

Quick answers to common problems

Kubernetes Cookbook

Learn how to automate and manage your Linux containers and improve the overall performance of your system

Hideto Saito
Hui-Chuan Chloe Lee
Ke-Jou Carol Hsu

[PACKT] open source
PUBLISHING
community experience distilled



Quick answers to common problems

Kubernetes Cookbook

Learn how to automate and manage your Linux containers and improve the overall performance of your system

Hideto Saito
Hui-Chuan Chloe Lee
Ke-Jou Carol Hsu

[PACKT] open source
PUBLISHING
community experience distilled

Table of Contents

Kubernetes Cookbook
Credits
About the Authors
About the Reviewer
www.PacktPub.com
    eBooks, discount offers, and more
        Why Subscribe?
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Sections
        Getting ready
        How to do it…
        How it works…
        There's more…
        See also
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Errata
        Piracy
        Questions
1. Building Your Own Kubernetes
    Introduction
    Exploring architecture
        Getting ready
        How to do it…
            Kubernetes master
                API server (kube-apiserver)
                Scheduler (kube-scheduler)
                Controller manager (kube-controller-manager)

- Command Line Interface (kubectl)
- Kubernetes node
  - kubelet
  - Proxy (kube-proxy)
- How it works…
  - etcd
  - Overlay network
    - Flannel
- See also

Preparing your environment
- Getting ready
  - Hardware resource
  - Operating system
- How to do it…
  - Kubernetes master
  - Kubernetes nodes
  - etcd
- See also

Building datastore
- How to do it…
  - Red Hat Enterprise Linux 7 or CentOS 7
  - Ubuntu Linux 15.10 Wily Werewolf
  - Other Linux
    - Download a binary
    - Creating a user
    - Install etcd
- How it works…
  - Auto startup script
    - Startup script (systemd)
    - Startup script (init)
  - Configuration
- See also

Creating an overlay network
- Getting ready
  - Installation
    - CentOS 7 or Red Hat Enterprise Linux 7
    - Other Linux options

- How to do it…
  - Flannel networking configuration
  - Integrating with Docker
- How it works…
- See also
- Configuring master
  - Getting ready
  - How to do it…
    - Installation
      - CentOS 7 or Red Hat Enterprise Linux 7
      - Adding daemon dependency
      - Other Linux options
    - Verification
  - See also
- Configuring nodes
  - Getting ready
  - How to do it…
    - Installation
      - CentOS 7 or Red Hat Enterprise Linux 7
      - Other Linux options
    - Verification
  - See also
- Run your first container in Kubernetes
  - Getting ready
  - How to do it…
    - Running an HTTP server (nginx)
    - Exposing the port for external access
    - Stopping the application
  - How it works…
  - See also

2. Walking through Kubernetes Concepts
- Introduction
- An overview of Kubernetes control
  - Getting ready
  - How to do it…
  - How it works…
  - See also

- Working with pods
  - Getting ready
  - How to do it…
  - How it works…
  - See also
- Working with a replication controller
  - Getting ready
  - How to do it…
    - Creating a replication controller
    - Getting information of a replication controller
    - Changing the configuration of a replication controller
    - Removing a replication controller
  - How it works…
  - See also
- Working with services
  - Getting ready
  - How to do it…
    - Creating services for different resources
      - Creating a service for a pod
      - Creating a service for the replication controller and adding an external IP
      - Creating a no-selector service for an endpoint
      - Creating a service with session affinity based on another service
    - Creating a service in a different type
      - Creating a service in NodePort type
    - Deleting a service
  - How it works…
  - See also
- Working with volumes
  - Getting ready
  - How to do it…
    - emptyDir
    - hostPath
    - nfs
    - glusterfs
    - iscsi

flocker
rbd
gitRepo
awsElasticBlockStore
gcePersistentDisk
downwardAPI
There's more…
PersistentVolume
See also
Working with secrets
Getting ready
How to do it…
Creating a secret
Picking up secret in the container
Deleting a secret
How it works…
There's more…
See also
Working with names
Getting ready
How to do it…
How it works…
See also
Working with namespaces
Getting ready
How to do it…
Changing the default namespace
Deleting a namespace
There's more…
Deleting LimitRange
See also
Working with labels and selectors
Getting ready
How to do it…
Linking service with a replication controller by using label selectors
There's more…

- See also
3. Playing with Containers
  - Introduction
  - Scaling your containers
    - Getting ready
    - How to do it…
    - How it works…
    - See also
  - Updating live containers
    - Getting ready
    - How to do it…
    - How it works…
    - There's more…
    - See also
  - Forwarding container ports
    - Getting ready
    - How to do it…
      - Container-to-container communications
      - Pod-to-pod communications
      - Pod-to-service communications
      - External-to-internal communications
    - See also
  - Ensuring flexible usage of your containers
    - Getting ready
    - How to do it…
      - Pod as a job
        - Creating a job with multiple pods running
      - Pod as a daemon set
        - Running the daemon set only on specific nodes
    - How it works…
    - See also
  - Working with configuration files
    - Getting ready
      - YAML
      - JSON
    - How to do it…
    - How it works…

Pods
Replication controllers
Services
See also
4. Building a High Availability Cluster
Introduction
Clustering etcd
Getting ready
How to do it…
Static
etcd discovery
See also
Building multiple masters
Getting ready
How to do it…
Preparing multiple master nodes
Setting up kubelet in master
Getting the configuration files ready
Starting the kubelet service and turning daemons on!
How it works…
See also
5. Building a Continuous Delivery Pipeline
Introduction
Moving monolithic to microservices
Getting ready
How to do it…
Microservices
Frontend WebUI
How it works…
Microservices
Frontend WebUI
See also
Integrating with Jenkins
Getting ready
Install a Jenkins server which can build a Docker program
How to do it…
Create your Jenkins project

Run a program testing
Deploying a program
How it works…
There's more…
See also
Working with the private Docker registry
Getting ready
How to do it…
How it works…
Alternatives
Docker Trusted Registry
Nexus Repository Manager
Amazon EC2 Container Registry
See also
Setting up the Continuous Delivery pipeline
Getting ready
How to do it…
How it works…
There's more…
See also
6. Building Kubernetes on AWS
Introduction
Building the Kubernetes infrastructure in AWS
Getting ready
How to do it…
VPC and subnets
Internet Gateway and NAT
Route Table
Security group
How it works…
See also
Managing applications using AWS OpsWorks
Getting ready
How to do it…
The OpsWorks stack
The OpsWorks layer
Adjusting the IAM role

The OpsWorks instance
How it works…
See also
Auto-deploying Kubernetes through Chef recipes
Getting ready
Creating ELB and its security groups
Creating an OpsWorks stack
Creating application layers
How to do it…
Stack configuration for custom recipes
Recipes for etcd
Recipes for the Kubernetes master
Recipes for the Kubernetes node
Starting the instances
See also
Using AWS CloudFormation for fast provisioning
Getting ready
How to do it…
Creating a network infrastructure
Creating OpsWorks for application management
See also
7. Advanced Cluster Administration
Introduction
Advanced settings in kubeconfig
Getting ready
How to do it…
Setting a new credential
Setting a new cluster
Setting and changing the current context
Cleaning up kubeconfig
See also
Setting resource in nodes
Getting ready
How to do it…
Managing node capacity
Managing computing resources in a pod
See also

- Playing with WebUI
  - Getting ready
  - How to do it…
  - How it works…
  - See also
- Working with a RESTful API
  - Getting ready
  - How to do it…
  - How it works…
  - See also
- Authentication and authorization
  - Getting ready
  - How to do it…
    - Enabling authentication for an API call
      - Basic authentication of etcd
      - Basic authentication of the Kubernetes master
    - Making use of user authorization
  - See also

8. Logging and Monitoring
- Introduction
- Collecting application logs
  - Getting ready
    - Elasticsearch
  - How to do it…
    - Logstash
    - Startup script
    - Dockerfile
    - Docker build
    - Kubernetes replication controller and service
    - Kibana
  - How it works…
  - See also
- Working with Kubernetes logs
  - Getting ready
  - How to do it…
  - See also
- Working with etcd log

Getting ready
How to do it…
See also
Monitoring master and node
Getting ready
Updating Kubernetes to the latest version: 1.2.1
Setting up the DNS server
Start the server using templates
Enable Kubernetes DNS in kubelet
How to do it…
Installing a monitoring cluster
Introducing the Grafana dashboard
Creating a new metric to monitor pods
See also
Index

Kubernetes Cookbook

Kubernetes Cookbook

Copyright © 2016 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: June 2016

Production reference: 1270616

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78588-006-3

www.packtpub.com

Credits

Authors

Hideto Saito

Hui-Chuan Chloe Lee

Ke-Jou Carol Hsu

Reviewer

Matt Ma

Commissioning Editor

Kartikey Pandey

Acquisition Editor

Divya Poojari

Content Development Editor

Sachin Karnani

Technical Editor

Pranav Kukreti

Copy Editor

Akshata Lobo

Project Coordinator

Nikhil Nair

Proofreader

Safis Editing

Indexer

Hemangini Bari

Graphics

Jason Monteiro

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Authors

Hideto Saito has around 20 years of experience in the computer industry. In 1998, while working for Sun Microsystems Japan, he was impressed with Solaris OS, OPENSTEP, and Sun Ultra Enterprise 10000 (AKA StarFire). Then, he decided to pursue the UNIX and MacOS X operation systems.

In 2006, he relocated to Southern California as a software engineer to develop products and services running on Linux and MacOS X. He was especially renowned for his quick Objective-C code when he was drunk.

He is also an enthusiast of Japanese anime, drama, and motor sports, and loves Japanese Otaku culture.

> There were a lot of difficulties while writing this book. I believe it was the busiest moment in my life. But I was lucky to have two talented friends, Chloe and Carol, to support this project. I hope I have a chance to work with them again.
>
> Lastly, I appreciate my wife and children for their support. Their support and understanding brings me success and happiness.

Hui-Chuan Chloe Lee has worked in the software industry for over 5 years. She has a master's degree in CS from the National Taiwan University and is an AWS-certified associate solution architect. Chloe is also a technology enthusiast who has extended interest and experiences in different topics, such as application development, container technology, and Continuous Delivery.

In her free time, she enjoys reading, traveling, and spending time with the people she loves.

This book is dedicated to the people I love. I feel so lucky enough to have you all in my life. Without your support, this would never happen.

Especially, thanks to the other two amazing coauthors, Hideto and Carol, for your suggestions and guidance along the way.

Ke-Jou Carol Hsu is an engineer at Trend Micro. As a developer working in the Data Center Service group, Carol helps to write programs for deploying or managing internal-facing systems. She has both a bachelor's and a master's degree from the National Tsing Hua University. While studying and doing research, Carol focused on the area of high performance computing and virtualization technology. The experience made her more and more interested in system software, especially distributed systems and cloud environments.

Many thanks to my family and friends! You covered most of the house chores and job duties. Sometimes, you just bore my bad temper caused by the pressure while writing. Thanks to all of you! I am good to come back to my original life now!

For the other two authors, Hideto and Chloe, you are definitely the ones I truly appreciate. You guys know both the hard times and the happy hours during the writing of this book. Without your guide and support, it would have been impossible for me to finish this book at such a rapid pace and still come out with this careful, creative work. Looking forward to another cooperation in a short time.

About the Reviewer

Matt Ma is a multitalented and highly motivated full stack software engineer. He is a JavaScript ninja with over 8 years of experience. He has won over two dozen of CSS awards, Webby awards, and other web development awards. He is proud of being a Node.js contributor, a certified MongoDB developer, and an earlier adopter of Docker and Kubernetes.

Matt Ma have over 6 years of Linux experience. He is a long-time user of Ubuntu and CentOS. He uses an open source lightweight operating system, such as CoreOS, and systemd init daemon along with its eco-system tools. He advocates the microservices architecture.

When he is not working, he likes to go to the beach or a rock concert, hike, or spend time with his wife and two kids.

He likes meeting new people at conferences and meetups. You can find him on Twitter (`@bigmabig`) or GitHub (https://github.com/mattma). Drop him a line or just say hi to him.

www.PacktPub.com

eBooks, discount offers, and more

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 <customercare@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

Preface

Docker has been getting popular in recent years. It makes application deployment so efficient, we could easily build, ship, and run the application containers everywhere. With the trend of microservices, many people built a lot of services wrapped and deployed by containers, so container management and orchestration became a problem. Kubernetes solves this. However, building Kubernetes can be complex. Setting up Kubernetes nodes and control planes can be cumbersome. Furthermore, many people want leverage and integrate it with their own Continuous Delivery pipeline, but getting to know the whole story and making it work well can be time-consuming.

This is a practical guide that provides you step-by-step tips and examples to help you build and run your own Kubernetes cluster with the required components. This helpful guide will then lead you to understand how to deploy your application and services using the command line and a configuration file. You will also get a deep understanding of how to scale and update live containers and how to do port forwarding and network routing in Kubernetes. Next, you will learn how to build a robust high availability cluster with the book's hands-on examples. Finally, you will build and integrate the Continuous Delivery pipeline with Jenkins and Docker registry and learn how to build your own cluster in the cloud automatically. The book will also cover important topics about logging and monitoring.

What this book covers

Chapter 1, *Building Your Own Kubernetes*, explains how to build Kubernetes from scratch and run our very first container in it.

Chapter 2, *Walking through Kubernetes Concepts*, covers the basic and advance concepts we need to know before utilizing Kubernetes. Then, you will learn how to combine them to provide a suitable environment for running our applications.

Chapter 3, *Playing with Containers*, talks about how to scale your containers up and down and perform rolling update in order to provide better availability for your applications. Furthermore, you will learn how to run on-demand containers for handling different usages in the real world. It will also provide information on how to write a configuration file to make the deployment all together.

Chapter 4, *Building a High Availability Cluster*, will cover information on how to build High Availability Kubernetes master and etcd, which will act as an important datastore. This will prevent Kubernetes from becoming a single point of failure.

Chapter 5, *Building a Continuous Delivery Pipeline*, will talk about how to integrate Kubernetes with an existing Continuous Delivery pipeline, and the best application type for utilizing Kubernetes.

Chapter 6, *Building Kubernetes on AWS*, will show how to build Kubernetes step by step on AWS. You will also learn how to build it automatically from scratch.

Chapter 7, *Advanced Cluster Administration*, covers multicluster and resource management. You will learn how to use native WebUI and RESTful APIs to make audit and administration easier. It will include sections on setting up the authentication and authorization for your clusters.

Chapter 8, *Logging and Monitoring*, will explain how to collect Kubernetes, etcd, and even our application logs using ELK (Elasticsearch, Logstash, and Kibana). You will also learn how to integrate Heapster, influxDB, and Grafana to monitor your Kubernetes cluster.

What you need for this book

Throughout the book, we have used at least three servers with a Linux-based OS to build all the components in Kubernetes. You could use one system to install all of them at the beginning. As for the scalability point of view, we recommend you to start with three servers in order to scale out the components independently.

Who this book is for

If you've been playing with Docker containers for a while and wanted to orchestrate your containers in a modern way, this book is the right choice for you. This book is for those who already understand Docker and container technology and want to explore more for a better way to orchestrate, manage, and deploy containers. This book is perfect for going beyond a single container and working with container clusters, learning how to build your own Kubernetes, and making it work seamlessly with your Continuous Delivery pipeline.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

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: "On the Kubernetes master, we could use `kubectl run` to create a certain number of containers."

A block of code is set as follows:

```
[Unit]
Description=Etcd Server
After=network.target
```

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

```
$ cat /etc/etcd/etcd.conf
ETCD_NAME=myhappy-etcd
ETCD_DATA_DIR="/var/lib/etcd/myhappy.etcd"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:8080"
ETCD_ADVERTISE_CLIENT_URLS="http://localhost:8080"
```

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: "After you finish the project configurations, you can click on **Save** and then click on **Build Now** to check the result."

> ## Note
>
> Warnings or important notes appear in a box like this.

Tip

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 for this book from your account at http://www.packtpub.com. 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.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac

- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Kubernetes-Cookbook. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

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.

Chapter 1. Building Your Own Kubernetes

In this chapter, we will cover the following topics:

- Exploring architecture
- Preparing your environment
- Building datastore
- Creating an overlay network
- Configuring master
- Configuring nodes
- Running your first container in Kubernetes

Introduction

Welcome to the journey of Kubernetes! In this very first section, you will learn how to build your own Kubernetes cluster. Along with understanding each component and connecting them together, you will learn how to run your first container on Kubernetes. Holding a Kubernetes cluster will help you continue the study in the chapters ahead.

Exploring architecture

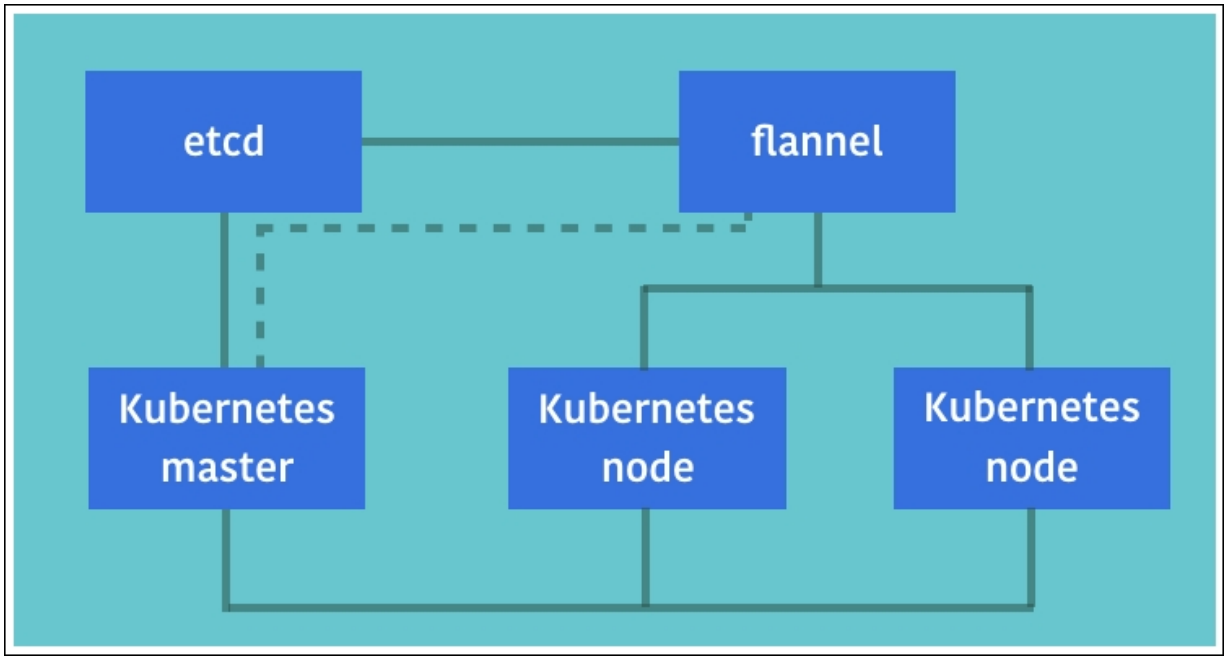
Kubernetes is an open source container management tool. It is a Go-Lang based (https://golang.org), lightweight, and portable application. You can set up a Kubernetes cluster on a Linux-based OS to deploy, manage, and scale the Docker container applications on multiple hosts.

Getting ready

Kubernetes is constructed using several components, as follows:

- Kubernetes master
- Kubernetes nodes
- etcd
- Overlay network (flannel)

These components are connected via network, as shown in the following screenshot:



The preceding image can be summarized as follows:

- **Kubernetes master** connects to **etcd** via HTTP or HTTPS to store the data. It also connects **flannel** to access the container application.
- Kubernetes nodes connect to the **Kubernetes master** via HTTP or HTTPS to get a command and report the status.
- Kubernetes nodes use an overlay network (for example, **flannel**) to make a connection of their container applications.

How to do it…

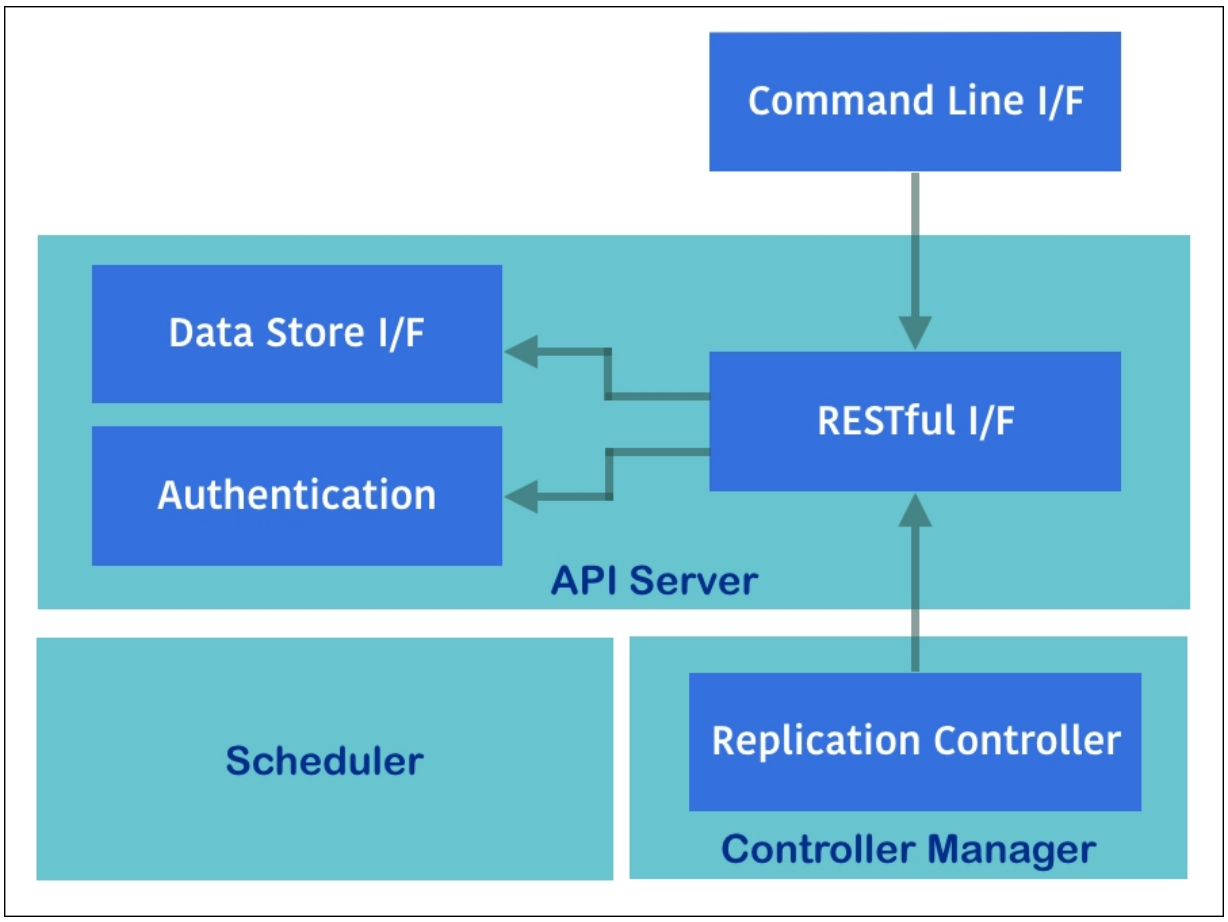
In this section, we are going to explain the features of Kubernetes master and nodes; both of them realize the main functions of the Kubernetes system.

Kubernetes master

Kubernetes master is the main component of Kubernetes cluster. It serves several functionalities, such as the following items:

- Authorization and authentication
- RESTful API entry point
- Container deployment scheduler to the Kubernetes nodes
- Scaling and replicating the controller
- Read and store the configuration
- Command Line Interface

The next image shows how master daemons worked together to fulfill the mentioned functionalities:



There are several daemon processes that make the Kubernetes master's functionality, such as **kube-apiserver**, **kube-scheduler**, and **kube-controller-manager**. Hypercube wrapper launched all of them.

In addition, the Kubernetes Command Line Interface kubectl can control the Kubernetes master functionality.

API server (kube-apiserver)

The API server provides an HTTP- or HTTPS-based RESTful API, which is the hub between Kubernetes components, such as kubectl, scheduler, replication controller, etcd datastore, and kubelet and kube-proxy, which runs on Kubernetes nodes and so on.

Scheduler (kube-scheduler)

Scheduler helps to choose which container runs by which nodes. It is a simple algorithm that defines the priority to dispatch and bind containers to nodes, for example:

- CPU
- Memory
- How many containers are running?

Controller manager (kube-controller-manager)

Controller manager performs cluster operations. For example:

- Manages Kubernetes nodes
- Creates and updates the Kubernetes internal information
- Attempts to change the current status to the desired status

Command Line Interface (kubectl)

After you install Kubernetes master, you can use the Kubernetes Command Line Interface kubectl to control the Kubernetes cluster. For example, `kubectl get cs` returns the status of each component. Also, `kubectl get nodes` returns a list of Kubernetes nodes:

```
//see the ComponentStatuses
# kubectl get cs
NAME                 STATUS    MESSAGE
ERROR
controller-manager   Healthy   ok
nil
scheduler            Healthy   ok
nil
etcd-0               Healthy   {"health": "true"}
nil

//see the nodes
# kubectl get nodes
NAME          LABELS
STATUS    AGE
kub-node1   kubernetes.io/hostname=kub-node1
Ready     26d
kub-node2   kubernetes.io/hostname=kub-node2
Ready     26d
```

Kubernetes node

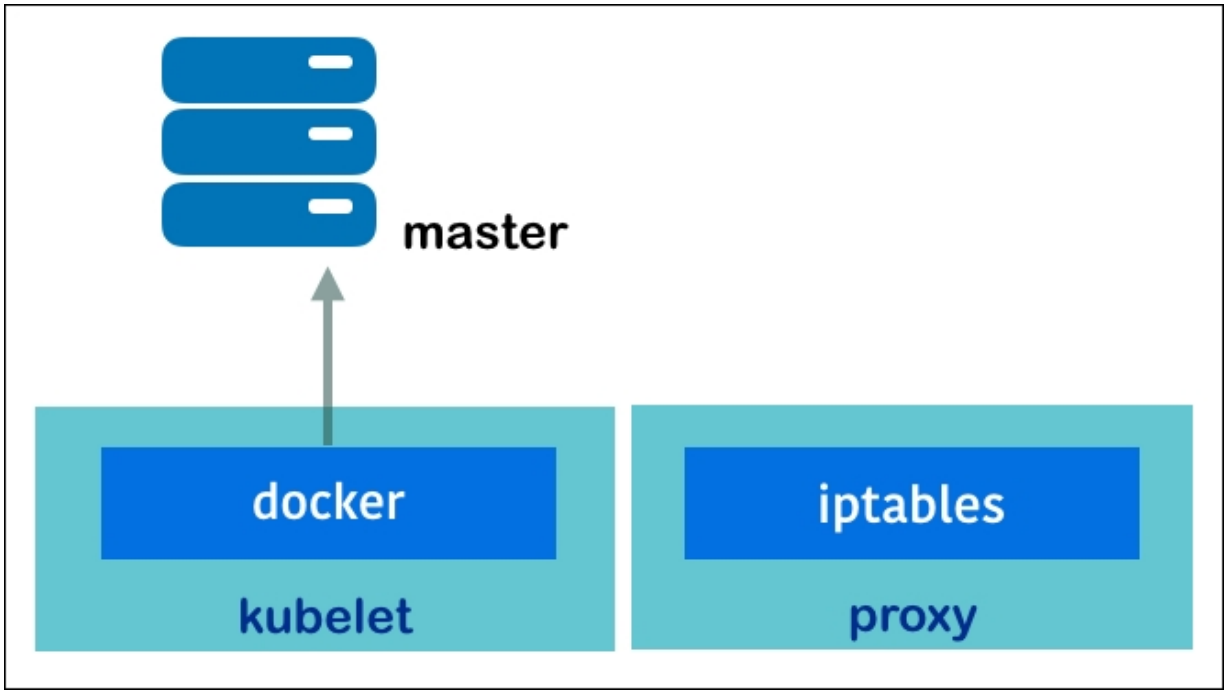
Kubernetes node is a slave node in the Kubernetes cluster. It is controlled by Kubernetes master to run the container application using Docker (http://docker.com) or rkt (http://coreos.com/rkt/docs/latest/) in this book; we will use the Docker container runtime as the default engine.

Tip

Node or slave?

The terminology of slave is used in the computer industry to represent the cluster worker node; however, it is also associated with discrimination. The Kubernetes project uses node instead.

The following image displays the role and tasks of daemon processes in node:



Node also has multiple daemon processes, named kubelet and kube-proxy, to support its functionalities.

kubelet

kubelet is the main process on Kubernetes node that communicates with Kubernetes master to handle the following operations:

- Periodically access the API Controller to check and report
- Perform container operations
- Runs the HTTP server to provide simple APIs

Proxy (kube-proxy)

Proxy handles the network proxy and load balancer for each container. It performs to change the Linux iptables rules (nat table) to control TCP and UDP packets across the containers.

After starting the `kube-proxy` daemon, it will configure iptables rules; you can see `iptables -t nat -L` or `iptables -t nat -S` to check the nat table rules, as follows:

```
 //the result will be vary and dynamically changed
by kube-proxy
# sudo iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER
-N FLANNEL
-N KUBE-NODEPORT-CONTAINER
-N KUBE-NODEPORT-HOST
-N KUBE-PORTALS-CONTAINER
-N KUBE-PORTALS-HOST
-A PREROUTING -m comment --comment "handle
ClusterIPs; NOTE: this must be before the NodePort
rules" -j KUBE-PORTALS-CONTAINER
-A PREROUTING -m addrtype --dst-type LOCAL -m
comment --comment "handle service NodePorts; NOTE:
this must be the last rule in the chain" -j KUBE-
```

```
NODEPORT-CONTAINER
-A PREROUTING -m addrtype --dst-type LOCAL -j
DOCKER
-A OUTPUT -m comment --comment "handle ClusterIPs;
NOTE: this must be before the NodePort rules" -j
KUBE-PORTALS-HOST
-A OUTPUT -m addrtype --dst-type LOCAL -m comment
--comment "handle service NodePorts; NOTE: this
must be the last rule in the chain" -j KUBE-
NODEPORT-HOST
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type
LOCAL -j DOCKER
-A POSTROUTING -s 192.168.90.0/24 ! -o docker0 -j
MASQUERADE
-A POSTROUTING -s 192.168.0.0/16 -j FLANNEL
-A FLANNEL -d 192.168.0.0/16 -j ACCEPT
-A FLANNEL ! -d 224.0.0.0/4 -j MASQUERADE
```

How it works…

There are two more components to complement the Kubernetes nodes' functionalities, the datastore etcd and the overlay network flannel. You can learn how they support the Kubernetes system in the following paragraphs.

etcd

The etcd (https://coreos.com/etcd/) is the distributed key-value datastore. It can be accessed via the RESTful API to perform the CRUD operation over the network. Kubernetes uses etcd as the main datastore.

You can explore the Kubernetes configuration and status in etcd (`/registry`) using the `curl` command as follows:

```
//example: etcd server is 10.0.0.1 and default
port is 2379
# curl -L "http://10.0.0.1:2379/v2/keys/registry"

{"action":"get","node":
{"key":"/registry","dir":true,"nodes":
```

```
[{"key":"/registry/namespaces","dir":true,"modifie
dIndex":15,"createdIndex":15},
{"key":"/registry/serviceaccounts","dir":true,"mod
ifiedIndex":16,"createdIndex":16},
{"key":"/registry/services","dir":true,"modifiedIn
dex":17,"createdIndex":17},
{"key":"/registry/ranges","dir":true,"modifiedInde
x":76,"createdIndex":76},
{"key":"/registry/nodes","dir":true,"modifiedIndex
":740,"createdIndex":740},
{"key":"/registry/pods","dir":true,"modifiedIndex"
:794,"createdIndex":794},
{"key":"/registry/controllers","dir":true,"modifie
dIndex":810,"createdIndex":810},
{"key":"/registry/events","dir":true,"modifiedInde
x":6,"createdIndex":6}],"modifiedIndex":6,"created
Index":6}}
```


Overlay network

Network communication between containers is the most difficult part. Because when you start to run the Docker, an IP address will be assigned dynamically; the container application needs to know the peer's IP address and port number.

If the container's network communication is only within the single host, you can use the Docker link to generate the environment variable to discover the peer. However, Kubernetes usually works as a cluster and ambassador pattern or overlay network could help to connect every node. Kubernetes uses overlay network to manage multiple containers' communication.

For overlay network, Kubernetes has several options, but using flannel is the easier solution.

Flannel

Flannel also uses etcd to configure the settings and store the status. You can also perform the `curl` command to explore the configuration (`/coreos.com/network`) and status, as follows:

```
//overlay network CIDR is 192.168.0.0/16
# curl -L
"http://10.0.0.1:2379/v2/keys/coreos.com/network/c
onfig"

{"action":"get","node":
{"key":"/coreos.com/network/config","value":"{
\"Network\": \"192.168.0.0/16\"
}","modifiedIndex":144913,"createdIndex":144913}}

//Kubernetes assigns some subnets to containers
# curl -L
"http://10.0.0.1:2379/v2/keys/coreos.com/network/s
ubnets"

{"action":"get","node":
{"key":"/coreos.com/network/subnets","dir":true,"n
odes":
[{"key":"/coreos.com/network/subnets/192.168.90.0-
24","value":"
{\"PublicIP\":\"10.97.217.158\"}","expiration":"20
15-11-
05T08:16:21.995749971Z","ttl":38993,"modifiedIndex
":388599,"createdIndex":388599},
{"key":"/coreos.com/network/subnets/192.168.76.0-
24","value":"
{\"PublicIP\":\"10.97.217.148\"}","expiration":"20
15-11-
05T04:32:45.528111606Z","ttl":25576,"modifiedIndex
":385909,"createdIndex":385909},
{"key":"/coreos.com/network/subnets/192.168.40.0-
24","value":"
{\"PublicIP\":\"10.97.217.51\"}","expiration":"201
5-11-
05T15:18:27.335916533Z","ttl":64318,"modifiedIndex
":393675,"createdIndex":393675}],"modifiedIndex":7
9,"createdIndex":79}}
```

See also

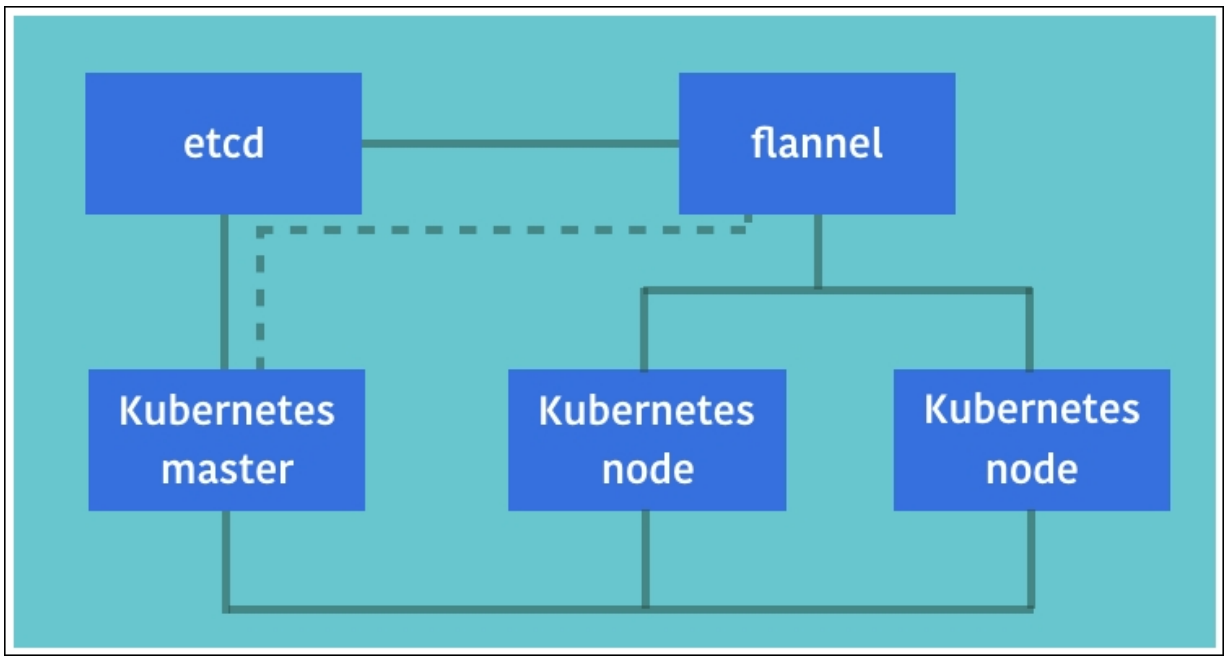
This section describes the basic architecture and methodology of Kubernetes and related components. Understanding Kubernetes is not easy, but a step-by-step lesson on how to setup, configure, and manage Kubernetes is really fun.

