network, and they are explained as follows.

Tenant network

Tenant network can be created by the tenants or the administrator for a tenant. The network can be used internally by the VMs of the tenant, and the physical world is not directly exposed to them. They need to go through one of the provider networks if they want to get out of the network.

Provider network

Provider network is administratively created and is used to bridge the tenant network and the physical network; tenants can be selectively given access to use the provider networks.

Implementations of virtual networks

The virtual networks can be implemented in one of the following ways depending on whether they are being used for the provider network or the tenant network. Each of the implementations has a different use case that it can be used for:

- **Local network**: A local network is a network that can only be realized on a single host. This is only used in proof of concept or development environments. Since we have a dedicated network node, we won't use it in our examples, but if we are doing a single node deployment or some tests, we can surely use it.
- **Flat network**: A flat network is a network that does not provide any segmentation options. A traditional L2 Ethernet network is a `flat` network. It is like a single switch without any VLANs. This should ideally be never used for tenants networks because the tenants will be able to see each other's traffic, unless of course that is the intent.
- **VLAN network**: A VLAN network is one that uses VLANs for segmentation. When you create a new network in Neutron, it will be assigned a VLAN ID from the range you have configured in your Neutron configuration. It actually sends the traffic out that is tagged to the physical switch, and hence needs the ports to the OpenStack servers that are trunked. Each of the tenants is separated, because each one is assigned to a VLAN. This is great for isolation where the different tenants may choose to have overlapping subnets and don't want to see each other's traffic, such as in a multi-tenanted situation.
- **GRE and VXLAN network**: These are the best networks to scale. They are very similar. They are both overlay networks that work by encapsulating network traffic. Similar to VLAN networks, each network you create receives a unique tunnel id, but unlike VLAN networks, we don't need to worry about the physical switch configuration, as the actual switch will never see these VLANs. In our configuration, we have GRE, but not VXLAN.

External network

The external network is a provider network, which will have access to the Internet connection. In our case, we have created the external network as an RFC1918 address. Please note that in production environments, RFC1918 (private IP addresses) are typically used when the range is routable inside the enterprise environment. If a public-facing cloud is required, this network is typically on public IP addresses.

Creating the network

We log into the controller node and export the admin credentials (since this is a provider network) sourcing the file (or export the variables manually):

```
source ~alokas/os.txt
```

```
neutron net-create ext-net --router:external True --provider:physical_
network external --provider:network_type flat
```

This will give the following output:

```
+---------------------------+--------------------------------------+
| Field                     | Value                                |
+---------------------------+--------------------------------------+
| admin_state_up            | True                                 |
| id                        | fea41df3-e54f-40d5-8027-11d9b00c18f4 |
| name                      | ext-net                              |
| provider:network_type     | flat                                 |
| provider:physical_network | external                             |
| provider:segmentation_id  |                                      |
| router:external           | True                                 |
| shared                    | False                                |
| status                    | ACTIVE                               |
| subnets                   |                                      |
| tenant_id                 | c95ff542161f494eb973fc9cc9977233     |
+---------------------------+--------------------------------------+
```

We create the network using the `net-create` command as shown in the preceding screenshot, since this is a provider network of the type `flat`. This will act as the Internet switch for all the different tenants. Let's take a look at the different parameters:

- `net-create <name>`: This is a Neutron subcommand to create a network with a particular name. In this case, we are unimaginatively calling it `ext-net`, short for external network.
- `--router:external True`: We are setting that an external router needs to be used for this network, so the Layer 3 process will kick in.

- `--provider:physical_network <name>`: This sets the physical network name. If you remember, we created an external bridge and we gave it a name in our ML2 plugin configuration We need this name. In our case, the name is `external`. If you need to locate this, please log in to the network node and open the `/etc/neutron/plugins/ml2/ml2_conf.ini` file and look for `bridge_mappings`.
- `--provider:network_type <network_type>`: We can choose from a range of types that we discussed previously, `flat`, `vlan`, `vxlan`, `gre`, and so on. Since we want this to be a flat switch where all the tenants can connect, we set this to `flat`.

Creating the subnet

The external network is created; we also need to create a subnet so that the IP addresses that are assigned to this network can be given out. In our case, our external network has an IP address `172.22.104.100/24`, so we will take some 20 IP addresses that are unassigned and create a subnet. Let's have the information handy to substitute in the command.

Name	Value
Network Address	172.22.104.0/24
Gateway	172.22.104.250
IP pool start address	172.22.104.150
IP pool end address	172.22.104.200

The command format to create a subnet is shown as follows:

```
neutron subnet-create <Network_Name> <NetworkID> --name <Subnet_name>
  --allocation-pool start=<IP_StartAddress>,end=<IP_End_Address>
  --disable-dhcp --gateway <GatewayAddress>
```

Substituting, we get the following command. (Please ensure not to leave any space between the comma in the start and end IP address.)

```
neutron subnet-create ext-net 172.22.104.0/24 --name ext-subnet \
  --allocation-pool start=172.22.104.150,end=172.22.104.200\
  --disable-dhcp --gateway 172.22.104.250
```

The subnet is then created and associated with `ext_net`. Please note that one or more networks can be created and associated with a network, but each subnet needs a network.

```
+-------------------+-----------------------------------------------------+
| Field             | Value                                               |
+-------------------+-----------------------------------------------------+
| allocation_pools  | {"start": "172.22.104.150", "end": "172.22.104.200"} |
| cidr              | 172.22.104.0/24                                     |
| dns_nameservers   |                                                     |
| enable_dhcp       | False                                               |
| gateway_ip        | 172.22.104.250                                      |
| host_routes       |                                                     |
| id                | c555f8e9-0221-4c6d-8e32-346eb323a16c                |
| ip_version        | 4                                                   |
| ipv6_address_mode |                                                     |
| ipv6_ra_mode      |                                                     |
| name              | ext-subnet                                          |
| network_id        | fea41df3-e54f-40d5-8027-11d9b00c18f4                |
| tenant_id         | c95ff542161f494eb973fc9cc9977233                    |
+-------------------+-----------------------------------------------------+
```

This step needs to be executed only once.

Tenant network

Our tenant needs a network for itself. For this, we will create a network, subnet, and router so that they can use the external network.

Create the tenant network

The following information should come in handy while creating the network:

Name	Value
Network name	cloud-network
Subnet name	cloud-subnet
Subnet	192.168.5.0/24
Subnet start	192.168.5.2
Subnet end	192.168.5.254
Subnet gateway	192.168.5.1
Router name	cloud-router

We source all the variables and admin credentials; we override the tenant name so that the new network is created in the new tenant:

```
source ~alokas/os.txt
export OS_TENANT_NAME=TestingCloud
neutron net-create cloud-network
```

```
root@OSControllerNode:~# neutron net-create cloud-network
Created a new network:
+---------------------------+--------------------------------------+
| Field                     | Value                                |
+---------------------------+--------------------------------------+
| admin_state_up            | True                                 |
| id                        | c83453d6-599f-4b01-aedf-7b8561eff67c |
| name                      | cloud-network                        |
| provider:network_type     | gre                                  |
| provider:physical_network |                                      |
| provider:segmentation_id  | 1                                    |
| router:external           | False                                |
| shared                    | False                                |
| status                    | ACTIVE                               |
| subnets                   |                                      |
| tenant_id                 | 0ca212a9eba346a3bca9b7216f3fd02b     |
+---------------------------+--------------------------------------+
```

Creating a subnet

We now create a subnet with the same format of the command that we used in the previous section with the exception that we don't give a start and an end IP address, thereby allowing the subnet to go as far and wide as possible:

```
neutron subnet-create --name cloud-subnet --gateway 192.168.5.1
cloud-network 192.168.5.0/24
```

```
root@OSControllerNode:~# neutron subnet-create --name cloud-subnet --gateway 192.168.5.1 cloud-network 192.168.5.0/24
Created a new subnet:
+-------------------+--------------------------------------------------+
| Field             | Value                                            |
+-------------------+--------------------------------------------------+
| allocation_pools  | {"start": "192.168.5.2", "end": "192.168.5.254"} |
| cidr              | 192.168.5.0/24                                   |
| dns_nameservers   |                                                  |
| enable_dhcp       | True                                             |
| gateway_ip        | 192.168.5.1                                      |
| host_routes       |                                                  |
| id                | 409a3361-8d8a-40a1-ac7b-2b89f66eec52             |
| ip_version        | 4                                                |
| ipv6_address_mode |                                                  |
| ipv6_ra_mode      |                                                  |
| name              | cloud-subnet                                     |
| network_id        | c83453d6-599f-4b01-aedf-7b8561eff67c             |
| tenant_id         | 0ca212a9eba346a3bca9b7216f3fd02b                 |
+-------------------+--------------------------------------------------+
```

As we can see, the subnet has used all the available IP addresses in the pool as we would have wanted it to.

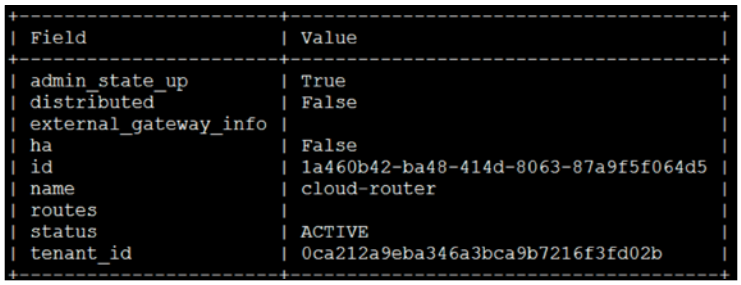
Creating a router

The router is an essential piece that will connect the tenant subnet to the external subnet. This is done in three easy steps:

1. Create the router
2. Add the tenant subnet
3. Add the gateway as external subnet

```
neutron router-create cloud-router
neutron router-interface-add cloud-router cloud-subnet
neutron router-gateway-set cloud-router ext-net
```

This will give the following output:



Field	Value
admin_state_up	True
distributed	False
external_gateway_info	
ha	False
id	1a460b42-ba48-414d-8063-87a9f5f064d5
name	cloud-router
routes	
status	ACTIVE
tenant_id	0ca212a9eba346a3bca9b7216f3fd02b

You will get a confirmation that the interface has been added and its gateway has been set.

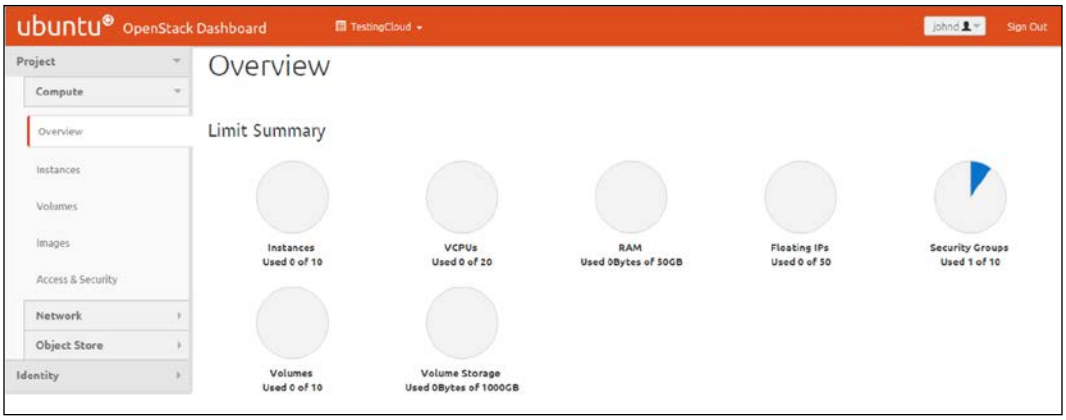
Since the images they need are already in the Glance repository, we are now ready to hand over the cloud to the development team for them to use. If more images need to be added in the repository for different operating systems, please follow the same steps as mentioned in *Chapter 3, Storing and Retrieving Data and Images using Glance, Cinder, and Swift.*

Now it is time to look at the cloud from a consumer's point of view. In our example, we have Jane and John Doe, who will be the users. John is the creative artist and hence prefers the GUI, and Jane being an extremely tech savvy user, prefers the CLI or a programmatic access route. Once we as IT administrators send them emails about the cloud being ready for their consumption, they decide to quickly test whether this works for their needs.

Requesting services

Let's first take a look at how John tries to request cloud services. He logs into the Horizon portal using his credentials that we created.

When he logs in, he will see a screen and the **TestingCloud** tenant, as this is the only one he is associated with. Also, he will only see the project section as he is only a member of the project. He is also presented with the dashboard showing the limits of his project and what has already been consumed:



Now, in order to request for services, he needs to go into the corresponding panel and request for them.

Before this is done, let's understand some base constructs of access and security that the user will be using in order to access his services. If you are familiar with AWS, the same constructs apply here.

Access and security

We can see the different components from the access and security panel in the project dashboard.

Security groups

Security groups are at a high level, a basic layer 3 firewall that is wrapped around every compute instance that is launched inside a group. Every instance needs to be launched inside a security group. This contains inbound and outbound rules.

It is also worthwhile to note that servers that are in the same security group also need to have rules permitting them to talk to each other, if they decide to do so. A common good practice is to create a security group per tier/application, whichever way the security needs of the enterprise are met. By default, a default security group is created, which has an – Allow rule. It is recommended that you do not use this security group.

Key pairs

Key pairs are in essence a public/private RSA key (asymmetric encryption) that helps you to log in to the instances that were created without a password. You can choose to create a key outside the OpenStack environment and then import it so that you can continue using your key pairs both inside and outside the OpenStack cloud.

In order to generate a key pair on any Linux machine with `ssh-keygen`, type the following command:

```
ssh-keygen -t rsa -f <filename>.key
```

And after this, just import the public part of your key pair. You should keep the private key somewhere safe.

If you are using Windows, download the Puttygen executable from the Internet say, `www.putty.org`, and generate the key pair as shown in the following by choosing SSH2-RSA and clicking on the **Generate** button.

OpenStack also lets you create a key pair by clicking on the **Create Keypair** button. It will download the private key to your desktop. Please ensure you keep this safe, because it is not backed up anywhere else and without it, you would lose access to any instances that are protected by the key.

Requesting your first VM

Now that we have understood the access and security mechanism, John is ready to request his first VM using the portal. As followers of best practices, we will create a security group for our VM and a key pair to go with it.

Creating a security group

Creating a security group is easy; what actually requires a little thought is the rules that need to go in it. So before we create our security group, let's for a moment think about the rules we would put in.

There are two types of rules—`ingress` and `egress` (in and out); this is from the perspective of the VM. Any traffic that we need to permit to the VM will be `ingress` and from the VM will be `egress`.

So, we need to think about this for a moment. Say we are creating a Linux web server, we will need just one port to be permitted in, Port 80? Maybe, but remember, this security group filters all the traffic, so if we only permit port 80 in, we won't be able to SSH into the machine. So the rules should include every type of traffic you want to be allowed to the machine: traffic to the applications, management, monitoring, everything.