The following recipes describe how to install and configure related components:

- *Building datastore*
- *Creating an overlay network*
- *Configuring master*
- *Configuring nodes*

Preparing your environment

Before heading to the journey of building our own cluster, we have to prepare the environment in order to build the following components:



There are different solutions of creating such a Kubernetes cluster, for example:

- Local-machine solutions that include:
    - Docker-based
    - Vagrant
    - Linux machine
- Hosted solution that includes:
    - Google Container Engine
- Custom solutions

A local-machine solution is suitable if we just want to build a development environment or do the proof of concept quickly. By

using **Docker** (https://www.docker.com) or **Vagrant** (https://www.vagrantup.com), we could easily build the desired environment in one single machine; however, it is not practical if we want to build a production environment. A hosted solution is the easiest starting point if we want to build it in the cloud.

Google Container Engine, which has been used by Google for many years, has the comprehensive support naturally and we do not need to care much about the installation and setting. Kubernetes can also run on different cloud and on-premises VMs by custom solutions. We will build the Kubernetes clusters from scratch on Linux-based virtual machines (CentOS 7.1) in the following chapters. The solution is suitable for any Linux machines in both cloud and on-premises environments.

Getting ready

It is recommended if you have at least four Linux servers for master, etcd, and two nodes. If you want to build it as a high availability cluster, more servers for each component are preferred. We will build three types of servers in the following sections:

- Kubernetes master
- Kubernetes node
- etcd

Flannel will not be located in one machine, which is required in all the nodes. Communication between containers and services are powered by flannel, which is an etcd backend overlay network for containers.

Hardware resource

The hardware spec of each component is suggested in the following table. Please note that it might cause a longer response time when manipulating the cluster if the amount of requests between the API

server and etcd is large. In a normal situation, increasing resources can resolve this problem:

Component Spec	**Kubernetes master**	**etcd**
CPU Count	1	1
Memory GB	2G	2G

For the nodes, the default maximum number of pods in one node is 40. However, a node capacity is configurable when adding a node. You have to measure how many resources you might need for the hosted services and applications to decide how many nodes should be there with a certain spec and with proper monitoring in production workload.

Tip

Check out your node capacity in node

In your master, you could install `jq` by `yum install jq` and use `kubectl get nodes -o json | jq '.items[] | {name: .metadata.name, capacity: .status.capacity}'` to check the capacity of each node, including CPU, memory, and the maximum capacity of pods:

```
// check out your node capacity
$ kubectl get nodes -o json | jq '.items[] |
{name: .metadata.name, capacity:
.status.capacity}'
{
```

```
      "name": "kub-node1",
      "capacity": {
        "cpu": "1",
        "memory": "1021536Ki",
        "pods": "40"
      }
    }
    {
      "name": "kub-node2",
      "capacity": {
        "cpu": "1",
        "memory": "1021536Ki",
        "pods": "40"
      }
    }
```

Operating system

The OS of nodes could be various, but the kernel version must be 3.10 or later. Following are the OSs that are using kernel 3.10+:

- CentOS 7 or later
- RHEL 7 or later
- Ubuntu Vivid 15.04 / Ubuntu Trusty 14.04 (LTS) / Ubuntu Saucy 13.10

Note

Beware of the Linux kernel version

Docker requires that your kernel must be 3.10 at minimum on CentOS or Red Hat Enterprise Linux, 3.13 kernel version on Ubuntu Precise 12.04 (LTS). It will cause data loss or kernel panic sometimes if using unsupported kernels. It is recommended you fully update the system before building Kubernetes. You can use `uname -r` to check the kernel you're currently using. For more information on checking the kernel

version, please refer to http://www.linfo.org/find_kernel_version.html.

How to do it…

To ensure each component works perfectly in Kubernetes cluster, we must install the correct packages on each machine of master, node, and etcd.

Kubernetes master

Kubernetes master should be installed on a Linux-based OS. For the examples listed in this book, we will use CentOS 7.1 as an OS. There are two packages required in master:

- Kubernetes
- Flannel (optional)
- iptables (at least 1.4.11+ is preferred)

Kubernetes (https://github.com/kubernetes/kubernetes/releases) has a couple of fast-paced releases. Flannel daemon is optional in master; if you would like to launch Kubernetes UI, flannel (https://github.com/coreos/flannel/releases) is required. Otherwise, Kubernetes UI will be failed to access via https://<kubernetes-master>/ui.

Note

Beware of iptables version

Kubernetes uses iptables to implement service proxy. iptables with version 1.4.11+ is recommended on Kubernetes. Otherwise, iptables rules might be out of control and keep increasing. You can use `yum info iptables` to check the current version of iptables.

Kubernetes nodes

On Kubernetes nodes, we have to prepare the following:

- Kubernetes
- Flannel daemon
- Docker (at least 1.6.2+ is preferred)
- iptables (at least 1.4.11+ is preferred)

Note

Beware of Docker version and dependencies

Sometimes, you'll get an unknown error when using the incompatible Docker version, such as target image is not found. You can always use the `docker version` command to check the current version you've installed. The recommended versions we tested are at least 1.7.1+. Before building the cluster, you can start the service by using the `service docker start` command and make sure it can be contacted using `docker ps`.

Docker has different package names and dependency packages in Linux distributions. In Ubuntu, you could use `curl -sSL https://get.docker.com/ | sh`. For more information, check out the Docker installation document (http://docs.docker.com/v1.8/installation) to find your preferred Linux OS.

etcd

etcd, which is a distributed reliable key-value store for shared configurations and service discovery, is powered by CoreOS. The release page is https://github.com/coreos/etcd/releases. The prerequisite we need is just the etcd package.

See also

After preparing the environment, it is time to build up your Kubernetes. Check out the following recipes for that:

- *Building datastore*
- *Creating an overlay network*
- *Configuring master*
- *Configuring nodes*
- The *Setting resource in nodes* recipe in Chapter 7, *Advanced Cluster Administration*
- The *Monitoring master and node* recipe in Chapter 8, *Logging and Monitoring*

Building datastore

In order to persist the Kubernetes cluster information, we need to set up datastore. Kubernetes uses etcd as a standard datastore. This section will guide you to build the etcd server.

How to do it…

The etcd database requires Linux OS; some Linux distributions provide the etcd package and some don't. This section describes how to install etcd.

Red Hat Enterprise Linux 7 or CentOS 7

Red Hat Enterprise Linux (RHEL) 7, CentOS 7 or later has an official package for etcd. You can install via the `yum` command, as follows:

```
//it will perform to install etcd package on RHEL/CentOS
Linux
sudo yum update -y
sudo yum install etcd
```

Ubuntu Linux 15.10 Wily Werewolf

Ubuntu 15.10 or later has an official package for etcd as well. You can install via the `apt-get` command as follows:

```
//it will perform to install etcd package on Ubuntu Linux
sudo apt-get update -y
sudo apt-get install etcd
```

Other Linux

If you are using a different Linux version, such as Amazon Linux, you can download a binary from the official website and install it as follows.

Download a binary

etcd is provided via `https://github.com/coreos/etcd/releases`. OS X (darwin-amd64), Linux, Windows binary, and source code are available for download.

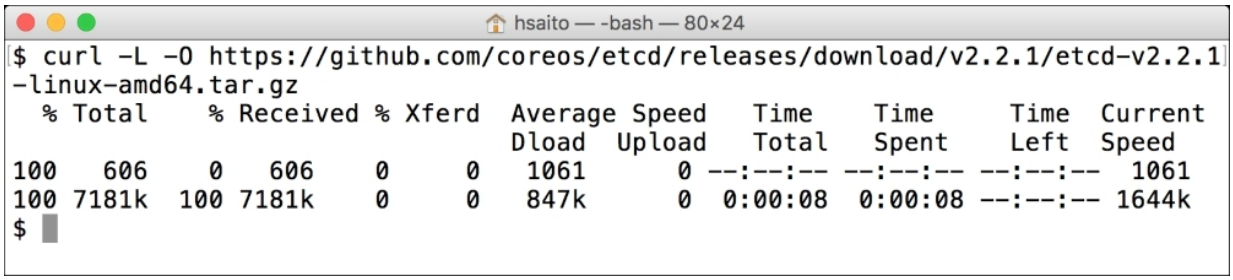
Tip

Note that there are no 32-bit binaries provided due to the Go runtime issue. You must prepare a 64-bit Linux OS.



On your Linux machine, use the `curl` command to download the `etcd-v2.2.1-linux-amd64.tar.gz` binary:

```
// follow redirection(-L) and use remote name (-O)
curl -L -O
https://github.com/coreos/etcd/releases/download/v2.2.1/etcd-
v2.2.1-linux-amd64.tar.gz
```



Creating a user

Due to security reasons, create a local user and group that can own etcd packages:

1. Run the following `useradd` command:

```
//options
//    create group(-U), home directory(-d), and create
it(-m)
//    name in GCOS field (-c), login shell(-s)
$ sudo useradd -U -d /var/lib/etcd -m -c "etcd user" -s
/sbin/nologin etcd
```

2. You can check `/etc/passwd` to see whether creating `etcd user` has created a user or not:

```
//search etcd user on /etc/passwd, uid and gid is vary
$ grep etcd /etc/passwd
etcd:x:997:995:etcd user:/var/lib/etcd:/sbin/nologin
```

> **Tip**
>
> You can delete a user any time; type `sudo userdel -r etcd` to delete `etcd user`.

Install etcd

1. After downloading an etcd binary, use the `tar` command to extract files:

```
$ tar xf etcd-v2.2.1-linux-amd64.tar.gz
$ cd etcd-v2.2.1-linux-amd64

//use ls command to see that there are documentation and
binaries
$ ls
Documentation  README-etcdctl.md  README.md  etcd
etcdctl
```

2. There are `etcd` daemon and `etcdctl` command that need to be copied to `/usr/local/bin`. Also, create `/etc/etcd/etcd.conf` as a setting file:

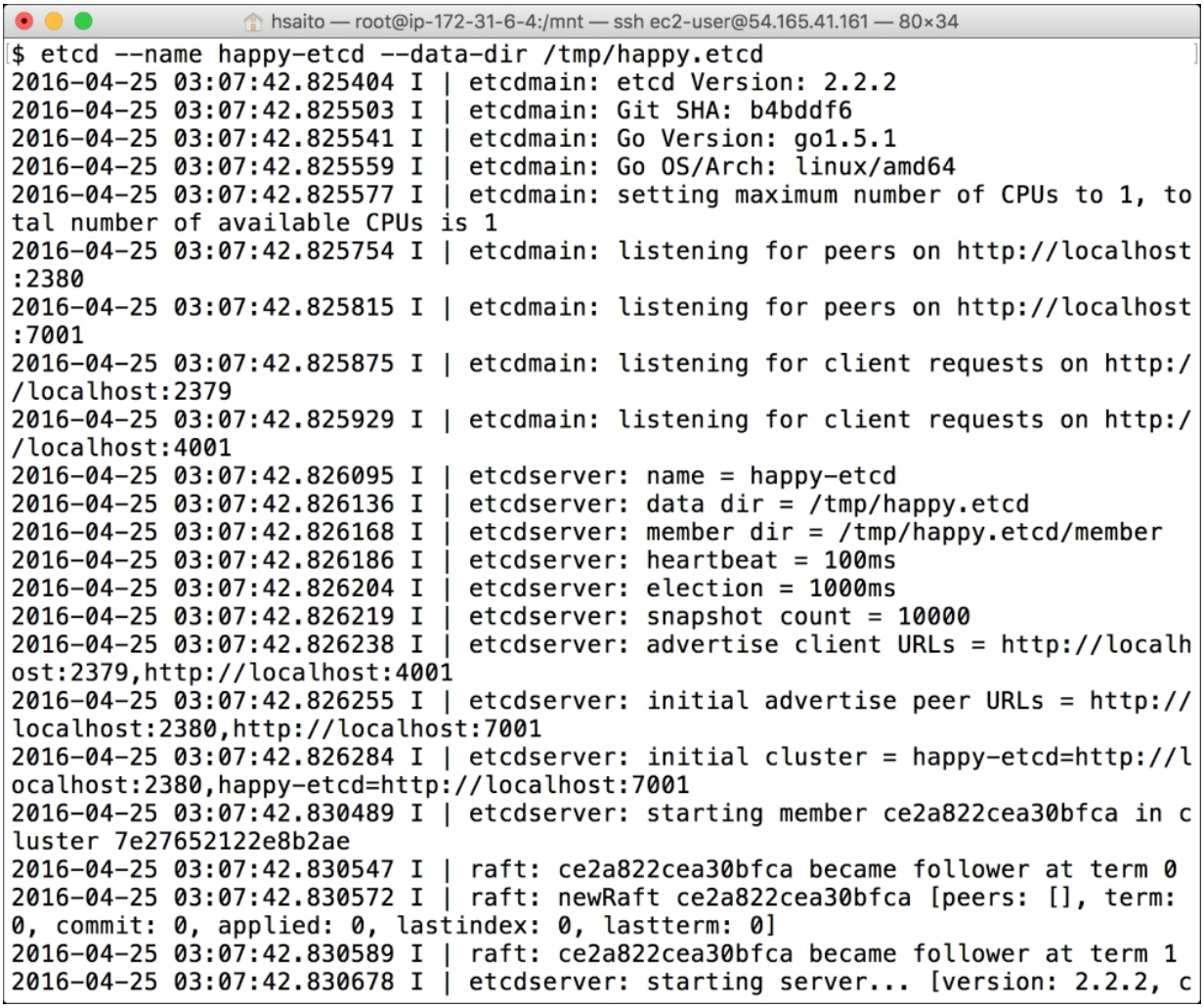
```
$ sudo cp etcd etcdctl /usr/local/bin/

//create etcd.conf
$ sudo mkdir -p /etc/etcd/
$ sudo touch /etc/etcd/etcd.conf
$ sudo chown -R etcd:etcd /etc/etcd
```

How it works…

Let's test run the `etcd` daemon to explorer the etcd functionalities. Type the `etcd` command with the `name` and `data-dir` argument as follows:

```
//for the testing purpose, create data file under /tmp
$ etcd --name happy-etcd --data-dir /tmp/happy.etcd &
```

Then, you will see several output logs as follows:



Now, you can try to use the `etcdctl` command to access etcd and to load and store the data as follows:

```
//set value "hello world" to the key /my/happy/data
$ etcdctl set /my/happy/data "hello world"
```

```
//get value for key /my/happy/data
$ etcdctl get /my/happy/data
hello world
```

In addition, by default, etcd opens TCP port `2379` to access the RESTful API, so you may also try to use an HTTP client, such as the `curl` command to access data as follows:

```
//get value for key /my/happy/data using cURL
$ curl -L http://localhost:2379/v2/keys/my/happy/data
{"action":"get","node":{"key":"/my/happy/data","value":"hello
world","modifiedIndex":4,"createdIndex":4}}

//set value "My Happy world" to the key /my/happy/data using
cURL
$ curl http://127.0.0.1:2379/v2/keys/my/happy/data -XPUT -d
value="My Happy world"

//get value for key /my/happy/data using etcdctl
$ etcdctl get /my/happy/data
My Happy world
```

Okay! Now, you can delete the key using the `curl` command as follows:

```
$ curl http://127.0.0.1:2379/v2/keys/my?recursive=true -
XDELETE

//no more data returned afterword
$ curl http://127.0.0.1:2379/v2/keys/my/happy/data
{"errorCode":100,"message":"Key not
found","cause":"/my","index":10}

$ curl http://127.0.0.1:2379/v2/keys/my/happy
{"errorCode":100,"message":"Key not
found","cause":"/my","index":10}

$ curl http://127.0.0.1:2379/v2/keys/my
{"errorCode":100,"message":"Key not
found","cause":"/my","index":10}
```

Auto startup script

Based on your Linux, either systemd or init, there are different ways to make an auto startup script.

If you are not sure, check the process ID `1` on your system. Type `ps -P 1` to see the process name as follows:

```
//This Linux is systemd based
$ ps -P 1
  PID PSR TTY      STAT   TIME COMMAND
    1   0 ?        Ss     0:03 /usr/lib/systemd/systemd --
switched-root -system


//This Linux is init based
# ps -P 1
  PID PSR TTY      STAT   TIME COMMAND
    1   0 ?        Ss     0:01 /sbin/init
```

Startup script (systemd)

If you are using systemd-based Linux, such as RHEL 7, CentOS 7, Ubuntu 15.4 or later, you need to prepare the `/usr/lib/systemd/system/etcd.service` file as follows:

```
[Unit]
Description=Etcd Server
After=network.target

[Service]
Type=simple
WorkingDirectory=/var/lib/etcd/
EnvironmentFile=/etc/etcd/etcd.conf
User=etcd
ExecStart=/usr/local/bin/etcd

[Install]
WantedBy=multi-user.target
```

After that, register to `systemd` using the `systemctl` command as follows:

```
# sudo systemctl enable etcd
```

Then, you restart the system or type `sudo systemctl start etcd` to launch the `etcd` daemon. You may check the etcd service status using `sudo systemctl status -l etcd`.

Startup script (init)

If you are using the init-based Linux, such as Amazon Linux, use the traditional way to prepare the `/etc/init.d/etcd` script as follows:

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

```
# etcd This shell script takes care of starting and stopping
etcd
#
# chkconfig: - 60 74
# description: etcd

### BEGIN INIT INFO
# Provides: etcd
# Required-Start: $network $local_fs $remote_fs
# Required-Stop: $network $local_fs $remote_fs
# Should-Start: $syslog $named ntpdate
# Should-Stop: $syslog $named
# Short-Description: start and stop etcd
# Description: etcd
### END INIT INFO

# Source function library.
. /etc/init.d/functions

# Source networking configuration.
. /etc/sysconfig/network

prog=/usr/local/bin/etcd
etcd_conf=/etc/etcd/etcd.conf
lockfile=/var/lock/subsys/`basename $prog`
hostname=`hostname`

start() {
  # Start daemon.
. $etcd_conf
  echo -n $"Starting $prog: "
  daemon --user=etcd $prog > /var/log/etcd.log 2>&1 &
  RETVAL=$?
  echo
  [ $RETVAL -eq 0 ] && touch $lockfile
  return $RETVAL
}
stop() {
  [ "$EUID" != "0" ] && exit 4
        echo -n $"Shutting down $prog: "
  killproc $prog
  RETVAL=$?
        echo
  [ $RETVAL -eq 0 ] && rm -f $lockfile
  return $RETVAL
}

# See how we were called.
case "$1" in
  start)
  start
  ;;
  stop)
  stop
  ;;
  status)
```

```
    status $prog
    ;;
    restart)
    stop
    start
    ;;
    reload)
    exit 3
    ;;
    *)
    echo $"Usage: $0 {start|stop|status|restart|reload}"
    exit 2
esac
```

After that, register to init script using the `chkconfig` command as follows:

```
//set file permission correctly
$ sudo chmod 755 /etc/init.d/etcd
$ sudo chown root:root /etc/init.d/etcd

//auto start when boot Linux
$ sudo chkconfig --add etcd
$ sudo chkconfig etcd on
```

Then, you restart the system or type `/etc/init.d/etcd start` to launch the `etcd` daemon.

Configuration

There is the file `/etc/etcd/etcd.conf` to change the configuration of etcd, such as data file path and TCP port number.

The minimal configuration is as follows:

NAME	Mean	Example	Note
`ETCD_NAME`	Instance name	`myhappy-etcd`	
`ETCD_DATA_DIR`	Data file path	`/var/lib/etcd/myhappy.etcd`	File path must be owned by etcd user

NAME	Mean	Example	Note
`ETCD_LISTEN_CLIENT_URLS`	TCP port number	`http://0.0.0.0:8080`	Specifying `0.0.0.0`, binds all IP address, otherwise use localhost to accept only same machine
`ETCD_ADVERTISE_CLIENT_URLS`	Advertise this etcd URL to other cluster instances	`http://localhost:8080`	Use for clustering configuration

Note that you need to use the `export` directive if you want to use the init-based Linux in order to set environment variables as follows:

```
$ cat /etc/etcd/etcd.conf

export ETCD_NAME=myhappy-etcd
export ETCD_DATA_DIR="/var/lib/etcd/myhappy.etcd"
export ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:8080"
export ETCD_ADVERTISE_CLIENT_URLS="http://localhost:8080"
```

On the other hand, systemd-based Linux doesn't need the export directive as follows:

```
$ cat /etc/etcd/etcd.conf
ETCD_NAME=myhappy-etcd
ETCD_DATA_DIR="/var/lib/etcd/myhappy.etcd"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:8080"
ETCD_ADVERTISE_CLIENT_URLS="http://localhost:8080"
```

See also

This section described how to configure etcd. It is easy and simple to operate via the RESTful API, but powerful. However, there's a need to be aware of its security and availability. The following recipes will describe how to ensure that etcd is secure and robust:

- *Exploring architecture*
- The *Clustering etcd* recipe in Chapter 4, *Building a High Availability Cluster*
- The *Authentication and authorization* recipe in Chapter 7, *Advanced Cluster Administration*
- The *Working with etcd log* recipe in Chapter 8, *Logging and Monitoring*

Creating an overlay network

Kubernetes abstracts the networking to enable communication between containers across nodes. The basic unit to make it possible is named pod, which is the smallest deployment unit in Kubernetes with a shared context in a containerized environment. Containers within a pod can communicate with others by port with the localhost. Kubernetes will deploy the pods across the nodes.

Then, how do pods talk to each other?

Kubernetes allocates each pod an IP address in a shared networking namespace so that pods can communicate with other pods across the network. There are a couple of ways to achieve the implementation. The easiest and across the platform way will be using flannel.

Flannel gives each host an IP subnet, which can be accepted by Docker and allocate the IPs to individual containers. Flannel uses etcd to store the IP mapping information, and has a couple of backend choices for forwarding the packets. The easiest backend choice would be using TUN device to encapsulate IP fragment in a UDP packet. The port is `8285` by default.

Flannel also supports in-kernel VXLAN as backend to encapsulate the packets. It might provide better performance than UDP backend while it is not running in user space. Another popular choice is using the advanced routing rule upon Google Cloud Engine (https://cloud.google.com/compute/docs/networking#routing). We'll use both UDP and VXLAN as examples in this section.

Flanneld is the agent of flannel used to watch the information from etcd, allocate the subnet lease on each host, and route the packets. What we will do in this section is let flanneld be up and running and allocate a subnet for each host.

> **Note**
>
> If you're struggling to find out which backend should be used, here is a simple performance test between UDP and VXLAN. We use qperf (http://linux.die.net/man/1/qperf) to measure packet transfer performance between containers. TCP streaming one way bandwidth through UDP is 0.3x slower than VXLAN when there are some loads on the hosts. If you prefer building Kubernetes on the cloud, GCP is the easiest choice.

Getting ready

Before installing flannel, be sure you have the etcd endpoint. Flannel needs etcd as its datastore. If Docker is running, stop the Docker service first and delete `docker0`, which is a virtual bridge created by Docker:

```
# Stop docker service
$ service docker stop

# delete docker0
$ ip link delete docker0
```

Installation

Using the `etcdctl` command we learned in the previous section on the etcd instance, insert the desired configuration into etcd with the key `/coreos.com/network/config`:

Configuration Key	Description
`Network`	IPv4 network for flannel to allocate to entire virtual network
`SubnetLen`	The subnet prefix length to each host, default is `24`.
`SubnetMin`	The beginning of IP range for flannel subnet allocation
`SubnetMax`	The end of IP range for flannel subnet allocation
`Backend`	Backend choices for forwarding the packets. Default is `udp`.

```
# insert desired CIDR for the overlay network Flannel creates
$ etcdctl set /coreos.com/network/config '{ "Network":
"192.168.0.0/16" }'
```

Flannel will assign the IP address within `192.168.0.0/16` for overlay network with `/24` for each host by default, but you could also overwrite its default setting and insert into etcd:

```
$ cat flannel-config-udp.json
{
    "Network": "192.168.0.0/16",
    "SubnetLen": 28,
    "SubnetMin": "192.168.10.0",
    "SubnetMax": "192.168.99.0",
    "Backend": {
        "Type": "udp",
        "Port": 7890
    }
}
```

Use the `etcdctl` command to insert the `flannel-config-udp.json` configuration:

```
# insert the key by json file
$ etcdctl set /coreos.com/network/config < flannel-config-
udp.json
```

Then, flannel will allocate to each host with `/28` subnet and only issue the subnets within `192.168.10.0` and `192.168.99.0`. Backend will be still `udp` and the default port will be changed from `8285` to `7890`.

We could also use VXLAN to encapsulate the packets and use `etcdctl` to insert the configuration:

```
$ cat flannel-config-vxlan.json
{
    "Network": "192.168.0.0/16",
    "SubnetLen": 24,
    "Backend": {
        "Type": "vxlan",
        "VNI": 1
    }
}

# insert the key by json file
$ etcdctl set /coreos.com/network/config < flannel-config-
vxlan.json
```

You might be able to see the configuration you get using `etcdctl`:

```
$ etcdctl get /coreos.com/network/config
{
    "Network": "192.168.0.0/16",
    "SubnetLen": 24,
    "Backend": {
        "Type": "vxlan",
        "VNI": 1
```

```
        }
    }
```

CentOS 7 or Red Hat Enterprise Linux 7

RHEL 7, CentOS 7, or later have an official package for flannel. You can install them via the `yum` command:

```
# install flannel package
$ sudo yum install flannel
```

After the installation, we have to configure the etcd server in order to use the flannel service:

```
$ cat /etc/sysconfig/flanneld

# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="<your etcd server>"

# etcd config key.  This is the configuration key that flannel
queries
# For address range assignment
FLANNEL_ETCD_KEY="/coreos.com/network"

# Any additional options that you want to pass
#FLANNEL_OPTIONS=""
```

We should always keep `flanneld` up and running all the time when we boot up the server. Using `systemctl` could do the trick:

```
# Enable flanneld service by default
$ sudo systemctl enable flanneld

# start flanneld
$ sudo service flanneld start

# check if the service is running
$ sudo service flannel status
```

Other Linux options

You can always download a binary as an alternative. The CoreOS flannel official release page is here: https://github.com/coreos/flannel/releases. Choose the packages with the **Latest release** tag; it will always include the latest bug fixes:

```
# download flannel package
$ curl -L -O
```

```
https://github.com/coreos/flannel/releases/download/v0.5.5/flanne
l-0.5.5-linux-amd64.tar.gz

# extract the package
$ tar zxvf flannel-0.5.5-linux-amd64.tar.gz

# copy flanneld to $PATH
$ sudo cp flannel-0.5.5/flanneld /usr/local/bin
```

If you use a startup script (`systemd`) in the etcd section, you might probably choose the same way to describe `flanneld`:

```
$ cat /usr/lib/systemd/system/flanneld.service
[Unit]
Description=Flanneld overlay address etcd agent
Wants=etcd.service
After=etcd.service
Before=docker.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/flanneld
EnvironmentFile=-/etc/sysconfig/docker-network
ExecStart=/usr/bin/flanneld -etcd-endpoints=${FLANNEL_ETCD} -
etcd-prefix=${FLANNEL_ETCD_KEY} $FLANNEL_OPTIONS
Restart=on-failure

RestartSec=5s

[Install]
WantedBy=multi-user.target
```

Then, enable the service on bootup using `sudo systemctl enable flanneld`.

Alternatively, you could use a startup script (init) under `/etc/init.d/flanneld` if you're using an init-based Linux:

```
#!/bin/bash

# flanneld  This shell script takes care of starting and stopping
flanneld
#

# Source function library.
. /etc/init.d/functions

# Source networking configuration.
. /etc/sysconfig/network

prog=/usr/local/bin/flanneld
lockfile=/var/lock/subsys/`basename $prog`
```

After you have sourced and set the variables, you should implement start, stop status, and restart for the service. The only thing you need to take care of is to ensure to add the `etcd` endpoint into the configuration when the daemon starts:

```
start() {
  # Start daemon.
  echo -n $"Starting $prog: "
  daemon $prog \
    --etcd-endpoints=="<your etcd server>" \
    -ip-masq=true \
    > /var/log/flanneld.log 2>&1 &
  RETVAL=$?
  echo
  [ $RETVAL -eq 0 ] && touch $lockfile
  return $RETVAL
}

stop() {
  [ "$EUID" != "0" ] && exit 4
        echo -n $"Shutting down $prog: "
  killproc $prog
  RETVAL=$?
        echo
  [ $RETVAL -eq 0 ] && rm -f $lockfile
  return $RETVAL
}

case "$1" in
  start)
  start
  ;;
  stop)
  stop
  ;;
  status)
  status $prog
  ;;
  restart|force-reload)
  stop
  start
  ;;
  try-restart|condrestart)
  if status $prog > /dev/null; then
      stop
      start
  fi
  ;;
  reload)
  exit 3
  ;;
  *)
  echo $"Usage: $0 {start|stop|status|restart|try-restart|force-
reload}"
  exit 2
esac
```

Tip

If flannel gets stuck when starting up

Check out your etcd endpoint is accessible and the key listed in `FLANNEL_ETCD_KEY` exists:

```
# FLANNEL_ETCD_KEY="/coreos.com/network/config"
$ curl -L http://<etcd
endpoint>:2379/v2/keys/coreos.com/network/config
```

You could also check out flannel logs using `sudo journalctl -u flanneld`.

After the flannel service starts, you should be able to see a file in `/run/flannel/subnet.env` and the `flannel0` bridge in `ifconfig`.

How to do it…

To ensure flannel works well and transmits the packets from the Docker virtual interface, we need to integrate it with Docker.

Flannel networking configuration

1. After `flanneld` is up and running, use the `ifconfig` or `ip` commands to see whether there is a `flannel0` virtual bridge in the interface:

   ```
   # check current ipv4 range
   $ ip a | grep flannel | grep inet
       inet 192.168.50.0/16 scope global flannel0
   ```

   We can see from the preceding example, the subnet lease of `flannel0` is `192.168.50.0/16`.

2. Whenever your `flanneld` service starts, flannel will acquire the subnet lease and save in etcd and then write out the environment variable file in `/run/flannel/subnet.env` by default, or you could change the default path using the `--subnet-file` parameter when launching it:

   ```
   # check out flannel subnet configuration on this host
   $ cat /run/flannel/subnet.env
   FLANNEL_SUBNET=192.168.50.1/24
   FLANNEL_MTU=1472
   FLANNEL_IPMASQ=true
   ```

Integrating with Docker

There are a couple of parameters that are supported by the Docker daemon. In `/run/flannel/subnet.env`, flannel already allocated one subnet with the suggested MTU and IPMASQ settings. The corresponding parameters in Docker are:

Parameters	Meaning
`--bip=""`	Specify network bridge IP (`docker0`)
`--mtu=0`	Set the container network MTU (for `docker0` and `veth`)
`--ip-masq=true`	(Optional) Enable IP masquerading

1. We could use the variables listed in `/run/flannel/subnet.env` into the Docker daemon:

```
# import the environment variables from subnet.env
$ . /run/flannel/subnet.env

# launch docker daemon with flannel information
$ docker -d --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU}
# Or if your docker version is 1.8 or higher, use subcommand
daemon instead
$ docker daemon --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU}
```

2. Alternatively, you can also specify them into `OPTIONS` of `/etc/sysconfig/docker`, which is the Docker configuration file in CentOS:

```
### in the file - /etc/sysconfig/docker
# set the variables into OPTIONS
$ OPTIONS="--bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU} --ip-
masq=${FLANNEL_IPMASQ}"
```

   In the preceding example, specify `${FLANNEL_SUBNET}` is replaced by `192.168.50.1/24` and `${FLANNEL_MTU}` is `1472` in the `/etc/sysconfig/docker`.

3. Start Docker using `service docker start` and type `ifconfig`; you might be able to see the virtual network device `docker0` and its allocated IP address from flannel.

How it works…

There are two virtual bridges named `flannel0` and `docker0` that are created in the previous steps. Let's take a look at their IP range using the `ip` command:

```
# checkout IPv4 network in local
$ ip -4 a | grep inet
    inet 127.0.0.1/8 scope host lo
    inet 10.42.1.171/24 brd 10.42.21.255 scope global dynamic
ens160
    inet 192.168.50.0/16 scope global flannel0
    inet 192.168.50.1/24 scope global docker0
```

Host IP address is `10.42.1.171/24`, `flannel0` is `192.168.50.0/16`, `docker0` is `192.168.50.1/24`, and the route is set for the full flat IP range:

```
# check the route
$ route -n
Destination     Gateway         Genmask         Flags Metric Ref
Use Iface
0.0.0.0         10.42.1.1       0.0.0.0         UG    100    0
0 ens160
192.168.0.0     0.0.0.0         255.255.0.0     U     0      0
0 flannel0
192.168.50.0    0.0.0.0         255.255.255.0   U     0      0
0 docker0
```

Let's go a little bit deeper to see how etcd stores flannel subnet information. You could retrieve the network configuration by using the `etcdctl` command in etcd:

```
# get network config
$ etcdctl get /coreos.com/network/config
{ "Network": "192.168.0.0/16" }

# show all the subnet leases
$ etcdctl ls /coreos.com/network/subnets
/coreos.com/network/subnets/192.168.50.0-24
```

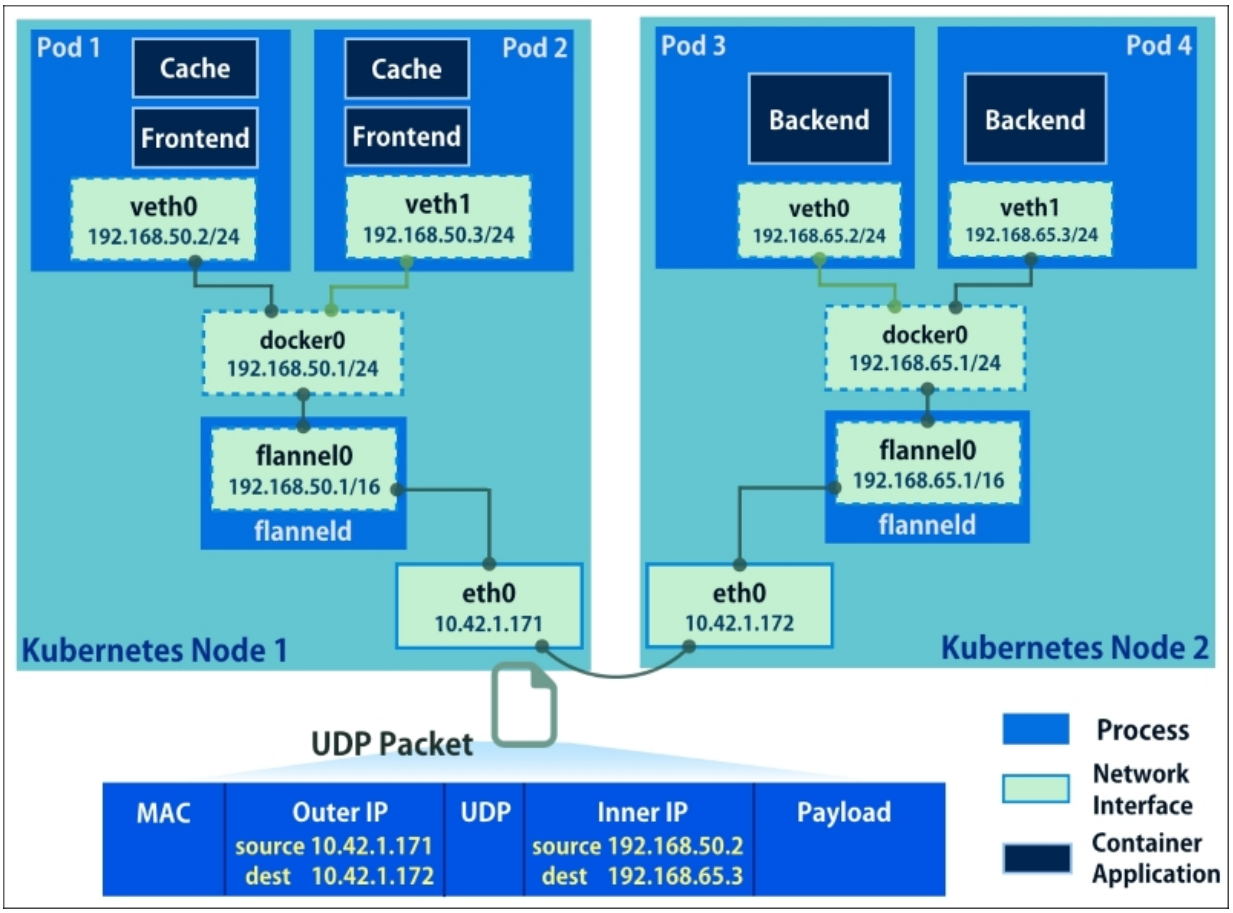
The preceding example shows that the network CIDR is `192.168.0.0/16`. There is one subnet lease. Check the value of the key; it's exactly the IP address of `eth0` on the host:

```
# show the value of the key of
`/coreos.com/network/subnets/192.168.50.0-24`
$ etcdctl get /coreos.com/network/subnets/192.168.50.0-24
{"PublicIP":"10.42.1.171"}
```

If you're using other backend solutions rather than simple UDP, you might see more configuration as follows:

```
# show the value when using different backend
$ etcdctl get /coreos.com/network/subnets/192.168.50.0-24
{"PublicIP":"10.97.1.171","BackendType":"vxlan","BackendData":
{"VtepMAC":"ee:ce:55:32:65:ce"}}
```

Following is an illustration about how a packet from **Pod1** goes through the overlay network to **Pod4**. As we discussed before, every pod will have its own IP address and the packet is encapsulated so that pod IPs are routable. The packet from **Pod1** will go through the **veth** (virtual network interface) device that connects to **docker0**, and routes to **flannel0**. The traffic is encapsulated by **flanneld** and sent to the host (**10.42.1.172**) of the target pod.