So, in our case, we will allow port `22`, `80`, and `443` for management and application, and then we will permit port `161` to monitor using SNMP. We will allow all ports from the machine outwards. (This might not be allowed in a secure environment, but for now this is alright). As discussed in the following table:

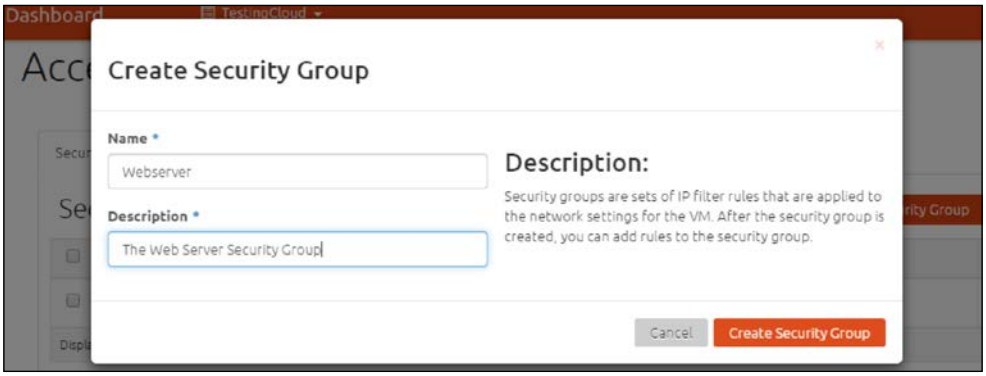
Direction	Protocol	Port	Remote
In	TCP	22	0.0.0.0/0
In	TCP	80	0.0.0.0/0
In	TCP	443	0.0.0.0/0
In	UDP	161	0.0.0.0/0
Out	ANY		0.0.0.0/0

In a production environment, the SSH access also needs to be locked down and the access needs to be allowed only from your administrative subnet.

As you can see, we can create the rules, and we have set any source/destination as the IP range. We could even control this depending on the application. It is also worthwhile knowing that the remote can be an IP CIDR or even a security group. So if we want to permit the servers in the web security group to talk to servers in the app security group, we can create these rules in that fashion.

So, now that we know our security group rules, let's create them. Click on **Access & Security**, choose the **Security Group** Tab, and click on **Create Security Group**.

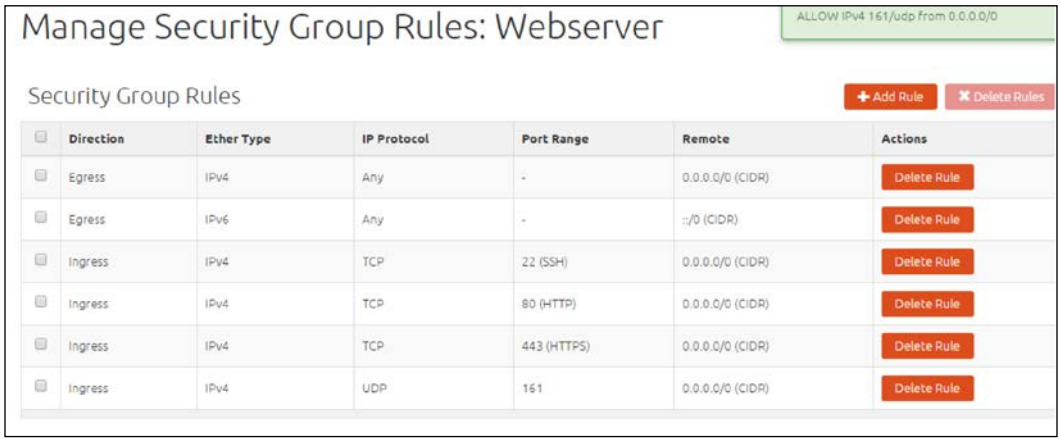
It will ask for a name and description, fill that in, and click on **Create Security Group**, you will see that the security group has been created.



By default, this will block all incoming traffic and permit outgoing. We will need to set the rules as we have planned, so let's go ahead and click on **Manage Rules**:

We will then add rules that we have planned for by clicking on the **Add Rule** button. There are some predefined rules for ports, we can use these or create custom ones.

After adding the rules, our configuration should look like the following:



The servers in the same security group, while communicating with each other, will also be subjected to the security group rules. So, if we want to permit these, we will have to explicitly permit them. As the best practice, we should use the security group as the remote source/destination.

Once our security group is created, we proceed to the next step.

Creating a key pair

We can choose to create a key pair in the OpenStack system or outside it, and then just import the public key. The import option is used in case we want to use an existing key that we used somewhere else, or we can use an external system for enterprise key management. For our purposes, we will just create a key pair.

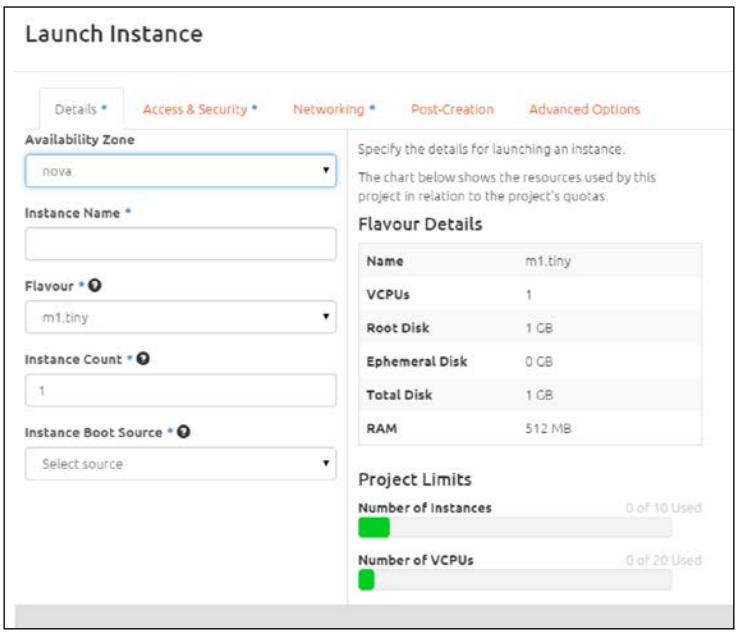
Click on **Access & Security** and then click on the **Key Pairs** tab and click on **Create Key Pair**:

Give the key pair a name, and it will be created. Please remember to download the private key and store it in a safe place.

Launching an instance

Now that we are ready, head over to the instances panel, which displays the currently running instances for the project. Click on **Launch Instance**.

We will have to fill in some information in all the tabs:



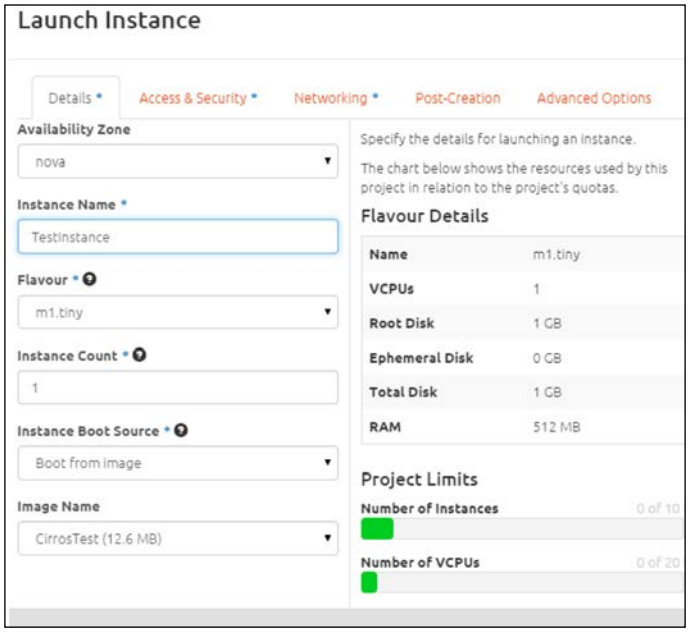
Since we only have one AZ created (by default), we will just use that. We will have to fill in the following details to launch an instance:

- **Instance Name**: The instance name as it will be shown in the console.
- **Flavor**: This is the size of instance that will be launched. There are some predefined values that we can see using the `nova flavor-list` command:

```
+----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
| ID | Name      | Memory_MB | Disk | Ephemeral | Swap | VCPUs | RXTX_Factor | Is_Public |
+----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
| 1  | m1.tiny   | 512       | 1    | 0         |      | 1     | 1.0         | True      |
| 2  | m1.small  | 2048      | 20   | 0         |      | 1     | 1.0         | True      |
| 3  | m1.medium | 4096      | 40   | 0         |      | 2     | 1.0         | True      |
| 4  | m1.large  | 8192      | 80   | 0         |      | 4     | 1.0         | True      |
| 5  | m1.xlarge | 16384     | 160  | 0         |      | 8     | 1.0         | True      |
+----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
```

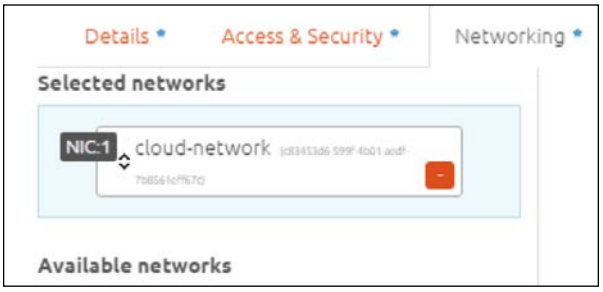
We can also create a new flavor using the `nova flavor-create` command.

- **Instance Boot Source**: The instance boot source specifies where to boot the instance from; since we are going to use the cirros test image that we created, we will choose that in this case.



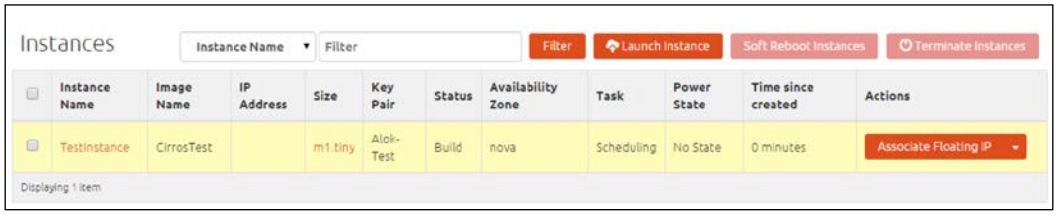
- **Key Pair**: Choose a key pair to use. (We choose the one we created.)
- **Security Groups**: Choose one or more security groups to use. (We choose **Webserver**.)

- **Networking**: We choose the tenant network to use. We can choose to add more than one NIC card, if there are multiple networks. However, in our case, as an administrator, we only created a single NIC:



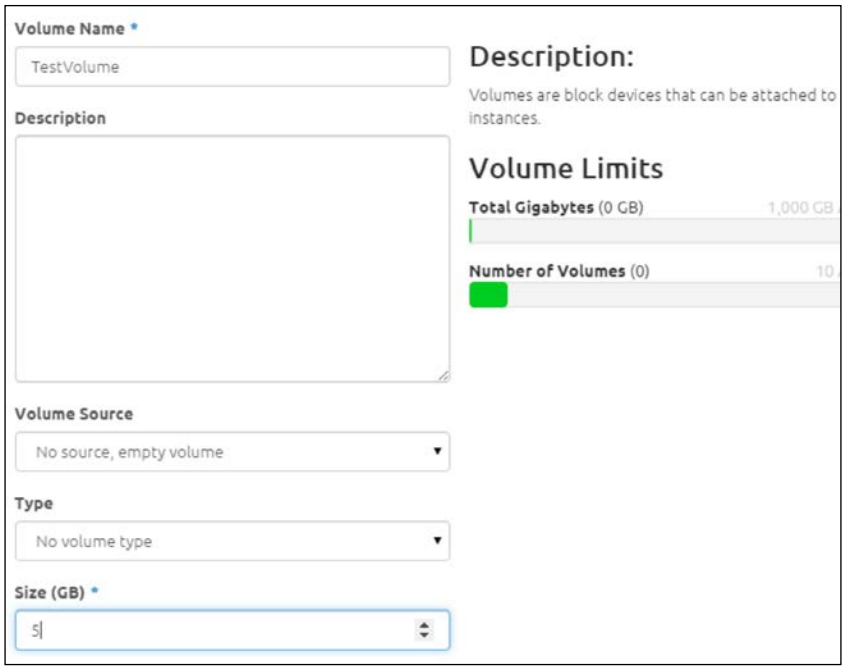
These are mandatory fields, but we can also inject a post creation customization script and even partitioning in the next tabs, but for our purposes, let's click on the **Launch** button.

Once the request is submitted, we will see the instance in the list and see the status as scheduled, shown in the following screenshot:



If we go back to where we installed Nova, the scheduler service is responsible for choosing the appropriate compute node on which the instance should reside. The instance powers on after a while and you can SSH into the instance using the private key that you created.

John can similarly request other services such as a block volume, by going to the volumes panel and clicking on **Create Volume**. So he requests a 5 GB volume so that he can store the images that he will work on, and this will be available even after the instance is terminated.



On clicking on **Create Volume**, a volume is created. It can then be attached to the instance that we just created. However, after the volume is mounted on the OS, the operating system tools need to be used (such as the fdisk/disk manager) to partition and format the volume and mount it.

Using CLI tools

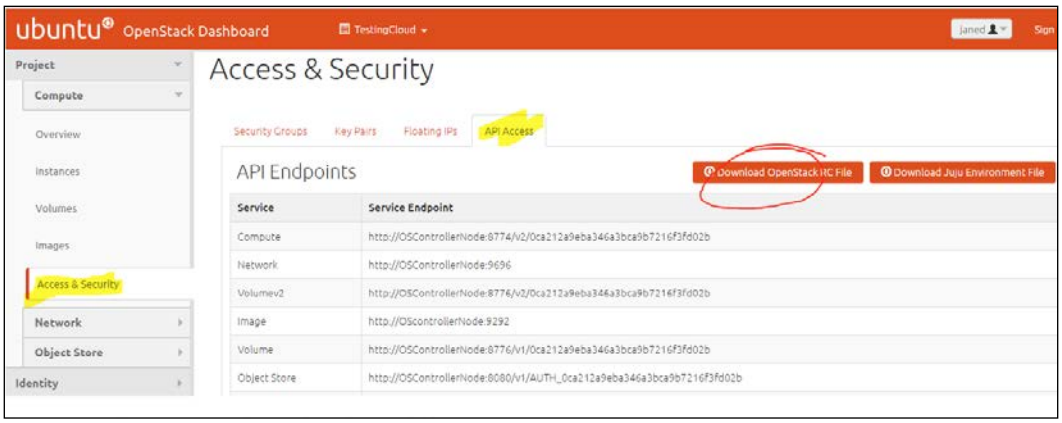
Jane, unlike John, is fond of command line (CLI) utilities and wants to perform the same tasks using either CLI tools or a scriptable method. She can choose to either use the APIs directly or install command line tools.

So, say Jane has a Linux machine that she uses as a development machine (running CentOS). She can install command line tools on her machine using either the Python `PiP`, or she can install them using the `yum` package manager.

The package name format is `python-<project>client`, so we can replace the project by Keystone, Glance, Nova, Cinder, and so on. For example, Jane will execute the following commands for the Nova, Glance and Cinder command line tools:

```
yum install python-novaclient
yum install python-cinderclient
yum install python-glanceclient
```

Once the tools are installed, Jane would log in to the GUI once, navigate to the **Access & Security** panel, click on the **API Access** tab, and click on **Download Openstack RC File**:



This will download a shell script that will help Jane to set the environment variables for the command line tools to work. After running the script, it will also prompt for Jane's password and will set the password as an environment variable. She will then have the ability to fire commands to perform the same functions as John, but using the command line tools instead.

Generating a key pair

In order to create a key pair using the CLI tools, Jane should execute the following command:

```
nova keypair-add jane-key
```

This command will create a key called `jane-key` and display the private key on screen, which Jane has to copy and store in a secure location.

Requesting a server

In order to request a server using the command line, Jane needs some information. She needs the flavor ID, image ID, and key name.

She can execute the following command:

```
nova flavor-list
```

She will get a list of flavors with their IDs and will see the list of images using the following command:

```
nova image-list
```

Jane would now see the following:

```
+----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
| ID | Name      | Memory_MB | Disk | Ephemeral | Swap | VCPUs | RXTX_Factor | Is_Public |
+----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
| 1  | m1.tiny   | 512       | 1    | 0         |      | 1     | 1.0         | True      |
| 2  | m1.small  | 2048      | 20   | 0         |      | 1     | 1.0         | True      |
| 3  | m1.medium | 4096      | 40   | 0         |      | 2     | 1.0         | True      |
| 4  | m1.large  | 8192      | 80   | 0         |      | 4     | 1.0         | True      |
| 5  | m1.xlarge | 16384     | 160  | 0         |      | 8     | 1.0         | True      |
+----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
root@OSControllerNode:~# nova image-list
+--------------------------------------+------------+--------+--------+
| ID                                   | Name       | Status | Server |
+--------------------------------------+------------+--------+--------+
| 34b205dc-63aa-4b66-a347-2ab98451252d | CirrosTest | ACTIVE |        |
+--------------------------------------+------------+--------+--------+
```

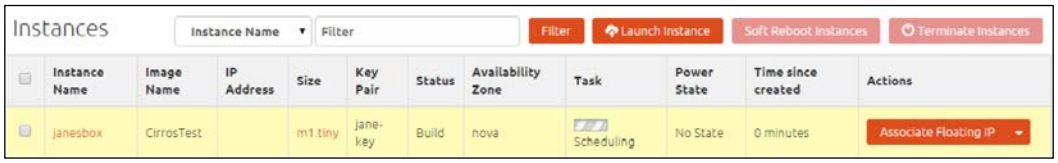
So if she wants to spin up a machine, which is `m1.tiny`, with the cirros test image and with the machine name of `janesbox`, she can use the following command:

```
nova boot --flavor 1 --key_name jane-key  --image 34b205dc-63aa-4b66-
a347-2ab98451252d janesbox
```

You have to replace the value of your image ID in the preceding command, and your key name, as shown here:

```
+--------------------------------------+------------------------------------------------------+
| Property                             | Value                                                |
+--------------------------------------+------------------------------------------------------+
| OS-DCF:diskConfig                    | MANUAL                                               |
| OS-EXT-AZ:availability_zone          | nova                                                 |
| OS-EXT-STS:power_state               | 0                                                    |
| OS-EXT-STS:task_state                | scheduling                                           |
| OS-EXT-STS:vm_state                  | building                                             |
| OS-SRV-USG:launched_at               | -                                                    |
| OS-SRV-USG:terminated_at             | -                                                    |
| accessIPv4                           |                                                      |
| accessIPv6                           |                                                      |
| adminPass                            | NEnTEdw7vv9w                                         |
| config_drive                         |                                                      |
| created                              | 2015-07-27T12:08:37Z                                 |
| flavor                               | m1.tiny (1)                                          |
| hostId                               |                                                      |
| id                                   | 0d4c3ef9-65a4-491b-81a5-72a78a424683                 |
| image                                | CirrosTest (34b205dc-63aa-4b66-a347-2ab98451252d)    |
| key_name                             | jane-key                                             |
| metadata                             | {}                                                   |
| name                                 | janesbox                                             |
| os-extended-volumes:volumes_attached | []                                                   |
| progress                             | 0                                                    |
| security_groups                      | default                                              |
| status                               | BUILD                                                |
| tenant_id                            | 0ca212a9eba346a3bca9b7216f3fd02b                     |
| updated                              | 2015-07-27T12:08:38Z                                 |
| user_id                              | 29d500bc4f7a4d0fb0449ef79d4bb428                     |
+--------------------------------------+------------------------------------------------------+
```

You can see that the machine starts to build like the one we built with the GUI:



So, with the CLI we can perform the same functions as that from the portal. More tech savvy people can make calls directly using a programming language of their choice using the APIs offered by the system.

Behind the scenes - how it all works

We have seen the entire process of requesting a service using both the GUI and the CLI, so as a recap let's take a look at how the process works behind the scenes. If we had to write our own toolsets using the RESTful API, these are the steps that we would take. The following ones show the major calls that are made in the process.

Knowing this, we should also be able to troubleshoot where a problem is, by isolating the step at which the machine fails to build and by looking at the appropriate log file:

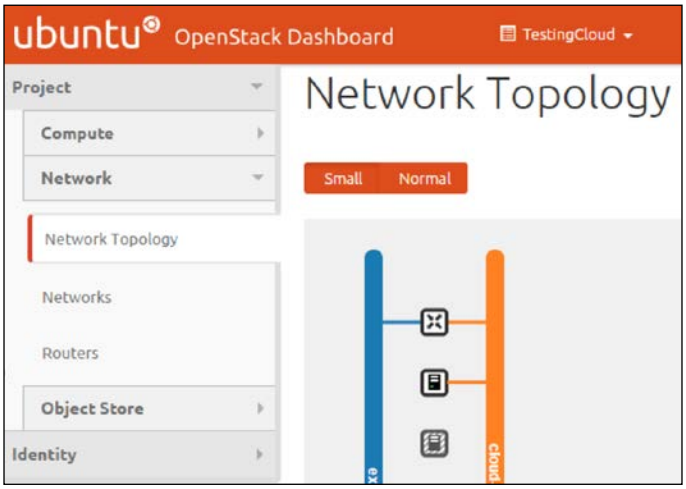
1. Authenticate and generate a token: As the first step, a call is made to the Keystone service, the credentials are presented, the tenant and the user are authenticated and a token is retrieved. This token is used in all the subsequent API calls, either directly or through Horizon that determines the level of access across all the services.
2. Make a call to Glance for images: As the second step, the user would make a call to the Glance service to retrieve the images that are allowed by the user, while using the token as the common authentication mechanism (if the token has not yet expired).
3. Make a call to Nova for a list of flavors: The next call is made to Nova to determine the list of flavors (sizes) of the machine.
4. Make a call to Nova for a list of available keys and security groups: The next call is made to Nova to determine the different security keys that are available, so we can choose the key name. Another call is made for the security groups and a list for these is obtained.
5. Make a call to Neutron for available networks: The next call is made to Neutron for the available networks that can be used and connected to the Nova instance.

   The previously mentioned calls can be made sequentially or in parallel or can even be prefetched and kept. Once the calls are made, the parameters can be passed when the request is made to Nova.

6. The Nova API accepts the call: The Nova API service accepts the call and puts the request in the messaging queue.
7. The Nova scheduler chooses compute: The Nova scheduler service picks up the message and finds the appropriate compute node where the instance can be brought online and puts the message back in the queue.
8. The Nova compute interacts with Hypervisor: The Nova compute service spins up the instance by sending the appropriate command to Hypervisor. The Nova compute can work with a variety of Hypervisors, such as Xen, KVM, and VMware.
9. A call is made to the Neutron API: Nova makes a call to the Neutron API for a port where the VM can be connected. Neutron allocates a port in the network defined in the call.

This way the entire machine is booted and then given to the user. The user can then choose to attach additional services such as a block storage using Cinder and an elastic IP address using Neutron. The process almost remains similar.

Once the machine is booted, we can take a look at its Network topology to see how it is connected to the environment:



Creating VM templates

Up until now, we have been using the cirros template that we downloaded from the Ubuntu website to testing our cloud. We now need to create our own virtual machine templates that we can use. These templates need to be created as per the standards of our organization and this is how we like them best.

The following tools may be used to create the images. There are others as well, just choose the one that works best for you depending on the images that you are trying to build. As an example VMBuilder that ships with Ubuntu can only create Ubuntu images:

- Oz
- Packer
- Disk image builder
- VM builder

In this chapter, we will use a tool called `Oz`. This nifty little tool can install the operating system and represents it as disk images. It actually installs the images using `libvirt`, so those packages are prerequisites to use the tool.

Installing Oz and its dependencies

On the machine where we intend to create the template, we can install Oz and its dependencies.

It is advised that these be separate from the OpenStack nodes.

RHEL/CentOS

Installing Oz on the RHEL or CentOS is easily accomplished using the following command:

```
yum -y install kvm libvirt oz qemu-kvm
```

The preceding command installs `Oz` and its dependencies.

Ubuntu

On Ubuntu, we need a few more packages. So we execute the following commands:

```
apt-get install build-essential kvm libguestfs-tools python-all
genisoimage mtools openssh-client
```

Once the pre-requisites are installed, we will use `git` to pull the packages, compile them, and install them:

```
mkdir ~/oz
cd ~/oz
git clone https://github.com/clalancette/oz.git oz-git
cd ~/oz/oz-git
dpkg-buildpackage -us –uc
cd ~/oz
dpkg -i oz_*_all.deb
```

This will install Oz on the system.

Oz templates

Oz uses template files in order to feed information to Oz as to what kind of machine needs to be built. This template is a kind of XML file, is readable, and easily understandable.

Let's take a look at a sample template file, which will use an ISO file in order to create a CentOS installation, which in effect will use a `kickstart` file to help with the installation:

```
<template>
  <name>centos64</name>
  <os>
    <name>CentOS-6</name>
    <version>4</version>
    <arch>x86_64</arch>
    <install type='iso'>
    <iso>http://mirror.rackspace.com/CentOS/6/isos/x86_64/CentOS-6.4-
    x86_64-bin-DVD1.iso</iso>
    </install>
  </os>
  <description>CentOS 6.4 x86_64</description>
</template>
```

The preceding template is the Oz template that is created. You can download sample templates for Oz from the following URL:

`https://github.com/rcbops/oz-image-build/tree/master/templates`

You can find Oz templates for several of the OSs that we normally build using Oz. However, for our purposes, we will use the previous template and save it as `centos64.tdl`, and we will now create a `kickstart` file to go along with this template.

You can choose to put all the customizations in the `kickstart` file itself.

Let's type the following in to a file and call it `myCentOS.ks`:

```
install
text
key --skip
keyboard us
lang en_US.UTF-8
skipx
network --device eth0 --bootproto dhcp
rootpw myRootPwd!
firewall --disabled
authconfig --enableshadow --enablemd5
```

```
selinux --disabled
timezone --utc America/Chicago
bootloader --location=mbr --append="console=tty0 console=ttyS0,115200"
zerombr yes
clearpart --all
part /boot --fstype ext4 --size=200
part swap --size=512
part / --fstype ext4 --size=1024 --grow
repo --name=epel --baseurl=http://ftp.nluug.nl/pub/os/Linux/distr/fedora-epel/6/x86_64/
reboot

%packages
@core
@base

%post
rpm -Uvh http://ftp.nluug.nl/pub/os/Linux/distr/fedora-epel/6/x86_64/epel-release-6-8.noarch.rpm
rm -f /etc/udev/rules.d/70-persistent-net.rules
sed -i '/HWADDR/d' /etc/sysconfig/network-scripts/ifcfg-eth0
sed -i '/UUID/d' /etc/sysconfig/network-scripts/ifcfg-eth0
sed -i 's,UUID=[^[:blank:]]* / ,/dev/vda3               / ,' /etc/fstab
sed -i 's,UUID=[^[:blank:]]* /boot,/dev/vda1               /boot,' /etc/fstab
sed -i 's,UUID=[^[:blank:]]* swap,/dev/vda2               swap,' /etc/fstab
rm -f /root/anaconda-ks.cfg
rm -f /root/install.log
rm -f /root/install.log.syslog
find /var/log -type f -delete
```

This will serve as my base file. You can change the root password and the time zone, and also add additional post-customization tasks.

Once we have created these files, we are now in a position to create the template.

Creating VM templates using Oz

We will now build a VM using the Oz tool and convert its disk by running the following command:

```
oz-install -p -u -d1 -a myCentOS.ks centos64.tdl
```

The system uses the `kickstart` file in order to complete the installation. We can use a similar method for Ubuntu, using a pre-seed file, and achieve the same thing.

The machine disk will be created in the `/var/lib/libvirt/images/` folder, with the file name `centos64.dsk`. (Please note the `<name>` tag in the `tdl` file.)

We will now need to convert this to the QCOW2 format. We do this using the `qemu-img` tool:

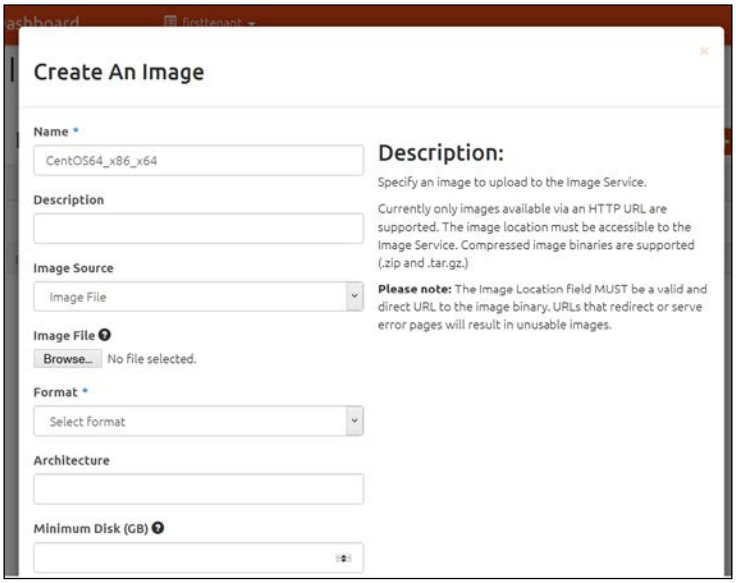
```
qemu-img convert /var/lib/libvirt/images/centos64.dsk -O qcow2
/root/centos64_x86_64.qcow2
```

Now we have an image that we can upload to our Glance repository, either using the command line tool or using Horizon.