Let's perform a simple test by running two individual containers to see whether flannel works well. Assume we have two hosts (`10.42.1.171` and `10.42.1.172`) with different subnets, which are allocated by Flannel with the same etcd backend, and have launched Docker run by `docker run -it ubuntu /bin/bash` in each host:

Container 1 on host 1 (10.42.1.171)	Container 2 on host 2 (10.42.1.172)
root@0cd2a2f73d8e:/# ifconfig eth0 eth0 Link encap:Ethernet HWaddr 02:42:c0:a8:3a:08 inet addr:192.168.50.2 Bcast:0.0.0.0 Mask:255.255.255.0 inet6 addr: fe80::42:c0ff:fea8:3a08/64 Scope:Link UP BROADCAST RUNNING MULTICAST MTU:8951 Metric:1 RX packets:8 errors:0 dropped:0 overruns:0 frame:0 TX packets:8 errors:0 dropped:0 overruns:0 carrier:0 collisions:0 txqueuelen:0 RX bytes:648 (648.0 B) TX bytes:648 (648.0 B) root@0cd2a2f73d8e:/# ping 192.168.65.2 PING 192.168.4.10 (192.168.4.10) 56(84) bytes of data. 64 bytes from 192.168.4.10: icmp_seq=2 ttl=62 time=0.967 ms 64 bytes from 192.168.4.10: icmp_seq=3 ttl=62 time=1.00 ms	root@619b3ae36d77:/# ifconfig eth0 eth0 Link encap:Ethernet HWaddr 02:42:c0:a8:04:0a inet addr:192.168.65.2 Bcast:0.0.0.0 Mask:255.255.255.0 inet6 addr: fe80::42:c0ff:fea8:40a/64 Scope:Link UP BROADCAST RUNNING MULTICAST MTU:8973 Metric:1 RX packets:8 errors:0 dropped:0 overruns:0 frame:0 TX packets:8 errors:0 dropped:0 overruns:0 carrier:0 collisions:0 txqueuelen:0 RX bytes:648 (648.0 B) TX bytes:648 (648.0 B)

We can see that two containers can communicate with each other using ping. Let's observe the packet using `tcpdump` in `host2`, which is a command-line tool that can help dump traffic on a network:

```
# install tcpdump in container
$ yum install -y tcpdump

# observe the UDP traffic from host2
```

```
$ tcpdump host 10.42.1.172 and udp
11:20:10.324392 IP 10.42.1.171.52293 > 10.42.1.172.6177: UDP,
length 106
11:20:10.324468 IP 10.42.1.172.47081 > 10.42.1.171.6177: UDP,
length 106
11:20:11.324639 IP 10.42.1.171.52293 > 10.42.1.172.6177: UDP,
length 106
11:20:11.324717 IP 10.42.1.172.47081 > 10.42.1.171.6177: UDP,
length 106
```

The traffic between the containers are encapsulated in UDP through port `6177` using flanneld.

See also

After setting up and understanding the overlay network, we have a good understanding of how flannel acts in Kubernetes. Check out the following recipes:

- The *Working with pods*, *Working with services* recipes in Chapter 2, *Walking through Kubernetes Concepts*
- The *Forwarding container ports* recipe in Chapter 3, *Playing with Containers*
- The *Authentication and authorization* recipe in Chapter 7, *Advanced Cluster Administration*

Configuring master

The master node of Kubernetes works as the control center of containers. The duties of which are taken charge by the master include serving as a portal to end users, assigning tasks to nodes, and gathering information. In this recipe, we will see how to set up Kubernetes master. There are three daemon processes on master:

- API Server
- Scheduler
- Controller Manager

We can either start them using the wrapper command, `hyperkube`, or individually start them as daemons. Both the solutions are covered in this section.

Getting ready

Before deploying the master node, make sure you have the etcd endpoint ready, which acts like the datastore of Kubernetes. You have to check whether it is accessible and also configured with the overlay network **Classless Inter-Domain Routing** (**CIDR** https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing). It is possible to check it using the following command line:

```
// Check both etcd connection and CIDR setting
$ curl -L <etcd endpoint
URL>/v2/keys/coreos.com/network/config
```

If connection is successful, but the etcd configuration has no expected CIDR value, you can push value through `curl` as well:

```
$ curl -L <etcd endpoint
URL>/v2/keys/coreos.com/network/config -XPUT -d
```

```
value="{ \"Network\": \"<CIDR of overlay
network>\" }"
```

> **Tip**
>
> Besides this, please record the following items: the URL of `etcd endpoint`, the port exposed by `etcd endpoint`, and the CIDR of the overlay network. You will need them while configuring master's services.

How to do it…

In order to build up a master, we propose the following steps for installing the source code, starting with the daemons and then doing verification. Follow the procedure and you'll get a practical master eventually.

Installation

Here, we offer two kinds of installation procedures:

- One is a RHEL-based OS with package manager; master daemons are controlled by `systemd`
- The other one is for other Linux distributions; we build up master with binary files and service init scripts

CentOS 7 or Red Hat Enterprise Linux 7

1. RHEL 7, CentOS 7, or later have an official package for Kubernetes. You can install them via the `yum` command:

```
// install Kubernetes master package
# yum install kubernetes-master kubernetes-
client
```

   The `kubernetes-master` package contains master daemons, while `kubernetes-client` installs a tool called `kubectl`, which is

the Command Line Interface for communicating with the Kubernetes system. Since the master node is served as an endpoint for requests, with `kubectl` installed, users can easily control container applications and the environment through commands.

> ## Note
>
> **CentOS 7's RPM of Kubernetes**
>
> There are five Kubernetes RPMs (the `.rpm` files, https://en.wikipedia.org/wiki/RPM_Package_Manager) for different functionalities: `kubernetes`, `kubernetes-master`, `kubernetes-client`, `kubernetes-node`, and `kubernetes-unit-test`.
>
> The first one, `kubernetes`, is just like a hyperlink to the following three items. You will install `kubernetes-master`, `kubernetes-client`, and `kubernetes-node` at once. The one named `kubernetes-node` is for node installation. And the last one, `kubernetes-unit-test` contains not only testing scripts, but also Kubernetes template examples.

2. Here are the files after `yum install`:

```
// profiles as environment variables for
services
# ls /etc/kubernetes/
apiserver  config  controller-manager
scheduler
// systemd files
# ls /usr/lib/systemd/system/kube-*
/usr/lib/systemd/system/kube-apiserver.service
/usr/lib/systemd/system/kube-scheduler.service
/usr/lib/systemd/system/kube-controller-
manager.service
```

3. Next, we will leave the `systemd` files as the original ones and modify the values in the configuration files under the directory `/etc/kubernetes` to build a connection with etcd. The file named `config` is a shared environment file for several Kubernetes daemon processes. For basic settings, simply change items in `apiserver`:

```
# cat /etc/kubernetes/apiserver
###
# kubernetes system config
#
# The following values are used to configure
the kube-apiserver
#

# The address on the local server to listen
to.
KUBE_API_ADDRESS="--address=0.0.0.0"

# The port on the local server to listen on.
KUBE_API_PORT="--insecure-port=8080"

# Port nodes listen on
# KUBELET_PORT="--kubelet_port=10250"

# Comma separated list of nodes in the etcd
cluster
KUBE_ETCD_SERVERS="--etcd_servers=<etcd
endpoint URL>:<etcd exposed port>"

# Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-
range=<CIDR of overlay network>"

# default admission control policies
KUBE_ADMISSION_CONTROL="--
admission_control=NamespaceLifecycle,Namespace
Exists,LimitRanger,SecurityContextDeny,Service
Account,ResourceQuota"

# Add your own!
KUBE_API_ARGS="--cluster_name=<your cluster
name>"
```

4. Then, start the daemon `kube-apiserver`, `kube-scheduler`, and `kube-controller-manager` one by one; the command `systemctl` can help for management. Be aware that `kube-apiserver` should always start first, since `kube-scheduler` and `kube-controller-manager` connect to the Kubernetes API server when they start running:

```
// start services
# systemctl start kube-apiserver
# systemctl start kube-scheduler
# systemctl start kube-controller-manager
// enable services for starting automatically
while server boots up.
# systemctl enable kube-apiserver
# systemctl enable kube-scheduler
# systemctl enable kube-controller-manager
```

Adding daemon dependency

1. Although `systemd` does not return error messages without the API server running, both `kube-scheduler` and `kube-controller-manager` get connection errors and do not provide regular services:

```
$ sudo systemctl status kube-scheduler -l—
output=cat kube-scheduler.service - Kubernetes
Scheduler Plugin
   Loaded: loaded
(/usr/lib/systemd/system/kube-
scheduler.service; enabled)
   Active: active (running) since Thu 2015-11-
19 07:21:57 UTC; 5min ago
     Docs:
https://github.com/GoogleCloudPlatform/kuberne
tes
 Main PID: 2984 (kube-scheduler)
   CGroup: /system.slice/kube-
scheduler.service
           └─2984 /usr/bin/kube-scheduler—
logtostderr=true—v=0 --master=127.0.0.1:8080
E1119 07:27:05.471102    2984
reflector.go:136] Failed to list *api.Node:
Get http://127.0.0.1:8080/api/v1/nodes?
```

```
fieldSelector=spec.unschedulable%3Dfalse: dial
tcp 127.0.0.1:8080: connection refused
```

2. Therefore, in order to prevent the starting order to affect performance, you can add two settings under the section of `systemd.unit` in `/usr/lib/systemd/system/kube-scheduler` and `/usr/lib/systemd/system/kube-controller-manager`:

```
[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/GoogleCloudPl
atform/kubernetes
After=kube-apiserver.service
Wants=kube-apiserver.service
```

   With the preceding settings, we can make sure `kube-apiserver` is the first started daemon.

3. Furthermore, if you expect the scheduler and the controller manager to always be running along with a healthy API server, which means if `kube-apiserver` is stopped, `kube-scheduler` and `kube-controller-manager` will be stopped as well; you can change `systemd.unit` item `Wants` to `Requires`, as follows:

```
Requires=kube-apiserver.service
```

   `Requires` has more strict restrictions. In case the daemon `kube-apiserver` has crashed, `kube-scheduler` and `kube-controller-manager` would also be stopped. On the other hand, configuration with `Requires` is hard for debugging master installation. It is recommended that you enable this parameter once you make sure every setting is correct.

Other Linux options

It is also possible that we download a binary file for installation. The official website for the latest release is here: https://github.com/kubernetes/kubernetes/releases:

1. We are going to install the version tagged as **Latest release** and start all the daemons with the wrapper command `hyperkube`:

```
// download Kubernetes package
# curl -L -O
https://github.com/GoogleCloudPlatform/kuberne
tes/releases/download/v1.1.2/kubernetes.tar.gz

// extract the tarball to specific local, here
we put it under /opt. the KUBE_HOME would be
/opt/kubernetes
# tar zxvf kubernetes.tar.gz -C /opt/

// copy all binary files to system directory
# cp /opt/kubernetes/server/bin/*
/usr/local/bin/
```

2. The next step is to create a startup script (init), which would cover three master daemons and start them individually:

```
# cat /etc/init.d/kubernetes-master
#!/bin/bash
#
# This shell script takes care of starting and
stopping kubernetes master

# Source function library.
. /etc/init.d/functions

# Source networking configuration.
. /etc/sysconfig/network

prog=/usr/local/bin/hyperkube
lockfile=/var/lock/subsys/`basename $prog`
hostname=`hostname`
logfile=/var/log/kubernetes.log

CLUSTER_NAME="<your cluster name>"
ETCD_SERVERS="<etcd endpoint URL>:<etcd
exposed port>"
CLUSTER_IP_RANGE="<CIDR of overlay network>"
MASTER="127.0.0.1:8080"
```

3. To manage your Kubernetes settings more easily and clearly, we will put the declaration of changeable variables at the beginning of this `init` script. Please double-check the etcd URL

and overlay network CIDR to confirm that they are the same as your previous installation:

```
start() {

  # Start daemon.
  echo $"Starting apiserver: "
  daemon $prog apiserver \
  --service-cluster-ip-
range=${CLUSTER_IP_RANGE} \
  --port=8080 \
  --address=0.0.0.0 \
  --etcd_servers=${ETCD_SERVERS} \
  --cluster_name=${CLUSTER_NAME} \
  > ${logfile}_apiserver 2>&1 &

  echo $"Starting controller-manager: "
  daemon $prog controller-manager \
  --master=${MASTER} \
  > ${logfile}_controller-manager 2>&1 &

  echo $"Starting scheduler: "
  daemon $prog scheduler \
  --master=${MASTER} \
  > ${logfile}_scheduler 2>&1 &

  RETVAL=$?
  [ $RETVAL -eq 0 ] && touch $lockfile
  return $RETVAL
}

stop() {
  [ "$EUID" != "0" ] && exit 4
        echo -n $"Shutting down $prog: "
  killproc $prog
  RETVAL=$?
        echo
  [ $RETVAL -eq 0 ] && rm -f $lockfile
  return $RETVAL
}
```

4. Next, feel free to attach the following lines as the last part in the script for general service usage:

```
# See how we were called.
case "$1" in
  start)
```

```
  start
  ;;
  stop)
  stop
  ;;
  status)
  status $prog
  ;;
  restart|force-reload)
  stop
  start
  ;;
  try-restart|condrestart)
  if status $prog > /dev/null; then
      stop
      start
  fi
  ;;
  reload)
  exit 3
  ;;
  *)
  echo $"Usage: $0
{start|stop|status|restart|try-restart|force-
reload}"
  exit 2
esac
```

5. Now, it is good to start the service named `kubernetes-master`:

```
$sudo service kubernetes-master start
```

> **Note**
>
> At the time of writing this book, the latest version of Kubernetes was 1.1.2. So, we will use 1.1.2 in the examples for most of the chapters.

Verification

1. After starting all the three daemons of the master node, you can verify whether they are running properly by checking the service

status. Both the commands, `systemd` and `service`, are able to get the logs:

```
# systemd status <service name>
```

2. For a more detailed log in history, you can use the command `journalctl`:

```
# journalctl -u <service name> --no-pager --
full
```

   Once you find a line showing `Started...` in the output, you can confirm that the service setup has passed the verification.

3. Additionally, the dominant command in Kubernetes, `kubectl`, can begin the operation:

```
// check Kubernetes version
# kubectl version
Client Version: version.Info{Major:"1",
Minor:"0.3",
GitVersion:"v1.0.3.34+b9a88a7d0e357b",
GitCommit:"b9a88a7d0e357be2174011dd2b127038c6e
a8929", GitTreeState:"clean"}
Server Version: version.Info{Major:"1",
Minor:"0.3",
GitVersion:"v1.0.3.34+b9a88a7d0e357b",
GitCommit:"b9a88a7d0e357be2174011dd2b127038c6e
a8929", GitTreeState:"clean"}
```

See also

From the recipe, you know how to create your own Kubernetes master. You can also check out the following recipes:

- *Exploring architecture*
- *Configuring nodes*
- The *Building multiple masters* recipe in Chapter 4, *Building a High Availability Cluster*

- The *Building the Kubernetes infrastructure in AWS* recipe in Chapter 6, *Building Kubernetes on AWS*
- The *Authentication and authorization* recipe in Chapter 7, *Advanced Cluster Administration*

Configuring nodes

Node is the slave in the Kubernetes cluster. In order to let master take a node under its supervision, node installs an agent called kubelet for registering itself to a specific master. After registering, daemon kubelet also handles container operations and reports resource utilities and container statuses to the master. The other daemon running on the node is kube-proxy, which manages TCP/UDP packets between containers. In this section, we will show you how to configure a node.

Getting ready

Since node is the worker of Kubernetes and the most important duty is running containers, you have to make sure that Docker and flanneld are installed at the beginning. Kubernetes relies on Docker helping applications to run in containers. And through flanneld, the pods on separated nodes can communicate with each other.

After you have installed both the daemons, according to the file `/run/flannel/subnet.env`, the network interface `docker0` should be underneath the same LAN as `flannel0`:

```
# cat /run/flannel/subnet.env
FLANNEL_SUBNET=192.168.31.1/24
FLANNEL_MTU=8973
FLANNEL_IPMASQ=true

// check the LAN of both flanneld0 and docker0
# ifconfig docker0 ; ifconfig flannel0
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu
1500
        inet 192.168.31.1  netmask 255.255.255.0
broadcast 0.0.0.0
        ether 02:42:6e:b9:a7:51  txqueuelen 0
(Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame
```

```
0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier
0  collisions 0
flannel0: flags=81<UP,POINTOPOINT,RUNNING>  mtu
8973
        inet 192.168.31.0  netmask 255.255.0.0
destination 192.168.11.0
        unspec 00-00-00-00-00-00-00-00-00-00-00-
00-00-00-00-00  txqueuelen 500  (UNSPEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame
0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier
0  collisions 0
```

If `docker0` is in a different CIDR range, you may take the following service scripts as a reference for a reliable Docker service setup:

```
# cat /etc/sysconfig/docker
# /etc/sysconfig/docker
#
# Other arguments to pass to the docker daemon
process
# These will be parsed by the sysv initscript and
appended
# to the arguments list passed to docker -d, or
docker daemon where docker version is 1.8 or
higher

. /run/flannel/subnet.env

other_args="--bip=${FLANNEL_SUBNET} --
mtu=${FLANNEL_MTU}"
DOCKER_CERT_PATH=/etc/docker
```

Alternatively, by way of `systemd`, the configuration also originally handles the dependency:

```
$ cat
/etc/systemd/system/docker.service.requires/flanne
```

```
ld.service
[Unit]
Description=Flanneld overlay address etcd agent
After=network.target
Before=docker.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/flanneld
EnvironmentFile=-/etc/sysconfig/docker-network
ExecStart=/usr/bin/flanneld -etcd-
endpoints=${FLANNEL_ETCD} -etcd-
prefix=${FLANNEL_ETCD_KEY} $FLANNEL_OPTIONS
ExecStartPost=/usr/libexec/flannel/mk-docker-
opts.sh -k DOCKER_NETWORK_OPTIONS -d
/run/flannel/docker

[Install]
RequiredBy=docker.service

$ cat /run/flannel/docker
DOCKER_OPT_BIP="--bip=192.168.31.1/24"
DOCKER_OPT_MTU="--mtu=8973"
DOCKER_NETWORK_OPTIONS=" --bip=192.168.31.1/24 --
mtu=8973 "
```

Once you have modified the Docker service script to a correct one, stop the Docker service, clean its network interface, and start it again.

For more details on the flanneld setup and Docker integration, please refer to the recipe *Creating an overlay network*.

You can even configure a master to the node; just install the necessary daemons.

How to do it…

Once you verify that Docker and flanneld are good to go on your node host, continue to install the Kubernetes package for the node. We'll cover both RPM and tarball setup.

Installation

This will be the same as the Kubernetes master installation, Linux OS having the command line tool `yum`, the package management utility, can easily install the node package. On the other hand, we are also able to install the latest version through downloading a tarball file and copy binary files to the specified system directory, which is suitable for every Linux distribution. You can try either of the solutions for your deployment.

CentOS 7 or Red Hat Enterprise Linux 7

1. First, we will install the package `kubernetes-node`, which is what we need for the node:

```
// install kubernetes node package
$ yum install kubernetes-node
```

   The package `kubernetes-node` includes two daemon processes, `kubelet` and `kube-proxy`.

2. We need to modify two configuration files to access the master node:

```
# cat /etc/kubernetes/config
###
# kubernetes system config
#
# The following values are used to configure
various aspects of all
# kubernetes services, including
#
#   kube-apiserver.service
#   kube-controller-manager.service
#   kube-scheduler.service
#   kubelet.service
#   kube-proxy.service
# logging to stderr means we get it in the
systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"
```

```
# Should this cluster be allowed to run
privileged docker containers
KUBE_ALLOW_PRIV="--allow_privileged=false"

# How the controller-manager, scheduler, and
proxy find the apiserver
KUBE_MASTER="--master=<master endpoint>:8080"
```

3. In the configuration file, we will change the master location argument to the machine's URL/IP, where you installed master. If you specified another exposed port for the API server, remember to update it as well, instead of port `8080`:

```
# cat /etc/kubernetes/kubelet
###
# kubernetes kubelet (node) config

# The address for the info server to serve on
(set to 0.0.0.0 or "" for all interfaces)
KUBELET_ADDRESS="--address=0.0.0.0"

# The port for the info server to serve on
# KUBELET_PORT="--port=10250"

# You may leave this blank to use the actual
hostname
KUBELET_HOSTNAME="--
hostname_override=127.0.0.1"

# location of the api-server
KUBELET_API_SERVER="--api_servers=<master
endpoint>:8080"

# Add your own!
KUBELET_ARGS=""
```

   We open the kubelet address for all the interfaces and attached master location.

4. Then, it is good to start services using the command `systemd`. There is no dependency between `kubelet` and `kube-proxy`:

```
// start services
# systemctl start kubelet
# systemctl start kube-proxy
// enable services for starting automatically
while server boots up.
# systemctl enable kubelet
# systemctl enable kube-proxy
// check the status of services
# systemctl status kubelet
# systemctl status kube-proxy
```

Other Linux options

1. We can also download the latest Kubernetes binary files and write a customized service init script for node configuration. The tarball of Kubernetes' latest updates will be released at https://github.com/kubernetes/kubernetes/releases:

```
// download Kubernetes package
# curl -L -O
https://github.com/GoogleCloudPlatform/kuberne
tes/releases/download/v1.1.2/kubernetes.tar.gz

// extract the tarball to specific local, here
we put it under /opt. the KUBE_HOME would be
/opt/kubernetes
# tar zxvf kubernetes.tar.gz -C /opt/

// copy all binary files to system directory
# cp /opt/kubernetes/server/bin/*
/usr/local/bin/
```

2. Next, a file named `kubernetes-node` is created under `/etc/init.d` with the following content:

```
# cat /etc/init.d/kubernetes-node
#!/bin/bash
#
# kubernetes    This shell script takes care
of starting and stopping kubernetes

# Source function library.
. /etc/init.d/functions
```

```
# Source networking configuration.
. /etc/sysconfig/network

prog=/usr/local/bin/hyperkube
lockfile=/var/lock/subsys/`basename $prog`
MASTER_SERVER="<master endpoint>"
hostname=`hostname`
logfile=/var/log/kubernetes.log
```

3. Be sure to provide the master URL/IP for accessing the Kubernetes API server. If you're trying to install a node package on the master host as well, which means make master also work as a node, the API server should work on the local host. If so, you can attach `localhost` or `127.0.0.1` at `<master endpoint>`:

```
start() {
    # Start daemon.
    echo $"Starting kubelet: "
    daemon $prog kubelet \
        --
api_servers=http://${MASTER_SERVER}:8080 \
        --v=2 \
        --address=0.0.0.0 \
        --enable_server \
        --hostname_override=${hostname} \
        > ${logfile}_kubelet 2>&1 &

    echo $"Starting proxy: "
    daemon $prog proxy \
        --master=http://${MASTER_SERVER}:8080
\
        --v=2 \
        > ${logfile}_proxy 2>&1 &

    RETVAL=$?
    [ $RETVAL -eq 0 ] && touch $lockfile
    return $RETVAL
}
stop() {
    [ "$EUID" != "0" ] && exit 4
        echo -n $"Shutting down $prog: "
    killproc $prog
    RETVAL=$?
        echo
    [ $RETVAL -eq 0 ] && rm -f $lockfile
```

```
    return $RETVAL
}
```

4. The following lines are for general daemon management, attaching them in the script to get the functionalities:

```
# See how we were called.
case "$1" in
  start)
    start
    ;;
  stop)
    stop
    ;;
  status)
    status $prog
    ;;
  restart|force-reload)
    stop
    start
    ;;
  try-restart|condrestart)
    if status $prog > /dev/null; then
        stop
        start
    fi
    ;;
  reload)
    exit 3
    ;;
  *)
    echo $"Usage: $0
{start|stop|status|restart|try-restart|force-
reload}"
    exit 2
esac
```

5. Now, you can start the service with the name of your `init` script:

```
# service kubernetes-node start
```

Verification

In order to check whether a node is well-configured, the straightforward way would be to check it from the master side:

```
// push command at master
# kubelet get nodes
NAME                                     LABELS
STATUS
ip-10-97-217-56.sdi.trendnet.org
kubernetes.io/hostname=ip-10-97-217-
56.sdi.trendnet.org   Ready
```

See also

It is also recommended to read the recipes about the architecture of the cluster and system environment. Since the Kubernetes node is like a worker, who receives tasks and listens to the others; they should be built after the other components. It is good for you to get more familiar with the whole system before building up nodes. Furthermore, you can also manage the resource in nodes. Please check the following recipes for more information:

- *Exploring architecture*
- *Preparing your environment*
- The *Setting resource in nodes* recipe in Chapter 7, *Advanced Cluster Administration*

Run your first container in Kubernetes

Congratulations! You've built your own Kubernetes cluster in the previous sections. Now, let's get on with running your very first container nginx (http://nginx.org/), which is an open source reverse proxy server, load balancer, and web server.

Getting ready

Before we start running the first container in Kubernetes, it's better to check whether every component works as expected. Please follow these steps on master to check whether the environment is ready to use:

1. Check whether the Kubernetes components are running:

```
# check component status are all healthy
$ kubectl get cs
NAME                 STATUS    MESSAGE
ERROR
controller-manager   Healthy   ok
nil
scheduler            Healthy   ok
nil
etcd-0               Healthy   {"health":
"true"}   nil
```

> **Note**
>
> If any one of the components is not running, check out the settings in the previous sections. Restart the related services, such as `service kube-apiserver start`.

2. Check the master status:

```
# Check master is running
$ kubectl cluster-info
Kubernetes master is running at
http://localhost:8080
```

> **Note**
>
> If the Kubernetes master is not running, restart the service using `service kubernetes-master start` or `/etc/init.d/kubernetes-master start`.

3. Check whether all the nodes are ready:

```
# check nodes are all Ready
$ kubectl get nodes
NAME          LABELS
STATUS
kub-node1   kubernetes.io/hostname=kub-node1
Ready
kub-node2   kubernetes.io/hostname=kub-node2
Ready
```

> **Note**
>
> If one node is expected as `Ready` but is `NotReady`, go to that node to restart Docker and the node service using `service docker start` and `service kubernetes-node start`.

Before we go to the next section, make sure the nodes are accessible to the Docker registry. We will use the nginx image from Docker Hub (https://hub.docker.com/) as an example. If you want to

run your own application, be sure to dockerize it first! What you need to do for your custom application is to write a Dockerfile (https://docs.docker.com/v1.8/reference/builder), build, and push it into the public/private Docker registry.

Tip

Test your node connectivity with the public/private Docker registry

On your node, try `docker pull nginx` to test whether you can pull the image from Docker Hub. If you're behind a proxy, please add `HTTP_PROXY` into your Docker configuration file (normally, in `/etc/sysconfig/docker`). If you want to run the image from the private repository in Docker Hub, using `docker login` on the node to place your credential in `~/.docker/config.json`, copy the credentials into /var/lib/kubelet/.dockercfg in the json format and restart Docker:

```
# put the credential of docker registry
$ cat /var/lib/kubelet/.dockercfg

{
        "<docker registry endpoint>": {
                "auth": "SAMPLEAUTH=",
                "email": "noreply@sample.com"
        }
}
```

If you're using your own private registry, specify `INSECURE_REGISTRY` in the Docker configuration file.

How to do it…

We will use the official Docker image of nginx as an example. The image is prebuilt in Docker Hub (https://hub.docker.com/_/nginx/).

Many official and public images are available on Docker Hub so that you do not need to build it from scratch. Just pull it and set up your custom setting on top of it.

Running an HTTP server (nginx)

1. On the Kubernetes master, we could use `kubectl run` to create a certain number of containers. The Kubernetes master will then schedule the pods for the nodes to run:

```
$ kubectl run <replication controller name> --
image=<image name> --replicas=<number of
replicas> [--port=<exposing port>]
```

2. The following example will create two replicas with the name `my-first-nginx` from the nginx image and expose port `80`. We could deploy one or more containers in what is referred to as a pod. In this case, we will deploy one container per pod. Just like a normal Docker behavior, if the nginx image doesn't exist in local, it will pull it from Docker Hub by default:

```
# Pull the nginx image and run with 2
replicas, and expose the container port 80
$ kubectl run my-first-nginx --image=nginx --
replicas=2 --port=80
CONTROLLER       CONTAINER(S)     IMAGE(S)
SELECTOR            REPLICAS
my-first-nginx   my-first-nginx   nginx
run=my-first-nginx   2
```

> ### Tip
>
> **The name of replication controller <my-first-nginx> cannot be duplicate**
>
> The resource (pods, services, replication controllers, and so on) in one Kubernetes namespace cannot be duplicate. If

you run the preceding command twice, the following error will pop up:

```
Error from server: replicationControllers "my-first-nginx" already exists
```

3. Let's get and see the current status of all the pods using `kubectl get pods`. Normally, the status of the pods will hold on `Pending` for a while, since it takes some time for the nodes to pull the image from Docker Hub:

```
# get all pods
$ kubectl get pods
NAME                    READY     STATUS
RESTARTS   AGE
my-first-nginx-nzygc    1/1       Running   0
1m
my-first-nginx-yd84h    1/1       Running   0
1m
```

Tip

If the pod status is not running for a long time

You could always use `kubectl get pods` to check the current status of the pods and `kubectl describe pods $pod_name` to check the detailed information of a pod. If you make a typo of the image name, you might get the `Image not found` error message, and if you are pulling the images from a private repository or registry without proper credentials setting, you might get the `Authentication error` message. If you get the `Pending` status for a long time and check out the node capacity, make sure you don't run too many replicas that exceed the node capacity described in the *Preparing your environment* section. If there are other unexpected error messages, you could either stop the pods or the entire

replication controller to force master to schedule the tasks again.

4. After waiting a few seconds, there are two pods running with the `Running` status:

```
# get replication controllers
$ kubectl get rc
CONTROLLER         CONTAINER(S)        IMAGE(S)
SELECTOR
REPLICAS
my-first-nginx     my-first-nginx      nginx
run=my-first-nginx
2
```

Exposing the port for external access

We might also want to create an external IP address for the `nginx` replication controller. On cloud providers, which support an external load balancer (such as Google Compute Engine) using the `LoadBalancer` type, will provision a load balancer for external access. On the other hand, you can still expose the port by creating a Kubernetes service as follows, even though you're not running on the platforms that support an external load balancer. We'll describe how to access this externally later:

```
# expose port 80 for replication controller named
my-first-nginx
$ kubectl expose rc my-first-nginx --port=80 --
type=LoadBalancer
NAME              LABELS                SELECTOR
IP(S)     PORT(S)
my-first-nginx    run=my-first-nginx    run=my-
first-nginx             80/TCP
```

We can see the service status we just created:

```
# get all services
```

```
$ kubectl get service
NAME                          LABELS
SELECTOR
IP(S)              PORT(S)
my-first-nginx                run=my-first-nginx
run=my-first-nginx
192.168.61.150     80/TCP
```

Congratulations! You just ran your first container with a Kubernetes pod and exposed port `80` with the Kubernetes service.