Uploading the image

Let's upload the image using the Horizon portal. Log in to the portal as `admin` and navigate to the **Admin** dashboard and the **Images panel**.

Click on the **Create Image** button.



Once you fill in the details and click the **Upload** button, the image will be uploaded in the Glance repository.

Summary

In this chapter, we have put to use all that we learned in the previous chapters and saw the cloud in action. We have used both the GUI and the CLI method to achieve this. We have discussed tenant and user management, understood different types of networks in the context of OpenStack, and set up a provider network and a tenant network. We looked at security groups and key pairs. Finally, we launched our first VM (IaaS) in a self-service fashion. We also got a flavor of what really happens behind the scenes and how all of this work together, which is a very critical element that aids in troubleshooting, if the need arises.

Now that we have a basic cloud functioning, in the next chapter we will discuss some more advanced topics.

8
Taking Your Cloud to the Next Level

In the previous chapters, we looked at all the essential components required to build IaaS. In this chapter, we will look at how to install and configure two optional OpenStack services, namely Heat and Ceilometer.

Although these services are optional, in a production environment, it is definitely recommended that we have at least Ceilometer installed to meter different components, which can then be used to integrate with a billing system of your choice (for example, Velvica).

Heat, on the other hand, can help the users to use a template in order to orchestrate the entire stack.

In this chapter, we will cover the following topics:

- Installing and configuring Heat
- Installing and configuring Ceilometer
- Testing the installation
- Billing and usage reporting

Working with Heat

Heat is the orchestration engine of OpenStack. A Heat system can take care of provisioning an entire stack by requesting various services from other services of OpenStack (Nova, Cinder, and so on.) based on a text template. It can also be used to modify a currently running stack and delete it when we are done with it, so the entire life cycle of the stack can be managed using Heat.

Heat is functionally similar to AWS's CloudFormation and is compatible with AWS CF template, in addition to its own template format—**Heat Orchestration Template** (**HOT**).

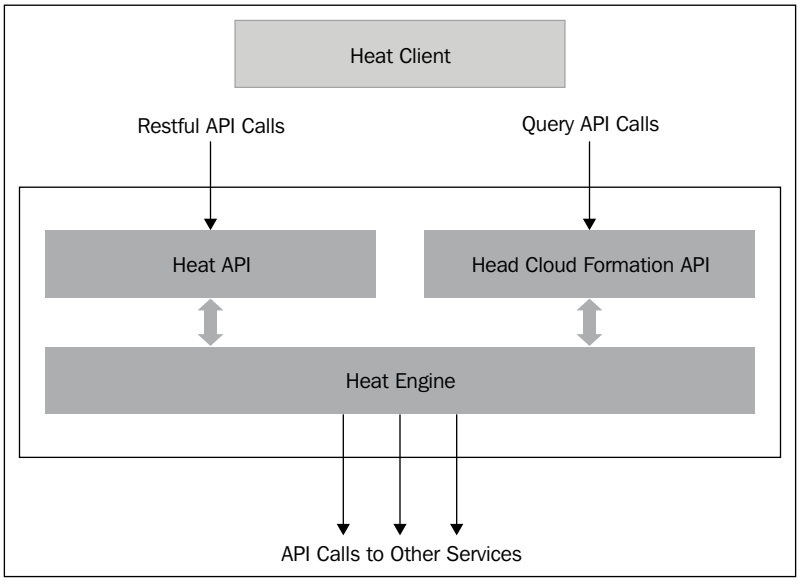
The Heat system supports the REST API, which is native to other OpenStack services and supports an AWS CloudFormation-style Query API.

The components of Heat

The Heat system does not have many subcomponents. The subcomponents and their functions are as follows:

- **Heat**: This is the command-line client, which is used to talk to the Heat API component to execute the API calls. This is mainly used by the administrators to execute a CloudFormation template.
- **Heat API**: This component accepts the API calls made to it and interfaces with the main subcomponent, the Heat engine. This accepts the REST-style API calls.
- **Heat CF API**: This component accepts the Query-style API that is native to the AWS CloudFormation. It performs the same functions as the Heat API.
- **Heat engine**: The Heat engine is the main subcomponent. It orchestrates the launch of the templates and passes the events back to the calling resource.

The following diagram demonstrates the architecture of the Heat system quite clearly:



Heat Orchestration Template (HOT)

While the format of the template is not yet set in stone, this format attempts to replace the CloudFormation (CFN) template that is currently in use. We shall briefly look at the template formats, just enough to understand how to read, modify, and create them.

Each HOT has the following fields:

- **Heat template version tag**: This is a mandatory field.
- **Description**: This is an optional field that is used to describe the template.
- **Resources**: This is a required field that describes the different resources that a stack will have, for example, a compute instance or a block storage for all the resources.
    - **Properties**: These are the properties of the resource that need to be passed.
    - **Parameters**: These are the properties that can be passed to the template during run time, allowing it to not be hard coded. These can have restrictions and constraints applied to them so that the user passes the same values to the template.

      Input parameters: These are the parameters that have to be filled in while requesting a use of the template, as an example, the input parameter can be the machine size that you want for a stack or a key you want to pass in.

      Output parameters: These are passed back to the user after the template is executed, as an example, the ID of a machine, its IP address, and so on.

A simple Heat Orchestration Template, which will request an instance with a predefined set of parameters, is shown in the following code. In this, we can see that the machine size needs to be passed during the runtime, and if it's not passed, it defaults to the `m1.small` size.

```
heat_template_version: 2013-05-23

description: Simple template to deploy a single compute instance

parameters:
  instance_type:
    type: string
    label: Instance Type
```

```
    description: Type of instance (flavor) to be used
    default: m1.small


resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      key_name: my_key
      image: MyCustomTemplate
      flavor: { get_param: instance_type }
```

We can create multiple resources in a template and the Heat engine will orchestrate them in a sequence. We can even create nested templates in order to pass the variables between a child and a parent template allowing us to create an intricate orchestration process.

Writing HOT is beyond the purview of the book. However, after understanding the constructs of HOT, we should be able to now read and understand the different ones that are available freely on the Internet.

Installing Heat

The installation of Heat services is straightforward and we will be installing them on the controller node. The installation follows the same principles and processes that we followed for the other services:

- Creating the database
- Keystone configurations (Creating users and endpoints)
- Installing packages
- Configuring packages
- Cleaning up

Let's start with the familiar checklists that we have been using so far, by filling in the details:

Name	**Info**
Access to the Internet	Yes
Proxy needed	No
Proxy IP and port	Not Applicable

Name	Info
Node name	OSControllerNode
Node IP address	172.22.6.95
Node OS	Ubuntu 14.04.1 LTS
Heat DB password	`h3atpwd`
Heat Keystone password	`h3atkeypwd`
CloudFormation port	8000
Heat orchestration port	8004
Region	dataCenterOne

Creating the database

We create a blank database after logging in to the MySQL server:

```
mysql -u root -p
```

Enter the `dbr00tpassword` password. Once in the database, execute the following commands:

```
create database heat;
```

This will create an empty database called Heat. Let's now set up the `heat` database user credentials:

```
GRANT ALL PRIVILEGES ON heat.* TO 'heat'@'localhost' IDENTIFIED BY
'h3atpwd';
GRANT ALL PRIVILEGES ON heat.* TO 'heat'@'%' IDENTIFIED BY 'h3atpwd';
```

All this does is allows the username `heat`, using our password, to be able to access the database called `heat`.

Installing components

The Heat components are installed using the aptitude package manager with the following command:

```
sudo apt-get install heat-api heat-api-cfn heat-engine python-heatclient
```

This installs the components. We do need to ensure that the command completes successfully and the components are installed.

The initial configuration

Let's now look at configuring Heat, which includes similar steps followed for other OpenStack services, such as creating a Keystone user and Keystone services and endpoints. However, a small difference in comparison to the other services is that for Heat, we will need to create some additional roles as well.

Creating a Heat user in Keystone

We will create a user in Keystone, and by now, you are familiar with how to export credentials in order to use the different OpenStack command-line utilities:

```
keystone user-create --name heat --pass h3atkeypwd
```

We should see something like the following screenshot:

```
root@OSControllerNode:~# keystone user-create --name heat --pass h3atkeypwd
+----------+----------------------------------+
| Property |              Value               |
+----------+----------------------------------+
|  email   |                                  |
| enabled  |               True               |
|    id    | d27ea09833ff4fd3b64d65eecba821a5 |
|   name   |               heat               |
| username |               heat               |
+----------+----------------------------------+
```

We then add the user to the admin by running the following command:

```
keystone user-role-add --user heat --tenant service --role admin
```

Creating additional Heat stack roles

Up until now, we have been dealing with just two roles—admin and member, we will now create two new roles called `heat_stack_owner` and `heat_stack_user`.

The `heat_stack_owner` role needs to be assigned to the people who will manage the stacks themselves and the `heat_stack_user` role is for the people who use these stacks:

```
keystone role-create --name heat_stack_owner
keystone role-create --name heat_stack_user
```

We should not assign both the `heat_stack_user` and owner roles to the same user, as this will cause some conflicts. The `heat_stack_user` role is assigned automatically to any users that are created using HOT.

We hope that you remember our users, Jane and John Doe, from our previous chapter. We will now assign the owner role to the user John (and to our admin user) in the tenant called `TestingCloud` that we created in our previous chapter, in the following way:

```
keystone user-role-add --user johnd --tenant TestingCloud --role
heat_stack_owner
keystone user-role-add --user admin --tenant TestingCloud --role
heat_stack_owner
```

This will give John and the admin users permissions to manage the stacks. This can also be done using the Horizon portal, the same way we assigned the member roles in the previous chapter.

Creating Heat services in Keystone

There are two services that need to be created for the Heat system, one for the regular REST API and the second one for the Query API that will be used in case of a CloudFormation call, as follows:

```
keystone service-create --name heat-cfn --type cloudformation \
--description "Orchestration"
keystone service-create --name heat --type orchestration \
--description "Orchestration"
```

You should see something like the following screenshot:

```
root@OSControllerNode:~# keystone service-create --name heat-cfn --type cloudformation \
> --description "Orchestration"
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
| description |          Orchestration           |
|   enabled   |               True               |
|      id     | 70bae78b99be4fbeb03e96dbe66af189 |
|     name    |             heat-cfn             |
|     type    |          cloudformation          |
+-------------+----------------------------------+
root@OSControllerNode:~# keystone service-create --name heat --type orchestration \
>  --description "Orchestration"
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
| description |          Orchestration           |
|   enabled   |               True               |
|      id     | 7934f03d9ec944fd87fec4f5f65bc0e8 |
|     name    |               heat               |
|     type    |          orchestration           |
+-------------+----------------------------------+
```

We will have to note the ID of both the services, which we will use in the next section.

In our case, the IDs are as follows:

- **Heat**: `7934f03d9ec944fd87fec4f5f65bc0e8`
- **Heat-CFN**: `70bae78b99be4fbeb03e96dbe66af189`

Creating Heat endpoints in Keystone

We will create two endpoints, one for the cloud formation and the other for Heat. Please ensure you replace the `service-id` with what you got in your previous step.

Creating the endpoint can be done as shown here:

```
keystone endpoint-create \
  --service-id 7934f03d9ec944fd87fec4f5f65bc0e8\
  --publicurl http://OSControllerNode:8004/v1/%\(tenant_id\)s \
  --internalurl http://OSControllerNode:8004/v1/%\(tenant_id\)s \
  --adminurl http://OSControllerNode:8004/v1/%\(tenant_id\)s \
  --region dataCenterOne
```

The second endpoint is created with the `port` and the `service-id` of the CloudFormation service that was created earlier, as follows: (Substitute it with the `heat-cfn` endpoint ID.)

```
keystone endpoint-create \
  --service-id 70bae78b99be4fbeb03e96dbe66af189 \
  --publicurl http://OSControllerNode:8000/v1 \
  --internalurl http://OSControllerNode:8000/v1 \
  --adminurl http://OSControllerNode:8000/v1 \
  --region dataCenterOne
```

This will create the Heat endpoints in Keystone.

Modifying the configuration file

We will now modify a single configuration file, `/etc/heat/heat.conf`. The following changes are done in the configuration file:

- In the `[database]` section, the following are done:
    - We will set the database connection string as:

```
connection = mysql://heat:h3atpwd@OSControllerNode/heat
```

- In the `[default]` section, this is done:

    - Configure the RabbitMQ access as follows:

```
rpc_backend = rabbit
rabbit_host = OSControllerNode
rabbit_password = rabb1tmqpass
```

    - Set the Heat metadata server as:

```
heat_metadata_server_url = http://OSControllerNode:8000
heat_waitcondition_server_url =
http://OSControllerNode:8000/v1/waitcondition
```

- In the `[keystone_authtoken]` section, the following are done:
    - Set the Keystone configuration:

```
auth_uri = http://OSControllerNode:5000/v2.0
identity_uri = http://OSControllerNode:35357
admin_tenant_name = service
admin_user = heat
admin_password = h3atkeypwd
```

- In the `[ec2authtoken]` section, the following settings are made:
    - Set the authentication URL:

```
auth_uri = http://OSControllerNode:5000/v2.0
```

Populating the database

We can populate the database using the following command (under root):

```
/bin/sh -c "heat-manage db_sync" heat
```

Ensure that the database is created and there are no errors. If there are any errors, please check the connection string that is mentioned in the `heat.conf` file and ensure that the MySQL instance is up.

Finalizing the installation

We will delete the SQLite database file and restart all the services by running the following commands:

```
rm -f /var/lib/heat/heat.sqlite
service heat-api restart
service heat-api-cfn restart
service heat-engine restart
```

This concludes the installation steps that are needed for the Heat system to run.

Deploying your first HOT

Now that we have completed the hard work of setting up Heat, let's create a sample HOT and create a stack.

The template is very similar to what we saw in the previous section, and it simply requests a single server and takes the parameters as an image name, network ID, and machine size.

Create a file in directory `/var` and name it as `test-stack.yml`. The contents of the file are shown as follows. If we look carefully, we have set the default machine size to be `m1.tiny`, so if we do not override this, we will get an `m1.tiny` machine. The contents of the file are as shown:

```
heat_template_version: 2014-10-16
description: Testing stack with a single server
parameters:
  image_id:
    type: string
    description: Image use to boot a server
  net_id:
    type: string
    description: Network ID for the server
  instance_type:
    type: string
    label: Instance Type
```

```
    description: Type of instance (flavor) to be used
    default: m1.tiny
resources:
  server:
    type: OS::Nova::Server
    properties:
      image: { get_param: image_id }
      flavor: { get_param: instance_type }
      networks:
      - network: { get_param: net_id }
```

Once the template is created, we will export the credentials for the `TestingCloud` tenant. The only difference between the admin credentials that we have been exporting so far is the name of the tenant.

We have also added the admin user in the `heat_stack_owner` role, so this will work just fine. Alternatively, we can use John's credentials that we created. We will export the credentials as shown here:

```
export OS_TENANT_NAME=TestingCloud
export OS_USERNAME=admin
export OS_PASSWORD=h33l0world
export OS_AUTH_URL=http://OSControllerNode:5000/v2.0
```

We will need the following information to pass to the template:

- Network ID
- Image name
- Flavor name

We can then execute the following commands to view the available choices:

```
nova flavor-list
nova image-list
nova net-list
```

We have just one image that we have created, `CirrosTest`. We will use this. We can leave the flavor information to the default `m1.tiny` value, and we will need the ID for `cloud-network` so that our computer can be connected to this system:

```
root@OSControllerNode:/var# nova flavor-list
+----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
| ID | Name      | Memory_MB | Disk | Ephemeral | Swap | VCPUs | RXTX_Factor | Is_Public |
+----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
| 1  | m1.tiny   | 512       | 1    | 0         |      | 1     | 1.0         | True      |
| 2  | m1.small  | 2048      | 20   | 0         |      | 1     | 1.0         | True      |
| 3  | m1.medium | 4096      | 40   | 0         |      | 2     | 1.0         | True      |
| 4  | m1.large  | 8192      | 80   | 0         |      | 4     | 1.0         | True      |
| 5  | m1.xlarge | 16384     | 160  | 0         |      | 8     | 1.0         | True      |
+----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
root@OSControllerNode:/var# nova image-list
+--------------------------------------+------------+--------+--------+
| ID                                   | Name       | Status | Server |
+--------------------------------------+------------+--------+--------+
| 34b205dc-63aa-4b66-a347-2ab98451252d | CirrosTest | ACTIVE |        |
+--------------------------------------+------------+--------+--------+
root@OSControllerNode:/var# nova net-list
+--------------------------------------+---------------+------+
| ID                                   | Label         | CIDR |
+--------------------------------------+---------------+------+
| fea41df3-e54f-40d5-8027-11d9b00c18f4 | ext-net       | None |
| c83453d6-599f-4b01-aedf-7b8561eff67c | cloud-network | None |
+--------------------------------------+---------------+------+
```

So, we can have values as follows:

- `Parameter: net_id= c83453d6-599f-4b01-aedf-7b8561eff67c`
- `Parameter: image_id=CirrosTest`
- `Parameter: instance_type=m1.tiny`

We will now create the stack using the Heat client with the parameters as shown in the following. The stack will be called as `mystack`.

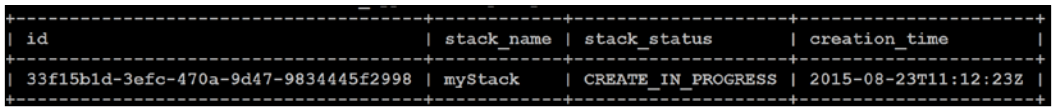
The format is as shown:

```
heat stack-create –f <filename of HOT template> -P <Parameters>
"Stack Name"
```

Substituting our values, we get the following: (We could even use the names, and they will be translated.)

```
heat stack-create -f /var/test-stack.yml \
  -P "image_id=CirrosTest;net_id=c83453d6-599f-4b01-aedf-
7b8561eff67c;instance_type=m1.tiny" myStack
```

The output would look as shown here:



```
+--------------------------------------+------------+--------------------+----------------------+
| id                                   | stack_name | stack_status       | creation_time        |
+--------------------------------------+------------+--------------------+----------------------+
| 33f15b1d-3efc-470a-9d47-9834445f2998 | myStack    | CREATE_IN_PROGRESS | 2015-08-23T11:12:23Z |
+--------------------------------------+------------+--------------------+----------------------+
```

We can see that the stack creation is in progress. We will need to wait until the stack is deployed, you can test the status using the following command:

```
heat stack-list
```

We have successfully tested the orchestration module of OpenStack, and we can manage our stacks using HOT, if need be. This enhances the functionality of the cloud that enables complicated stacks to be defined and easily spun up by users. This can be used in the cases where the platform team defines the stacks that are allowed and the templates to be shared with users to spin up new stacks at their will.

It is worthwhile to note that this is merely another abstraction layer, and the restrictions that are set by the other layers still apply. As an example, the user can only use the network that is assigned to a project, and they can only spin up the maximum number of instances allowed by the quota, so on and so forth.

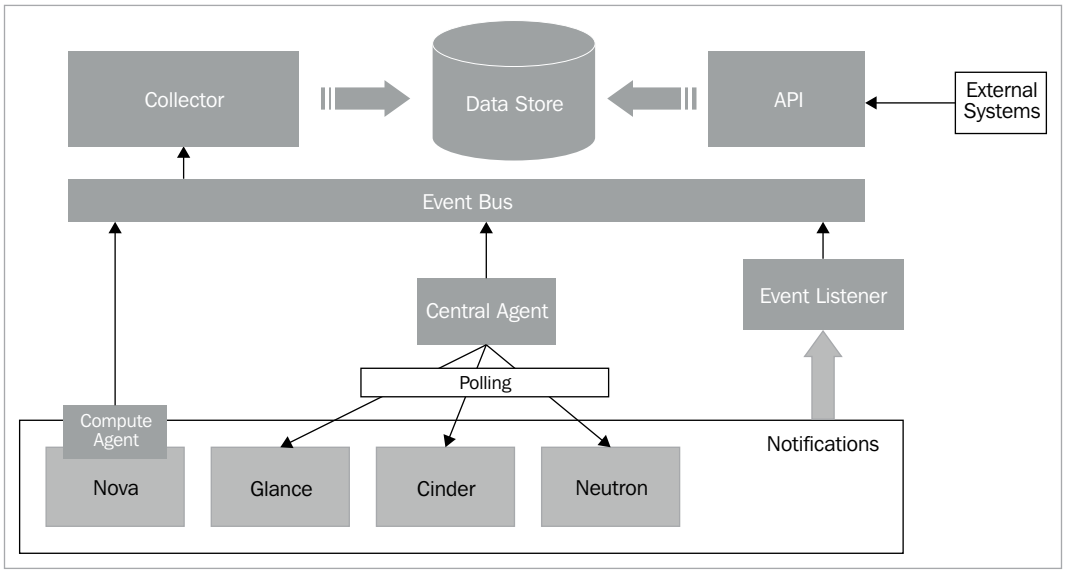
The next service that we will add to our cloud is Ceilometer, which is a telemetry module used to measure the different aspects of the cloud for the purposes of billing.

Ceilometer

Ceilometer is another optional module that can be installed in the OpenStack environment. This is a telemetry system, and its function is to collect the usage statistics from different components of OpenStack and to store and warn when the data breaches certain thresholds. The output can be used for billing, auditing, and capacity planning purposes, and whatever your need may be. It is a good idea to run a Ceilometer service in the OpenStack environment only if we need metering data for billing purposes. If we need to just perform capacity planning in a private cloud without any need to bill downstream users, then we can ignore this component.

Please note that this system does not provide billing information, and an external billing system needs to be used if you need to generate showback/chargeback reports for your company.

The following diagram shows the different major components of the Ceilometer system:



As shown in the diagram, the **Data Store** component is the heart of the system, where all data will finally reside. The key aspects of the system are, as follows:

- Data collection
- Data access
- Meters
- Alarms

The system allows us to write custom agents, if we need to. However, this is beyond the scope of our book, and therefore let's concentrate on the basics:

- **Data collection**: Data is collected from the various different services of the OpenStack system. There are three kinds of inputs that are fed to the system:
    - **User action**: A user performing a CRUD action on a resource generated.
    - **Polling**: The Ceilometer agent polls the different services for data.
    - **Audit**: The regular audit events generated by the service.

There are agents, such as the compute agent, which sit right on the the compute node, and they keep sending information. Another central agent's function is to poll the different services and gather information, and the OpenStack notifications goes to the event listener. All of these feed the data into the collector using the Ceilometer event bus.

This was done so that Ceilometer is turned into a distributed system. The collector component then dumps the data into the data store.

- **Data access**: The next important thing is how we fetch the data that has been stored in the data store by the collector accumulating it from various methods from various sources. The API element, which is Keystone authenticated, provides a REST API which we can then use to either fetch raw data or data in different meters.
- **Meters**: Ceilometer has three kinds of meters, or ways, in which it stores and interprets raw data:
    - **Cumulative**: Increasing over time (for example, instance hours).
    - **Delta**: Changing over time (for example, bandwidth).
    - **Gauge**: This includes discrete items like floating IPs and fluctuating values such as disk IO.

  These meters are for now sufficient to cater to the needs of the different OpenStack services.
- **Alarms**: Ceilometer also provides the capability to send out alarms when the collected data crosses a certain defined threshold conditions.

Installing Ceilometer

Ceilometer is installed on the controller node, the agent for compute is installed on the compute node, and Glance, Cinder and, Swift are configured with notifications, so we will be making modifications to the various nodes of our installation.

Installing Ceilometer on the controller node

The control components on Ceilometer are installed on the controller node. There is a little difference when it comes to Ceilometer. We will be using a MongoDB rather than the regular MySQL(MariaDB) installation that we have used for the other services.

This is because the MongoDB is a NoSQL database that is especially suited for data sources with high volume and where fast access is needed. This forms a perfect fit for the purposes of Ceilometer.

Installing and configuring MongoDB

We install the MongoDB and its client using the following command:

```
apt-get install mongodb-server mongodb-clients python-pymongo
```

We will now configure the MongoDB to listen on the controller node's IP address. We will make the following changes to the `/etc/mongodb.conf` file:

- Change `bind_ip` to the controller node's IP address (Please set this to your controller node's IP address). This will allow MongoDB to listen to the requests from outside the nodes as well:

  ```
  bind_ip = 172.22.6.95
  ```

- Make the MongoDB journal space small, since this is just a learning server. In the production server, you may skip this step:

  ```
  smallfiles = true
  ```

Once the configuration changes have been made, we will stop the service, delete the old journal files, and start the service using this command:

```
service mongodb stop
rm /var/lib/mongodb/journal/prealloc.*
service mongodb start
```

You should be able to see the MongoDB service listening on the IP address, using the `netstat -lnp` command.

Before continuing any further, let's fill our checklist so that we have the values handy:

Name	**Info**
Access to the Internet	Yes
Proxy needed	No
Proxy IP and port	Not applicable
Node name	`OSControllerNode`
Node IP address	172.22.6.95
Node OS	Ubuntu 14.04.1 LTS
Ceilometer DB password	`c3ilpwd`
Ceilometer Keystone password	`c3ilkeypwd`

Creating the database

The database will be created in the MongoDB and will have a different format:

```
mongo --host OSControllerNode --eval '
  db = db.getSiblingDB("ceilometer");
  db.addUser({user: "ceilometer",
  pwd: "c3ilpwd",
  roles: [ "readWrite", "dbAdmin" ]})'
```

Executing the preceding command creates the database and adds the user with read/write and DB admin roles. The following output can be seen upon the successful creation of the database in MongoDB:

```
root@OSControllerNode:~# mongo --host OSControllerNode --eval '
>   db = db.getSiblingDB("ceilometer");
>   db.addUser({user: "ceilometer",
>   pwd: "c3ilpwd",
>   roles: [ "readWrite", "dbAdmin" ]})'
MongoDB shell version: 2.4.9
connecting to: OSControllerNode:27017/test
{
        "user" : "ceilometer",
        "pwd" : "0c5a28f73791ee509010a51d37dfefd8",
        "roles" : [
                "readWrite",
                "dbAdmin"
        ],
        "_id" : ObjectId("55d9e3eab90f661e1b4c6798")
```

Installing packages

We will install packages using the Aptitude package manager with the following command:

```
apt-get install ceilometer-api ceilometer-collector ceilometer-agent-
central \
  ceilometer-agent-notification ceilometer-alarm-evaluator
ceilometer-alarm-notifier \
  python-ceilometerclient
```

Let's look at the various components and the functions they perform:

- `ceilometer-api`: This forms the frontend that accepts API calls from external systems and taps into the data source to respond to queries.
- `ceilometer-collector`: This is the collector module, which aggregates the inputs from different sources and saves the data in the data source, or it can also send the data to an external consumer.
- `ceilometer-agent-central`: This performs polling functionality and keeps polling the different services of OpenStack. We can have multiple central agents for scalability and performance purposes in an installation.
- `ceilometer-agent-notification`: This service consumes messages from the message queue and builds event and metering data.
- `ceilometer-alarm-evaluator`: This service determines which alarms to fire due to the values crossing the threshold over a sliding time window.
- `ceilometer-alarm-notifier`: This is used to set the alarms based on threshold for v.
- `ceilometer-client`: This is used to fire away the Ceilometer commands to test and configure the system.

These packages are installed on the controller server.

Initial configuration

Let's now look at configuring Ceilometer!

Creating the Ceilometer user in Keystone

After exporting the admin credentials, either using individual export commands or sourcing the file, as we have done in the past, let's create Keystone user using the command:

```
keystone user-create --name ceilometer --pass c3ilkeypwd
```

You should get an output like the following screenshot:

```
+----------+----------------------------------+
| Property |              Value               |
+----------+----------------------------------+
|  email   |                                  |
| enabled  |               True               |
|    id    | 688f1364fd9343e59e6e48246d49bfad |
|   name   |            ceilometer            |
| username |            ceilometer            |
+----------+----------------------------------+
```

The user account is created.

We will then add the created account as an admin in the service tenant using the following command:

```
keystone user-role-add --user ceilometer --tenant service --role
admin
```

Creating the Ceilometer service

We will create the service for Ceilometer in Keystone using the following command with the type as `metering`:

```
keystone service-create --name ceilometer --type metering \
--description "Telemetry"
```

You should get the following screen:

```
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
| description |            Telemetry             |
|   enabled   |               True               |
|      id     | b843a8c5485b4ad7a4f038cdfea1e668 |
|     name    |            ceilometer            |
|     type    |             metering             |
+-------------+----------------------------------+
```

We will note down the ID, as it will be needed in the next step. In our case, it is `b843a8c5485b4ad7a4f038cdfea1e668`.

Creating the Ceilometer endpoint

We will create the endpoint using the following command, please be sure to replace your service ID here:

```
keystone endpoint-create \
  --service-id b843a8c5485b4ad7a4f038cdfea1e668 \
  --publicurl http://OSControllerNode:8777 \
  --internalurl http://OSControllerNode:8777 \
  --adminurl http://OSControllerNode:8777 \
  --region dataCenterOne
```

You should see something like the following screenshot:

```
root@OSControllerNode:~# keystone endpoint-create \
>   --service-id b843a8c5485b4ad7a4f038cdfea1e668 \
>   --publicurl http://OSControllerNode:8777 \
>   --internalurl http://OSControllerNode:8777 \
>   --adminurl http://OSControllerNode:8777 \
>   --region dataCenterOne
+-------------+----------------------------------+
|   Property  |              Value               |
+-------------+----------------------------------+
|   adminurl  |   http://OSControllerNode:8777   |
|      id     | b680928408794bef83480fa7fe5a52e4 |
| internalurl |   http://OSControllerNode:8777   |
|  publicurl  |   http://OSControllerNode:8777   |
|    region   |          dataCenterOne           |
|  service_id | b843a8c5485b4ad7a4f038cdfea1e668 |
+-------------+----------------------------------+
```

The service is now created successfully.

Generating a random password

We will now generate a random password using the following command, which we will use as a metering secret:

```
openssl rand -hex 10
```

Note the output down, that will be similar to the following screenshot, and we will use this in place of the metering secret:

```
root@OSControllerNode:~# openssl rand -hex 10
a17a568977c28b18b8ec
```

In our case, it is `a17a568977c28b18b8ec`.

Editing the configuration files

We need to edit the configuration file located at `/etc/ceilometer/ceilometer.conf`. We need to make the following changes:

- In `[database]` section, this has to be done:
    - We will set the database `connection` string, as follows:

```
connection = mongodb://ceilometer:c3ilpwd@
OSControllerNode:27017/ceilometer
```

- In `[default]` section, we will do the following:
  - Configure the RabbitMQ access as:

```
rpc_backend = rabbit
rabbit_host = OSControllerNode
rabbit_password = rabb1tmqpass
```

  - Set the authentication strategy to `keystone` as:

```
auth_strategy = keystone
```

- In the `[keystone_authtoken]` section, the following is done:
  - Set the Keystone configuration as:

```
auth_uri = http://OSControllerNode:5000/v2.0
identity_uri = http://OSControllerNode:35357
admin_tenant_name = service
admin_user = ceilometer
admin_password = c3ilkeypwd
```

- In the `[service_credentials]` section, the following settings are done:
  - Set the service credentials as:

```
os_auth_url = http://OSControllerNode:5000/v2.0
os_username = ceilometer
os_tenant_name = service
os_password = c3ilkeypwd
```

- In the `[publisher]` section, we will do the following:
  - Set the `metering_secret`: (this is the random password that we generated earlier.)

```
metering_secret = a17a568977c28b18b8ec
```

Enabling the Glance notification

We need to edit the `/etc/glance/glance-api.conf` and `/etc/glance/glance-registry.conf` configuration files. In both of them, we will enable the notification as follows:

- In the `[DEFAULT]` section, the following configurations are done:
    - Configure notification and RabbitMQ, as follows:

```
notification_driver = messagingv2
rpc_backend = rabbit
rabbit_host = controller
rabbit_password = rabb1tmqpass
```

Ensure both the files are updated with the configurations. If there are any conflicting values, then comment out the old configurations. (For example, you may see that `notification_driver` may be set to no operation.)

Enabling the Cinder notification

We will need to configure a notification in the configuration file of Cinder located at `/etc/cinder/cinder.conf`:

- In the `[DEFAULT]` section, do the following:
    - Configure notifications, as follows:

```
control_exchange = cinder
notification_driver = messagingv2
```

This will enable the notification in the Cinder system; we will then restart the Cinder services.

Enabling the Swift notification

Enabling the Swift notification will need some additional steps:

We have to create a `ResellerAdmin` role in Keystone, which will be used in order to federate the authentication with Keystone. In addition, Ceilometer will require access to the Swift folders, and hence the group modification needs to be done on Ceilometer's account.

Creating the ResellerAdmin role

In order to create a `ResellerAdmin` role, we will export our admin credentials using the export commands or by sourcing the file where the export commands are set, as we have done in the past.

We then execute the following command:

```
keystone role-create --name ResellerAdmin
```

```
root@OSControllerNode:~# source ~alokas/os.txt
root@OSControllerNode:~# keystone role-create --name ResellerAdmin
+----------+----------------------------------+
| Property |              Value               |
+----------+----------------------------------+
|    id    | 259165da71bc40e983e0d3218ffe31d0 |
|   name   |          ResellerAdmin           |
+----------+----------------------------------+
```

We have to note down the ID that is generated so that we can use this in the next command. In our case, this is `259165da71bc40e983e0d3218ffe31d0`.

We will then associate the role with the Ceilometer user account in the service tenant using the following command: (Remember to substitute your role ID appropriately.)

```
keystone user-role-add --tenant service --user ceilometer \
  --role 259165da71bc40e983e0d3218ffe31d0
```

This will associate the role. We could also do this using the Horizon frontend, as we have seen in the previous chapter.

Enabling notifications

We need to perform these steps on all the Swift proxy nodes. In our case, we have only installed it on the controller, and hence we will configure it only here.

The file that we will be modifying is `/etc/swift/proxy-server.conf`. We will make the following changes:

- In the `[filter:keystoneauth]` section, we will do the following:
  - Add the `ResellerAdmin` role, as follows:

    ```
    operator_roles = admin,_member_, swiftoperator,
    ResellerAdmin
    ```

- In the [pipeline:main] section, we will do this:
    - Add `ceilometer` just before the `proxy-server`, as shown here:

```
pipeline = healthcheck cache authtoken keystoneauth proxy-logging ceilometer proxy-server
```

- In the `[filter:ceilometer]` section (create this if it doesn't exist), we configure as follows:
    - Configure the notifications, as follows:

```
use = egg:ceilometer#swift
log_level = WARN
```

Once the configuration changes are made, we will need to restart the Swift proxy service as explained in the *Finalizing the installation* subsection.

Allowing Swift access to Ceilometer files

We will allow Swift user access to the Ceilometer configuration files by adding the Swift user to the Ceilometer group:

```
usermod -a -G ceilometer swift
```

This way the Swift user also has permissions for Ceilometer.

Finalizing the installation

As a final step, we will restart all the services:

```
service ceilometer-agent-central restart
service ceilometer-agent-notification restart
service ceilometer-api restart
service ceilometer-collector restart
service ceilometer-alarm-evaluator restart
service ceilometer-alarm-notifier restart
service glance-registry restart
service glance-api restart
service cinder-api restart
service cinder-scheduler restart
service swift-proxy restart
```

Once we complete this, we move on to the next step of installing the compute agents.

Installing Ceilometer on the compute node

On the compute node, we will install the compute agent, which will send the details to the collector via the message bus.

Log in to `OSComputeNode` (in our case, `172.22.6.97`).

Installing the packages

We will install one package called the compute agent using the package manager:

```
apt-get install ceilometer-agent-compute
```

Let's look at the function of the component

Ceilometer-Agent-Compute

The `ceilometer-agent-compute` function polls for the resource utilization statistics and sends it to the collector.

Once we have installed the package, we will need to follow some steps to configure this.

Initial configuration

We will need to configure the Ceilometer agent, and we will enable notifications in the configuration for Nova.

In the file located at `/etc/ceilometer/ceilometer.conf`, we need to make the following changes:

- In the `[default]` section, we will do the following:
  - Configure the RabbitMQ access, as follows:

```
rpc_backend = rabbit
rabbit_host = OSControllerNode
rabbit_password = rabb1tmqpass
```

- In the `[keystone_authtoken]` section, we will do the following settings:
  - Set the Keystone configuration, as shown here:

```
auth_uri = http://OSControllerNode:5000/v2.0
identity_uri = http://OSControllerNode:35357
admin_tenant_name = service
admin_user = ceilometer
admin_password = c3ilkeypwd
```

- In the `[service_credentials]` section, we will do as shown here:
    - Set the service credentials, as follows:

```
os_auth_url = http://OSControllerNode:5000/v2.0
os_username = ceilometer
os_tenant_name = service
os_password = c3ilkeypwd
os_endpoint_type = internalURL
os_region_name = dataCenterOne
```

- In the `[publisher]` section, we will do the following:
    - Set the `metering_secret` (it's the random password that we generated earlier):

```
metering_secret= a17a568977c28b18b8ec
```

Enable Nova notification

In the Nova configuration file, we will enable the notifications. In the `/etc/nova/nova.conf` file, we will do the following:

- In the `[default]` section, we will do this:
    - Set the notifications, as follows:

```
instance_usage_audit = True
instance_usage_audit_period = hour
notify_on_state_change = vm_and_task_state
notification_driver = messagingv2
```

Finalizing the installation

The preceding changes conclude the installation process in the compute node. This needs to be done in every compute node in the environment. We will restart the services in order for the configurations to take effect, as shown:

```
service ceilometer-agent-compute restart
service nova-compute restart
```

Installing Ceilometer on the storage node

On the storage node, we will enable the notifications for Cinder.

Enabling Cinder notification

We will need to configure the notification in the Cinder configuration file located at `/etc/cinder/cinder.conf`:

- In the `[DEFAULT]` section, we will do the following:
    - Configure notifications, as follows:

```
control_exchange = cinder
notification_driver = messagingv2
```

This will enable the notification for the Cinder service; we will then restart the Cinder services.

Finalizing the installation

The services need to be restarted to complete the installation:

```
service cinder-volume restart
```

The notifications on the Cinder system are now enabled.

Testing the installation

Now that Ceilometer has been installed, we need to be able to test it. In order to do so, we will use the controller node and export the admin credentials.

In order to see the current meters, we will execute the following command:

```
ceilometer meter-list
```

This will show the current metered units. Since we do not have anything metered, it will show something like the following screenshot:

```
+------------+-------+-------+--------------------------------------+
| Name       | Type  | Unit  | Resource ID                          |
+------------+-------+-------+--------------------------------------+
| image      | gauge | image | 34b205dc-63aa-4b66-a347-2ab98451252d |
| image.size | gauge | B     | 34b205dc-63aa-4b66-a347-2ab98451252d |
+------------+-------+-------+--------------------------------------+
```

We see the image and the image size at the two meters that are available.

We will now perform some functions and ensure that Ceilometer picks them up. One such simple task is downloading an image from Glance. So, we have one image that we called `CirrosTest`, let's just download it and ensure that the metering is correct.

We can download the image using the following command:

```
glance image-download "CirrosTest" > sample.img
```

This just downloads the image from Glance and stores it locally as a file called `sample.img`.

Once this is done, we execute the same command one more time and we will see that the download now appears, as shown:

Name	Type	Unit	Resource ID	User ID
image	gauge	image	34b205dc-63aa-4b66-a347-2ab98451252d	None
image.download	delta	B	34b205dc-63aa-4b66-a347-2ab98451252d	None
image.serve	delta	B	34b205dc-63aa-4b66-a347-2ab98451252d	None
image.size	gauge	B	34b205dc-63aa-4b66-a347-2ab98451252d	None

Now, since this is a metering system, we should be able to see the amount using the following command (you can substitute any meter name to get the statistics of that meter):

```
ceilometer statistics -m image.download
```

You should be able to see the following screen:

```
root@OSControllerNode:/var/log# ceilometer statistics -m image.download
```

Period	Period Start	Period End	Max	Min
0	2015-08-24T18:43:39.233000	2015-08-24T18:43:39.233000	13200896.0	13200896.0

We notice that the `Max` amount shows the value `13200896.0`. We can verify the size of the image by executing the `ls -l` command on the downloaded file:

```
ls -l sample.img
```

This will indeed confirm that a file of that size was downloaded. You should see something like the following screenshot:

```
root@OSControllerNode:/var/log# ls -l sample.img
-rw-r--r-- 1 root root 13200896 Aug 25 00:30 sample.img
```

If we download the file multiple times, we can verify that the count in the Ceilometer statistics will keep incrementing.

Billing and usage reporting

After Ceilometer has been configured, we will need a billing system that can use this data and present it to the user in a usable format.

Some of the meters that are used in case of the guest virtual machine would typically be as follows:

Name	Type	Unit	Meaning
`cpu`	cumulative	ns	The total CPU time used
`cpu_util`	gauge	%	The average CPU utilization. of the instance
`disk.ephemeral.size`	gauge	GB	The ephemeral disk size
`disk.read.bytes`	cumulative	B	The total number of bytes read
`disk.read.requests`	cumulative	request	The total number of read requests made
`disk.root.size`	gauge	GB	The root disk size for the instance
`disk.write.bytes`	cumulative	B	The total number of bytes written to the disk
`disk.write.requests`	cumulative	request	The total number of write requests to the disk
`instance`	gauge	instance	The number of instances in existence
`instance:m1.large`	gauge	instance	The number of large instances in existence
`instance:m1.medium`	gauge	instance	The number of medium instances in existence
`instance:m1.small`	gauge	instance	The number of small instances in existence

Name	Type	Unit	Meaning
`instance:m1.xlarge`	gauge	instance	The number of extra large instances in existence
`memory`	gauge	MB	The memory allocated by the hypervisor to the instance
`network.incoming.bytes`	cumulative	B	The total number of bytes incoming to a network interface
`network.incoming.packets`	cumulative	packet	The total number of packets incoming to a network interface
`network.outgoing.bytes`	cumulative	B	The total number of bytes outgoing from a network interface
`network.outgoing.packets`	cumulative	packet	The total number of packets outgoing from a network interface
`vcpus`	gauge	vcpu	The number of virtual CPU's

The meter definitions are stored in a file called `/etc/ceilometer/meter/data/meter.yaml`, a standard meter definition looks like the following:

```
metric:
  - name: 'meter name'
    event_type: 'event name'
    type: 'type of meter eg: gauge, cumulative or delta'
    unit: 'name of unit eg: MB'
    volume: 'path to a measurable value eg: $.payload.size'
    resource_id: 'path to resouce id eg: $.payload.id'
    project_id: 'path to project id eg: $.payload.owner'
```

We can use the preceding format and create new meters as required in Ceilometer.

The OpenStack foundation currently has a project, which functions as a billing system called the CloudKitty.

The CloudKitty project installers are not yet available, but one can easily install it from the source. It also has a basic Horizon integration to show the billing aspects. We can use any billing system as long as it can make REST requests to the system. It should be able to pull data from Ceilometer and present it in any format it deems fit to the end user.

Summary

Having looked at all the basic services of OpenStack in the previous chapters, in this chapter we have discussed some optional services such as orchestration (using Heat) and metering (using Ceilometer), which will definitely be the next step for any enterprise when adopting OpenStack.

There are several other modules that are available in Juno, but we haven't discussed them here. These modules include Designate, Manila, and so on, which can be explored as you start playing with your OpenStack installation and get more comfortable.

With this, we come to the end of the installation and configuration of OpenStack components. In the next chapter, we shall look at various distributions and use cases of OpenStack.

9
Looking Ahead

After having travelled together in this wonderful OpenStack journey so far, we are now nearing the close. We have seen in the previous chapters how a private or a public cloud can be set up with OpenStack, and how we can choose to offer its flexibility beyond just the standard compute, storage, and networking aspects of an IaaS cloud by adding several other OpenStack components.

In this chapter, we will look at where OpenStack currently stands in the market and in the near future. We will also look at some of the major deployments and who is backing them. We will look at how OpenStack clouds will look in the future. We will conclude this book by looking at a more important question: What is in it for you?

To be specific, we will discuss the following major topics in this chapter:

- The different OpenStack distributions
- Seeing OpenStack in action via several use cases

OpenStack distributions

In this book, we have installed the OpenStack components using the aptitude package manager after adding the Ubuntu repositories. We could have installed the components in several other ways, and there are several distributions available, which provide either the packages or scripts needed to install the system. We can, of course, install the components from the source by cloning the Git repository for the packages and using the install script, but several vendors have come up with their own distributions.

Devstack

Devstack is a development distribution, as the name suggests, and it can install all the components either in a single box or multiple ones that can be chosen at installation. This is a good and quick way to test the features of the different components apart from developing them further and fixing bugs, if you choose to contribute. Devstack is available for installation on Ubuntu, Fedora, and RHEL currently, but can also be installed on the other distros as well.

Operating system distributions

The operating system distribution vendors are creating distributions of OpenStack, which can be easily installed on their respective operating systems by packaging and making the distributions available as repositories. In this very book, we have used an OS distribution of OpenStack (Ubuntu). Let's look at the major ones.

Ubuntu OpenStack

Ubuntu OpenStack is designed to run on the **Long Term Support** (**LTS**) editions of Ubuntu and has releases for Kilo, Juno, and Icehouse. It also has the support of most popular hypervisors right from ESXi, Hyper-V, KVM, Qemu, and LxC (Linux Containers). You can also get a production-grade 24x7 support.

RedHat OpenStack

RedHat OpenStack is designed to work on Enterprise Linux edition and has support only for ESXi and KVM. It currently is on the Juno release, and Kilo is soon launching.

Oracle OpenStack

Oracle OpenStack is installed on top of Oracle Enterprise Linux. The key difference is that this is possibly the only distribution to support Solaris. Xen and KVM hypervisors are supported by this distribution.

Vendor offerings

Several vendors offer their products and services along with OpenStack. These vendors range from Hypervisor vendors with tight integration to their platform to cloud providers providing public, private, or hybrid offerings. There are certain other offerings based on just a few components of the stack, but not the full stack, such as the storage service alone.

Since an exhaustive list will be pointless as new vendors are being added very often, the list will soon be obsolete. We will, therefore, look at some key and famous offerings.

VMware integrated OpenStack

At the time of writing the book, VMware-integrated OpenStack was the only distribution that came from a hypervisor vendor. You can get this for free, if you have an enterprise plus license from VMware. This is integrated nicely into the vSphere web client and deploys a production grade, highly available OpenStack in just a matter of a few clicks, and is tightly integrated with the ESXi Hypervisor. The latest available version is based on the Kilo release.

This is slowly but surely becoming quite popular among enterprises that have already made huge investments in terms of VMware licenses for hypervisors.

Rackspace cloud

Rackspace needs a special mention in this book, not only because they run a famous public cloud based on OpenStack, but also because if it was not for them, we would not even have OpenStack, as Rackspace and NASA actually started this project in 2010. They are still, however, on Icehouse with the Xen hypervisor.

HP Helion

In the FOSS segment for cloud products, OpenStack and Eucalyptus were two products that were solving the same problems. HP acquired Eucalyptus and has added it to its Helion cloud offerings. However, what most people do not realize is that there are two offerings, and if someone wants AWS-like APIs they can choose HP Helion Eucalyptus, or if they prefer OpenStack, they can choose the OpenStack version of it.

Cisco OpenStack

Cisco has an OpenStack distribution running on its UCS chassis and provides mostly private cloud solutions for enterprises that enable an easy deployment of a supported OpenStack installation in their datacenters.

Mirantis OpenStack

Mirantis OpenStack is one of the most flexible and at the same time, an open distribution of OpenStack and more importantly, you can purchase support here. In addition, the hypervisor support for this distribution ranges from Xen, Docker, Hyper-V, ESXi, LXC (Linux containers), QEMU, and KVM. Actually, this is an exhaustive list of what OpenStack supports (other than the bare metal provisioning that is code named Ironic). So if you want more supported choices in terms of hypervisors, look no further.

SwiftStack

SwiftStack is an example of a partial OpenStack implementation. It only implements as the name implies, Swift, the object store service of OpenStack. Therefore, this forms a choice if you want to use an object store (AWS S3 equivalent). You can also connect this with your in-house Swift deployment of OpenStack.

IBM Cloud manager

The IBM Cloud manager is from the tech giant IBM, which provides integration with the z/VM hypervisor running on mainframes. It also provides some management toolsets along with their distribution. Their current release is based on Juno.

Suse Cloud

Based on OpenStack and Crowbar, this private cloud offering supports mixed hypervisor cloud deployments based on the Icehouse release of OpenStack.

Other public clouds

There are several vendors (aside from the ones we previously discussed), whose public clouds are designed using OpenStack. Here are a few to name (in no particular order):

- Internap (`www.internap.com`)
- Anchor Cloud (`www.anchor.com.au`)
- Ultimum Cloud (`ultimum.io`)
- Dream compute (`www.dreamhost.com`)
- Kloud open (`https://kionetworks.com`)
- Elastx ops (`elastx.com`)

These are a few in a long list of service providers working with OpenStack. This list continues to change very frequently, and in order to see the most updated list, be sure to visit the OpenStack market place for the different distributions:

`https://www.openstack.org/marketplace/distros/`.

Choosing a distribution

After having understood the services, the purposes of the services, and the architecture, here are a few things to keep in mind while choosing a distribution if you are looking to implement OpenStack for your company's private cloud:

- **Service Level Agreements** (**SLA**): You need to take a look at the kind of support and the SLAs that the different providers offer.
- **Hypervisor support**: Not all the distributions support all hypervisors, even if the underlying OpenStack does. You should choose the distribution that supports the ones you are using currently and intend to use in the future. Also, read the fine print as to what happens when you configure OpenStack with the hypervisors that the distribution does not support—it would work, but it might invalidate the entire support contract, or just the hypervisor that is not supported.
- **Operating system support**: Almost all the distributions support Windows and Linux, but say if you need Solaris support, you may have to get a distribution that does this.
- **Update speed**: We should also check the frequency at which a distribution is updated and how quickly it is updated after a release cycle of OpenStack, patches, and so on, especially if you are anxious to get the new features and like to be on the cutting-edge side of things.
- **Lock in**: Check whether the distribution locks into some proprietary technology, such as hardware or software. You may have to decide whether this is something your company can live with.
- **Additional tools**: Many of the distributions sometimes package additional tools that help to manage or administer OpenStack better. For example, Suse offers a **Suse Studio**, which makes it easy to manage templates and converts them from one format to the other.

In order to help with the choosing, let's look at the following table, which lists the hypervisor and operating system support for a few distributions normally used in the private cloud world.

Distribution name	Hypervisor support	OS support
Ubuntu OpenStack	Hyper-V, QEMU, KVM, ESXi, LXC	Linux, Windows
Red Hat OpenStack	ESXi, KVM	Linux, Windows
VMWare Integrated OpenStack	ESXi	Linux, Windows
Mirantis OpenStack	Xen, Docker, Hyper-V, ESXi, LXC, QEMU, KVM	Linux, Windows
Oracle OpenStack for Solaris	Solaris	Solaris
Oracle OpenStack	Xen, KVM	Linux, Windows, Solaris
Cisco OpenStack	QEMU, KVM	Linux, Windows
IBM Cloud Manager	z/VM, PowerVM, ESXi, Hyper-V, KVM	Linux, Windows
Suse Cloud	Xen, Hyper-V, ESXi, KVM	Linux, Windows
Dell Red Hat Cloud	KVM	Linux, Windows
HP Helion OpenStack	ESXi, KVM	Linux, Windows

OpenStack in action

OpenStack being the most successful FOSS segment cloud product, the implementations of this are too many to detail. In this section, we will look at some of the most common use cases where you might see OpenStack in action.

Enterprise Private Cloud

Enterprise Private Cloud is one of most common use cases, and the most likely reason why you are reading this book. If you are in the IT organization of any enterprise that chooses to offer private cloud to its different business units, which now demand agility, flexibility, lower time-to-market, and so on, OpenStack is the right choice wherein you can offer an Amazon Web Services style of services while exposing APIs to support cloud native applications.

Some of the enterprises that have adopted this are eBay, Alcatel-Lucent, BMW, PayPal, NASA, and Sony, among several others.

Service providers

If you are in a service provider line of business, such as a datacenter provider, you may want to start offering cloud services. OpenStack's distributed nature can easily help you build a service provider-grade cloud for multiple tenants, and you can throw some additional integrations over and above the stock OpenStack offering, such as toolsets and SLAs.

There are several service providers that use OpenStack today, some of them are already mentioned in the previous sections. Some additional names are AT&T, Telstra CCS (Cisco Cloud Services), Korea Telecom, Dream Host, and so on .

Schools/Research centers

Even schools use OpenStack in their labs to quickly provision different kinds of workloads for students, staff, and research faculty who are conducting projects or research in various fields of their study. Not being dependent on the IT team greatly reduces the time required to start a project.

Some of the notable examples are CERN, MIT CSAIL, and so on.

Web/SaaS providers

These sorts of companies need agility. There are several hundreds of them and their success criteria depends on how quickly they can bring in new features to their products, hence the Dev/Test and the entire DevOps paradigm for them becomes key to survival, and OpenStack can help them achieve just that. These companies inevitably use OpenStack or an equivalent to help them in this area.

Some examples in this segment would be `MercadoLibre.com`, Platform 9, and so on.

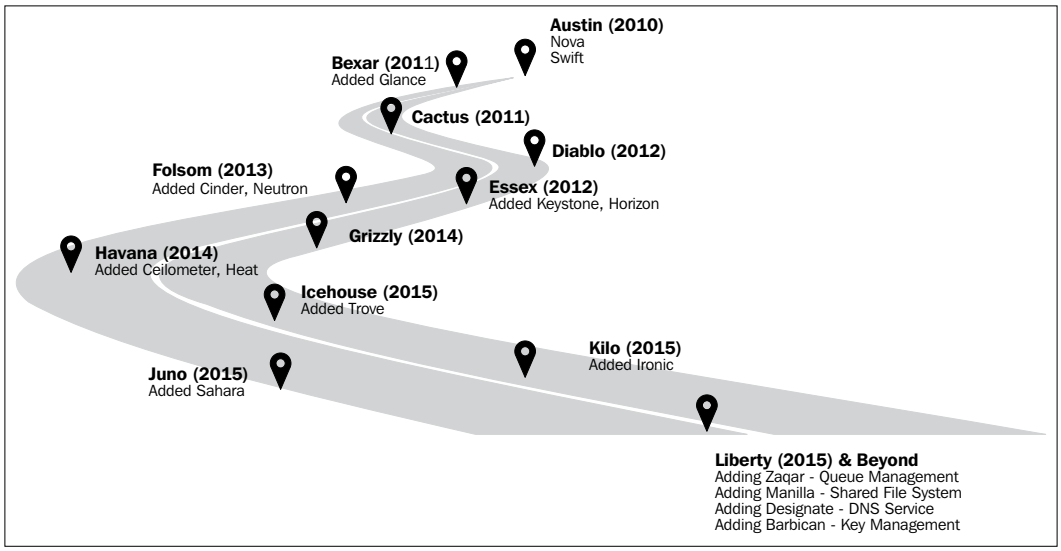
These are a few segments where OpenStack saves the day for various organizations and their IT staff.

The roadmap

The OpenStack roadmap is nothing short of exciting when we look at it. There are over a hundred companies backing OpenStack, it has over 6000 contributors and is slowly becoming the de-facto cloud standard. There are several new OpenStack services that have been added in the newly-released Kilo and several others that are planned for the next major release of Liberty and beyond that.

With the complete suite of the different OpenStack products, OpenStack is currently quite ahead in the market compared to any of its competitors in the FOSS segment. Coming second in this segment is OpenNebula, which is far behind in terms of adoption numbers.

The following diagram shows the full roadmap for OpenStack since its inception in 2010. At this point, there are 12 programs that are already part of OpenStack and four that are in active development (they already have had interim beta releases that can be used in Kilo) to be included in the next release of Liberty.



We have already covered in the first chapter of the book, the services and the functions they perform that are being added to Liberty, as the developer versions in Kilo have been released as well. The previous diagram shows the addition of the stable releases of the projects to the corresponding OpenStack releases.

With all the projects, both the stable ones and the ones under development, OpenStack comes remarkably close to Amazon Web Services and its offering. So when deployed, OpenStack can provide both the feel and functionality of the public cloud giant in-house and can run in a controlled environment.

What is in it for you?

Our belief is that this is possibly the most exciting time to be associated with OpenStack, as the demand for professionals in this field is ginormous and the adoption of OpenStack keeps increasing with every new core service that gets added to the stack.

We have also seen some large backing from companies such as Google, and right now, OpenStack is rightly being called the **Linux of the Cloud**.

Earlier this year (2015), a survey of 3,000 IT decision makers found that about a third were deploying private clouds to enhance enterprise operations. Half of these were using OpenStack as the underlying platform. Most of the enterprise IT teams worth their salt are using OpenStack or are surely considering it. However, the real challenge is, many companies are not moving their most critical operations, including native applications, to OpenStack when building private or hybrid cloud environments. Why? The primary reason is that there is a massive gap in the skills required to properly implement and manage platforms such as OpenStack.

We believe that this is the right kind of opportunity for all of you who have made a start by reading this book to go ahead, build on your skills, and enhance your career in the next generation cloud world. "May the force be with you".

Summary

In this chapter, we looked at the different distributions of OpenStack and vendor cloud offerings based on OpenStack. We then looked at the different use cases and the roadmap.

With this chapter, we have finally concluded our journey in this book. At the end of this, we hope that you now have enough knowledge to install and configure OpenStack to set up your first private cloud.

New Releases

As you are aware by now, our book is based on the Juno release of OpenStack. However, there have been newer releases during the writing of this book. In this section, we shall look at the major differences among the last three releases of OpenStack. The latest release is Liberty, which was launched on October 15, 2015, right before the publication of this book.

The core concepts and services of the releases were very similar in all the three releases, and most changes have been related to advanced concepts and a new "Big Tent" approach to services. The focus of our book has been primarily around the basic or mandatory services. Nevertheless, it is worthwhile to know the differences among the releases so as to keep our knowledge level up to date.

"Big Tent" is the new approach to OpenStack services. Earlier, we had an "integrated release" that included multiple projects and was growing. However, the challenge in this was that not every new project made sense for every OpenStack cloud and not every new project was actually being deployed. Therefore, a new model called "Big Tent" has been adopted, in which there are projects that are considered to be core and mature and then there is everything else. Among the projects that are core include the Nova compute, Swift storage, Glance image, Horizon dashboard, Neutron network, and Keystone identity.

We will cover the following important topics in this appendix:

- Understanding the newer releases
- Changes in the installation procedures of the new releases
- Upgrading from Juno

The releases

Kilo release is the 11th OpenStack release, and has multiple improvements and bug fixes mainly to enhance the core and Big Tent services of the OpenStack. This adds a few new services (as shown in the first chapter) in the area of infrastructure components.

Liberty is the 12th release of OpenStack, and has major improvements in the area of the governance, manageability, and extensibility aspects of OpenStack. The Liberty release added new services mainly in the field of governance and operationalization of OpenStack. This was done effectively to cater to the needs of enterprises and help them to efficiently and effectively manage their OpenStack-based private clouds.

The table in the next section summarizes the major additions/changes to the services of OpenStack that we have covered in this book. Most of these changes are on the advanced aspects of OpenStack and build upon the same concepts that were discussed earlier in the book. This is not an exhaustive list, and just shows the changes that were made to the top of the list based on the demand (or expectations) from the community itself.

If you are interested in all the new features and changes that were made in the releases, you should look at the release notes located at the following:

Kilo: `https://wiki.openstack.org/wiki/ReleaseNotes/Kilo`

Liberty: `https://wiki.openstack.org/wiki/ReleaseNotes/Liberty`

Features and differences

The following table shows a snapshot of the new features and the difference they made to the last two releases.

OpenStack Project	Kilo	Liberty
Nova	• Nova scheduler has been improved, and now Kilo is preparing to split it as its own project. • NUMA has been completed. • The EC2 API has been deprecated and a new Stack Forge project has been introduced to do the EC2 API translation. • The v2.1 API layer has been introduced. • Cells were introduced. This construct, when used, allows a multiregion style of Nova deployment.	• NFV: Network Functions Virtualization. • Improvement to schedulers has been made. This includes customizable schedulers. • v2.1 API has seen substantial modifications.
Swift	• Support for erasure codes to store large infrequently-used data. This is fundamentally similar to RAID. • Composite tokens allow data deletion with the consent of both user and service.	• The overall performance has been improved on slower drives and latency spikes has been reduced. • Ring operations have been made easier with ring-builder-analyzer to test ring-related operation. • Erasure coding has been improved. • Per object metadata has been included to explode archives.

OpenStack Project	Kilo	Liberty
Glance	• Supports multiple swift containers. • Artifact repository was introduced to store different artifacts such as the Heat templates and the Murano application packages.	• Image verification has been added by allowing the signing of a Glance image with a private key for integrity checks. • S3 Proxy was introduced to allow images to be stored to and retrieved from the S3 backend using a HTTP proxy.
Cinder	• Supports rolling upgrades. • New volume types have been introduced. • New drivers namely Dell Storage Center and Cloud Founders Open vStorage have been added.	• Quota enforcement in hierarchal tenants/ projects has been added. • A nondisruptive backup has been added.
Neutron	• Advanced services have been split into their own repositories. • MTU and path advertisement. • IPv6 Router: More support for IPv6.	• The pluggable IPAM (IP Address Manager) layer has been added. • IPv6 prefix delegation has been added. • Bandwidth quotas at port level have been added. • A new reference implementation of LBaaS (Load Balancing as a Service) has been added.

OpenStack Project	Kilo	Liberty
Horizon	• Based on Django 1.8. • Improvements have been made to support Federated Identity. • Support for themes has been added.	• New views have been added such as the new Launch instance and network topology pages. • Control of the IDP-specific (Identity provider) WebSSO configuration has been enabled.
Keystone	• The WSGI architecture implementation has been made. • Fernet tokens, which are non-persistent tokens with symmetric encryption keys, have been added. • Hierarchical multi-tenancy: The projects/ tenants nested under another tenant feature have been added. • Support has been given to and improvement made to the identity federation.	• Multi cloud authentication improvements have been made. • Greater control over IDP has been enabled.
Ceilometer	• Additional meters have been added. • A role base access control has been added for the API. • IP v6 support has been added.	• New alarm service called Aodh added. • There has been improved performance. • New meters can easily be added.

OpenStack Project	Kilo	Liberty
Heat	• New template functions have been added. • Multiregion stacks can now be deployed. • There has been improved scaling using nested stacks.	• New resources such as Barbican, Designate, and Keystone have been added.

Most of these features are configured by modifying the service configuration files. For the purpose of this book, most of these fall under the realm of advanced configuration, and hence are beyond the purview of the book.

Changes in the installation procedure

The installation mainly remains the same, except for the differences mentioned in the following sections.

Adding the repository

We will need to add the appropriate repository for the version that we need to install, so if we were to install the Kilo release, we will add the Kilo repository as shown (we added the repository in *Chapter 2*, *Authentication and Authorization Using Keystone*, before installing Keystone):

```
apt-get install ubuntu-cloud-keyring
echo "deb http://ubuntu-cloud.archive.canonical.com/ubuntu trusty-updates/kilo main" > /etc/apt/sources.list.d/cloudarchive-kilo.list
```

For a Liberty release, the repository can be added as follows:

```
apt-get install software-properties-common
add-apt-repository cloud-archive:liberty
```

The OpenStack client

The Juno release had individual clients with the `python-<projectname>client` format, for example, `python-keystoneclient`, `python-novaclient`, and so on. In the Kilo and Liberty releases, this is being replaced by a single client called `python-openstackclient`. This replaces all of the previous clients.

We can install the client by executing the following command:

```
apt-get install python-openstackclient
```

Once this is done, we no longer need to install the individual clients.

In the Kilo release, you still use the older clients, but it will throw a deprecation notice on the screen. With the new client, we can use the same commands as with the older clients.

Installing Keystone

In both Kilo and Liberty, the Keystone service needs to be installed a little differently. The older method will still work, but it is being deprecated in favor of using the WSGI Apache methodology.

So, in order to enable that, after we have installed the Keystone service, we will execute the following steps:

1. Disable the Keystone service from starting automatically, as this will now be proxied via the Apache server:

   ```
   echo "manual" > /etc/init/keystone.override
   ```

2. Install the Apache and Apache WSGI modules:

   ```
   apt-get install keystone apache2 libapache2-mod-wsgi \
   memcached python-memcache
   ```

3. Edit the `/etc/apache2/apache2.conf` file and add the `ServerName` directive to the hostname of the controller node.

4. Create a new file, `/etc/apache2/sites-available/wsgi-keystone.conf`, and paste the following content in it:

   ```
   Listen 5000
   Listen 35357

   <VirtualHost *:5000>
       WSGIDaemonProcess keystone-public processes=5 threads=1
   user=keystone group=keystone display-name=%{GROUP}
       WSGIProcessGroup keystone-public
       WSGIScriptAlias / /usr/bin/keystone-wsgi-public
       WSGIApplicationGroup %{GLOBAL}
       WSGIPassAuthorization On
   ```

```
    <IfVersion >= 2.4>
      ErrorLogFormat "%{cu}t %M"
    </IfVersion>
    ErrorLog /var/log/apache2/keystone.log
    CustomLog /var/log/apache2/keystone_access.log combined

    <Directory /usr/bin>
        <IfVersion >= 2.4>
            Require all granted
        </IfVersion>
        <IfVersion < 2.4>
            Order allow,deny
            Allow from all
        </IfVersion>
    </Directory>
</VirtualHost>

<VirtualHost *:35357>
    WSGIDaemonProcess keystone-admin processes=5 threads=1
user=keystone group=keystone display-name=%{GROUP}
    WSGIProcessGroup keystone-admin
    WSGIScriptAlias / /usr/bin/keystone-wsgi-admin
    WSGIApplicationGroup %{GLOBAL}
    WSGIPassAuthorization On
    <IfVersion >= 2.4>
      ErrorLogFormat "%{cu}t %M"
    </IfVersion>
    ErrorLog /var/log/apache2/keystone.log
    CustomLog /var/log/apache2/keystone_access.log combined

    <Directory /usr/bin>
        <IfVersion >= 2.4>
            Require all granted
        </IfVersion>
        <IfVersion < 2.4>
            Order allow,deny
```

```
            Allow from all
        </IfVersion>
    </Directory>
</VirtualHost>
```

5. Create a soft link of the file, as follows:

```
ln -s /etc/apache2/sites-available/wsgi-keystone.conf \
/etc/apache2/sites-enabled
```

6. Restart the Apache service. This way, Apache will be the frontend for all the requests coming to Keystone. The Keystone service doesn't need to be started automatically.

This is a major installation difference when it comes to the Kilo or the Liberty release compared to Juno. The older installations of Keystone will still work in the Kilo release.

Service configurations

In the service configuration, there is only one notable difference. In all the configuration files for the different services, wherever we have the `[keystone_authtoken]` section, we will have to make the following modifications:

Juno	Kilo/Liberty

`[keystone_authtoken]`
`auth_uri = http://controller:5000/v2.0`
`identity_uri = http://controller:35357`
`admin_tenant_name = service`
`admin_user = <Service_UserName>`
`admin_password = <Service keystone pwd>`
	`[keystone_authtoken]`
	`auth_uri = http://controller:5000`
	`auth_url = http://controller:35357`
	`auth_plugin = password`
	`project_domain_id = default`
	`user_domain_id = default`
	`project_name = service`
	`username = <Service Username>`
	`password = <Service keystone pwd>`

As we can see, there are three new fields to be added (`auth_plugin`, `project_domain_id`, and `user_domain_id`) and `identity_uri` is replaced with `auth_url` and `admin_tenant_name` with `project_name`.

This configuration is found in all the different service files and needs to be replaced to use the Apache WSGI configuration.

You can find the install guides for the Kilo and Liberty releases for the Ubuntu OpenStack distribution at the following:

`http://docs.openstack.org/kilo/install-guide/install/apt/content/ch_preface.html`

`http://docs.openstack.org/liberty/install-guide-ubuntu/overview.html`

Upgrading from Juno

The Juno install can be upgraded to the Kilo release and then finally can be upgraded to the Liberty release with the following procedure.

The controller node needs to be upgraded first followed by the storage, compute, and finally network nodes. The configuration of the services needs to be done in the same order as the installation, so in our book, we have followed this order: Keystone, Glance, Cinder, Swift, Nova, Neutron, and Horizon.

Cleanup

Any instances or volumes, which are in an inconsistent state needs to be purged or cleaned. This should be done from the CLI or Horizon portal. For some rogue entities, we may need to remove them from the database directly (please note that it is not recommended that the database be touched directly).

We will remove the null UUIDs of the Nova instances with the following command:

```
nova-manage db null_instance_uuid_scan
```

Backup

Create a backup of all the nodes, if they are virtual, using a snapshot. The MySQL database backup can be performed using the `mysqldump` command. The following command format is used:

```
mysqldump -u root -p[root_password] [database_name] >
/backup/path/dbname.sql
```

In order to restore the nodes, we use the `mysql` command line as follows:

```
mysql -u root -p[root_password] [database_name] < dumpfilename.sql
```

Adding the repositories

We will add the new repository as shown in the previous section. After adding the repository, execute this command:

```
apt-get update
```

Running the upgrade

Running the upgrade is performed by the following command:

```
apt-get upgrade
```

During the upgrade, you will be asked whether you want to keep the existing configuration files or replace them with samples. We recommend that you keep your existing configurations.

Installing additional components

We will install the OpenStack client as described in previous sections and also install the Apache and WSGI modules for Apache as shown previously.

Updating the DB schema

The database schema has changed in the new version, so we will now update it using the commands that we used during the installation. For example, look at the following:

- **Keystone**: `keystone-manage db_sync`
- **Nova**: `nova-manage db_sync`
- **Glance**: `glance-manage db_sync`
- **Cinder**: `cinder-manage db_sync`

Modifying configuration files

We will modify the configuration files as shown previously (by modifying the `keystone_authtoken` section) and also make the WSGI configuration for Apache for Keystone as shown in the previous section of this appendix.

Restarting services

We will need to restart every service in the same order that we restarted them during the installation. We also recommend a full clean reboot of the OpenStack nodes.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Thank you for buying
Learning OpenStack

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.

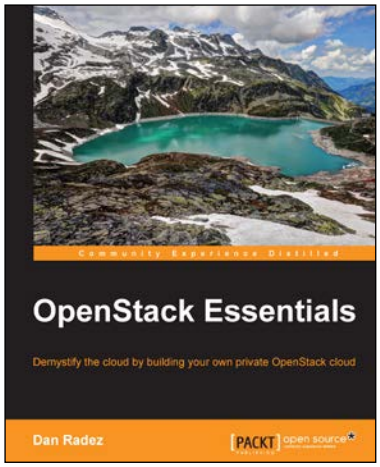
[PACKT] open source
community experience distilled
PUBLISHING

OpenStack Cloud Computing Cookbook

ISBN: 978-1-84951-732-4 Paperback: 318 pages

Over 100 recipes to successfully set up and manage your OpenStack cloud environments with complete coverage of Nova, Swift, Keystone, Glance, and Horizon

1. Learn how to install and configure all the core components of OpenStack to run an environment that can be managed and operated just like AWS or Rackspace.
2. Master the complete private cloud stack from scaling out compute resources to managing swift services for highly redundant, highly available storage.



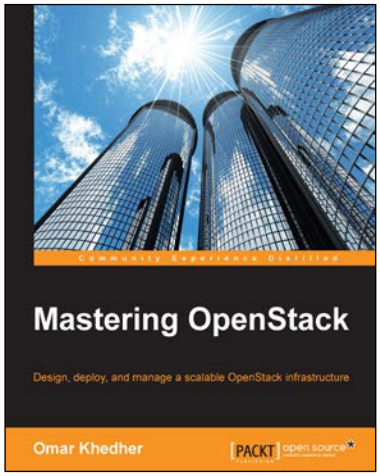
OpenStack Essentials

ISBN: 978-1-78398-708-5 Paperback: 182 pages

Demystify the cloud by building your own private OpenStack cloud

1. Set up a powerful cloud platform using OpenStack.
2. Learn about the components of OpenStack and how they interact with each other.
3. Follow a step-by-step process that exposes the inner details of an OpenStack cluster.

[PACKT] open source
community experience distilled
PUBLISHING

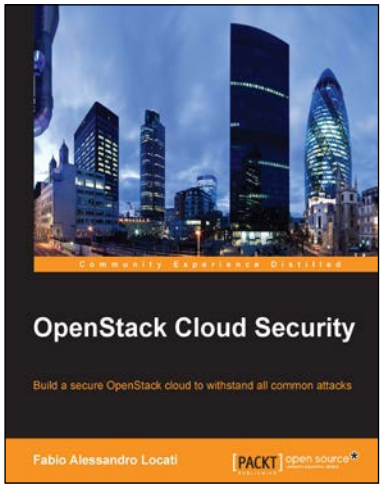


Mastering OpenStack

ISBN: 978-1-78439-564-3 Paperback: 400 pages

Design, deploy, and manage a scalable OpenStack infrastructure

1. Learn how to design and deploy an OpenStack private cloud using automation tools and best practices.
2. Gain valuable insight into OpenStack components and new services.
3. Explore the opportunities to build a scalable OpenStack infrastructure with this comprehensive guide.



OpenStack Cloud Security

ISBN: 978-1-78217-098-3 Paperback: 160 pages

Build a secure OpenStack cloud to withstand all common attacks

1. Design, implement, and deliver a safe and sound OpenStack cluster using best practices.
2. Create a production-ready environment and make sure your cloud storage and other resources are secure.
3. A step-by-step tutorial packed with real-world solutions that helps you learn easily and quickly.

Please check **www.PacktPub.com** for information on our titles