Stopping the application

We could stop the application using commands such as the `stop` replication controller and service. Before this, we suggest you read through the following introduction first to understand more about how it works:

```
# stop replication controller named my-first-nginx
$ kubectl stop rc my-first-nginx
replicationcontrollers/my-first-nginx

# stop service named my-first-nginx
$ kubectl stop service my-first-nginx
services/my-first-nginx
```

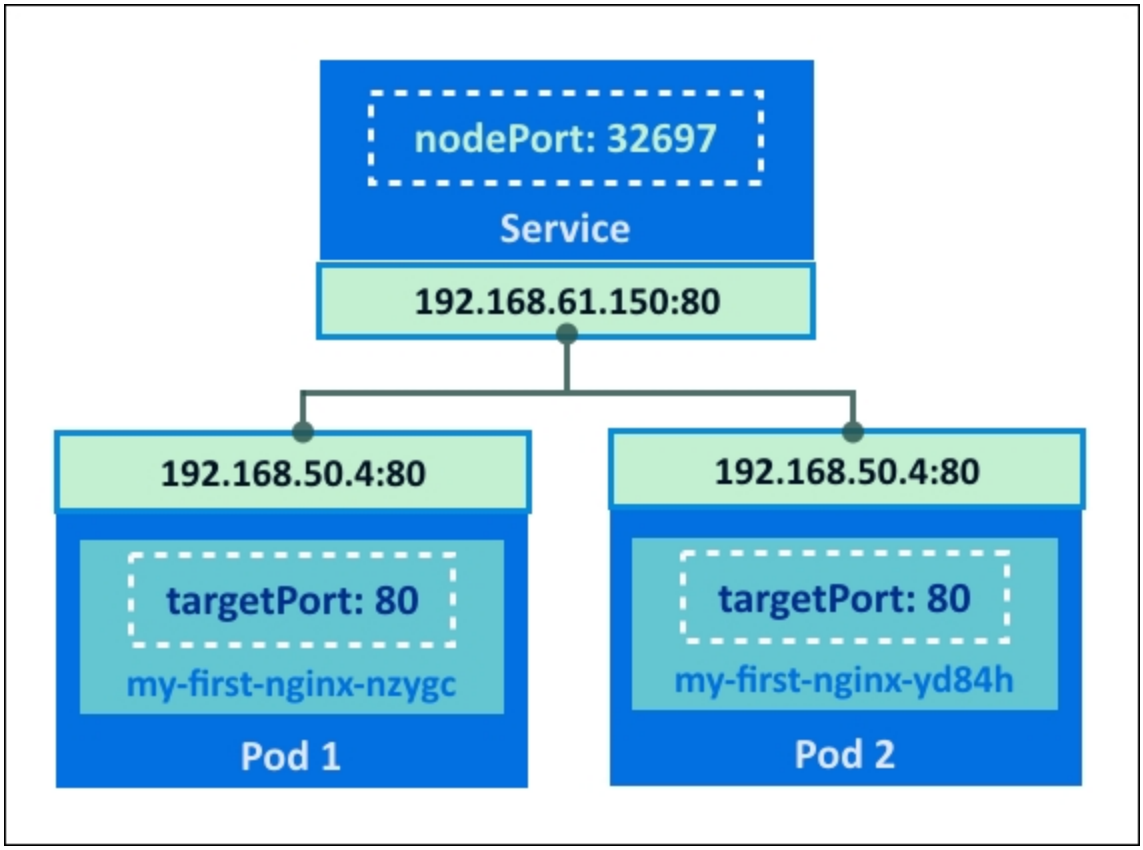
How it works…

Let's take a look at the insight of the service using `describe` in the `kubectl` command. We will create one Kubernetes service with the type `LoadBalancer`, which will dispatch the traffic into two `Endpoints` `192.168.50.4` and `192.168.50.5` with port `80`:

```
$ kubectl describe service my-first-nginx
Name:       my-first-nginx
Namespace:     default
Labels:       run=my-first-nginx
Selector:     run=my-first-nginx
Type:       LoadBalancer
```

```
IP:       192.168.61.150
Port:       <unnamed>  80/TCP
NodePort:     <unnamed>  32697/TCP
Endpoints:     192.168.50.4:80,192.168.50.5:80
Session Affinity:  None
No events.
```

`Port` here is an abstract service port, which will allow any other resources to access the service within the cluster. The `nodePort` will be indicating the external port for allowing external access. The `targetPort` is the port the container allows traffic into; by default, it will be the same with `Port`. The illustration is as follows. External access will access service with `nodePort`. Service acts as a load balancer to dispatch the traffic to the pod using `Port 80`. The pod will then pass through the traffic into the corresponding container using `targetPort 80`:



In any nodes or master (if your master has flannel installed), you should be able to access the nginx service using ClusterIP

192.168.61.150 with port 80:

```
# curl from service IP
$ curl 192.168.61.150:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-
serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is
successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please
refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

It will be the same result if we curl to the target port of the pod directly:

```
# curl from endpoint
$ curl 192.168.50.4:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

```
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-
serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is
successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please
refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

If you'd like to try out external access, use your browser to access the external IP address. Please note that the external IP address depends on which environment you're running in.

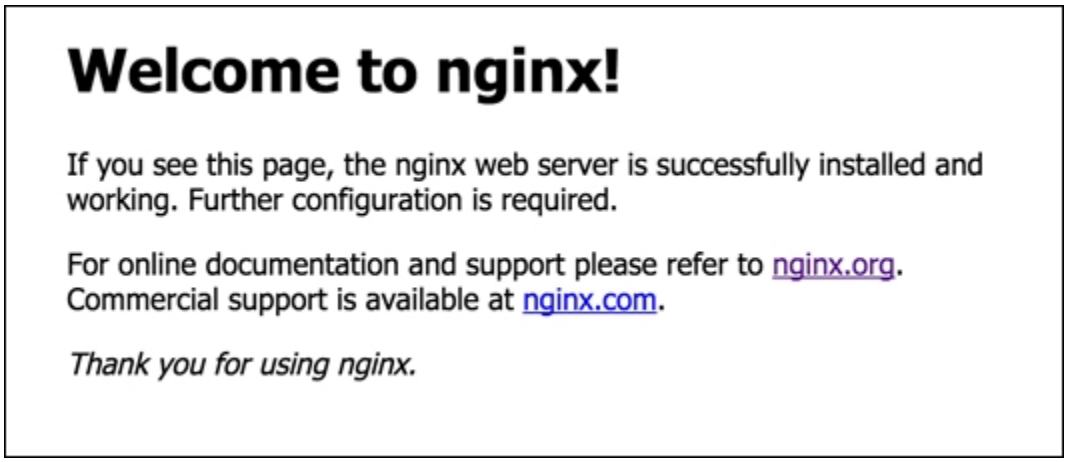
In Google Compute Engine, you could access it via a ClusterIP with proper firewall rules setting:

```
$ curl http://<clusterIP>
```

In a custom environment, such as on a premise datacenter, you could go through the IP address of nodes to access to:

```
$ curl http://<nodeIP>:<nodePort>
```

You should be able to see the following page using a web browser:



See also

We have run our very first container in this section. Now:

- To explore more of the concepts in Kubernetes, refer to Chapter 2, *Walking through Kubernetes Concepts*

Chapter 2. Walking through Kubernetes Concepts

In this chapter, we will cover:

- An overview of Kubernetes control
- Working with pods
- Working with a replication controller
- Working with services
- Working with volumes
- Working with secrets
- Working with names
- Working with namespaces
- Working with labels and selectors

Introduction

In this chapter, we will start by creating different kinds of resources on the Kubernetes system. To realize your application in a microservices structure, reading the recipes in this chapter will be a good start to understanding the concepts of the Kubernetes resources and consolidating them. After you deploy applications in Kubernetes, you can work on its scalable and efficient container management, and also fulfill the DevOps delivering procedure of microservices.

An overview of Kubernetes control

Working with Kubernetes is quite easy, using either **Command Line Interface** (**CLI**) or API (RESTful). This section will describe Kubernetes control by CLI. The CLI we used in this chapter is version 1.1.3.

Getting ready

After you install Kubernetes master, you can run a `kubectl` command as follows. It shows the `kubectl` and Kubernetes master versions (both 1.1.3).

```
% kubectl version
Client Version: version.Info{Major:"1", Minor:"1",
GitVersion:"v1.1.3",
GitCommit:"6a81b50c7e97bbe0ade075de55ab4fa34f049dc
2", GitTreeState:"clean"}
Server Version: version.Info{Major:"1", Minor:"1",
GitVersion:"v1.1.3",
GitCommit:"6a81b50c7e97bbe0ade075de55ab4fa34f049dc
2", GitTreeState:"clean"}
```

How to do it…

`kubectl` connects the Kubernetes API server using RESTful API. By default it attempts to access the localhost, otherwise you need to specify the API server address using the `--server` parameter. Therefore, it is recommended to use `kubectl` on the API server machine for practice.

Tip

If you use `kubectl` over the network, you need to consider authentication and authorization for the API server. See Chapter 7, *Advanced Cluster Administration*.

How it works…

`kubectl` is the only command for Kubernetes clusters, and it controls the Kubernetes cluster manager. Find more information at http://kubernetes.io/docs/user-guide/kubectl-overview/. Any container, or Kubernetes cluster operation, can be performed by a `kubectl` command.

In addition, `kubectl` allows the inputting of information by either the command line's optional arguments, or by file (use `-f` option), but it is highly recommended to use file, because you can maintain the Kubernetes cluster as code:

```
kubectl [command] [TYPE] [NAME] [flags]
```

The attributes of the preceding command are explained as follows:

- `command`: Specifies the operation that you want to perform on one or more resources.
- `TYPE`: Specifies the resource type. Resource types are case-sensitive and you can specify the singular, plural, or abbreviated forms.
- `NAME`: Specifies the name of the resource. Names are case-sensitive. If the name is omitted, details for all resources are displayed.
- `flags`: Specifies optional flags.

For example, if you want to launch nginx, you can use the `kubectl run` command as the following:

```
# /usr/local/bin/kubectl run my-first-nginx --
```

```
image=nginx
replicationcontroller "my-first-nginx"
```

However, you can write either a YAML file or a JSON file to perform similar operations. For example, the YAML format is as follows:

```
# cat nginx.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-first-nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: my-first-nginx
        image: nginx
```

Then specify the `create -f` option to execute the `kubectl` command as follows:

```
# kubectl create -f nginx.yaml
replicationcontroller "my-first-nginx" created
```

If you want to see the status of the replication controller, type the `kubectl get` command as follows:

```
# kubectl get replicationcontrollers
CONTROLLER       CONTAINER(S)      IMAGE(S)
SELECTOR    REPLICAS   AGE
my-first-nginx   my-first-nginx   nginx
app=nginx   1          12s
```

If you also want the support abbreviation, type the following:

```
# kubectl get rc
CONTROLLER        CONTAINER(S)      IMAGE(S)
SELECTOR    REPLICAS    AGE
my-first-nginx    my-first-nginx    nginx
app=nginx    1           1m
```

If you want to delete these resources, type the `kubectl delete` command as follows:

```
# kubectl delete rc my-first-nginx
replicationcontroller "my-first-nginx" deleted
```

The `kubectl` command supports many kinds of sub-commands, use `-h` option to see the details, for example:

```
//display whole sub command options
# kubectl -h

//display sub command "get" options
# kubectl get -h

//display sub command "run" options
# kubectl run -h
```

See also

This recipe describes how to use the `kubectl` command to control the Kubernetes cluster. The following recipes describe how to set up Kubernetes components:

- The *Building datastore*, *Creating an overlay network*, *Configuring master*, and *Configuring nodes* recipes in Chapter 1, *Building Your Own Kubernetes*

Working with pods

The pod is a group of one or more containers and the smallest deployable unit in Kubernetes. Pods are always co-located and co-scheduled, and run in a shared context. Each pod is isolated by the following Linux namespaces:

- Process ID (PID) namespace
- Network namespace
- Interprocess Communication (IPC) namespace
- Unix Time Sharing (UTS) namespace

In a pre-container world, they would have been executed on the same physical or virtual machine.

It is useful to construct your own application stack pod (for example, web server and database) that are mixed by different Docker images.

Getting ready

You must have a Kubernetes cluster and make sure that the Kubernetes node has accessibility to the Docker Hub (https://hub.docker.com) in order to download Docker images. You can simulate downloading a Docker image by using the `docker pull` command as follows:

```
//run as root on node machine

# docker pull centos
latest: Pulling from centos

47d44cb6f252: Pull complete
168a69b62202: Pull complete
812e9d9d677f: Pull complete
4234bfdd88f8: Pull complete
ce20c473cd8a: Pull complete
```

```
Digest:
sha256:c96eeb93f2590858b9e1396e808d817fa0ba4076c68
b59395445cb957b524408
Status: Downloaded newer image for centos:latest
```

How to do it…

1. Log in to the Kubernetes master machine and prepare the following YAML file. It defines the launch nginx container and the CentOS container.
2. The nginx container opens the HTTP port (TCP/80). On the other hand, the CentOS container attempts to access the `localhost:80` every three seconds using the `curl` command:

```
# cat my-first-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: my-first-pod
spec:
  containers:
  - name: my-nginx
    image: nginx
  - name: my-centos
    image: centos
    command: ["/bin/sh", "-c", "while : ;do
curl http://localhost:80/; sleep 3; done"]
```

3. Then, execute the `kubectl create` command to launch `my-first-pod` as follows:

```
# kubectl create -f my-first-pod.yaml
pod "my-first-pod" created
```

   It takes between a few seconds and minutes, depending on the network bandwidth to the Docker Hub and Kubernetes nodes spec.

4. You can check `kubectl get pods` to see the status as follows:

```
//still downloading Docker images (0/2)
# kubectl get pods
NAME           READY     STATUS    RESTARTS
AGE
my-first-pod   0/2       Running   0
6s

//it also supports shorthand format as "po"
# kubectl get po
NAME           READY     STATUS    RESTARTS
AGE
my-first-pod   0/2       Running   0
7s


//my-first-pod is running (2/2)

# kubectl get pods
NAME           READY     STATUS    RESTARTS
AGE
my-first-pod   2/2       Running   0
8s
```

Now both the nginx container (`my-nginx`) and the CentOS container (`my-centos`) are ready.

5. Let's check whether the CentOS container can access nginx or not. You can check the stdout (standard output) by using the `kubectl logs` command and specifying the CentOS container (`my-centos`) as follows:

```
//it shows last 30 lines output (--tail=30)

# kubectl logs my-first-pod -c my-centos --
tail=30
</body>
</html>
  % Total    % Received % Xferd  Average Speed
Time    Time     Time  Current
                                 Dload  Upload
Total   Spent    Left  Speed
<!DOCTYPE html>
<html>
<head>
```

```
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial,
sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server
is successfully installed and
working. Further configuration is required.
</p>

<p>For online documentation and support please
refer to
<a href="http://nginx.org/">nginx.org</a>.
<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
100   612  100   612    0     0   4059      0
--:--:-- --:--:-- --:--:--  4080
```

As you can see, the pod links two different containers, nginx and CentOS into the same Linux namespace.

How it works…

When launching a pod, the Kubernetes scheduler dispatches to the kubelet process to handle all the operations to launch both nginx and CentOS containers.

If you have two or more nodes, you can check the `-o wide` option to find a node which runs a pod:

```
//it indicates Node ip-10-96-219-25 runs my-first-
```

```
pod

# kubectl get pods -o wide
NAME           READY     STATUS    RESTARTS   AGE
NODE
my-first-pod   2/2       Running   0          2m
ip-10-96-219-25
```

Log in to that node, then you can check the `docker ps` command to see the running containers as follows:

```
# docker ps
CONTAINER ID        IMAGE
COMMAND                  CREATED             STATUS
PORTS               NAMES
b7eb8d0925b2        centos
"/bin/sh -c 'while :   2 minutes ago       Up 2
minutes                                    k8s_my-
centos.704bf394_my-first-pod_default_a3b78651-
a061-11e5-a7fb-06676ae2a427_f8b61e2b
55d987322f53        nginx
"nginx -g 'daemon of   2 minutes ago       Up 2
minutes                                    k8s_my-
nginx.608bdf36_my-first-pod_default_a3b78651-a061-
11e5-a7fb-06676ae2a427_10cc491a
a90c8d2d40ee
gcr.io/google_containers/pause:0.8.0   "/pause"
2 minutes ago       Up 2 minutes
k8s_POD.6d00e006_my-first-pod_default_a3b78651-
a061-11e5-a7fb-06676ae2a427_dfaf502a
```

You may notice that three containers – CentOS, nginx and pause – are running instead of two. Because each pod we need to keep belongs to a particular Linux namespace, if both the CentOS and nginx containers die, the namespace will also destroyed. Therefore, the pause container just remains in the pod to maintain Linux namespaces.

Let's launch a second pod, rename it as `my-second-pod` and run the `kubectl create` command as follows:

```
//just replace the name from my-first-pod to my-
second-pod

# cat my-first-pod.yaml | sed -e 's/my-first-
pod/my-second-pod/' > my-second.pod.yaml

# cat my-second.pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-second-pod
spec:
  containers:
  - name: my-nginx
    image: nginx
  - name: my-centos
    image: centos
    command: ["/bin/sh", "-c", "while : ;do curl
http://localhost:80/; sleep 3; done"]

# kubectl create -f my-second.pod.yaml
pod "my-second-pod" created

# kubectl get pods
NAME            READY     STATUS    RESTARTS   AGE
my-first-pod    2/2       Running   0          49m
my-second-pod   2/2       Running   0          5m
```

If you have two or more nodes, `my-second-pod` was probably launched by another node, because the Kubernetes scheduler chose the most suitable node.

Tip

Note that, if you would like to deploy more of the same pod, consider using a replication controller instead.

After your testing, you can run the `kubectl delete` command to delete your pod from the Kubernetes cluster:

```
//running both my-first-pod and my-second-pod
# kubectl get pods
NAME            READY     STATUS    RESTARTS   AGE
my-first-pod    2/2       Running   0          49m
my-second-pod   2/2       Running   0          5m


//delete my-second-pod
# kubectl delete pod my-second-pod
pod "my-second-pod" deleted
# kubectl get pods
NAME           READY     STATUS    RESTARTS   AGE
my-first-pod   2/2       Running   0          54m


//delete my-first-pod
# kubectl delete pod my-first-pod
pod "my-first-pod" deleted
# kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
```

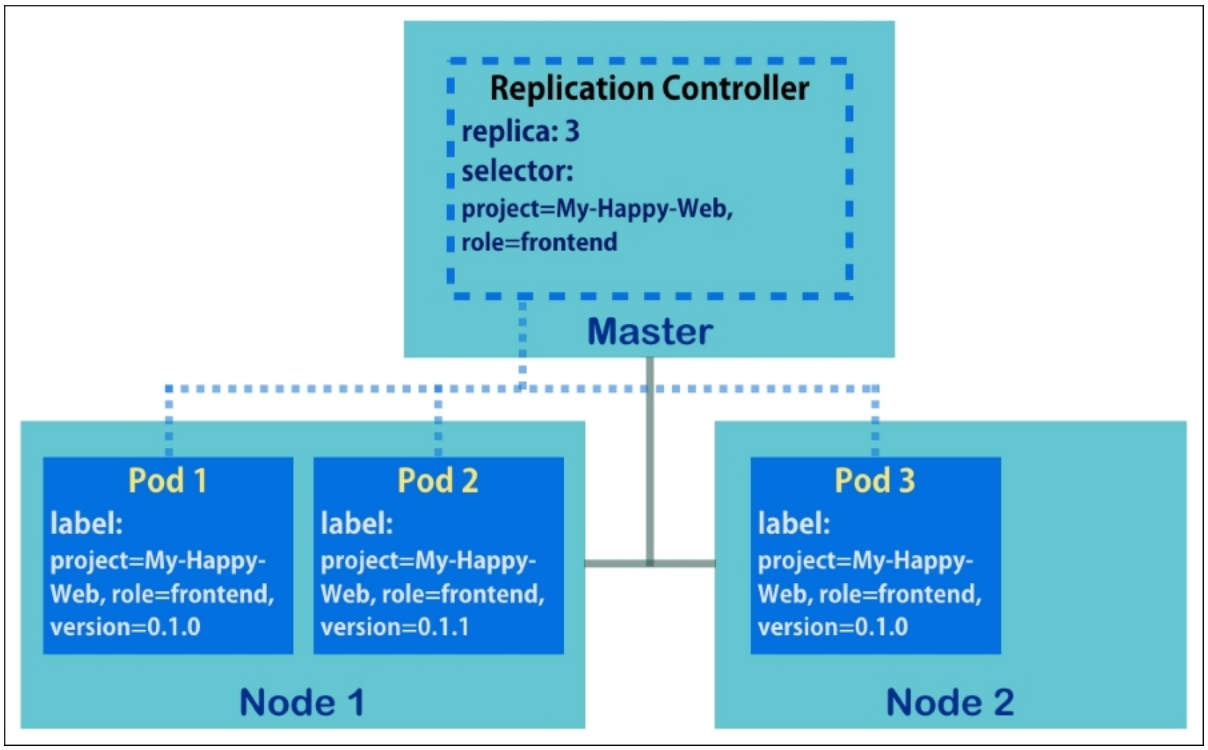
See also

This recipe described how to control pods. It is the basic component and operation of Kubernetes. The following recipes will describe advanced operation of pods using a replication controller, services and so on:

- *Working with a replication controller*
- *Working with services*
- *Working with labels and selectors*

Working with a replication controller

A replication controller is a term for API objects in Kubernetes that refers to pod replicas. The idea is to be able to control a set of pods' behaviors. The replication controller ensures that the pods, in a user-specified number, are running all the time. If some pods in the replication controller crash and terminate, the system will recreate pods with the original configurations on healthy nodes automatically, and keep a certain amount of processes continuously running. According to this feature, no matter whether you need replicas of pods or not, you can always shield the pods with the replication controller for autorecovery. In this recipe, you're going to learn how to manage your pods by using the replication controller:



The replication controller usually handles a tier of applications. As you see in the preceding image, we launch a replication controller with three pod replicas. Some mechanism details are listed as follows:

- The daemon in the master is called the controller manager and helps to maintain the resource running in its desired state. For example, the desired state of the replication controller in the previous image is three pod replicas.
- The daemon scheduler in the master takes charge of assigning tasks to healthy nodes.
- The selector of the replication controller is used for deciding which pods it covers. If the key-value pairs in the pod's label include all items in the selector of the replication controller, this pod belongs to this replication controller. As you will see, the previous image shows three pods are under the charge of the replication controller. Since the selector is covered and labelled **project** and **role**, the pod with the different minor version number, **(Pod 2)**, could still be one of the members.

Getting ready

We demonstrated the management of the replication controller in the Kubernetes master, when we installed the Kubernetes client package. Please login to the master first and make sure your environment is able to create replication controllers.

Tip

The evaluation of replication controller creation from the master

You can verify whether your Kubernetes master is a practical one through checking the following items:

Check whether the daemons are running or not. There should be three working daemon processes on the master node: `apiserver`, `scheduler` and `controller-manager`.

Check the command `kubectl` exists and is workable. Try the command `kubectl get componentstatuses` or `kubectl get cs`, so you can verify not only the components' status but also the feasibility of `kubectl`.

Check the nodes are ready to work. You can check them by the command `kubectl get nodes` for their status.

In case some of the preceding items are invalid, please refer to Chapter 1, *Building Your Own Kubernetes* for proper guidelines of the installation.

How to do it…

A replication controller can be created directly through CLI, or through a template file. We will express the former solution in this section. For template one, please refer to the recipe *Working with configuration files* in Chapter 3, *Playing with Containers*.

Creating a replication controller

To create replication controllers, we use the subcommand `run` after `kubectl`. The basic command pattern looks like the following:

```
//  kubectl run <REPLICATION CONTROLLER NAME> --images=<IMAGE NAME>
[OPTIONAL_FLAGS]
# kubectl run my-first-rc --image=nginx
CONTROLLER    CONTAINER(S)    IMAGE(S)    SELECTOR      REPLICAS
my-first-rc   my-first-rc     nginx       run=my-first-rc   1
```

This simple command is creating a replication controller by image nginx from the Docker Hub (https://hub.docker.com). The name, `my-first-rc`, must be unique in all replication controllers. Without specified number of replicas, the system will only build one pod as its default value. There are more optional flags you may integrate together to create a qualified replication controller:

--Flag=[Default Value]	Description	Example
`--replicas=1`	The number of pod replicas	`--replicas=3`
`--port=-1`	The exposed port of the	`--port=80`

--Flag=[Default Value]	Description	Example
	container	
`--hostport=-1`	The host port mapping for the container port. Make sure that if using the flag, you do not run multiple pod replicas on a node in order to avoid port conflicts.	`--hostport=8080`
`--labels=""`	Key-value pairs as the labels for pods. Separate each pair with a comma.	`--labels="ProductName=HappyCloud,ProductionState=staging,ProjectOwner`
`--command[=false]`	After the container boots up, run a different command through this flag. Two dashed lines append the flag as segmentation.	`--command -- /myapp/run.py -o logfile`
`--env=[]`	Set environment variables in containers. This flag could be used multiple times in a single command.	`--env="USERNAME=amy" --env="PASSWD=pa$$w0rd"`
`--overrides=""`	Use this flag to override	`--overrides='{"apiVersion": "v1"}'`

--Flag=[Default Value]	Description	Example
	some generated objects of the system in JSON format. The JSON value uses a single quote to escape the double quote. The field `apiVersion` is necessary. For detailed items and requirements, please refer to the *Working with configuration files* recipe in Chapter 3, *Playing with Containers*.	
`--limits=""`	The upper limit of resource usage in the container. You can specify how much CPU and memory could be used. The unit of CPU is the number of cores in the format `NUMBERm`. `m` indicates `milli` (10-3). The unit of memory is byte. Please also check `--requests`.	`--limits="cpu=1000m,memory=512Mi"`

--Flag=[Default Value]	Description	Example
`--requests=""`	The minimum requirement of resource for container. The rule of value is the same as `--limits`.	`--requests="cpu=250m,memory=256Mi"`
`--dry-run[=false]`	Display object configuration without sending it out for being created.	`--dry-run`
`--attach[=false]`	For the replication controller, your terminal will attach to one of the replicas, and view the runtime log coming from the program. The default is to attach the first container in the pod, in the same way as Dockers attach. Some logs from the system show the container's pending status.	`--attach`
`-i , --stdin[=false]`	Enable the interactive mode of the container. The replica must be 1.	`-i`

--Flag=[Default Value]	Description	Example
`--tty=[false]`	Allocate a `tty` (new controlling terminal) to each container. You must enable the interactive mode by attaching the flag `-i` or `--stdin`.	`--tty`

The subcommand `run` will create a replication controller by default because of the flag `--restart`, which is preset as `Always`, meaning that the generated objects will always be triggered and run to meet the desired numbers of the replication controller.

For example, you can launch a replication controller, and then add new features or modify configurations:

```
// Run a replication controller with some flags for Nginx nodes. Try to verify the
settings with "--dry-run" first!
# kubectl run nginx-rc-test --image=nginx --
labels="Owner=Amy,ProductionState=test" --replicas=2 --port=80 --dry-run
CONTROLLER      CONTAINER(S)    IMAGE(S)    SELECTOR
REPLICAS
nginx-rc-test   nginx-rc-test   nginx      Owner=Amy,ProductionState=test   2

// Send out the request
# kubectl run nginx-rc --image=nginx --labels="Owner=Amy,ProductionState=test" --
replicas=2 --port=80

// Try to override container name while generating, which name is the same as the
name of replication controller in default
# kubectl run nginx-rc-override --image=nginx --
overrides='{"apiVersion":"v1","spec":{"template":{"spec": {"containers":[{"name":
"k8s-nginx","image":"nginx"}]}}}}'
CONTROLLER       CONTAINER(S)   IMAGE(S)   SELECTOR             REPLICAS
nginx-rc-override   k8s-nginx      nginx      run=nginx-rc-override   1
//Interact with container after create a pod in replication controller
# kubectl run nginx-bash --image=nginx --tty -i --command -- /bin/bash
Waiting for pod to be scheduled
Waiting for pod default/nginx-bash-916y6 to be running, status is Running, pod
ready: false
Waiting for pod default/nginx-bash-916y6 to be running, status is Running, pod
ready: false
ls
bin   dev  home  lib64  mnt  proc  run  srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@nginx-bash-916y6:/#
```

Getting information of a replication controller

After we create a replication controller, the subcommand `get` and `describe` can help us to capture the information and pod status. In the CLI of Kubernetes, we usually use the abbreviation `rc` for resource type, instead of the full name replication controller:

First, we can check any replication controller in the system:

```
// Use subcommand "get" to list replication controllers
# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR          REPLICAS   AGE
check-rc-1   check-rc-1     nginx      run=check-rc-1    5          7m
check-rc-2   check-rc-2     nginx      app=nginx         2          6m
```

As it displays, the special column items are `SELECTOR` and `REPLICAS`. The selector must be the pods' labels, which indicate that the pods are controlled by this replication controller. We may specify the selector by the flag `--labels` while creating the replication controller with `kubectl run`. The default selector assigned to the replication controller, created by CLI, is in the form of `run=<REPLICATION CONTROLLER NAME>`:

```
// We can also get status of pod through selector/labels
# kubectl get pod -l app=nginx
NAME               READY     STATUS    RESTARTS   AGE
check-rc-2-95851   1/1       Running   0          6m
check-rc-2-mjezz   1/1       Running   0          6m
```

Furthermore, the subcommand `describe` helps users to get detailed items and logs of the resources:

```
# kubectl describe rc check-rc-2
Name:    check-rc-2
Namespace:  default
Image(s):  nginx
Selector:  app=nginx
Labels:    app=nginx
Replicas:  2 current / 2 desired
Pods Status:  2 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen  LastSeen  Count  From        SubobjectPath  Reason      Message
  ─────────  ────────  ─────  ────        ─────────────  ──────      ───────
  6m      6m    1  {replication-controller }      SuccessfulCreate  Created pod:
check-rc-2-95851
  6m    6m    1  {replication-controller }      SuccessfulCreate  Created pod:
check-rc-2-mjezz
```

Changing the configuration of a replication controller

The subcommands known as `edit`, `patch` and `replace` can help to update live replication controllers. All these three change the settings by way of a configuration file. Here we just take `edit` for example.

The subcommand `edit` lets users modify resource configuration through the editor. Try to update your replication controller through the command `kubectl edit rc/<REPLICATION CONTROLLER NAME>` (to change to another resource type, you can change `rc`, for example, `po`, `svc`, `ns`), you will access this via the default editor with a YAML configuration file, except for resource type and resource name. Take a look at the *Working with configuration files* recipe in Chapter 3, *Playing with Containers* for reference, and try to change the other values:

```
// Try to update your by subcommand edit
# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR        REPLICAS   AGE
```

```
test-edit    test-edit      nginx      run=test-edit   1          5m
# kubectl edit rc/test-edit
replicationcontroller "test-edit" edited
# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR                   REPLICAS   AGE
test-edit    nginx-rc       nginx      app=nginx,run=after-edit   3          7m
```

Removing a replication controller

In order to remove replication controllers from the system, you can rely on the subcommand `delete`. The similar subcommand `stop` is deprecated and covered by `delete`, so we just introduce `delete` here. While we use `delete` to remove the resource, it removes the target objects forcefully and ignores any requests for the target objects at the same time:

```
// A replication controller that we want to remove has 5 replicas
# kubectl get rc
CONTROLLER    CONTAINER(S)    IMAGE(S)    SELECTOR          REPLICAS   AGE
test-delete   test-delete     nginx       run=test-delete   5          19s
# kubectl get pod
NAME                 READY     STATUS    RESTARTS   AGE
test-delete-g4xyy    1/1       Running   0          34s
test-delete-px9z6    1/1       Running   0          34s
test-delete-vctnk    1/1       Running   0          34s
test-delete-vsikc    1/1       Running   0          34s
test-delete-ye07h    1/1       Running   0          34s
// timing the response of "delete" and check the state of pod directly
# time kubectl delete rc test-delete && kubectl get pod
replicationcontroller "test-delete" deleted
real  0m2.028s
user  0m0.014s
sys  0m0.007s
NAME                 READY     STATUS        RESTARTS   AGE
test-delete-g4xyy    1/1       Terminating   0          1m
test-delete-px9z6    0/1       Terminating   0          1m
test-delete-vctnk    1/1       Terminating   0          1m
test-delete-vsikc    0/1       Terminating   0          1m
test-delete-ye07h    1/1       Terminating   0          1m
```

We find that the response time is quite short and the effect is also instantaneous.

Tip

Removing pods from the replication controller

It is impossible to remove or scale down the replication controller by deleting pods on it, because while a pod is removed, the replication controller is out of its desired status, and the controller manager will ask it to create another one. This concept is shown in the following commands:

```
// Check replication controller and pod first
# kubectl get rc,pod
CONTROLLER        CONTAINER(S)      IMAGE(S)    SELECTOR              REPLICAS
AGE
test-delete-pod   test-delete-pod   nginx       run=test-delete-pod   3
12s
NAME                    READY     STATUS    RESTARTS   AGE
test-delete-pod-8hooh   1/1       Running   0          14s
test-delete-pod-jwthw   1/1       Running   0          14s
test-delete-pod-oxngk   1/1       Running   0          14s
// Remove certain pod and check pod status to see what happen
# kubectl delete pod test-delete-pod-8hooh && kubectl get pod
NAME                    READY     STATUS        RESTARTS   AGE
test-delete-pod-8hooh   0/1       Terminating   0          1m
```

```
test-delete-pod-8nryo   0/1       Running       0          3s
test-delete-pod-jwthw   1/1       Running       0          1m
test-delete-pod-oxngk   1/1       Running       0          1m
```

How it works…

The replication controller defines a set of pods by a pod template and labels. As you know from previous sections, the replication controller only manages the pods by their labels. It is possible that the pod template and the configuration of the pod, are different. And it also means that standalone pods can be added into a controller's group by label modification. According to the following commands and results, let's evaluate this concept on selectors and labels:

```
// Two pod existed in system already, they have the same label app=nginx
# kubectl get pod -L app -L owner
NAME       READY     STATUS    RESTARTS   AGE       APP       OWNER
web-app1   1/1       Running   0          6m        nginx     Amy
web-app2   1/1       Running   0          6m        nginx     Bob
```

Then, we create a three-pod replication controller with the selector `app=nginx:`

```
# kubectl run rc-wo-create-all --replicas=3 --image=nginx --labels="app=nginx"
replicationcontroller "rc-wo-create-all" created
```

We can find that the replication controller meets the desired state of three pods but only needs to boot one pod. The pod `web-app1` and `web-app2` are now controlled by running `rc-wo-create-all`:

```
# kubectl get pod -L app -L owner
NAME                     READY     STATUS    RESTARTS   AGE       APP       OWNER
rc-wo-create-all-jojve   1/1       Running   0          5s        nginx     <none>
web-app1                 1/1       Running   0          7m        nginx     Amy
web-app2                 1/1       Running   0          7m        nginx     Bob

# kubectl describe rc rc-wo-create-all
Name:    rc-wo-create-all
Namespace:  default
Image(s):  nginx
Selector:  app=nginx
Labels:    app=nginx
Replicas:  3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen  LastSeen  Count  From        SubobjectPath  Reason      Message
  _________  ________  _____  ____        _____________  ______      _______
  1m    1m    1  {replication-controller }      SuccessfulCreate  Created pod: rc-
wo-create-all-jojve
```

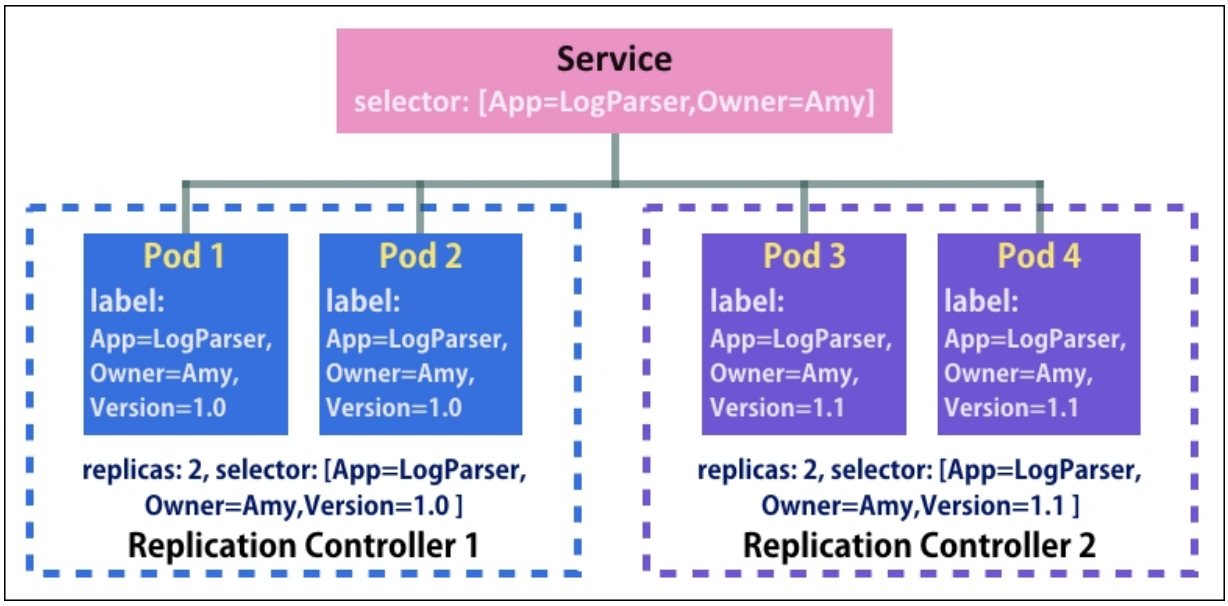
See also

In this chapter, there are some recipes for getting more ideas about the replication controller:

- *Working with pods*
- *Working with services*
- *Working with labels and selectors*
- The Working with configuration files recipe in Chapter 3, *Playing with Containers*

Working with services

The network service is an application that receives requests and provides a solution. Clients access the service by a network connection. They don't have to know the architecture of the service or how it runs. The only thing that clients have to verify is whether the end point of the service is contactable, and then follow its usage policy to solve problems. The Kubernetes service has similar ideas. It is not necessary to understand every pod before reaching their functionalities. For components outside the Kubernetes system, they just access the Kubernetes service with an exposed network port to communicate with running pods. It is not necessary to be aware of the containers' IPs and ports. Therefore, we can fulfill a zero downtime update for our container programs without struggling:



The preceding image shows the basic structure of the service and realizes the following concepts:

- As with the replication controller, the service directs the pods that have labels containing the service's selector. In other words, the pods selected by the service are based on their labels.

- The load of requests sent to the services will distribute to four pods.
- The replication controller ensures that the number of running pods meets its desired state. It monitors the pods for the service, making sure someone will take over duties from the service.

In this recipe, you will learn how to create services along with your pods.

Getting ready

Prior to applying services, it is important to verify whether all your nodes in the system are running `kube-proxy`. Daemon `kube-proxy` works as a network proxy in node. It helps to reflect service settings like IPs or ports on each node. To check whether `kube-proxy` is enabled or not, you can inspect the status of the daemon or search running processes on the node with a specific name:

```
// check the status of service kube-proxy
# service kube-proxy status
```

or

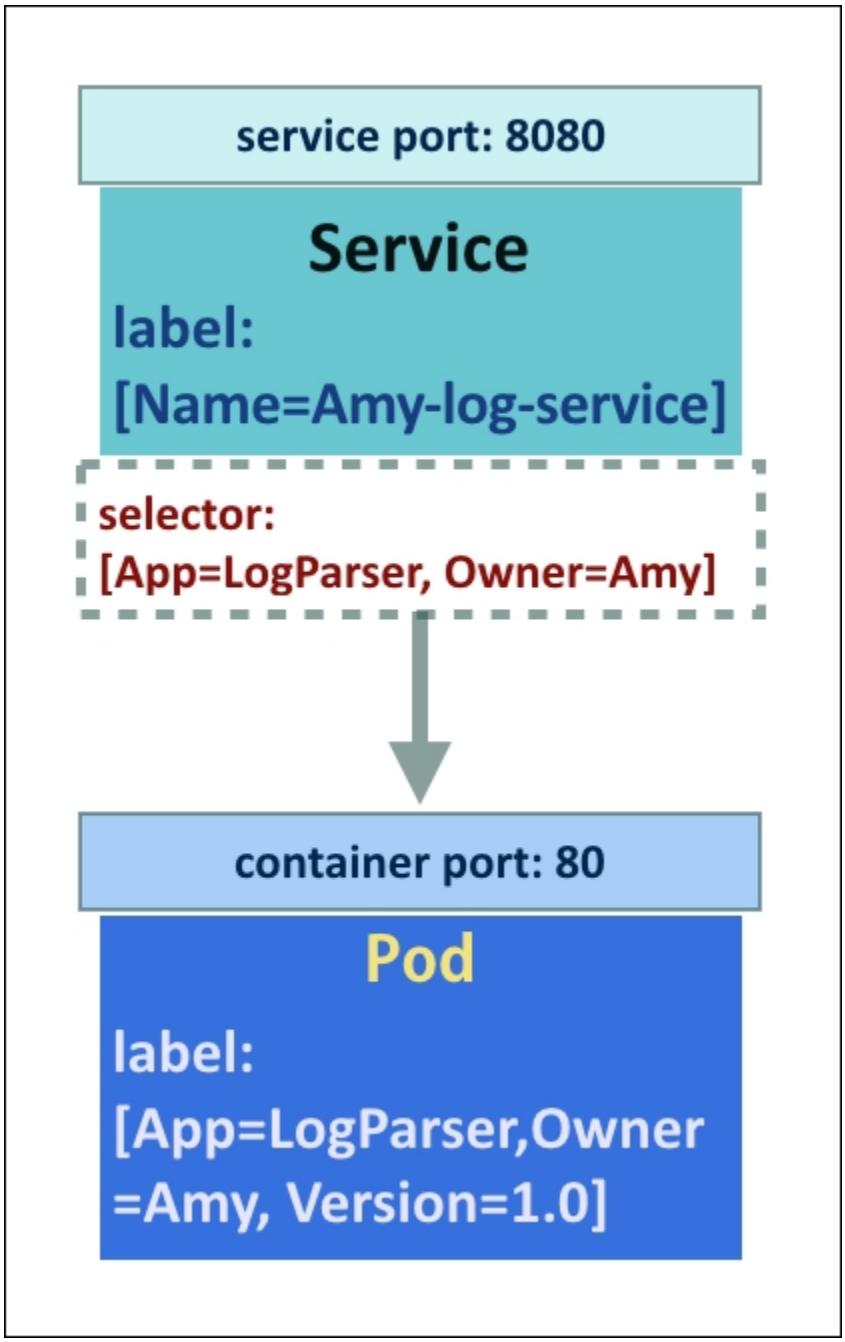
```
// Check processes on each node, and focus on
kube-proxy
// grep "kube-proxy" or "hyperkube proxy"
# ps aux | grep "kube-proxy"
```

For demonstration in later sections, you can also install a private network environment on the master node. The daemons related to the network settings are `flanneld` and `kube-proxy`. It is easier for you to do the operation and verification on a single machine. Otherwise, please verify Kubernetes services on a node, which by default, has an internal network ready.

How to do it…

We can define and create a new Kubernetes service through the CLI or a configuration file. Here we are going to explain how to deploy the services by command. The subcommands `expose` and `describe` are utilized in the following commands for various scenarios. The version of Kubernetes we used in this recipe is 1.1.3. For file-format creation, please go to *Working with configuration files* recipe in Chapter 3, *Playing with Containers* for a detailed discussion.

When creating services, there are two configurations with which we have to take care: one is the label, the other is the port. As the following image indicates, the service and pod have their own key-value pair labels and ports. Be assured to use correct tags for these settings:



To create a service like this one, push the following command:

```
# kubectl expose pod <POD_NAME> --
labels="Name=Amy-log-service" --
selector="App=LogParser,Owner=Amy" --port=8080 --
target-port=80
```

The `--labels` tag `expose` in the subcommand is for labeling the services with key-value pairs. It is used to mark the services. For defining the selector of the service, use tag `--selector`. Without setting the selector for service, the selector would be the same as the labels of resource. In the preceding image, the selector would have an addition label: **Version=1.0**.

To expose the service port, we send out a port number with the tag `--port` in the subcommand `expose`. The service will take the container port number as its exposed port if no specific number is assigned. On the other hand, the tag `--target-port` points out the container port for service. While the target port is different from the exposed port of the container, users will get an empty response. At the same time, if we only assign the service port, the target port will copy it. Taking the previous image as an example, the traffic will be directed to container port `8080` supposing we don't use the tag `--target-port`, which brings out a refused connection error.

Creating services for different resources

You can attach a service to a pod, a replication controller and an endpoint outside the Kubernetes system, or even another service. We will show you these, one by one, in the next pages. The service creation is in the format: `kubectl expose RESOURCE_TYPE RESOURCE_NAME [TAGS]` or `kubectl expose -f CONFIGURATION_FILE [TAGS]`. Simply put, the resource types pod, replication controller and service are supported by the subcommand `expose`. So is the configuration file which follows the type limitation.

Creating a service for a pod

The pods shielded by the service need to contain labels, because the service takes this as a necessary condition based on the selector:

```
// Create a pod, and add labels to it for the
selector of service.
```

```
# kubectl run nginx-pod --image=nginx --port=80 --
restart="Never" --labels="app=nginx"
pod "nginx-pod" created
# kubectl expose pod nginx-pod --port=8000 --
target-port=80 --name="service-pod"
service "service-pod" exposed
```

Tip

The abbreviation of Kubernetes resources

While managing resources through CLI, you can type their abbreviations instead of the full names to save time and avoid typing errors.

Resource type	Abbreviated alias
Componentstatuses	`cs`
Events	`ev`
Endpoints	`ep`
Horizontalpodautoscaler	`hpa`
Limitranges	`limits`
Nodes	`no`

Resource type	Abbreviated alias
Namespaces	`ns`
Pods	`po`
Persistentvolumes	`pv`
Persistentvolumesclaims	`pvc`
Resourcequotas	`qotas`
Replicationcontrollers	`rc`
Services	`svc`
Ingress	`ing`

```
// "svc" is the abbreviation of service
# kubectl get svc service-pod
NAME          CLUSTER_IP        EXTERNAL_IP
PORT(S)    SELECTOR    AGE
service-pod   192.168.195.195   <none>
8000/TCP   app=nginx   11s
```

As you see in these commands, we open a service with port `8000` exposed. The reason why we specify the container port is so that the

service doesn't take `8000` as the container port, by default. To verify whether the service is workable or not, go ahead with the following command in an internal network environment (which has been installed with the Kubernetes cluster CIDR).

```
// accessing by services CLUSTER_IP and PORT
# curl 192.168.195.195:8000
```

Creating a service for the replication controller and adding an external IP

A replication controller is the ideal resource type for a service. For pods supervised by the replication controller, the Kubernetes system has a controller manager to look over the lifecycle of them. It is also helpful for updating the version or state of program by binding existing services to another replication controller:

```
// Create a replication controller with container
port 80 exposed
# kubectl run nginx-rc --image=nginx --port=80 --
replicas=2
replicationcontroller "nginx-rc" created
# kubectl expose rc nginx-rc --name="service-rc" -
-external-ip="<USER_SPECIFIED_IP>"
service "service-rc" exposed
```

In this case, we can provide the service with another IP address, which doesn't need to be inside the cluster network. The tag `--external-ip` of the subcommand `expose` can realize this static IP requirement. Be aware that the user-specified IP address could be contacted, for example, with the master node public IP:

```
// EXTERNAL_IP has Value shown on
# kubectl get svc service-rc
NAME          CLUSTER_IP      EXTERNAL_IP
PORT(S)      SELECTOR        AGE
service-rc    192.168.126.5    <USER_SPECIFIED_IP>
80/TCP     run=nginx-rc    4s
```

Now, you can verify the service by `192.168.126.5:80` or `<USER_SPECIFIED_IP>:80`:

```
// Take a look of service in details
# kubectl describe svc service-rc
Name:       service-rc
Namespace:    default
Labels:       run=nginx-rc
Selector:     run=nginx-rc
Type:       ClusterIP
IP:       192.168.126.5
Port:       <unnamed>  80/TCP
Endpoints:    192.168.45.3:80,192.168.47.2:80
Session Affinity:  None
No events.
```

You will find that the label and selector of a service is the default of the replication controller. In addition, there are multiple endpoints, which are replicas of the replication controller, available for dealing with requests from the service.

Creating a no-selector service for an endpoint

First, you should have an endpoint with an IP address. For example, we can generate an individual container in an instance, where it is located outside our Kubernetes system but is still contactable:

```
// Create an nginx server on another instance with
IP address <FOREIGN_IP>
# docker run -d -p 80:80 nginx
2a17909eca39a543ca46213839fc5f47c4b5c78083f0b067b2
df334013f62002
# docker ps
CONTAINER ID        IMAGE
COMMAND                  CREATED
STATUS              PORTS
NAMES
2a17909eca39        nginx
"nginx -g 'daemon off"   21 seconds ago      Up 20
seconds       0.0.0.0:80->80/tcp, 443/tcp
goofy_brown
```

Then, in the master, we can create a Kubernetes endpoint by using the configuration file. The endpoint is named `service-foreign-ep`. We could configure multiple IP addresses and ports in the template:

```
# cat nginx-ep.json
{
    "kind": "Endpoints",
    "apiVersion": "v1",
    "metadata": {
        "name": "service-foreign-ep"
    },
    "subsets": [
        {
            "addresses": [
                { "ip": "<FOREIGN_IP>" }
            ],
            "ports": [
                { "port": 80 }
            ]
        }
    ]
}
# kubectl create -f nginx-ep.json
endpoints "service-foreign-ep" created
# kubectl get ep service-foreign-ep
NAME                    ENDPOINTS
AGE
service-foreign-ep      <FOREIGN_IP>:80
16s
```

As mentioned in the previous section, we can start a service for a resource-configured template with the subcommand `expose`. However, the CLI is unable to support exposing an endpoint in the file format:

```
// Give it a try!
# kubectl expose -f nginx-ep.json
error: invalid resource provided: Endpoints, only
a replication controller, service or pod is
accepted
```

Therefore, we create the service through a configuration file:

```
# cat service-ep.json
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "service-foreign-ep"
    },
    "spec": {
        "ports": [
            {
                "protocol": "TCP",
                "port": 80,
                   "targetPort" : 80
            }
        ]
    }
}
```

The most important thing of all is that there is no selector defined in the template. This is quite reasonable since the endpoints are not in the Kubernetes system. The relationship between endpoints and service is built up by resource name. As you can see, the name of the service must be identical to the name of the endpoint:

```
# kubectl create -f service-ep.json
service "service-foreign-ep" created
// Check the details in service
# kubectl describe svc service-foreign-ep
Name:       service-ep
Namespace:    default
Labels:       <none>
Selector:     <none>
Type:       ClusterIP
IP:       192.168.234.21
Port:       <unnamed>  80/TCP
Endpoints:    <FOREIGN_IP>:80
Session Affinity:  None
No events.
```

Finally, the no-selector service is created for the external endpoint. Verify the result with `<FOREIGN_IP>:80`.

Creating a service with session affinity based on another service

Through the subcommand `expose`, we can also copy the settings of one service to another:

```
// Check the service we created for replication
controller in previous section
# kubectl describe svc service-rc
Name:       service-rc
Namespace:     default
Labels:       run=nginx-rc
Selector:     run=nginx-rc
Type:       ClusterIP
IP:       192.168.126.5
Port:       <unnamed>  80/TCP
Endpoints:     192.168.45.3:80,192.168.47.2:80
Session Affinity:  None
No events.
//Create a new service with different name and
service port
# kubectl expose svc service-rc --port=8080 --
target-port=80 --name=service-2nd --session-
affinity="ClientIP"
service "service-2nd" exposed
```

The new service named `service-2nd` is reset with service port `8080` and session affinity is enabled:

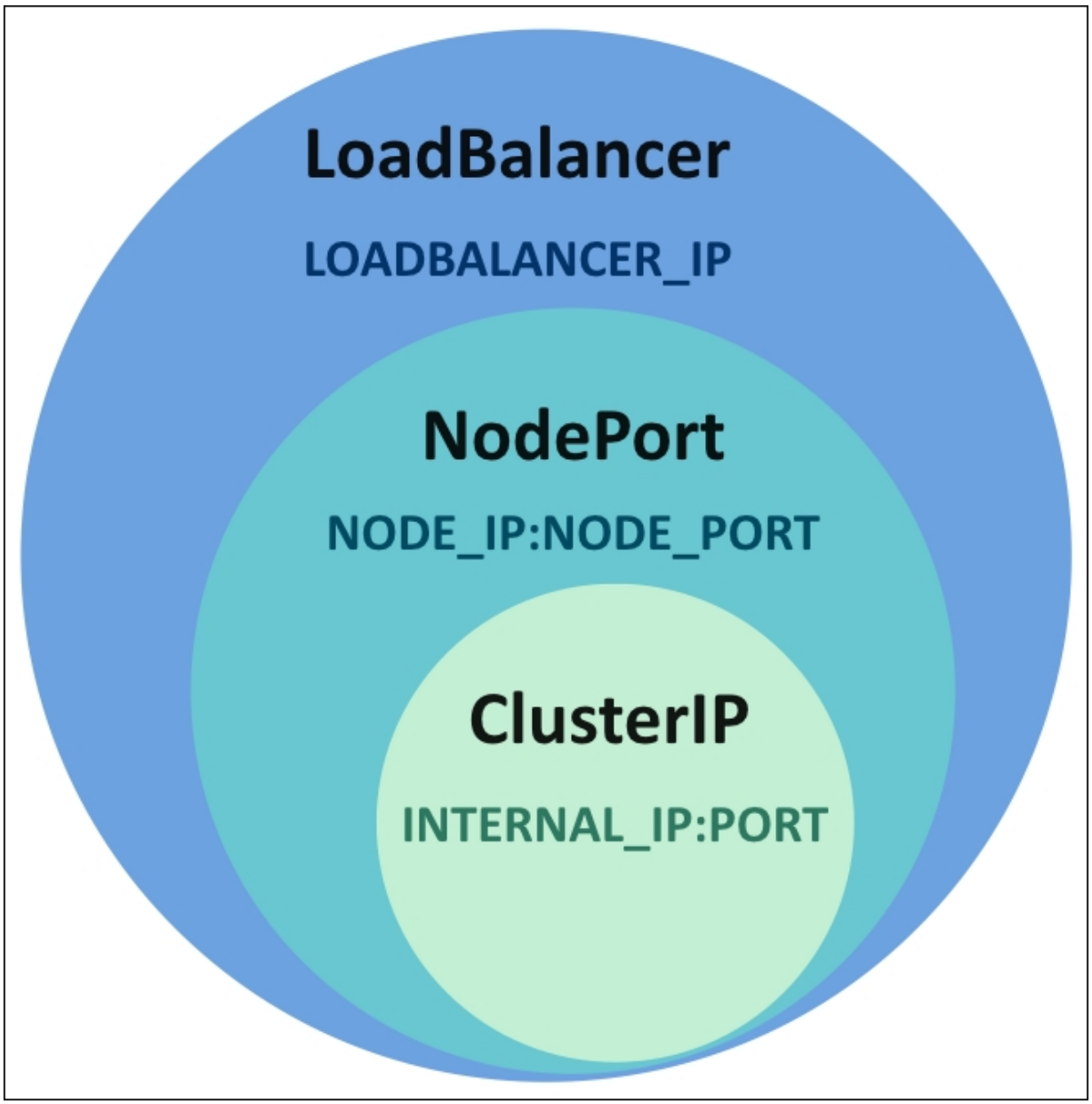
```
# kubectl describe svc service-2nd
Name:       service-2nd
Namespace:     default
Labels:       run=nginx-rc
Selector:     run=nginx-rc
Type:       ClusterIP
IP:       192.168.129.65
Port:       <unnamed>  8080/TCP
Endpoints:     192.168.45.3:80,192.168.47.2:80
```

```
Session Affinity:  ClientIP
No events.
```

Currently, the `ClientIP` is the only valued setting for the tag `--session-affinity`. While session affinity to the `ClientIP` is enabled, instead of round robin, the request of which endpoint the service should be sent to would be decided by the `ClientIP`. For example, if the requests from the client in the CIDR range `192.168.45.0/24` are sent to service `service-2nd`, they will be transferred to the endpoint `192.168.45.3:80`.

Creating a service in a different type

There are three types of service: **ClusterIP**, **NodePort** and **LoadBalancer**:



By default, every service is created as a ClusterIP type. The service in the ClusterIP type would be assigned an internal IP address randomly. For the NodePort type, it covers the ClusterIP's feature, and also allows the user to expose services on each node with the same port. The LoadBalancer is on the top of the other two types. The LoadBalancer service would be exposed internally and on the node. Besides this, if your cloud provider supports external load balancing servers, you can bind the load balancer IP to the service and this will become another exposing point.

Creating a service in NodePort type

Next, we are going to show you how to create a `NodePort` service. The tag `--type` in the subcommand `expose` helps to define the service type:

```
// Create a service with type NodePort, attaching
to the replication controller we created before
# kubectl expose rc nginx-rc --name=service-
nodeport --type="NodePort"
service "service-nodeport" exposed
# kubectl describe svc service-nodeport
Name:       service-nodeport
Namespace:     default
Labels:       run=nginx-rc
Selector:     run=nginx-rc
Type:       NodePort
IP:       192.168.57.90
Port:       <unnamed>  80/TCP
NodePort:     <unnamed>  31841/TCP
Endpoints:     192.168.45.3:80,192.168.47.2:80
Session Affinity:  None
No events.
```

In the preceding case, the network port `31841` exposed on a node is randomly assigned by the system; the default port range is `30000` to `32767`. Notice that the port is exposed on every node in the system, so it is fine to access the service through `<NODE_IP>:31841`, for example, through the domain name of a node, like `kube-node1:31841`.

Deleting a service

You can simply work with the subcommand `delete` in cases where you want to stop a service:

```
# kubectl delete svc <SERVICE_NAME>
service "<SERVICE_NAME>" deleted
```

How it works…

The main actors in the Kubernetes system that perform the service environment are flanneld and kube-proxy. Daemon flanneld builds up a cluster network by allocating a subnet lease out of a preconfigured address space, and storing the network configuration in etcd, while kube-proxy directs the endpoints of services and pods.

See also

To get the best use of services, the following recipes are suggested to be read as well:

- *Working with a replication controller*
- *Working with labels and selectors*
- The *Working with configuration files* recipe in Chapter 3, *Playing with Containers*
- The *Moving monolithic to microservices* recipe in Chapter 5, *Building a Continuous Delivery Pipeline*

Working with volumes

Files in a container are ephemeral. When the container is terminated, the files are gone. Docker has introduced data volumes and data volume containers to help us manage the data by mounting from the host disk directory or from other containers. However, when it comes to a container cluster, it is hard to manage volumes across hosts and their lifetime by using Docker.

Kubernetes introduces volume, which lives with a pod across container restarts. It supports the following different types of network disks:

- emptyDir
- hostPath
- nfs
- iscsi
- flocker
- glusterfs
- rbd
- gitRepo
- awsElasticBlockStore
- gcePersistentDisk
- secret
- downwardAPI

In this section, we'll walk through the details of emptyDir, hostPath, nfs and glusterfs. Secret, which is used to store credentials, will be introduced in the next section. Most of them have similar Kubernetes syntax with a different backend.

Getting ready

The storage providers are required when you start to use volume in Kubernetes except for emptyDir, which will be erased when the pod

is removed. For other storage providers, folders, servers or clusters have to be built before using them in the pod definition.

Different volume types have different storage providers:

Volume Type	**Storage Provider**
`emptyDir`	Local host
`hostPath`	Local host
`nfs`	NFS server
`iscsi`	iSCSI target provider
`flocker`	Flocker cluster
`glusterfs`	GlusterFS cluster
`rbd`	Ceph cluster
`gitRepo`	Git repository
`awsElasticBlockStore`	AWS EBS

Volume Type	Storage Provider
`gcePersistentDisk`	GCE persistent disk
`secret`	Kubernetes configuration file
`downwardAPI`	Kubernetes pod information

How to do it…

Volumes are defined in the volumes section of the pod definition with a unique name. Each type of volume has a different configuration to be set. Once you define the volumes, you can mount them in the `volumeMounts` section in container `spec`. `volumeMounts.name` and `volumeMounts.mountPath` are required, which indicate the name of the volumes you defined and the mount path inside the container.

We'll use the Kubernetes configuration file with the YAML format to create a pod with volumes in the following examples.

emptyDir

`emptyDir` is the simplest volume type, which will create an empty volume for containers in the same pod to share. When the pod is removed, the files in `emptyDir` will be erased as well. `emptyDir` is created when a pod is created. In the following configuration file, we'll create a pod running Ubuntu with commands to sleep for 3600 seconds. As you can see, one volume is defined in the volumes section with name `data`, and the volumes will be mounted under `/data-mount` path in the Ubuntu container:

```
// configuration file of emptyDir volume
# cat emptyDir.yaml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu
labels:
  name: ubuntu
spec:
  containers:
    -
      image: ubuntu
      command:
        - sleep
        - "3600"
      imagePullPolicy: IfNotPresent
      name: ubuntu
      volumeMounts:
        -
          mountPath: /data-mount
          name: data
  volumes:
    -
      name: data
      emptyDir: {}

// create pod by configuration file emptyDir.yaml
# kubectl create -f emptyDir.yaml
```

Tip

Check which node the pod is running on

By using the `kubectl describe pod <Pod name> | grep Node` command you could check which node the pod is running on.

After the pod is running, you could use `docker inspect <container ID>` on the target node and you could see the detailed mount points inside your container:

```
    "Mounts": [
        {
            "Source":
"/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~
empty-dir/data",
            "Destination": "/data-mount",
            "Mode": "",
            "RW": true
        },
       ...
     ]
```

Here, you can see Kubernetes simply create an empty folder with the path `/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~empty-dir/<volumeMount name>` for the pod to use. If you create a pod with more than one container, all of them will mount the same destination `/data-mount` with the same source.

`emptyDir` could be mounted as `tmpfs` if we set the `emptyDir.medium` setting to `Memory` in the previous configuration file `emptyDir.yaml`:

```
  volumes:
    -
      name: data
      emptyDir:
        medium: Memory
```

We could also check the `Volumes` information by `kubectl describe pods ubuntu` to see whether it's set successfully:

```
# kubectl describe pods ubuntu
Name:        ubuntu
Namespace:      default
Image(s):     ubuntu
Node:        ip-10-96-219-192/
Status:        Running
...
Volumes:
  data:
```

```
    Type:  EmptyDir (a temporary directory that
shares a pod's lifetime)
    Medium:  Memory
```

hostPath

`hostPath` acts as data volume in Docker. The local folder on a node listed in `hostPath` will be mounted into the pod. Since the pod can run on any nodes, read/write functions happening in the volume could explicitly exist in the node on which the pod is running. In Kubernetes, however, the pod should not be node-aware. Please note that configuration and files might be different on different nodes when using `hostPath`. Therefore, the same pod, created by same command or configuration file, might act differently on different nodes.

By using `hostPath`, you're able to read and write the files between containers and local host disks of nodes. What we need for volume definition is for `hostPath.path` to specify the target mounted folder on the node:

```
// configuration file of hostPath volume
# cat hostPath.yaml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu
spec:
  containers:
    -
      image: ubuntu
      command:
        - sleep
        - "3600"
      imagePullPolicy: IfNotPresent
      name: ubuntu
      volumeMounts:
        -
          mountPath: /data-mount
          name: data
  volumes:
```

```
-
  name: data
  hostPath:
    path: /target/path/on/host
```

Using `docker inspect` to check the volume details, you will see the volume on the host is mounted in `/data-mount` destination:

```
"Mounts": [
    {
        "Source": "/target/path/on/host",
        "Destination": "/data-mount",
        "Mode": "",
        "RW": true
    },
   ...
]
```

Tip

Touching a file to validate that the volume is mounted successfully

Using `kubectl exec <pod name> <command>` you could run the command inside a pod. In this case, if we run `kubectl exec ubuntu touch /data-mount/sample`, we should be able to see one empty file named `sample` under `/target/path/on/host`.

nfs

You can mount the **Network File System** (**NFS**) to your pod as a `nfs` volume. Multiple pods can mount and share the files in the same `nfs` volume. The data stored in the `nfs` volume will be persistent across the pod's lifetime. You have to create your own NFS server before using `nfs` volume, and make sure that the `nfs-utils` package is installed on the Kubernetes nodes.

Note

Checking that the nfs server works before you go

You should check out that the `/etc/exports` file has proper sharing parameters and directory, and is using the `mount -t nfs <nfs server>:<share name> <local mounted point>` command to check whether it could be mounted locally.

The configuration file of a volume type with `nfs` is similar to others, but the `nfs.server` and `nfs.path` are required in the volume definition to specify NFS server information, and the path mounting from. `nfs.readOnly` is an optional field for specifying whether the volume is read-only or not (default is false):

```
// configuration file of nfs volume
# cat nfs.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nfs
spec:
  containers:
    -
      name: nfs
      image: ubuntu
      volumeMounts:
          - name: nfs
            mountPath: "/data-mount"
  volumes:
  - name: nfs
    nfs:
      server: <your nfs server>
      path: "/"
```

After you run `kubectl create -f nfs.yaml`, you can describe your pod by using `kubectl describe <pod name>` to check the mounting

status. If it's mounted successfully, it should show conditions. It's ready if it shows `true` and the target `nfs` you have mounted:

```
Conditions:
  Type    Status
  Ready   True
Volumes:
  nfs:
    Type:  NFS (an NFS mount that lasts the
lifetime of a pod)
    Server:  <your nfs server>
    Path:  /
    ReadOnly:  false
```

If we inspect the container by using the Docker command, you will see the volume information in the `Mounts` section:

```
    "Mounts": [
 {
            "Source":
"/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~
nfs/nfs",
            "Destination": "/data-mount",
            "Mode": "",
            "RW": true
        },
       ...
     ]
```

Actually, Kubernetes just mounts your `<nfs server>:<share name>` into `/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~nfs/nfs` and then mounts it into a container as its destination in the `/data-mount`. You could also use `kubectl exec` to touch the file, as the previous tip mentions, to test whether it's perfectly mounted.

glusterfs

GlusterFS (https://www.gluster.org) is a scalable network-attached storage file system. The `glusterfs` volume type allows you mount

the GlusterFS volume into your pod. Just like NFS volume, the data in the GlusterFS volume is persistent across the pod's lifetime. If the pod is terminated, the data is still accessible in the GlusterFS volume. You should build a GlusterFS system before using a GlusterFS volume.

Note

Checking GlusterFS works before you go

By using `gluster volume info` on GlusterFS servers, you can see currently available volumes. By using `mount -t glusterfs <glusterfs server>:/<volume name> <local mounted point>` locally, you can check whether the GlusterFS system can be successfully mounted.

Since the volume replica in GlusterFS must be greater than 1, let's assume we have two replicas in servers `gfs1` and `gfs2` and the volume name is `gvol`.

First, we need to create an endpoint acting as a bridge for `gfs1` and `gfs2`:

```
# cat gfs-endpoint.yaml
kind: Endpoints
apiVersion: v1
metadata:
  name: glusterfs-cluster
subsets:
  -
    addresses:
      -
        ip: <gfs1 server ip>
    ports:
      -
        port: 1
  -
    addresses:
```

```
      -
        ip: <gfs2 server ip>
    ports:
      -
        port: 1

// create endpoints
# kubectl create -f gfs-endpoint.yaml
```

Then we could use `kubectl get endpoints` to check the endpoint is created properly:

```
# kubectl get endpoints
NAME                ENDPOINTS
AGE
glusterfs-cluster   <gfs1>:1,<gfs2>:1
12m
```

After that, we should be able to create the pod with the GlusterFS volume by `glusterfs.yaml`. The parameters of the `glusterfs` volume definition are `glusterfs.endpoints`, which specify the endpoint name we just created, and the `glusterfs.path` which is the volume name `gvol`. `glusterfs.readOnly` and is used to set whether the volume is mounted in read-only mode:

```
# cat glusterfs.yaml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu
spec:
  containers:
    -
      image: ubuntu
      command:
        - sleep
        - "3600"
      imagePullPolicy: IfNotPresent
      name: ubuntu
      volumeMounts:
        -
```

```
            mountPath: /data-mount
            name: data
    volumes:
      -
        name: data
        glusterfs:
          endpoints: glusterfs-cluster
          path: gvol
```

Let's check the volume setting by `kubectl describe`:

```
    Volumes:
      data:
        Type:    Glusterfs (a Glusterfs mount on the
    host that shares a pod's lifetime)
        EndpointsName:  glusterfs-cluster
        Path:    gvol
        ReadOnly:    false
```

Using `docker inspect` you should be able to see the mounted source is `/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~glusterfs/data` to the destination `/data-mount`.

iscsi

The `iscsi` volume is used to mount the existing iSCSI to your pod. Unlike `nfs` volume, the `iscsi` volume is only allowed to be mounted in a single container in read-write mode. The data will be persisted across the pod's lifecycle:

Field Name	Field Definition
`targetPortal`	IP Address of iSCSI target portal

Field Name	Field Definition
`Iqn`	IQN of the target portal
`Lun`	Target LUN for mounting
`fsType`	File system type on LUN
`readOnly`	Specify read-only or not, default is false

flocker

Flocker is an open-source container data volume manager. The `flocker` volume will be moved to the target node when the container moves. Before using Flocker with Kubernetes, the Flocker cluster (Flocker control service, Flocker dataset agent, Flocker container agent) is required. Flocker's official website (https://docs.clusterhq.com/en/1.8.0/install/index.html) has detailed installation instructions.

After you get your Flocker cluster ready, create a dataset and specify the dataset name in the Flocker volume definition in the Kubernetes configuration file:

Field Name	Field Definition
`datasetName`	Target dataset name in Flocker

rbd

Ceph RADOS Block Device (http://docs.ceph.com/docs/master/rbd/rbd/) could be mounted into your pod by using rbd volume. You need to install Ceph before using the rbd volume. The definition of rbd volume support is secret in order to keep authentication secrets:

Field Name	Field Definition	Default Value
`monitors`	Cepth monitors	
`pool`	The name of RADOS pool	`rbd`
`image`	The image name rbd created	
`user`	RADOS user name	`admin`
`keyring`	The path of keyring, will be overwritten if secret name is provided	`/etc/ceph/keyring`
`secretName`	Secret name	
`fsType`	File system type	

Field Name	Field Definition	Default Value
`readOnly`	Specify read-only or not	`False`

gitRepo

The `gitRepo` volume will mount as an empty dictionary and Git clone a repository with certain revision in a pod for you to use:

Field Name	Field Definition
`repository`	Your Git repository with SSH or HTTPS
`Revision`	The revision of repository
`readOnly`	Specify read-only or not

awsElasticBlockStore

`awsElasticBlockStore` volume mounts an AWS EBS volume into a pod. In order to use it, you have to have your pod running on AWS EC2 with the same availability zone with EBS. For now, EBS only supports attaching to an EC2 in nature, so it means you cannot attach a single EBS volume to multiple EC2 instances:

Field Name	Field Definition
`volumeID`	EBS volume info - `aws://<availability-zone>/<volume-id>`
`fsType`	File system type
`readOnly`	Specify read-only or not

gcePersistentDisk

Similar to `awsElasticBlockStore`, the pod using the `gcePersistentDisk` volume must be running on GCE with the same project and zone. The `gcePersistentDisk` supports only a single writer when `readOnly = false`:

Field Name	Field Definition
`pdName`	GCE persistent disk name
`fsType`	File system type
`readOnly`	Specify read-only or not

downwardAPI

`The downwardAPI` volume is a Kubernetes volume plugin with the ability to save some pod information in a plain text file into a container. The current supporting metadata of the `downwardAPI` volume is:

- metadata.annotations
- metadata.namespace
- metadata.name
- metadata.labels

The definition of the `downwardAPI` is a list of items. An item contains a `path` and `fieldRef`. Kubernetes will then dump the specified metadata listed in the `fieldRef` to a file named `path` under `mountPath` and mount the `<volume name>` into the destination you specified:

```
        {
            "Source":
"/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~
downward-api/<volume name>",
            "Destination": "/tmp",
            "Mode": "",
            "RW": true
        }
```

For the IP of the pod, using the environment variable to propagate in the pod spec would be much easier:

```
spec:
  containers:
    - name: envsample-pod-info
      env:
        - name: MY_POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
```

For more examples, look at the sample folder in Kubernetes GitHub (https://github.com/kubernetes/kubernetes/tree/master/docs/user-

guide/downward-api) which contains more examples for both environment variables and `downwardAPI` volume.

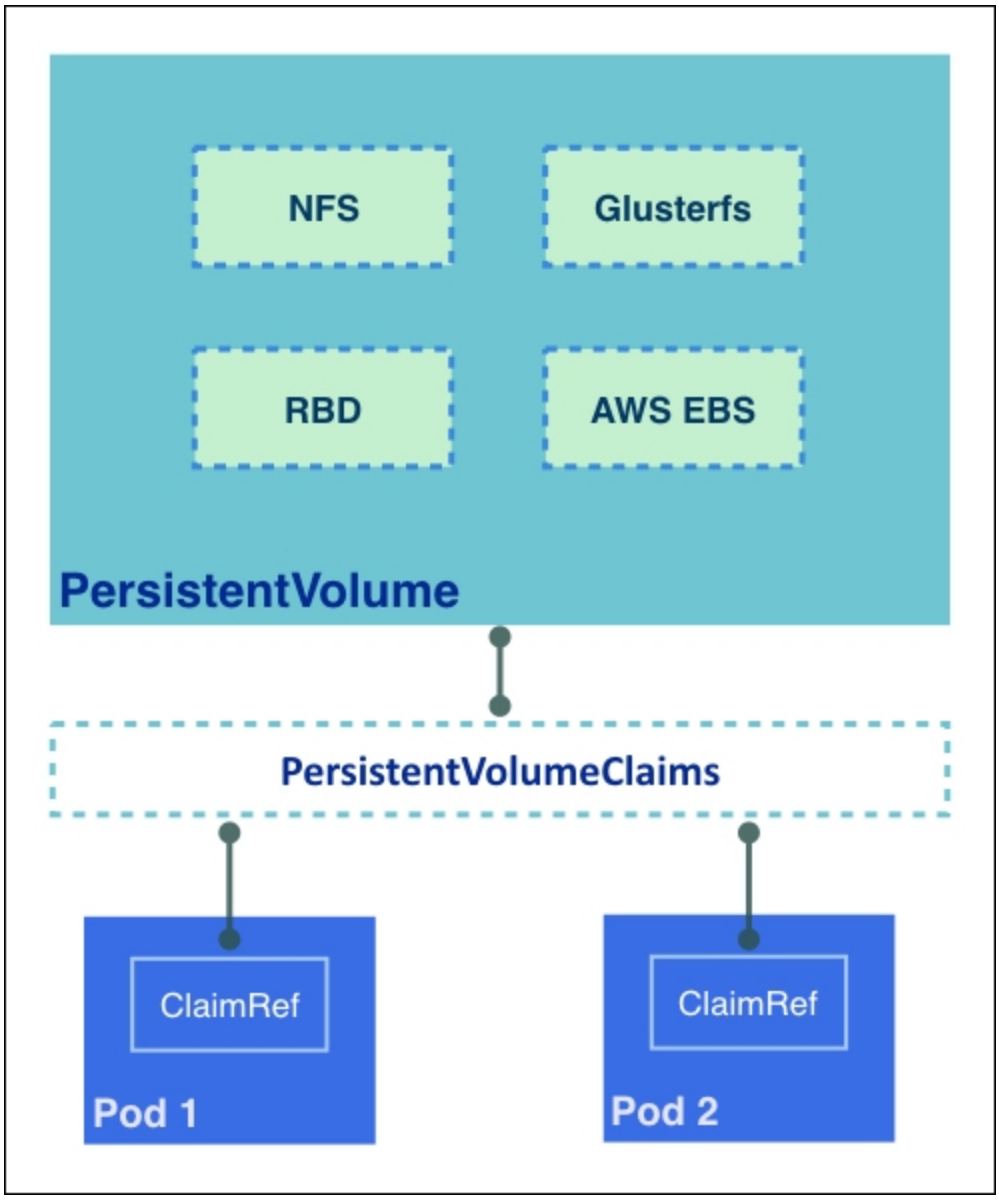
There's more…

In previous cases, the user needed to know the details of the storage provider. Kubernetes provides **PersistentVolume** (**PV**) to abstract the details of the storage provider and storage consumer. Kubernetes currently supports the PV types as follows:

- GCEPersistentDisk
- AWSElasticBlockStore
- NFS
- iSCSI
- RBD (Ceph Block Device)
- GlusterFS
- HostPath (not workable in multi-node cluster)

PersistentVolume

The illustration of persistent volume is shown in the following graph. At first, administrator provisions the specification of a `PersistentVolume`. Second, they provision consumer requests for storage by `PersistentVolumeClaim`. Finally, the pod mounts the volume by the reference of the `PersistentVolumeClaim`:



The administrator needs to provision and allocate the persistent volume first.

Here is an example using NFS:

```
// example of PV with NFS
# cat pv.yaml
  apiVersion: v1
  kind: PersistentVolume
  metadata:
    name: pvnfs01
```

```
  spec:
    capacity:
      storage: 3Gi
    accessModes:
      - ReadWriteOnce
    nfs:
      path: /
      server: <your nfs server>
    persistentVolumeReclaimPolicy: Recycle

// create the pv
# kubectl create -f pv.yaml
persistentvolume "pvnfs01" created
```

We can see there are three parameters here: `capacity`, `accessModes` and `persistentVolumeReclaimPolicy`. `capacity` is the size of this PV. `accessModes` is based on the capability of the storage provider, and can be set to a specific mode during provision. For example, NFS supports multiple readers and writers simultaneously, thus we could specify the `accessModes` as `ReadWriteOnce`, `ReadOnlyMany` or `ReadWriteMany`. The `accessModes` of one volume could be set to one mode at a time. `persistentVolumeReclaimPolicy` is used to define the behavior when PV is released. Currently, the supported policy is `Retain` and `Recycle` for `nfs` and `hostPath`. You have to clean the volume by yourself in `Retain` mode; on the other hand, Kubernetes will scrub the volume in `Recycle` mode.

PV is a resource like node. We could use `kubectl get pv` to see current provisioned PVs:

```
// list current PVs
# kubectl get pv
NAME      LABELS    CAPACITY   ACCESSMODES
STATUS    CLAIM               REASON    AGE
pvnfs01   <none>    3Gi         RWO           Bound
default/pvclaim01             37m
```

Next, we will need to bind `PersistentVolume` with `PersistentVolumeClaim` in order to mount it as a volume into the pod:

```
// example of PersistentVolumeClaim
# cat claim.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvclaim01
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi

// create the claim
# kubectl create -f claim.yaml
persistentvolumeclaim "pvclaim01" created

// list the PersistentVolumeClaim (pvc)
# kubectl get pvc
NAME         LABELS    STATUS    VOLUME    CAPACITY
ACCESSMODES   AGE
pvclaim01    <none>    Bound     pvnfs01   3Gi
RWO           59m
```

The constraints of `accessModes` and `storage` could be set in the `PersistentVolumeClaim`. If the claim is bound successfully, its status will turn to `Bound`; conversely, if the status is `Unbound`, it means that currently no PV matches the requests.

Then we are able to mount the PV as a volume by using `PersistentVolumeClaim`:

```
// example of mounting into Pod
# cat nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    project: pilot
    environment: staging
    tier: frontend
```

```
spec:
  containers:
    -
      image: nginx
      imagePullPolicy: IfNotPresent
      name: nginx
      volumeMounts:
      - name: pv
        mountPath: "/usr/share/nginx/html"
      ports:
      - containerPort: 80
  volumes:
    - name: pv
      persistentVolumeClaim:
        claimName: "pvclaim01"

// create the pod
# kubectl create -f nginx.yaml
pod "nginx" created
```

The syntax is similar to the other volume type. Just add the `claimName` of the `persistentVolumeClaim` in the volume definition. We are all set! Let's check the details to see whether we have mounted it successfully:

```
// check the details of a pod
# kubectl describe pod nginx
...
Volumes:
  pv:
    Type:  PersistentVolumeClaim (a reference to a
PersistentVolumeClaim in the same namespace)
    ClaimName:  pvclaim01
    ReadOnly:  false
...
```

We can see that we have a volume mounted in the pod nginx with type `pv` `pvclaim01`. Use `docker inspect` to see how it is mounted:

```
        "Mounts": [
            {
                "Source":
```

```
"/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~
nfs/pvnfs01",
            "Destination":
"/usr/share/nginx/html",
            "Mode": "",
            "RW": true
        },
      ...
    ]
```

Kubernetes mounts `/var/lib/kubelet/pods/<id>/volumes/kubernetes.io~nfs/<persistentvolume name>` into the destination in the pod.

See also

Volumes are put in container specs in pods or replication controllers. Check out the following recipes to jog your memory:

- *Working with pods*
- *Working with a replication controller*

Working with secrets

Kubernetes secrets manage information in key-value formats with the value encoded. With secrets, users don't have to set values in the configuration file or type them in CLI. When secrets are used properly, they can reduce the risk of credential leak and make our resource configurations more organized.

Currently, there are three types of secret:

- Opaque: https://en.wikipedia.org/wiki/Opaque_data_type
- Service account token
- Docker authentication

Opaque is the default type. We will put service account tokens and the authentication of Docker in the remark part.

Getting ready

Before using our credentials with secrets, some precautions must be taken First, secrets have a 1 MB size limitation. It works fine for defining several key-value pairs in a single secret. But, be aware that the total size should not exceed 1 MB. Next, secret acts like a volume for containers, so secrets should be created prior to dependent pods.

How to do it…

We can only generate secrets by using configuration files. In this recipe, we will deliver a simple template file and focus on the functionality. For various template designs, please take a look at the *Working with configuration files* recipe in Chapter 3, *Playing with Containers*.

Creating a secret

The configuration file of secrets contains secret type and data:

```
// A simple file for configuring secret
# cat secret-test.json
{
  "kind": "Secret",
  "apiVersion": "v1",
  "metadata": {
    "name": "secret-test"
  },
  "type": "Opaque",
  "data": {
    "username": "YW15Cg==",
    "password": " UGEkJHcwcmQhCg=="
  }
}
```

The secret type `Opaque` is the default one, which simply indicates that the data is not shown. The other types, service account token and Docker authentication, are applied when using the values `kubernetes.io/service-account-token` and `kubernetes.io/dockercfg` at the type item stage, respectively.

The data `username` and `password` are customized keys. Their corresponding values are base64-encoded string. You can get your encoded value through these pipe commands:

```
# echo "amy" | base64
YW15Cg==
```

The resource annotation and management of secrets is similar to other resource types. Feel free to create a secret and check its status by using common subcommands:

```
# kubectl create -f secret-test.json
secret "secret-test" created
# kubectl get secret
NAME          TYPE      DATA      AGE
```

```
secret-test   Opaque    2         39s
# kubectl describe secret secret-test
Name:    secret-test
Namespace:  default
Labels:    <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  10 bytes
username:  4 bytes
```

As you can see, although secret hides the information, we can get the amount of data, the data name and also the size of the value.

Picking up secret in the container

In order to let the pod get the secret information, secret data is mounted as a file in the container. The key-value pair data will be shown in plain-text file format, which takes a key name as the file name and the decoded value as the file content. Therefore, we create the pod by configuration file, where the container's mounting volume is pointed to secret:

```
# cat pod-secret.json
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "pod-with-secret"
  },
  "spec": {
    "volumes": [
      {
        "name": "secret-volume",
        "secret": {
          "secretName": "secret-test"
        }
      }
    ],
```

```
    "containers": [
      {
  "name": "secret-test-pod",
        "image": "nginx",
        "volumeMounts": [
          {
            "name": "secret-volume",
            "readOnly": true,
            "mountPath": "/tmp/secret-volume"
          }
        ]
      }
    ]
  }
}
```

For the previous template, we defined a volume called `secret-volume` which includes physical files with the content of the secret `secret-test`; the containers' mounting point is also defined along with the location, where to put secret files, and bound with `secret-volume`. In this case, the container could access secrets in its local file system by using `/tmp/secrets/<SECRET_KEY>`.

To verify the content is decrypted for the usage of the container program, let's take a look at the specific container on the node:

```
// login to node and enable bash process with new
tty
# docker exec -it <CONTAINER_ID> bash
root@pod-with-secret:/# ls /tmp/secrets/
password  username
root@pod-with-secret:/# cat /tmp/secrets/password
Pa$$w0rd!
root@pod-with-secret:/# cat /tmp/secrets/username
amy
```

Deleting a secret

Secret, like other resources, can be stopped by the subcommand `delete`. Both methods, deleting according to configuration file or deleting by resource name are workable:

```
# kubectl delete -f secret-test.json
secret "secret-test" deleted
```

How it works…

In order to reduce the risk of leaking the secrets' content, the Kubernetes system never saves the data of secrets on disk. Instead, secrets are stored in the memory. For a more accurate statement, the Kubernetes API server pushes secret to the node on which the demanded container is running. The node stores the data in `tmpfs`, which will be flashed if the container is destroyed.

Go and check the node, which has container with secrets running on it:

```
// check the disk
df -h --type=tmpfs
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           920M     0  920M   0% /dev/shm
tmpfs           920M   17M  903M   2% /run
tmpfs           920M     0  920M   0%
/sys/fs/cgroup
tmpfs           184M     0  184M   0%
/run/user/2007
tmpfs           920M  8.0K  920M   1%
/var/lib/kubelet/pods/2edd4eb4-b39e-11e5-9663-
0200e755981f/volumes/kubernetes.io~secret/secret-
volume
```

Furthermore, I suggest that you avoid creating a large-size secret or many small-size secrets. Since secrets are kept in the memory of nodes, reducing the total size of secrets could help to save resources and maintain good performance.

There's more…

In the previous sections, secret is configured in the default service account. The service account can make processes in containers in

contact with the API server. You could have different authentication by creating different service accounts.

Let's see how many service accounts we currently have:

```
$ kubectl get serviceaccounts
NAME          SECRETS   AGE
default       0         18d
```

Kubernetes will create a default service account. Let's see how to create our own one:

```
# example of service account creation
$ cat serviceaccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: test-account

# create service account named test-account
$ kubectl create -f serviceaccount.yaml
serviceaccount "test-account" created
```

After creation, let's list the accounts by `kubectl`:

```
$ kubectl get serviceaccounts
NAME           SECRETS   AGE
default        0         18d
test-account   0         37s
```

We can see there is a new service account named `test-account` in the list now.

Each service account could have its own API token, image pull secrets and mountable secrets.

Similarly, we could delete the service account by using `kubectl`:

```
$ kubectl delete serviceaccount test-account
serviceaccount "test-account" deleted
```

On the other hand, Docker authentication can also be saved as a secret data for pulling images. We will discuss the usage in *Working with the private Docker registry* recipe in Chapter 5, *Building a Continuous Delivery Pipeline*.

See also

- The *Working with configuration files* recipe in Chapter 3, *Playing with Containers*
- The *Moving monolithic to microservices*, *Working with the private Docker registry* recipes in Chapter 5, *Building a Continuous Delivery Pipeline*
- The *Advanced settings in kubeconfig* recipe in Chapter 7, *Advanced Cluster Administration*

Working with names

When you create any Kubernetes objects such as a pod, replication controller and service, you can assign a name to it. The names in Kubernetes are spatially unique, which means you cannot assign the same name in the pods.

Getting ready

Kubernetes allows us to assign a name with the following restrictions:

- Up to 253 characters
- Lowercase of alphabet and numeric characters
- May contain special characters in the middle but only dash (-) and dot (.)

How to do it…

The following example is the pod definition that assigns the pod name as `my-pod`, to the container name as `my-container`, you can successfully create it as follows:

```
# cat my-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx

# kubectl create -f my-pod.yaml
pod "my-pod" created

# kubectl get pods
```

```
NAME      READY     STATUS    RESTARTS   AGE
my-pod    0/1       Running   0          4s
```

You can use the `kubectl describe` command to see the container name `my-container` as follows:

```
# kubectl describe pod my-pod
Name:         my-pod
Namespace:       default
Image(s):       nginx
Node:         ip-10-96-219-25/10.96.219.25
Start Time:       Wed, 16 Dec 2015 00:46:33 +0000
Labels:         <none>
Status:         Running
Reason:
Message:
IP:         192.168.34.35
Replication Controllers:  <none>
Containers:
  my-container:
    Container ID:
docker://5501d115703e334ae44c1541b990a7e22ce4f3102
26eafea206594e4c85c90d9
    Image:    nginx
    Image ID:
docker://6ffc02088cb870652eca9ccd4c4fb582f75b29af2
879792ed09bb46fd1c898ef
    State:    Running
      Started:    Wed, 16 Dec 2015 00:46:34 +0000
    Ready:    True
    Restart Count:  0
    Environment Variables:
```

On the other hand, the following example contains two containers, but assigns the same name as `my-container`, therefore the `kubectl` command returns an error and can't create the pod.

```
//delete previous pods
# kubectl delete pods --all
pod "my-pod" deleted
# cat duplicate.yaml
apiVersion: v1
```

```
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx
  - name: my-container
    image: centos
    command: ["/bin/sh", "-c", "while : ;do curl
http://localhost:80/; sleep 3; done"]
# kubectl create -f duplicate.yaml
The Pod "my-pod" is invalid.
spec.containers[1].name: duplicate value 'my-
container'
```

Tip

You can add the `-validate` flag

For example: `kubectl create -f duplicate.yaml -validate`

Use a schema to validate the input before sending it

In another example, the YAML contains a replication controller and service, both of which are using the same name `my-nginx`, but it is successfully created because the replication controller and service are different:

```
# cat nginx.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  selector:
      sel : my-selector
  template:
    metadata:
        labels:
```

```
          sel : my-selector
    spec:
      containers:
      - name: my-container
        image: nginx
---
apiVersion: v1
kind: Service
metadata:
  name: my-nginx

spec:
  ports:
    - protocol: TCP
      port: 80
      nodePort: 30080
  type: NodePort
  selector:
     sel: my-selector

# kubectl create -f nginx.yaml
replicationcontroller "my-nginx" created
service "my-nginx" created

# kubectl get rc
CONTROLLER   CONTAINER(S)    IMAGE(S)    SELECTOR
REPLICAS   AGE
my-nginx     my-container   nginx       sel=my-
selector   2           8s

# kubectl get service
NAME         CLUSTER_IP       EXTERNAL_IP
PORT(S)    SELECTOR          AGE
kubernetes   192.168.0.1      <none>
443/TCP   <none>            6d
my-nginx     192.168.38.134   nodes         80/TCP
sel=my-selector   14s
```

How it works…

Name is just a unique identifier, all naming conventions are good, however it is recommended to look up and identify the container image. For example:

- memcached-pod1

- `haproxy.us-west`
- `my-project1.mysql`

On the other hand, the following examples do not work because of Kubernetes restrictions:

- Memcache-pod1 (contains uppercase)
- haproxy.us_west (contains underscore)
- my-project1.mysql. (dot in the last)

Note that Kubernetes supports a label that allows to assign a key=value style identifier. It also allows duplication. Therefore, if you want to assign something like the information below, use a label instead:

- environment (for example: staging, production)
- version (for example: v1.2)
- application role (for example: frontend, worker)

In addition, Kubernetes also supports namespaces which have isolated namespaces. This means that you can use the same name in different namespaces (for example: nginx). Therefore, if you want to assign just an application name, use namespaces instead.

See also

This section described how to assign and find the name of objects. This is just a basic methodology, but Kubernetes has more powerful naming tools such as namespace and selectors to manage clusters:

- *Working with pods*
- *Working with a replication controller*
- *Working with services*
- *Working with namespaces*
- *Working with labels and selectors*

Working with namespaces

The name of a resource is a unique identifier within a namespace in the Kubernetes cluster. Using a Kubernetes namespace could isolate namespaces for different environments in the same cluster. It gives you the flexibility of creating an isolated environment and partitioning resources to different projects and teams.

Pods, services, replication controllers are contained in a certain namespace. Some resources, such as nodes and PVs, do not belong to any namespace.

Getting ready

By default, Kubernetes has created a namespace named `default`. All the objects created without specifying namespaces will be put into default namespaces. You could use `kubectl` to list namespaces:

```
// check all namespaces
# kubectl get namespaces
NAME      LABELS    STATUS    AGE
default   <none>    Active    8d
```

Kubernetes will also create another initial namespace called `kube-system` for locating Kubernetes system objects, such as a Kubernetes UI pod.

The name of a namespace must be a DNS label and follow the following rules:

- At most 63 characters
- Matching regex [a-z0-9]([-a-z0-9]*[a-z0-9])

How to do it…

1. After selecting our desired name, let's create a namespace named `new-namespace` by using the configuration file:

```
# cat newNamespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: new-namespace

// create the resource by kubectl
# kubectl create -f newNamespace.yaml
```

2. After the namespace is created successfully, list the namespace again:

```
// list namespaces
# kubectl get namespaces
NAME            LABELS    STATUS    AGE
default         <none>    Active    8d
new-namespace   <none>    Active    12m
```

   You can see now that we have two namespaces.

3. Let's run the nginx replication controller described in Chapter 1, *Building Your Own Kubernetes* in a new namespace:

```
// run a nginx RC in namespace=new-namespace
# kubectl run nginx --image=nginx --
namespace=new-namespace
```

4. Then let's list the pods:

```
# kubectl get pods
NAME                                    READY
STATUS        RESTARTS   AGE
```

5. There are no pods running! Let's run again with the `--namespace` parameter:

```
// to list pods in all namespaces
```

```
# kubectl get pods --all-namespaces
NAMESPACE      NAME           READY     STATUS
RESTARTS   AGE
new-namespace  nginx-ns0ig    1/1       Running
0          17m

// to get pods from new-namespace
# kubectl get pods --namespace=new-namespace
NAME           READY     STATUS    RESTARTS
AGE
nginx-ns0ig    1/1       Running   0
18m
```

We can see our pods now.

6. By default, if you don't specify any namespace in the command line, Kubernetes will create the resources in the default namespace. If you want to create resources by configuration file, just simply specify it when doing `kubectl create`:

```
# kubectl create -f myResource.yaml --
namespace=new-namespace
```

Changing the default namespace

It is possible to switch the default namespace in Kubernetes:

1. Find your current context:

```
# kubectl config view | grep current-context
current-context: ""
```

It reveals that we don't have any context setting now.

2. No matter whether there is current context or not, using `set-context` could create a new one or overwrite the existing one:

```
# kubectl config set-context <current context
or new context name> --namespace=new-namespace
```

3. After setting the context with a new namespace, we can check the current configuration:

```
# kubectl config view
apiVersion: v1
clusters: []
contexts:
- context:
    cluster: ""
    namespace: new-namespace
    user: ""
  name: new-context
current-context: ""
kind: Config
preferences: {}
users: []
```

   We can see the namespace is set properly in the contexts section.

4. Switch the context to the one we just created:

```
# kubectl config use-context new-context
```

5. Then check the current context again:

```
# kubectl config view | grep current-context
current-context: new-context
```

   We can see that `current-context` is `new-context` now.

6. Let's list the current pods again. There's no need to specify the `Namespace` parameter, as we can list the pods in `new-namespace`:

```
# kubectl get pods
NAME          READY     STATUS    RESTARTS
AGE
nginx-ns0ig   1/1       Running   0
54m
```

7. Namespace is listed in the pod description as well:

```
# kubectl describe pod nginx-ns0ig
Name:        nginx-ns0ig
Namespace:      new-namespace
Image(s):     nginx
Node:        ip-10-96-219-156/10.96.219.156
Start Time:      Sun, 20 Dec 2015 15:03:40
+0000
Labels:        run=nginx
Status:        Running
```

Deleting a namespace

1. Using `kubectl delete` could delete the resources including the namespace. Deleting a namespace will erase all the resources under that namespace:

```
# kubectl delete namespaces new-namespace
namespace "new-namespace" deleted
```

2. After the namespace is deleted, our nginx pod is gone:

```
# kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
```

3. However, the default namespace in the context is still set as `new-namespace`:

```
# kubectl config view | grep current-context
current-context: new-context
```

   Will it be a problem?

4. Let's run an `nginx` replication controller again.

```
# kubectl run nginx --image=nginx
```

```
Error from server: namespaces "new-namespace"
not found
```

It will try to create an `nginx` replication controller and replica pod in the current namespace we just deleted. Kubernetes will throw out an error if the namespace is not found.

5. Let's switch back to the default namespace.

```
# kubectl config set-context new-context --
namespace=""
context "new-context" set.
```

6. Let's run an `nginx` again.

```
# kubectl run nginx --image=nginx
replicationcontroller "nginx" created
Does it real run in default namespace? Let's
describe the pod.
# kubectl describe pods nginx-ymqeh
Name:        nginx-ymqeh
Namespace:      default
Image(s):      nginx
Node:        ip-10-96-219-156/10.96.219.156
Start Time:      Sun, 20 Dec 2015 16:13:33
+0000
Labels:        run=nginx
Status:        Running
...
```

We can see the pod is currently running in `Namespace: default`. Everything looks fine.

There's more…

Sometimes you'll need to limit the resource quota for each team by distinguishing the namespace. After you create a new namespace, the details look like this:

```
$ kubectl describe namespaces new-namespace
Name:  new-namespace
```

```
Labels:  <none>
Status:  Active

No resource quota.

No resource limits.
```

Resource quota and limits are not set by default. Kubernetes supports constraint for a container or pod. `LimitRanger` in the Kubernetes API server has to be enabled before setting the constraint. You could either use a command line or configuration file to enable it:

```
// using command line-
# kube-apiserver --admission-control=LimitRanger

// using configuration file
# cat /etc/kubernetes/apiserver
...
# default admission control policies
KUBE_ADMISSION_CONTROL="--
admission_control=NamespaceLifecycle,NamespaceExis
ts,LimitRanger,SecurityContextDeny,ResourceQuota"
...
```

The following is a good example for creating a limit in a namespace.

We will then limit the resources in a pod with the values `2` as `max` and `200m` as `min` for `cpu`, and `1Gi` as `max` and `6Mi` as `min` for `memory`. For the container, the `cpu` is limited between `100m` - `2` and the memory is between `3Mi` - `1Gi`. If the `max` is set, then you have to specify the limit in the pod/container spec during the resource creation; if the `min` is set then the request has to be specified during the pod/container creation. The `default` and `defaultRequest` section in `LimitRange` is used to specify the default limit and request in the container spec:

```
# cat limits.yaml
apiVersion: v1
kind: LimitRange
```

```
metadata:
  name: limits
  namespace: new-namespace
spec:
  limits:
  - max:
      cpu: "2"
      memory: 1Gi
    min:
      cpu: 200m
      memory: 6Mi
    type: Pod
  - default:
      cpu: 300m
      memory: 200Mi
    defaultRequest:
      cpu: 200m
      memory: 100Mi
    max:
      cpu: "2"
      memory: 1Gi
    min:
      cpu: 100m
      memory: 3Mi
    type: Container

// create LimitRange
# kubectl create -f limits.yaml
limitrange "limits" created
```

After the `LimitRange` is created, we can list these down just like with any other resource:

```
// list LimitRange
# kubectl get LimitRange --namespace=new-namespace
NAME      AGE
limits    22m
```

When you describe the new namespace you will now be able to see the constraint:

```
# kubectl describe namespace new-namespace
```

```
Name:  new-namespace
Labels:  <none>
Status:  Active

No resource quota.

Resource Limits
 Type      Resource  Min   Max   Request  Limit
Limit/Request
 ----      --------  ---   ---   -------  -----
-------------
 Pod       memory    6Mi   1Gi   -        -
-
 Pod       cpu       200m  2     -        -
-
 Container  cpu       100m  2     200m     300m
-
 Container  memory    3Mi   1Gi   100Mi
200Mi   -
```

All the pods and containers created in this namespace have to follow the resource limits listed here. If the definitions violate the rule, a validation error will be thrown accordingly.

Deleting LimitRange

We could delete the `LimitRange` resource via:

```
# kubectl delete LimitRange <limit name> --
namespace=<namespace>
```

Here, the limit name is `limits` and the namespace is `new-namespace`. After that when you describe the namespace, the constraint is gone:

```
# kubectl describe namespace <namespace>
Name:  new-namespace
Labels:  <none>
Status:  Active

No resource quota.
```

```
No resource limits.
```

See also

Many resources are running under a namespace, check out the following recipes:

- *Working with pods*
- *Working with names*
- The *Setting resource in nodes* recipe in Chapter 7, *Advanced Cluster Administration*

Working with labels and selectors

Labels are a set of key/value pairs, which are attached to object metadata. We could use labels to select, organize and group objects, such as pods, replication controllers and services. Labels are not necessarily unique. Objects could carry the same set of labels.

Label selectors are used to query objects via labels. Current supported selector types are:

- Equality-based label selector
- Set-based label selector
- Empty label selector
- Null label selector

An equality-based label selector is a set of equality requirements, which could filter labels by equal or non-equal operation. A set-based label selector is used to filter labels by a set of values, and currently supports `in` and `notin` operators. When a label value matches the values in the `in` operator, it will be returned by the selector; conversely, if a label value does not match the values in the `notin` operator, it will be returned. Empty label selectors select all objects and null labels select no objects. Selectors are combinable. Kubernetes will return the objects that match all the requirements in selectors.

Getting ready

Before you get to set labels into the objects, you should consider the valid naming convention of key and value.

A valid key should follow these rules:

- A name with an optional prefix, separated by a slash.
- A prefix must be a DNS subdomain, separated by dots, no longer than 253 characters.
- A name must be less than 63 characters with the combination of [a-z0-9A-Z] and dashes, underscores and dots. Note that symbols are illegal if put at the beginning and the end.

A valid value should follow the following rules:

- A name must be less than 63 characters with the combination of [a-z0-9A-Z] and dashes, underscores and dots. Note that symbols are illegal if put at the beginning and the end.

You should also consider the purpose, too. For example, we have a service in the pilot project under different development environments which contain multiple tiers. Then we could make our labels:

- `project: pilot`
- `environment: development`, `environment: staging`, `environment: production`
- `tier: frontend`, `tier: backend`

How to do it…

Let's try to create an nginx pod with the previous labels in both a staging and production environment:

1. We will create the same staging for pod and production as that for the replication controller (RC):

```
# cat staging-nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    project: pilot
    environment: staging
    tier: frontend
spec:
```

```
  containers:
    -
      image: nginx
      imagePullPolicy: IfNotPresent
      name: nginx
      ports:
      - containerPort: 80

// create the pod via configuration file
# kubectl create -f staging-nginx.yaml
pod "nginx" created
```

2. Let's see the details of the pod:

```
# kubectl describe pod nginx
Name:         nginx
Namespace:       default
Image(s):       nginx
Node:         ip-10-96-219-231/
Start Time:       Sun, 27 Dec 2015 18:12:31
+0000
Labels:
environment=staging,project=pilot,tier=frotend
Status:         Running
...
```

   We could then see the labels in the pod description as `environment=staging`,`project=pilot`,`tier=frontend`.

   Good. We have a staging pod now.

3. Now, get on with creating the RC for a production environment by using the command line:

```
$ kubectl run nginx-prod --image=nginx --
replicas=2 --port=80 --
labels="environment=production,project=pilot,t
ier=frontend"
```

   This will then create an RC named `nginx-prod` with two replicas, an opened port `80`, and with the labels `environment=production`,`project=pilot`,`tier=frontend`.

4. We can see that we currently have a total three pods here. One pod is created for staging, the other two are for production:

```
# kubectl get pods
NAME                READY     STATUS
RESTARTS   AGE
nginx               1/1       Running   0
8s
nginx-prod-50345    1/1       Running   0
19s
nginx-prod-pilb4    1/1       Running   0
19s
```

5. Let's get some filters for the selecting pods. For example, if I wanted to select production pods in the pilot project:

```
# kubectl get pods -l
"project=pilot,environment=production"
NAME                READY     STATUS
RESTARTS   AGE
nginx-prod-50345    1/1       Running   0
9m
nginx-prod-pilb4    1/1       Running   0
9m
```

   By adding `-l` followed by key/value pairs as filter requirements, we could see the desired pods.

Linking service with a replication controller by using label selectors

Service in Kubernetes is used to expose the port and for load-balancing:

1. In some cases, you'll need to add a service in front of the replication controller in order to expose the port to the outside world or balance the load. We will use the configuration file to create services for the staging pod and command line for production pods in the following example:

```
// example of exposing staging pod
# cat staging-nginx-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    project: pilot
    environment: staging
    tier: frontend
spec:
  ports:
    -
      protocol: TCP
      port: 80
      targetPort: 80
  selector:
    project: pilot
    environment: staging
    tier: frontend
  type: LoadBalancer
// create the service by configuration file
# kubectl create -f staging-nginx-service.yaml
service "nginx" created
```

2. Using `kubectl describe` to describe the details of the service:

```
// describe service
# kubectl describe service nginx
Name:      nginx
Namespace:    default
Labels:
environment=staging,project=pilot,tier=fronten
d
Selector:
environment=staging,project=pilot,tier=fronten
d
Type:      LoadBalancer
IP:         192.168.167.68
Port:      <unnamed>  80/TCP
Endpoints:    192.168.80.28:80
Session Affinity:  None
No events.
```

Using `curl` for the ClusterIP could return the welcome page of nginx.

3. Next, let's add the service for RC with label selectors:

```
// add service for nginx-prod RC
# kubectl expose rc nginx-prod --port=80 --
type=LoadBalancer --
selector="project=pilot,environment=production
,tier=frontend"
```

4. Using `kubectl describe` to describe the details of service:

```
# kubectl describe service nginx-prod
Name:       nginx-prod
Namespace:    default
Labels:
environment=production,project=pilot,tier=fron
tend
Selector:
environment=production,project=pilot,tier=fron
tend
Type:       LoadBalancer
IP:       192.168.200.173
Port:       <unnamed>  80/TCP
NodePort:    <unnamed>  32336/TCP
Endpoints:
192.168.80.31:80,192.168.80.32:80
Session Affinity:  None
No events.
```

When we use `curl 192.168.200.173`, we can see the welcome page of nginx just like the staging one.

> **Note**
>
> It will return a `Connection reset by peer` error if you specify the empty pod set by the selector.

There's more…

In some cases, we might want to tag the resources with some values just for reference in the programs or tools. The non-identifying tags could use annotations instead, which are able to use structured or unstructured data. Unlike labels, annotations are not for querying and selecting. The following example will show you how to add an annotation into a pod and how to leverage them inside the container by downward API:

```
# cat annotation-sample.yaml
apiVersion: v1
kind: Pod
metadata:
  name: annotation-sample
  labels:
    project: pilot
    environment: staging
  annotations:
    git: 6328af0064b3db8b913bc613876a97187afe8e19
    build: "20"
spec:
  containers:
    -
      image: busybox
      imagePullPolicy: IfNotPresent
      name: busybox
      command: ["sleep", "3600"]
```

You could then use downward API, which we discussed in volumes, to access annotations in containers:

```
# cat annotation-sample-downward.yaml
apiVersion: v1
kind: Pod
metadata:
  name: annotation-sample
  labels:
    project: pilot
    environment: staging
  annotations:
    git: 6328af0064b3db8b913bc613876a97187afe8e19
    build: "20"
```

```
spec:
  containers:
    -
      image: busybox
      imagePullPolicy: IfNotPresent
      name: busybox
      command: ["sh", "-c", "while true; do if [[
-e /etc/annotations ]]; then cat /etc/annotations;
fi; sleep 5; done"]
      volumeMounts:
      - name: podinfo
        mountPath: /etc
volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
```

In this way, `metadata.annotations` will be exposed in the container as a file format under `/etc/annotations`. We could also check the pod logs are printing out the file content into stdout:

```
// check the logs we print in command section
# kubectl logs -f annotation-sample
build="20"
git="6328af0064b3db8b913bc613876a97187afe8e19"
kubernetes.io/config.seen="2015-12-
28T12:23:33.154911056Z"
kubernetes.io/config.source="api"
```

See also

You can practice labels and selectors through the following recipes:

- *Working with pods*
- *Working with a replication controller*
- *Working with services*
- *Working with volumes*
- The *Working with configuration files* recipe in Chapter 3, *Playing with Containers*

Chapter 3. Playing with Containers

In this chapter, we will cover the following topics:

- Scaling your containers
- Updating live containers
- Forwarding container ports
- Ensuring flexible usage of your containers
- Working with configuration files

Introduction

Talking about container management, you need to know some differences to compare it with application package management, such as rpm/dpkg, because you can run multiple containers on the same machine. You also need to care of the network port conflicts. This chapter covers how to update, scale, and launch the container application using Kubernetes.

Scaling your containers

Kubernetes has a scheduler to assign the container to the right node. In addition, you can easily scale out and scale down the number of containers. The Kubernetes scaling function will conduct the replication controller to adjust the number of containers.

Getting ready

Prepare the following YAML file, which is a simple replication controller to launch two nginx containers. Also, service will expose the TCP port `30080`:

```
# cat nginx-rc-svc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  selector:
      sel : my-selector
  template:
    metadata:
        labels:
          sel : my-selector
    spec:
      containers:
      - name: my-container
        image: nginx
---
apiVersion: v1
kind: Service
metadata:
name: my-nginx

spec:
  ports:
    - protocol: TCP
      port: 80
```

```
    nodePort: 30080
  type: NodePort
  selector:
    sel: my-selector
```

> **Tip**
>
> `NodePort` will bind all the Kubernetes nodes; therefore, make sure `NodePort` is not used by other processes.

Use the `kubectl` command to create resources as follows:

```
# kubectl create -f nginx-service.yaml
replicationcontroller "my-nginx" created
service "my-nginx" created
```

Wait for a moment to completely launch two nginx containers as follows:

```
# kubectl get pods
NAME             READY     STATUS    RESTARTS
AGE
my-nginx-iarzy   1/1       Running   0          7m
my-nginx-ulkvh   1/1       Running   0          7m

# kubectl get services
NAME         CLUSTER_IP       EXTERNAL_IP
PORT(S)    SELECTOR          AGE
kubernetes   192.168.0.1      <none>
443/TCP    <none>            44d
my-nginx     192.168.95.244   nodes          80/TCP
sel=my-selector   7m
```

How to do it…

Kubernetes has a command that changes the number of replicas for service:

1. Type the `kubectl scale` command as follows to specify the desired replicas:

```
# kubectl scale --replicas=4 rc my-nginx
replicationcontroller "my-nginx" scaled
```

   This example indicates that the replication controller, which is named `my-nginx`, changes the replicas to `4`.

2. Type `kubectl get pods` to confirm the result as follows:

```
# kubectl get pods
NAME              READY     STATUS    RESTARTS
AGE
my-nginx-iarzy    1/1       Running   0
20m
my-nginx-r5lnq    1/1       Running   0
1m
my-nginx-uhe8r    1/1       Running   0
1m
my-nginx-ulkvh    1/1       Running   0
20m
```

How it works…

The `kubectl scale` feature can change the number of replicas; not only increase, but also decrease. For example, you can change back to two replicas as follows:

```
# kubectl scale --replicas=2 rc my-nginx
replicationcontroller "my-nginx" scaled

# kubectl get pods
NAME              READY     STATUS        RESTARTS
AGE
my-nginx-iarzy    0/1       Terminating   0
40m
my-nginx-r5lnq    1/1       Running       0
21m
my-nginx-uhe8r    1/1       Running       0
```

```
21m
my-nginx-ulkvh    0/1       Terminating   0
40m
# kubectl get pods
NAME              READY     STATUS    RESTARTS
AGE
my-nginx-r5lnq    1/1       Running   0
25m
my-nginx-uhe8r    1/1       Running   0
25m
```

There is an option `--current-replicas` that specifies the expected current replicas. If it doesn't match, Kubernetes doesn't perform the scale function as follows:

```
//abort scaling, because current replica is 2, not
3
# kubectl scale --current-replicas=3 --replicas=4
rc my-nginx
Expected replicas to be 3, was 2

# kubectl get pods
NAME              READY     STATUS    RESTARTS
AGE
my-nginx-r5lnq    1/1       Running   0
27m
my-nginx-uhe8r    1/1       Running   0
27m
```

It will help prevent human error. By default, `--current-replicas` equals -1, which means bypass to check the current number of replicas:

```
//no matter current number of replicas, performs
to change to 4
# kubectl scale --current-replicas=-1 --replicas=4
 rc my-nginx
replicationcontroller "my-nginx" scaled

# kubectl get pods
NAME              READY     STATUS    RESTARTS
AGE
```

```
my-nginx-dimxj   1/1       Running   0          5s
my-nginx-eem3a   1/1       Running   0          5s
my-nginx-r5lnq   1/1       Running   0
35m
my-nginx-uhe8r   1/1       Running   0
35m
```

See also

This recipe described how to change the number of pods using the scaling option by the replication controller. It is useful to scale up and scale down your application quickly. To know more about how to update your container, refer to the following recipes:

- *Updating live containers*
- *Ensuring flexible usage of your containers*

Updating live containers

For the benefit of containers, we can easily publish new programs by executing the latest image, and reduce the headache of environment setup. But, what about publishing the program on running containers? Using native Docker commands, we have to stop the running containers prior to booting up new ones with the latest images and the same configurations. There is a simple and efficient zero-downtime method to update your program in the Kubernetes system. It is called rolling-update. We will show this solution to you in this recipe.

Getting ready

Rolling-update works on the units of the replication controller. The effect is to create new pods one by one to replace the old one. The new pods in the target replication controller are attached to the original labels. Therefore, if any service exposes this replication controller, it will take over the newly created pods directly.

For a later demonstration, we are going to update a new nginx image. In addition to this, we are going to make sure that nodes get your customized image, pushing it to Docker Hub, the public Docker registry, or private registry.

For example, you can create the image by writing your own `Dockerfile`:

```
$ cat Dockerfile
FROM nginx
RUN echo "Happy Programming!" >
/usr/share/nginx/html/index.html
```

In this Docker image, we changed the content of the default `index.html` page. Then, you can build your image and push it with

the following commands:

```
// push to Docker Hub
$ docker build -t <DOCKERHUB_ACCOUNT>/common-nginx
. && docker push <DOCKERHUB_ACCOUNT>/common-nginx
// Or, you can also push to your private docker
registry
$ docker build -t <RESITRY_NAME>/common-nginx . &&
docker push <RESITRY_NAME>/common-nginx
```

To add nodes' access authentications of the private Docker registry, please take the *Working with the private Docker registry* recipe in Chapter 5, *Building a Continuous Delivery Pipeline*, as a reference.

How to do it…

You'll now learn how to publish a Docker image. The following steps will help you successfully publish a Docker image:

1. At the beginning, create a pair of replication controller and service for rolling-update testing. As shown in the following statement, a replication controller with five replicas will be created. The nginx program exposed port `80` to the container, while the Kubernetes service transferred the port to `8080` in the internal network:

```
// Create a replication controller named
nginx-rc
# kubectl run nginx-rc --image=nginx --
replicas=5 --port=80 --
labels="User=Amy,App=Web,State=Testing"
replicationcontroller "nginx-rc" created
// Create a service supporting nginx-rc
# kubectl expose rc nginx-rc --port=8080 --
target-port=80 --name="nginx-service"
service "nginx-service" exposed
# kubectl get service nginx-service
NAME            CLUSTER_IP       EXTERNAL_IP
PORT(S)    SELECTOR
AGE
nginx-service   192.168.163.46   <none>
```

```
8080/TCP    App=Web,State=Testing,User=Amy
35s
```

You can evaluate whether the components work fine or not by examining `<POD_IP>:80` and `<CLUSTER_IP>:8080`.

2. Now, we are good to move on to the container update step! The Kubernetes subcommand `rolling-update` helps to keep the live replication controller up to date. In the following command, users have to specify the name of the replication controller and the new image. Here, we will use the image that is being uploaded to Docker Hub:

```
# kubectl rolling-update nginx-rc --image=
<DOCKERHUB_ACCOUNT>/common-nginx
Created nginx-rc-
b6610813702bab5ad49d4aadd2e5b375
Scaling up nginx-rc-
b6610813702bab5ad49d4aadd2e5b375 from 0 to 5,
scaling down nginx-rc from 5 to 0 (keep 5 pods
available, don't exceed 6 pods)
Scaling nginx-rc-
b6610813702bab5ad49d4aadd2e5b375 up to 1
```

3. You may see that the process is hanging. Because `rolling-update` will start a single new pod at a time and wait for a period of time; the default is one minute to stop an old pod and create a second new pod. From this idea, while updating, there always is one more pod on the serving, one more pod than the desired state of the replication controller. In this case, there would be six pods. While updating the replication controller, please access another terminal for a brand-new process.
4. Check the state of the replication controller for more concepts:

```
# kubectl get rc
CONTROLLER
CONTAINER(S)       IMAGE(S)
SELECTOR
REPLICAS   AGE
nginx-rc
nginx-rc           nginx
App=Web,State=Testing,User=Amy,deployment=313d
```

```
a350dea9227b89b4f0340699a388   5          1m
nginx-rc-b6610813702bab5ad49d4aadd2e5b375
nginx-rc          <DOCKERHUB_ACCOUNT>/common-
nginx
App=Web,State=Testing,User=Amy,deployment=b661
0813702bab5ad49d4aadd2e5b375   1          16s
```

5. As you will find, the system creates an almost identical replication controller with a postfix name. A new label key `deployment` is added to both the replication controllers for discriminating. On the other hand, new `nginx-rc` is attached to the other original labels. Service will also take care of the new pods at the same time:

```
// Check service nginx-service while updating
# kubectl describe service nginx-service
Name:      nginx-service
Namespace:    default
Labels:      App=Web,State=Testing,User=Amy
Selector:    App=Web,State=Testing,User=Amy
Type:      ClusterIP
IP:      192.168.163.46
Port:      <unnamed>  8080/TCP
Endpoints:
192.168.15.5:80,192.168.15.6:80,192.168.15.7:8
0 + 3 more...
Session Affinity:  None
No events.
```

   There are six endpoints of pods covered by `nginx-service`, which is supported by the definition of rolling-update.

6. Go back to the console running the update process. After it completes the update, you can find procedures as follows:

```
Created nginx-rc-
b6610813702bab5ad49d4aadd2e5b375
Scaling up nginx-rc-
b6610813702bab5ad49d4aadd2e5b375 from 0 to 5,
scaling down nginx-rc from 5 to 0 (keep 5 pods
available, don't exceed 6 pods)
Scaling nginx-rc-
b6610813702bab5ad49d4aadd2e5b375 up to 1
```

```
Scaling nginx-rc down to 4
Scaling nginx-rc-
b6610813702bab5ad49d4aadd2e5b375 up to 2
Scaling nginx-rc down to 3
Scaling nginx-rc-
b6610813702bab5ad49d4aadd2e5b375 up to 3
Scaling nginx-rc down to 2
Scaling nginx-rc-
b6610813702bab5ad49d4aadd2e5b375 up to 4
Scaling nginx-rc down to 1
Scaling nginx-rc-
b6610813702bab5ad49d4aadd2e5b375 up to 5
Scaling nginx-rc down to 0
Update succeeded. Deleting old controller:
nginx-rc
Renaming nginx-rc-
b6610813702bab5ad49d4aadd2e5b375 to nginx-rc
replicationcontroller "nginx-rc" rolling
updated
```

Old `nginx-rc` is gradually taken out of service by scaling down.

7. At the final steps of the update, the new replication controller is scaled up to five pods to meet the desired state and replace the old one eventually:

```
// Take a look a current replication
controller
// The new label "deployment" is remained
after update
# kubectl get rc nginx-rc
CONTROLLER   CONTAINER(S)   IMAGE(S)
SELECTOR
REPLICAS   AGE
nginx-rc     nginx-rc
<DOCKERHUB_ACCOUNT>/common-nginx
App=Web,State=Testing,User=Amy,deployment=b661
0813702bab5ad49d4aadd2e5b375   5          40s
```

8. Checking service with ClusterIP and port, we can now have all our pods in the replication controller updated:

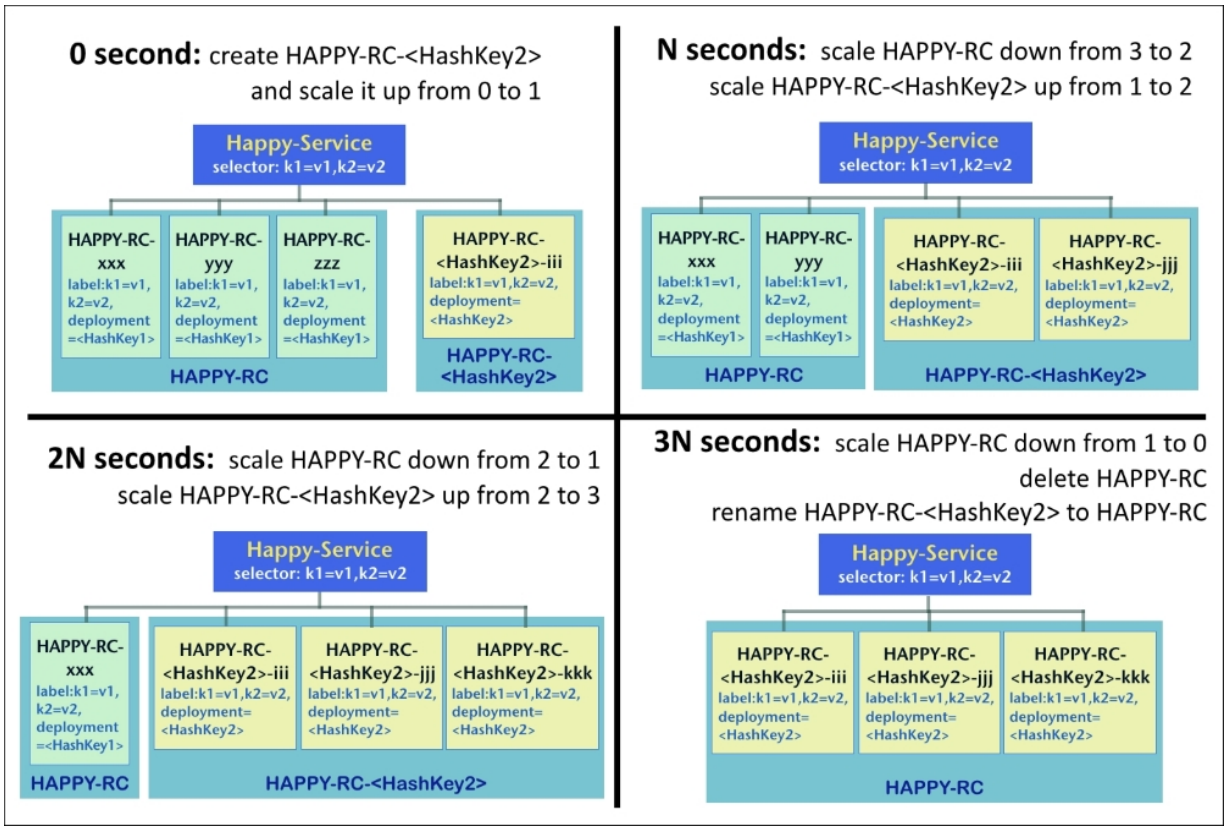
```
# curl 192.168.163.46:8080
Happy Programming!
```

9. According to the previous demonstration, it costs about five minutes to publish a new Docker image. It is because the updating time is set to one minute by default for the procedure of scaling up and down. It is possible for you to have a faster or slower pace of update by counting on the tag `--update-period`. The valid time units are `ns`, `us`, `ms`, `s`, `m`, and `h`. For example, `--update-period=1m0s`:

```
// Try on this one!
# kubectl rolling-update
<REPLICATION_CONTROLLER_NAME> --image=
<IMAGE_NAME> --update-period=10s
```

How it works…

In this section, we will discuss rolling-update in detail. How about renewing a replication controller with *N* seconds as the period of updating? See the following image:



The previous image indicates each step of the updating procedure. We may get some important ideas from rolling-update:

- Each pod in both the replication controllers has a new label, but an unequal value to point out the difference. Besides, the other labels are the same, so service can still cover both the replication controllers by selectors while updating.
- We would spend *# pod in replication controller * update period* time for migrating a new configuration.
- For zero-downtime updating, the total number of pods covered by the service should meet the desired state. For example, in the preceding image, there should be always three pods running at a time for the service.
- Rolling-update procedure doesn't assure users when the newly created pod, in **HAPPY-RC-<HashKey2>**, is in running state. That's why we need an update period. After a period of time, *N* seconds in the preceding case, a new pod should be ready to take the place of an old pod. Then, it is good to terminate one old pod.
- The period of updating time should be the worst case of the time required by a new pod from pulling an image to running.

There's more…

While doing rolling-update, we may specify the image for a new replication controller. But sometimes, we cannot update the new image successfully. It is because of container's image pull policy.

To update with a specific image, it will be great if users provide a tag so that what version of the image should be pulled is clear and accurate. However, most of the time, the latest one to which users look for and the latest tagged image could be regarded as the same one in local, since they are called the latest as well. Like the command `<DOCKERHUB_ACCOUNT>/common-nginx:latest image` will be used in this update:

```
# kubectl rolling-update nginx-rc --image=
```

```
<DOCKERHUB_ACCOUNT>/common-nginx --update-
period=10s
```

Still, nodes will ignore to pull the latest version of `common-nginx` if they find an image labeled as the same request. For this reason, we have to make sure that the specified image is always pulled from the registry.

In order to change the configuration, the subcommand `edit` can help in this way:

```
# kubectl edit rc <REPLICATION_CONTROLLER_NAME>
```

Then, you can edit the configuration of the replication controller in the YAML format. The policy of image pulling could be found in the following class structure:

```
apiVersion: v1
kind: replicationcontroller
spec:
  template:
    spec:
      containers:
      - name: <CONTAINER_NAME>
        image: <IMAGE_TAG>
                  imagePullPolicy: IfNotPresent
:
```

The value `IfNotPresent` tells the node to only pull the image not presented on the local disk. By changing the policy to `Always`, users will be able to avoid updating failure. It is workable to set up the key-value item in the configuration file. So, the specified image is guaranteed to be the one in the image registry.

See also

Pod is the basic computing unit in the Kubernetes system. You can learn how to use pods even more effectively through the following

recipes:

- *Scaling your containers*
- The *Moving monolithic to microservices*, *Integrating with Jenkins*, *Working with the private Docker registry*, and *Setting up the Continuous Delivery pipeline* recipes in Chapter 5, *Building a Continuous Delivery Pipeline*

Forwarding container ports

In the previous chapters, you learned how to work with the Kubernetes services to forward the container port internally and externally. Now, it's time to take it a step further to see how it works.

There are four networking models in Kubernetes, and we'll explore the details in the following sections:

- Container-to-container communications
- Pod-to-pod communications
- Pod-to-service communications
- External-to-internal communications

Getting ready

In this section, we will run two nginx web apps in order to demonstrate how these four models work. The default port of nginx in Docker Hub (https://hub.docker.com/_/nginx) is `80`. We will then create another nginx Docker image by modifying the nginx configuration file and Dockerfile from `80` to `8800`. The following steps will show you how to build it from scratch, but you are free to skip it and use our prebuilt image (https://hub.docker.com/r/msfuko/nginx_8800) as well.

Let's create one simple nginx configuration file first. Note that we need to listen to the `8800` port:

```
// create one nginx config file
# cat nginx.conf
server {
    listen       8800;
    server_name  localhost;

    #charset koi8-r;
    #access_log  /var/log/nginx/log/host.access.log
main;

    location / {
```

```
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }

    #error_page  404              /404.html;

    # redirect server error pages to the static page
/50x.html
    #
    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }
}
```

Next, we need to change the default nginx `Dockerfile` from expose `80` to `8800`:

```
// modifying Dockerfile as expose 8800 and add
config file inside
# cat Dockerfile
FROM debian:jessie

MAINTAINER NGINX Docker Maintainers "docker-
maint@nginx.com"

RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --
recv-keys 573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62
RUN echo "deb
http://nginx.org/packages/mainline/debian/ jessie
nginx" >> /etc/apt/sources.list

ENV NGINX_VERSION 1.9.9-1~jessie

RUN apt-get update && \
    apt-get install -y ca-certificates
nginx=${NGINX_VERSION} && \
        rm -rf /var/lib/apt/lists/*
COPY nginx.conf /etc/nginx/conf.d/default.conf

# forward request and error logs to docker log
collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log
```

```
VOLUME ["/var/cache/nginx"]

EXPOSE 8800

CMD ["nginx", "-g", "daemon off;"]
```

Then, we'll need to build it using the Docker command:

```
// build docker image
# docker build -t
$(YOUR_DOCKERHUB_ACCOUNT)/nginx_8800 .
```

Finally, we can push to our Docker Hub repository:

```
// be sure to login via `docker login` first
# docker push $(YOUR_DOCKERHUB_ACCOUNT)/nginx_8800
```

After this, you should be able to run the container by the pure Docker command: `docker run -d -p 8800:8800 msfuko/nginx_8800`. Using `curl $IP:8800`, you should be able to see the welcome page of nginx.

> ## Tip
>
> **How to find my $IP?**
>
> If you are running on Linux, then `ifconfig` could help to figure out the value of $IP. If you are running on another platform via Docker machine, `docker-machine ip` could help you with that.

How to do it...

Pod contains one or more containers, which run on the same host. Each pod has their own IP address; all the containers inside a pod see each other as on the same host. Containers inside a pod will be created, deployed, and deleted almost at the same time.

Container-to-container communications

We'll create two nginx containers in one pod that will listen to port `80` and `8800`, individually. In the following configuration file, you should change the second image path as the one you just built and pushed:

```
// create 2 containers inside one pod
# cat nginxpod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginxpod
spec:
  containers:
    -
      name: nginx80
      image: nginx
      ports:
        -
          containerPort: 80
          hostPort: 80
    -
      name: nginx8800
      image: msfuko/nginx_8800
      ports:
        -
          containerPort: 8800
          hostPort: 8800
// create the pod
# kubectl create -f nginxpod.yaml
pod "nginxpod" created
```

After the image is pulled and run, we can see that the status becomes running using the `kubectl` command:

```
// list nginxpod pod
# kubectl get pods nginxpod
NAME       READY     STATUS    RESTARTS   AGE
nginxpod   2/2       Running   0          12m
```

We could find the count in the `READY` column become `2/2`, since there are two containers inside this pod. Using the `kubectl describe`

command, we can see the details of the pod:

```
// show the details of nginxpod
# kubectl describe pod nginxpod
Name:        nginxpod
Namespace:      default
Image(s):      nginx,msfuko/nginx_8800
Node:        kube-node1/10.96.219.33
Start Time:      Sun, 24 Jan 2016 10:10:01 +0000
Labels:        <none>
Status:        Running
Reason:
Message:
IP:        192.168.55.5
Replication Controllers:  <none>
Containers:
  nginx80:
    Container ID:
docker://3b467d8772f09c57d0ad85caa66b8379799f3a60da0
55d7d8d362aee48dfa832
    Image:    nginx
    ...
  nginx8800:
    Container ID:
docker://80a77983f6e15568db47bd58319fad6d22a330c1c4c
9263bca9004b80ecb6c5f
    Image:    msfuko/nginx_8800
   ...
```

We could see the pod is run on `kube-node1` and the IP of the pod is `192.168.55.5`. Let's log in to `kube-node1` to inspect these two containers:

```
// list containers
# docker ps
CONTAINER ID        IMAGE
COMMAND                  CREATED             STATUS
PORTS                                        NAMES
80a77983f6e1        msfuko/nginx_8800
"nginx -g 'daemon off"   32 minutes ago      Up 32
minutes
k8s_nginx8800.645004b9_nginxpod_default_a08ed7cb-
c282-11e5-9f21-025a2f393327_9f85a41b
3b467d8772f0        nginx
```

```
"nginx -g 'daemon off"   32 minutes ago      Up 32
minutes
k8s_nginx80.5098ff7f_nginxpod_default_a08ed7cb-c282-
11e5-9f21-025a2f393327_9922e484
71073c074a76
gcr.io/google_containers/pause:0.8.0
"/pause"                 32 minutes ago      Up 32
minutes       0.0.0.0:80->80/tcp, 0.0.0.0:8800-
>8800/tcp
k8s_POD.5c2e23f2_nginxpod_default_a08ed7cb-c282-
11e5-9f21-025a2f393327_77e79a63
```

We know that the ID of the two containers we created are `3b467d8772f0` and `80a77983f6e1`.

We will use `jq` as the JSON parser to reduce the redundant information. For installing `jq`, simply download the binary file from https://stedolan.github.io/jq/download:

```
// inspect the nginx container and use jq to parse
it
# docker inspect 3b467d8772f0 | jq '.[]|
{NetworkMode: .HostConfig.NetworkMode,
NetworkSettings: .NetworkSettings}'
{
  "NetworkMode":
"container:71073c074a761a33323bb6601081d44a79ba7de3d
d59345fc33a36b00bca613f",
  "NetworkSettings": {
    "Bridge": "",
    "SandboxID": "",
    "HairpinMode": false,
    "LinkLocalIPv6Address": "",
    "LinkLocalIPv6PrefixLen": 0,
    "Ports": null,
    "SandboxKey": "",
    "SecondaryIPAddresses": null,
    "SecondaryIPv6Addresses": null,
    "EndpointID": "",
    "Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "IPAddress": "",
    "IPPrefixLen": 0,
    "IPv6Gateway": "",
```

```
        "MacAddress": "",
        "Networks": null
    }
}
```

We can see that the network mode is set as mapped container mode. The network bridge container is `container:71073c074a761a33323bb6601081d44a79ba7de3dd59345fc33a36b00bca613f`.

Let's see another setting about container `nginx_8800`:

```
// inspect nginx_8800
# docker inspect 80a77983f6e1 | jq '.[]|
{NetworkMode: .HostConfig.NetworkMode,
NetworkSettings: .NetworkSettings}'
{
  "NetworkMode":
"container:71073c074a761a33323bb6601081d44a79ba7de3d
d59345fc33a36b00bca613f",
  "NetworkSettings": {
    "Bridge": "",
    "SandboxID": "",
    "HairpinMode": false,
    "LinkLocalIPv6Address": "",
    "LinkLocalIPv6PrefixLen": 0,
    "Ports": null,
    "SandboxKey": "",
    "SecondaryIPAddresses": null,
    "SecondaryIPv6Addresses": null,
    "EndpointID": "",
    "Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "IPAddress": "",
    "IPPrefixLen": 0,
    "IPv6Gateway": "",
    "MacAddress": "",
    "Networks": null
  }
}
```

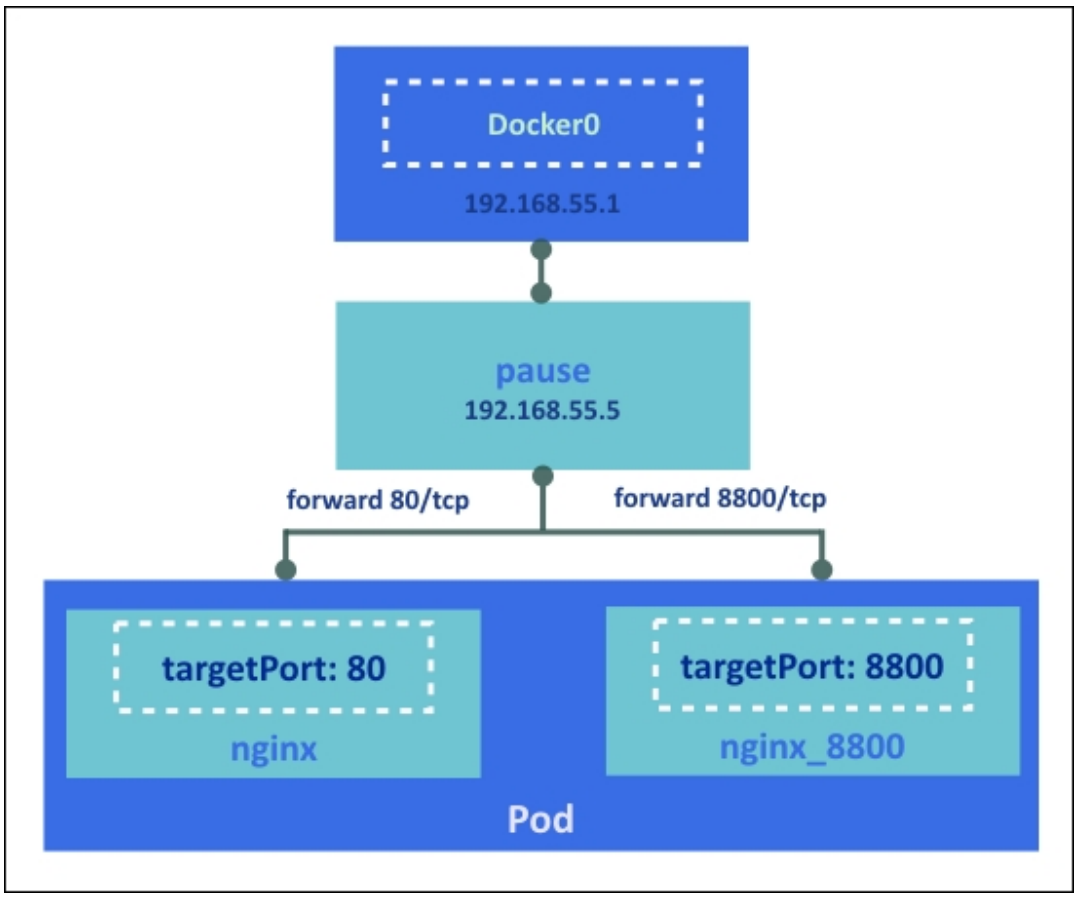
The network mode is also set to container: `71073c074a761a33323bb6601081d44a79ba7de3dd59345fc33a36b00bca613`.

Which container is this? We will then find out that this network container is created by Kubernetes when your pod starts. The container is named `gcr.io/google_containers/pause`:

```
// inspect network container `pause`
# docker inspect 71073c074a76 | jq '.[]|
{NetworkMode: .HostConfig.NetworkMode,
NetworkSettings: .NetworkSettings}'
{
  "NetworkMode": "default",
  "NetworkSettings": {
    "Bridge": "",
    "SandboxID":
"59734bbe4e58b0edfc92db81ecda79c4f475f6c8433e17951e9
c9047c69484e8",
    "HairpinMode": false,
    "LinkLocalIPv6Address": "",
    "LinkLocalIPv6PrefixLen": 0,
    "Ports": {
      "80/tcp": [
        {
          "HostIp": "0.0.0.0",
          "HostPort": "80"
        }
      ],
      "8800/tcp": [
        {
          "HostIp": "0.0.0.0",
          "HostPort": "8800"
        }
      ]
    },
    "SandboxKey":
"/var/run/docker/netns/59734bbe4e58",
    "SecondaryIPAddresses": null,
    "SecondaryIPv6Addresses": null,
    "EndpointID":
"d488fa8d5ee7d53d939eadda106e97ff01783f0e9dc9e4625d9
e69500e1fa451",
    "Gateway": "192.168.55.1",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "IPAddress": "192.168.55.5",
    "IPPrefixLen": 24,
    "IPv6Gateway": "",
    "MacAddress": "02:42:c0:a8:37:05",
```

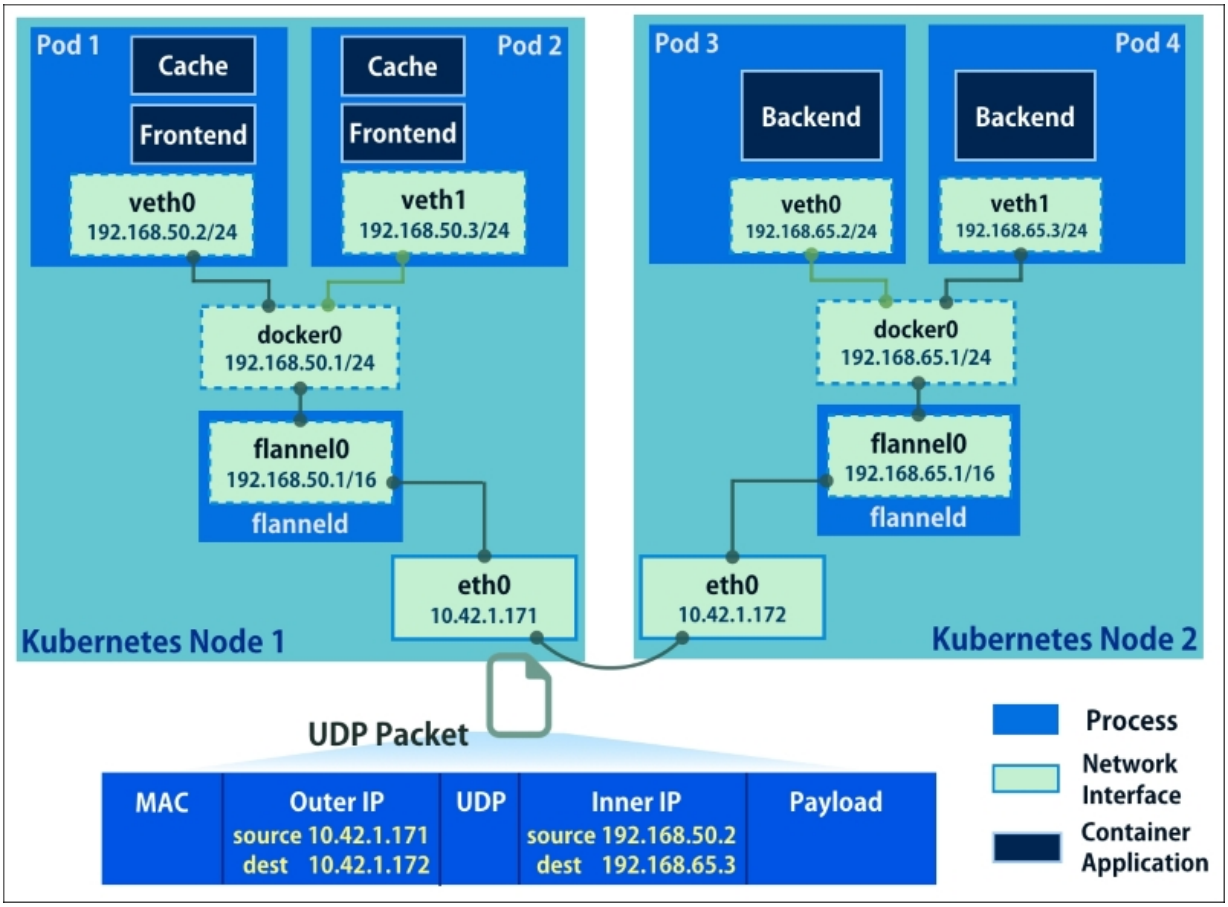
```
    "Networks": {
      "bridge": {
        "EndpointID":
"d488fa8d5ee7d53d939eadda106e97ff01783f0e9dc9e4625d9
e69500e1fa451",
        "Gateway": "192.168.55.1",
        "IPAddress": "192.168.55.5",
        "IPPrefixLen": 24,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:c0:a8:37:05"
      }
    }
  }
}
```

We will find out that the network mode is set to `default` and its IP address is set to the IP of the pod `192.168.55.5`; the gateway is set to `docker0` of the node. The routing illustration will be as shown in the following image. The network container pause will be created when a pod is created, which will be used to handle the route of the pod network. Then, two containers will share the network interface with pause; that's why they see each other as localhost:



Pod-to-pod communications

Since each pod has its own IP address, it makes the communication between pods easy. In the previous chapter, we use flannel as the overlay network, which will define different network segments for each node. Each packet is encapsulated to a UDP packet so that each pod IP is routable. The packet from **Pod1** will go though the **veth** (virtual network interface) device, which connects to **docker0** and routes to **flannel0**. The traffic is encapsulated by flanneld and sent to the host (**10.42.1.172**) of the target pod:



Pod-to-service communications

Pods could be stopped accidentally, so the IP of the pod could be changed. When we expose the port for a pod or a replication controller, we create a Kubernetes service acting as a proxy or a load balancer. Kubernetes will create a virtual IP, which will receive the request from clients and proxy the traffic to the pods in a service. Let's review how to do this. At first, we will create a replication controller named `my-first-nginx`:

```
// create a rc named my-first-nginx
# kubectl run my-first-nginx --image=nginx --
replicas=2 --port=80
replicationcontroller "my-first-nginx" created
```

Then, let's list the pods to ensure two pods are created by this `rc`:

```
// two pods will be launched by this rc
# kubectl get pod
NAME                          READY     STATUS
RESTARTS   AGE
my-first-nginx-hsdmz          1/1       Running   0
17s
my-first-nginx-xjtxq          1/1       Running   0
17s
```

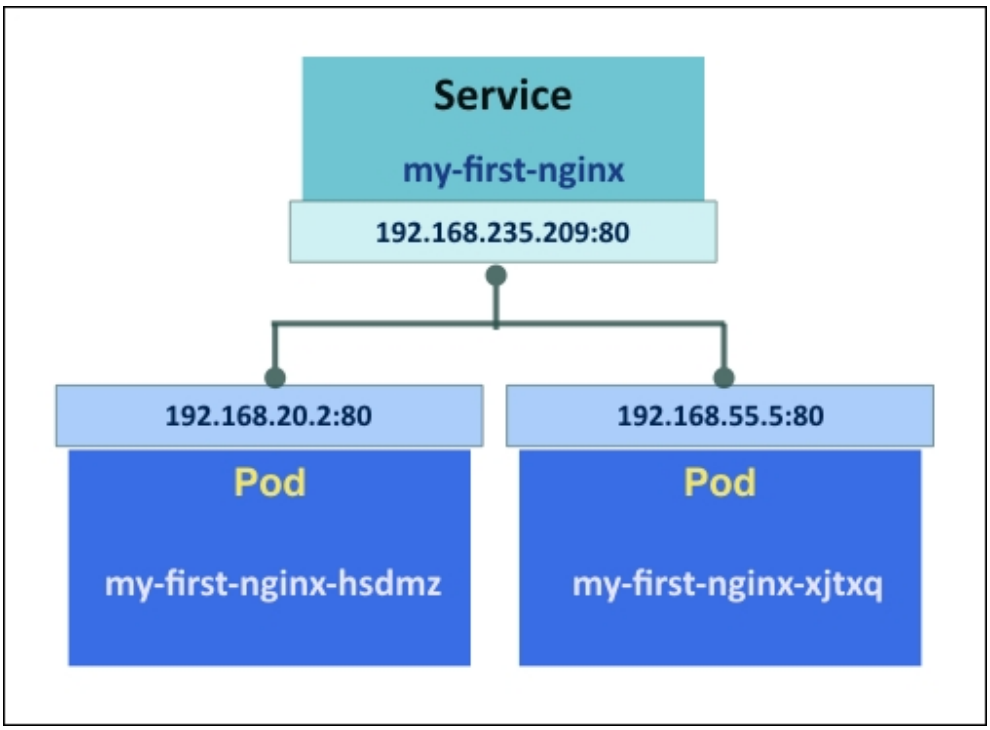
Next, let's expose one port `80` for the pod, which is the default port of the `nginx` app:

```
// expose port 80 for my-first-nginx
# kubectl expose rc my-first-nginx --port=80
service "my-first-nginx" exposed
```

Use `describe` to see the details of the service. The service type we create is a ClusterIP:

```
// check the details of the service
# kubectl describe service my-first-nginx
Name:      my-first-nginx
Namespace:    default
Labels:      run=my-first-nginx
Selector:    run=my-first-nginx
Type:      ClusterIP
IP:      192.168.235.209
Port:      <unnamed>  80/TCP
Endpoints:    192.168.20.2:80,192.168.55.5:80
Session Affinity:  None
No events.
```

The virtual IP of the service is `192.168.235.209`, which exposes the port `80`. The service will then dispatch the traffic into two endpoints `192.168.20.2:80` and `192.168.55.5:80`. The illustration is as follows:



`kube-proxy` is a daemon that works as a network proxy on every node. It helps to reflect the settings of services, such as IPs or ports, on each node. It will create the corresponding iptables rules:

```
// list iptables rule by filtering my-first-nginx
# iptables -t nat -L | grep my-first-nginx
REDIRECT   tcp  --  anywhere
192.168.235.209      /* default/my-first-nginx: */
tcp dpt:http redir ports 43321
DNAT       tcp  --  anywhere
192.168.235.209      /* default/my-first-nginx: */
tcp dpt:http to:10.96.219.33:43321
```

These two rules are under the `KUBE-PORTALS-CONTAINER` and `KUBE-PORTALS-HOST` chains, which represent any traffic destined for the virtual IP with port `80` that will be redirected to the localhost on port `43321` no matter whether the traffic is from containers or hosts. The kube-proxy programs the iptables rule to make the pod and service communication available. You should be able to access `localhost:43321` on the target node or `$nodeIP:43321` inside the cluster.

Note

Using the environment variables of the Kubernetes service in your program

Sometimes, you'll need to access the Kubernetes service in your program inside the Kubernetes cluster. You could use environment variables or DNS to access it. Environment variables are the easiest way and are supported naturally. When a service is created, `kubelet` will add a set of environment variables about this service:

- `$SVCNAME_SERVICE_HOST`
- `$SVCNAME_SERVICE_PORT`

Here, `$SVNNAME` is uppercase and the dashes are converted into underscores. The service that a pod wants to access must be created before the pod, otherwise the environment variables will not be populated. For example, the environment variables that `my-first-nginx` populate are:

- `MY_FIRST_NGINX_PORT_80_TCP_PROTO=tcp`
- `MY_FIRST_NGINX_SERVICE_HOST=192.168.235.209`
- `MY_FIRST_NGINX_SERVICE_PORT=80`
- `MY_FIRST_NGINX_PORT_80_TCP_ADDR=192.168.235.209`
- `MY_FIRST_NGINX_PORT=tcp://192.168.235.209:80`
- `MY_FIRST_NGINX_PORT_80_TCP_PORT=80`
- `MY_FIRST_NGINX_PORT_80_TCP=tcp://192.168.235.209:80`

External-to-internal communications

The external-to-internal communications could be set up using the external load balancer, such as GCE's ForwardingRules or AWS's ELB, or by accessing node IP directly. Here, we will introduce how accessing node IP could work. First, we'll run a replication controller with two replicas named `my-second-nginx`:

```
// create a rc with two replicas
# kubectl run my-second-nginx --image=nginx --
replicas=2 --port=80
```

Next, we'll expose the service with port `80` with the type `LoadBalancer`. As we discussed in the service section, `LoadBalancer` will also expose a node port:

```
// expose port 80 for my-second-nginx rc with type
LoadBalancer
# kubectl expose rc my-second-nginx --port=80 --
type=LoadBalancer
```

We could now check the details of `my-second-nginx` service. It has a virtual IP `192.168.156.93` with port `80`. It also has a node port `31150`:

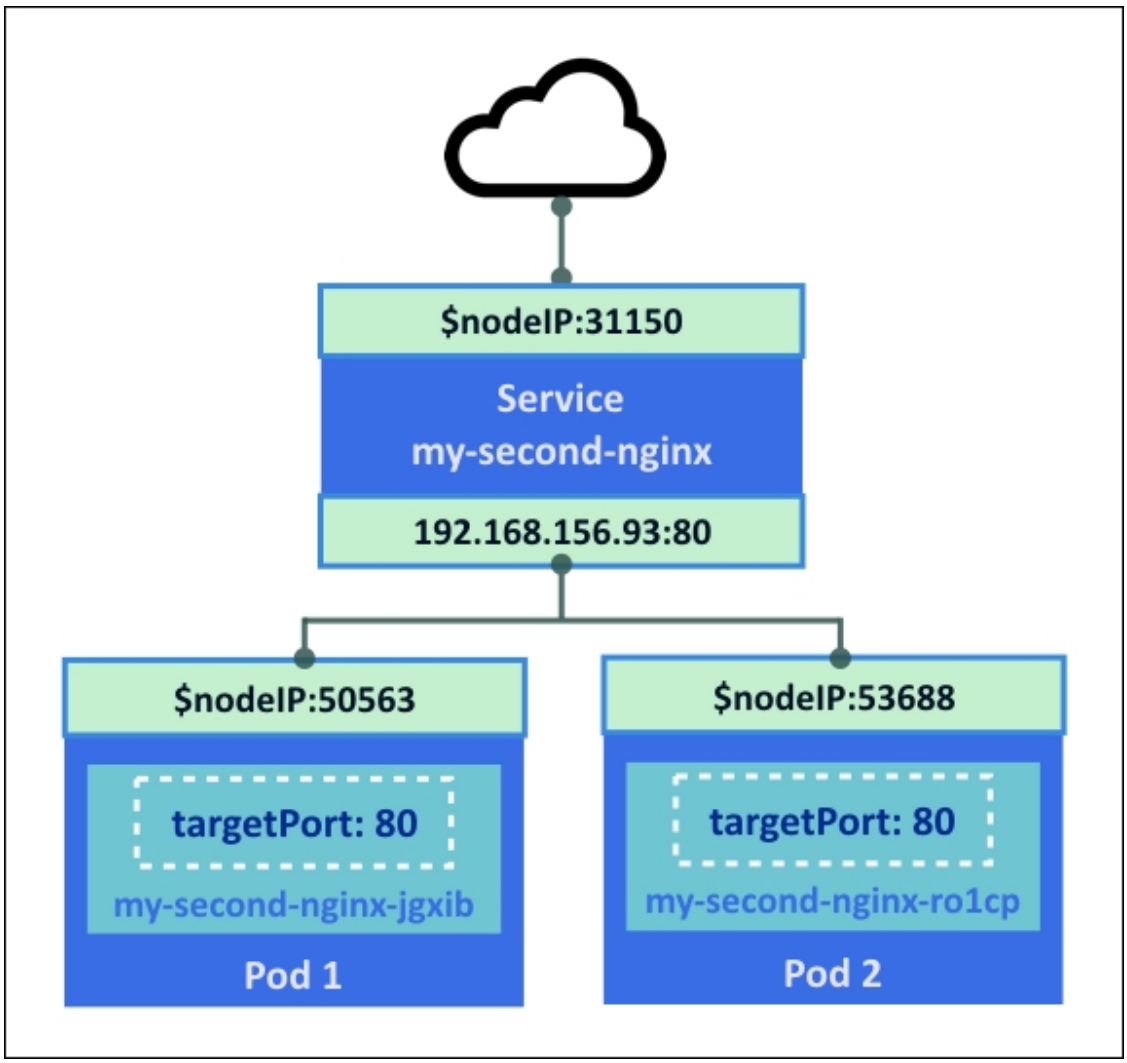
```
// list the details of service
# kubectl describe service my-second-nginx
Name:      my-second-nginx
Namespace:    default
Labels:      run=my-second-nginx
Selector:    run=my-second-nginx
Type:      LoadBalancer
IP:      192.168.156.93
Port:      <unnamed>  80/TCP
NodePort:    <unnamed>  31150/TCP
Endpoints:    192.168.20.3:80,192.168.55.6:80
Session Affinity:  None
No events.
```

Let's list the iptables rules to see the differences between the different types of service:

```
// list iptables rules and filtering my-seconde-
nginx on node1
# iptables -t nat -L | grep my-second-nginx
REDIRECT   tcp  --  anywhere             anywhere
/* default/my-second-nginx: */ tcp dpt:31150 redir
ports 50563
```

```
DNAT       tcp  --  anywhere             anywhere
/* default/my-second-nginx: */ tcp dpt:31150
to:10.96.219.141:50563
REDIRECT   tcp  --  anywhere
192.168.156.93       /* default/my-second-nginx: */
tcp dpt:http redir ports 50563
DNAT       tcp  --  anywhere
192.168.156.93       /* default/my-second-nginx: */
tcp dpt:http to:10.96.219.141:50563

// list iptables rules and filtering my-second-nginx
on node2
# iptables -t nat -L | grep my-second-nginx
REDIRECT   tcp  --  anywhere             anywhere
/* default/my-second-nginx: */ tcp dpt:31150 redir
ports 53688
DNAT       tcp  --  anywhere             anywhere
/* default/my-second-nginx: */ tcp dpt:31150
to:10.96.219.33:53688
REDIRECT   tcp  --  anywhere
192.168.156.93       /* default/my-second-nginx: */
tcp dpt:http redir ports 53688
DNAT       tcp  --  anywhere
192.168.156.93       /* default/my-second-nginx: */
tcp dpt:http to:10.96.219.33:53688
```

We have four rules related to `my-second-nginx` now. They are under the `KUBE-NODEPORT-CONTAINER`, `KUBE-NODEPORT-HOST`, `KUBE-PORTALS-CONTAINER`, and `KUBE-PORTALS-HOST` chains. Since we expose the node port in this example, if the traffic is from the outside world to node port `31150`, the traffic will be redirected to the target pod locally or across nodes. Following is an illustration of routing:



The traffic from node port (`x.x.x.x:31150`) or from ClusterIP (`192.168.156.93:80`) will be redirected to target pods by providing a load balancing mechanism across nodes. The ports `50563` and `53688` are dynamically assigned by Kubernetes.

See also

Kubernetes forwards port based on the overlay network. In this chapter, we also run pods and services with nginx. Reviewing the previous sections will help you to understand more about how to manipulate it. Also, look at the following recipes:

- The *Creating an overlay network*, *Running your first container in Kubernetes* recipes in Chapter 1, *Building Your Own Kubernetes*

- The *Working with pods*, *Working with services* in Chapter 2, *Walking through Kubernetes Concepts*

Ensuring flexible usage of your containers

Pod, in Kubernetes, means a set of containers, which is also the smallest computing unit. You may have known about the basic usage of pod in the previous recipes. Pods are usually managed by replication controllers and exposed by services; they work as applications with this scenario.

In this recipe, we will discuss two new features: `job` and `daemon set`. These two features can make the usage of pods more effective.

Getting ready

What are a job-like pod and a daemon-like pod? Pods in a Kubernetes job will be terminated directly after they complete their work. On the other hand, a daemon-like pod will be created in every node, while users want it to be a long running program served as a system daemon.

Both the job and daemon set belong to the extension of the Kubernetes API. Furthermore, the daemon set type is disabled in the default API settings. Then, you have to enable the usage later for testing. Without starting the setting of the daemon set, you will get an error about the unknown type:

```
# kubectl create -f daemonset-test.yaml
error validating "daemonset-test.yaml": error
validating data: couldn't find type:
v1beta1.DaemonSet; if you choose to ignore these
errors, turn validation off with --validate=false
Or error of this one
Error from server: error when creating "daemonset-
free.yaml": the server could not find the
requested resource
```

To enable the daemon set in the Kubernetes system, you should update a tag in the daemon of the Kubernetes apiserver: `--runtime-config=extensions/v1beta1/daemonsets=true`. Modify your service scripts or configuration options:

```
// For service init.d scripts, attached the tag
after command hyperkube apiserver or kube-
apiserver
# cat /etc/init.d/kubernetes-master
(heading lines ignored)
:
# Start daemon.
echo $"Starting apiserver: "
daemon $apiserver_prog \
--service-cluster-ip-range=${CLUSTER_IP_RANGE} \
--insecure-port=8080 \
--secure-port=6443 \
--basic-auth-file="/root/ba_file" \
--address=0.0.0.0 \
--etcd_servers=${ETCD_SERVERS} \
--cluster_name=${CLUSTER_NAME} \
--runtime-
config=extensions/v1beta1/daemonsets=true \
> ${logfile}-apiserver.log 2>&1 &
:
(ignored)
// For systemd service management, edit
configuration files by add the tag as optional one
# cat /etc/kubernetes/apiserver
(heading lines ignored)
:
# Add your own!
KUBE_API_ARGS="--cluster_name=Happy-k8s-cluster --
runtime-config=extensions/v1beta1/daemonsets=true"
```

After you set up the tag, remove the directory `/tmp/kubectl.schema`, which caches API schemas. Then, it is good to restart the Kubernetes apiserver:

```
// Remove the schema file
# rm -f /tmp/kubectl.schema
// The method to restart apiserver for init.d
```

```
script
# service kubernetes-master restart
// Or, restart the daemon for systemd service
management
# systemd restart kube-apiserver
// After restart daemon apiserver, you can find
daemonsets is enable in your API server
# curl
http://localhost:8080/apis/extensions/v1beta1
{
  "kind": "APIResourceList",
  "groupVersion": "extensions/v1beta1",
  "resources": [
    {
      "name": "daemonsets",
      "namespaced": true
    },
    {
      "name": "daemonsets/status",
      "namespaced": true
    },
:
```

Next, for the following sections, we are going to demonstrate how to create a job and daemon set using configuration files. Take a look at the recipe *Working with configuration files* in this chapter to know more about other concepts.

How to do it…

There is no command-line interface for us to create a job or a daemon set. Therefore, we will build these two resource types by writing all the configurations in a template file.

Pod as a job

A job-like pod is suitable for testing your containers, which can be used for unit test or integration test; or, it can be used for static program. Like in the following example, we will write a job template to check the packages installed in image `ubuntu`:

```
# cat job-dpkg.yaml
apiVersion: extensions/v1beta1
kind: Job
metadata:
  name: package-check
spec:
  selector:
    matchLabels:
      image: ubuntu
      test: dpkg
  template:
    metadata:
      labels:
        image: ubuntu
        test: dpkg
        owner: Amy
    spec:
      containers:
      - name: package-check
        image: ubuntu
        command: ["dpkg-query", "-l"]
      restartPolicy: Never
```

A job resource needs a selector to define which pods should be covered as this job. If no selector is specified in template, it will just be the same as the labels of the job. The restart policy for pods created in a job should be set to `Never` or `OnFailure`, since a job goes to termination once it is completed successfully.

Now, you are ready to create a job using your template:

```
# kubectl create -f job-dpkg.yaml
job "package-check" created
```

After pushing the requested file, it is possible to verify the status of both the pod and job:

```
# kubectl get job
JOB             CONTAINER(S)    IMAGE(S)
SELECTOR                           SUCCESSFUL
```

```
package-check   package-check   ubuntu     image
in (ubuntu),test in (dpkg)   1
// Check the job as well
# kubectl get pod
NAME                          READY     STATUS
RESTARTS   AGE
package-check-jrry1           0/1       Pending   0
6s
```

You will find that a pod is booting up for handling this task. This pod is going to be stopped very soon at the end of the process. The subcommand `logs` helps to get the result:

```
# kubectl logs package-check-gtyrc
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-
conf/Half-inst/trig-aWait/Trig-pend
|/ Err?=(none)/Reinst-required (Status,Err:
uppercase=bad)
||/ Name                              Version
Architecture Description
+++-=================================-
================================-============-
================================================
==============================
ii  adduser
3.113+nmu3ubuntu3                 all          add
and remove users and groups
ii  apt
1.0.1ubuntu2.10                   amd64
commandline package manager
ii  apt-utils
1.0.1ubuntu2.10                   amd64
package management related utility programs
ii  base-files                       7.2ubuntu5.3
amd64        Debian base system miscellaneous
files
:
(ignored)
```

Please go ahead and check the job `package-check` using the subcommand `describe`; the confirmation for pod completion and other messages are shown as system information:

```
# kubectl describe job package-check
```

Later, to remove the job you just created, stop it with the name:

```
# kubectl stop job package-check
job "package-check" deleted
```

Creating a job with multiple pods running

User can also decide the number of tasks that should be finished in a single job. It is helpful to solve some random and sampling problems. Let's try it on the same template in the previous example. We have to add the `spec.completions` item to indicate the pod number:

```
# cat job-dpkg.yaml
apiVersion: extensions/v1beta1
kind: Job
metadata:
  name: package-check
spec:
  completions: 3
  template:
    metadata:
      name: package-check-amy
      labels:
        image: ubuntu
        test: dpkg
        owner: Amy
    spec:
      containers:
      - name: package-check
        image: ubuntu
        command: ["dpkg-query", "-l"]
      restartPolicy: Never
```

Then, check how the job looks like using the subcommand `describe`:

```
# kubectl describe job package-check
Name:    package-check
Namespace:  default
Image(s):  ubuntu
Selector:  image in (ubuntu),owner in (Amy),test
in (dpkg)
Parallelism:  3
Completions:  3
Labels:    image=ubuntu,owner=Amy,test=dpkg
Pods Statuses:  3 Running / 0 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen  LastSeen  Count  From  SubobjectPath
Reason      Message
  _________  ________  _____  ____  _____________
______      _______
  6s    6s    1  {job }      SuccessfulCreate
Created pod: package-check-dk184
  6s    6s    1  {job }      SuccessfulCreate
Created pod: package-check-3uwym
  6s    6s    1  {job }      SuccessfulCreate
Created pod: package-check-eg4nl
```

As you can see, three pods are created to solve this job. Also, since the selector part is removed, the selector is copied from the labels.

Pod as a daemon set

If a Kubernetes daemon set is created, the defined pod will be deployed in every single node. It is guaranteed that the running containers occupy equal resources in each node. In this scenario, the container usually works as the daemon process.

For example, the following template has an `ubuntu` image container that keeps checking its memory usage half minute a time. We are going to build it as a daemon set:

```
# cat daemonset-free.yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
```

```
  name: ram-check
spec:
  selector:
    app: checkRam
    version: v1
  template:
    metadata:
      labels:
        app: checkRam
        owner: Amy
        version: v1
    spec:
      containers:
      - name: ubuntu-free
        image: ubuntu
        command: ["/bin/bash","-c","while true; do
free; sleep 30; done"]
      restartPolicy: Always
```

As the job, the selector could be ignored, but it takes the values of the labels. We will always configure the restart policy of the daemon set as `Always`, which makes sure that every node has a pod running.

The abbreviation of the daemon set is `ds`; use this shorter one in the command-line interface for convenience:

```
# kubectl create -f daemonset-free.yaml
daemonset "ram-check" created
# kubectl get ds
NAME        CONTAINER(S)   IMAGE(S)   SELECTOR
NODE-SELECTOR
ram-check   ubuntu-free    ubuntu
app=checkRam,version=v1   <none>
// Go further look at the daemon set by "describe"
# kubectl describe ds ram-check
Name:    ram-check
Image(s):  ubuntu
Selector:  app=checkRam,version=v1
Node-Selector:  <none>
Labels:    app=checkRam,owner=Amy,version=v1
Desired Number of Nodes Scheduled: 3
Current Number of Nodes Scheduled: 3
Number of Nodes Misscheduled: 0
Pods Status:  3 Running / 0 Waiting / 0 Succeeded
```

```
/ 0 Failed
Events:
  FirstSeen  LastSeen  Count  From
SubobjectPath  Reason      Message
  _________  ________  _____  ____
_____________  ______      _______
  3m    3m    1  {daemon-set }
SuccessfulCreate  Created pod: ram-check-bti08
  3m    3m    1  {daemon-set }
SuccessfulCreate  Created pod: ram-check-u9e5f
  3m    3m    1  {daemon-set }
SuccessfulCreate  Created pod: ram-check-mxry2
```

Here, we have three pods running in separated nodes. They can still be recognized in the channel of the pod:

```
# kubectl get pod --selector=app=checkRam
NAME               READY     STATUS    RESTARTS
AGE
ram-check-bti08    1/1       Running   0
4m
ram-check-mxry2    1/1       Running   0
4m
ram-check-u9e5f    1/1       Running   0
4m
// Get the evidence!
# kubectl describe pods --selector=app=checkRam -o
wide
NAME               READY     STATUS    RESTARTS
AGE   NODE
ram-check-bti08    1/1       Running   0
4m    kube-node1
ram-check-mxry2    1/1       Running   0
4m    kube-node3
ram-check-u9e5f    1/1       Running   0
4m    kube-node2
```

It is good for you to evaluate the result using the subcommand `logs`:

```
# kubectl logs ram-check-bti08
             total       used       free
shared    buffers     cached
```

```
Mem:       2051644    1256876     794768
148     136880     450620
-/+ buffers/cache:     669376    1382268
Swap:            0          0          0
             total       used       free
shared    buffers     cached
Mem:       2051644    1255888     795756
156     136912     450832
-/+ buffers/cache:     668144    1383500
Swap:            0          0          0
:
(ignored)
```

Next, delete this daemon set by the reference of template file or by the name of the resource:

```
# kubectl stop -f daemonset-free.yaml
// or
# kubectl stop ds ram-check
```

Running the daemon set only on specific nodes

There is also a solution for you to deploy daemon-like pods simply on specified nodes. First, you have to make nodes in groups by tagging them. We will only tag node `kube-node3` with the special key-value label, which indicates the one for running the daemon set:

```
# kubectl label nodes kube-node3 runDS=ok
node "kube-node3" labeled
# kubectl get nodes
NAME           LABELS
STATUS    AGE
kube-node1   kubernetes.io/hostname=kube-node1
Ready     27d
kube-node2   kubernetes.io/hostname=kube-node2
Ready     27d
kube-node3   kubernetes.io/hostname=kube-
node3,runDS=ok   Ready     4d
```

Then, we will select this one-member group in the template. The item `spec.template.spec.nodeSelector` can add any key-value pairs

for node selection:

```
# cat daemonset-free.yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: ram-check
spec:
  selector:
    app: checkRam
    version: v1
  template:
    metadata:
      labels:
        app: checkRam
        owner: Amy
        version: v1
    spec:
      nodeSelector:
        runDS: ok
      containers:
      - name: ubuntu-free
        image: ubuntu
        command: ["/bin/bash","-c","while true; do
free; sleep 30; done"]
      restartPolicy: Always
```

While assigning the daemon set to a certain node group, we can run it in a single node of the three-node system:

```
# kubectl describe pods --selector=app=checkRam |
grep "Node"
Node:        kube-node3/10.96.219.251
```

How it works…

Although job and daemon set are the special utilities of pods, the Kubernetes system has different managements between them and pods.

For job, its selector cannot point to the existing pod. It is for fear to take a pod controlled by the replication controller as a job. The replication controller has a desired number of pods running, which is against job's ideal situation: pods should be deleted once they finish the tasks. The pod in the replication controller won't get the state of end.

On the other hand, different from the general pod, a pod in a daemon set can be created without the Kubernetes scheduler. This concept is apparent because the daemon set only considers the labels of nodes, not their CPU or memory usages.

See also

In this recipe, we went deeply into the Kubernetes pod. Also, we used a bunch of Kubernetes configuration files. The recipe about configuration files will make you learn more about the following:

- The *Working with pods* recipe in Chapter 2, *Walking through Kubernetes Concepts*
- *Working with configuration files*

Working with configuration files

Kubernetes supports two different file formats YAML and JSON. Each format can describe the same function of Kubernetes.

Getting ready

Both YAML and JSON have official websites to describe the standard format.

YAML

The YAML format is very simple with less syntax rules; therefore, it is easy to read and write by a human. To know more about YAML, you can refer to the following website link:

http://www.yaml.org/spec/1.2/spec.html

The following example uses the YAML format to set up the `nginx` pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

JSON

The JSON format is also simple and easy to read by humans, but more program-friendly. Because it has data types (number, string, Boolean, and object), it is popular to exchange the data between systems. To know more about JSON, you can refer to the following website link:

http://json.org/

The following example of the pod is the same as the preceding YAML format, but using the JSON format:

```
{
    "apiVersion" : "v1",
    "kind" : "Pod",
    "metadata" : {
      "name" : "nginx",
      "labels": {
        "name": "nginx"
```

```
      }
    },
    "spec" : {
      "containers" : [
        {
            "name" : "nginx",
            "image" : "nginx",
            "ports" : [
               {
                 "containerPort": 80
               }
            ]
        }
      ]
    }
}
```

How to do it…

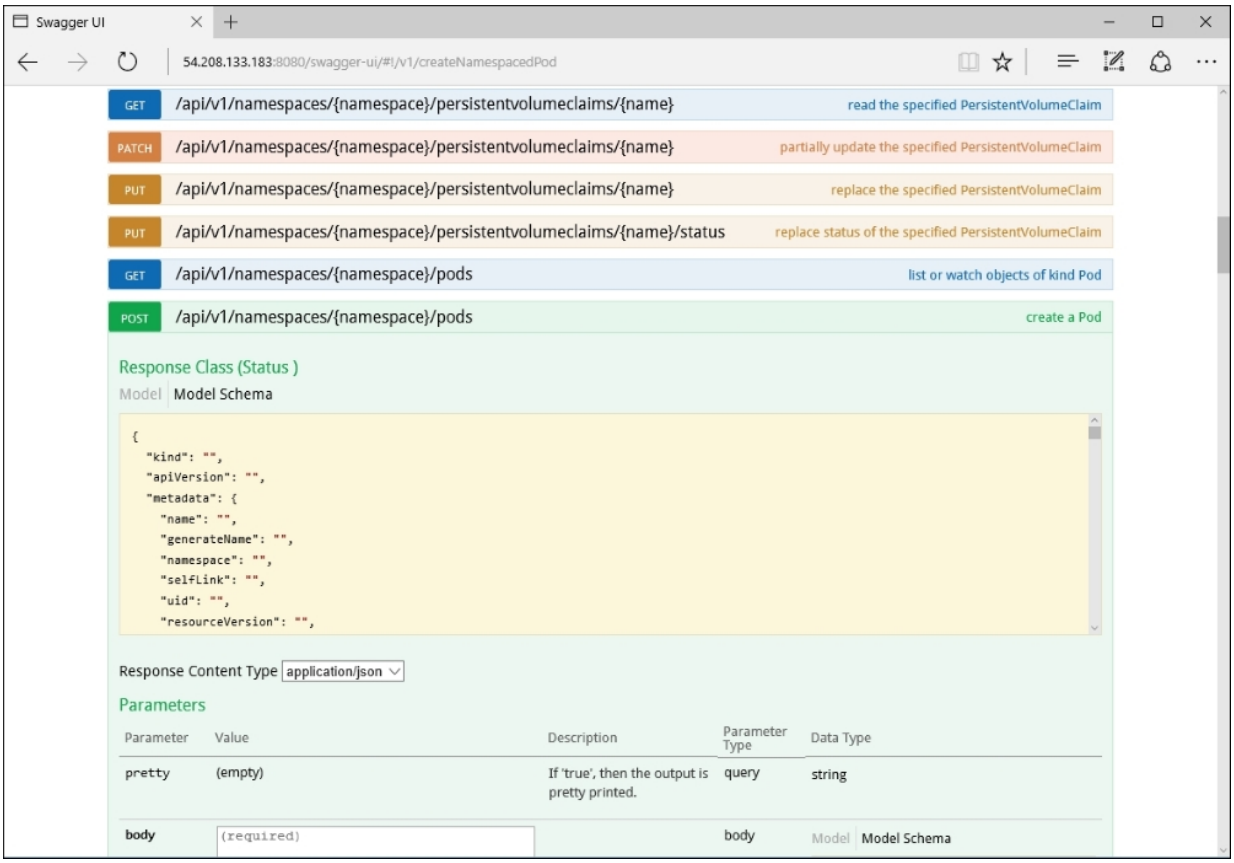
Kubernetes has a schema that is defined using the configuration format; schema can be generated in the `/tmp/kubectl.schema/` directory on executing the `kubectl create` command as follows:

```
# cat pod.json
{
    "apiVersion" : "v1",
    "kind" : "Pod",
    "metadata" : {
      "name" : "nginx",
      "labels": {
        "name": "nginx"
      }
    },
    "spec" : {
      "containers" : [
        {
            "name" : "nginx",
            "image" : "nginx",
            "ports" : [
               {
                 "containerPort": 80
               }
            ]
        }
      ]
    }
}

# kubectl create -f pod.json
pod "nginx" created

# ls -l /tmp/kubectl.schema/v1.1.3/api/v1/schema.json
-rw------- 2 root root 446224 Jan 24 04:50
/tmp/kubectl.schema/v1.1.3/api/v1/schema.json
```

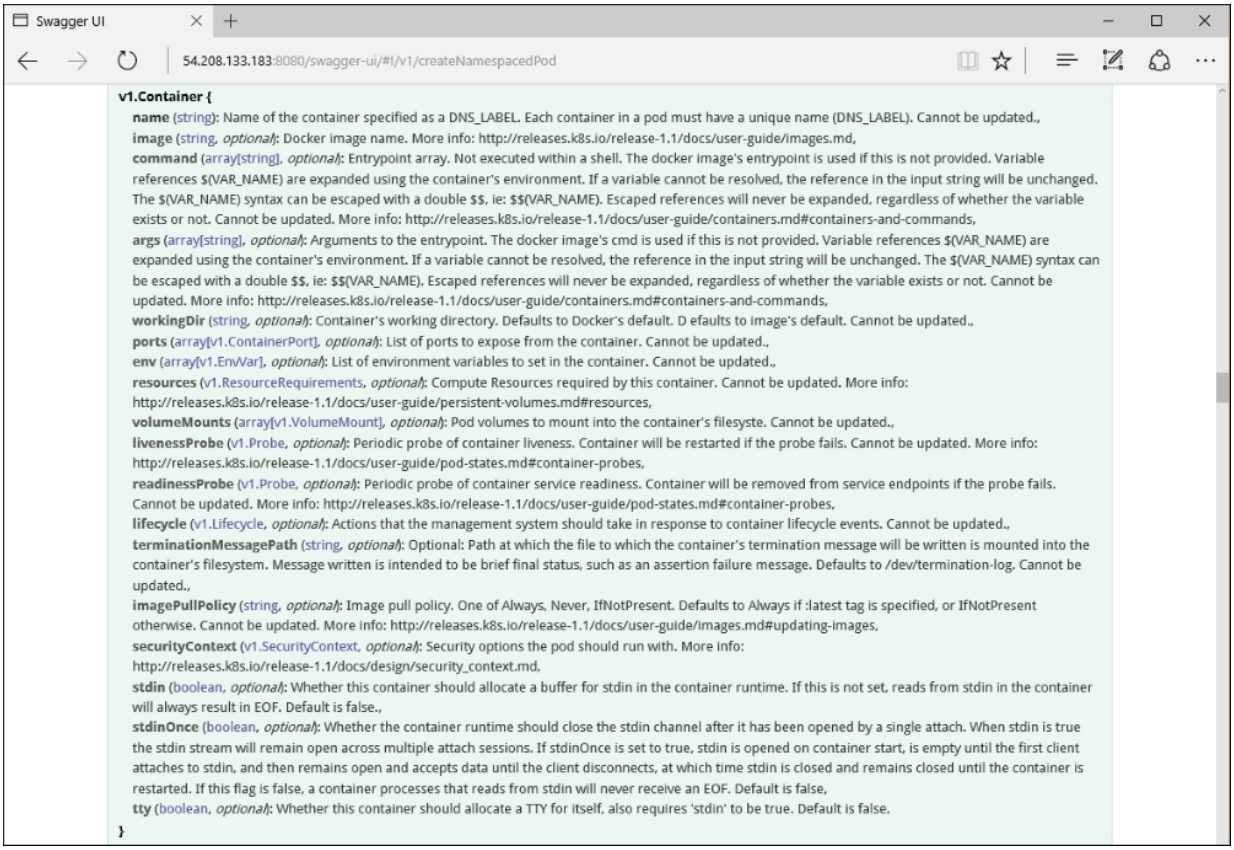
There is an alternative way, because Kubernetes is also using swagger (http://swagger.io/) to generate the REST API; therefore, you can access `swagger-ui` via

`http://<kubernetes-master>:8080/swagger-ui/`.

Each configuration, for example, pods, replication controllers, and services are described in the **POST** section, as shown in the following screenshot:



The preceding screenshot of `swagger-ui`, shows the pod's definition. Types of items, such as `string`, `array`, and `integer` are shown when you click on **Model** as follows:



The preceding screenshot shows the pod container definition. There are many items that are defined; however, some of them are indicated as `optional`, which is not necessary and applied as a default value or not set if you don't specify it.

> **Tip**
>
> Some of the items are indicated as `readonly`, such as UID. Kubernetes generates these items. If you specify this in the configuration file, it will be ignored.

How it works…

There are some mandatory items that need to be defined in each configuration file, as follows:

Pods

Item	Type	Example
apiVersion	String	v1

Item	Type	Example
kind	String	Pod
metadata.name	String	my-nginx
spec	v1.PodSpec	
v1.PodSpec.containers	array[v1.Container]	
v1.Container.name	String	my-nginx
v1.Container.image	String	nginx

Therefore, the minimal pod configuration is as follows (in the YAML format):

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx
spec:
  containers:
  - name: my-nginx
    image: nginx
```

Replication controllers

Item	Type	Example
apiVersion	String	v1
kind	String	ReplicationController
metadata.name	String	my-nginx-rc

Item	Type	Example
spec	v1.ReplicationControllerSpec	
v1.ReplicationControllerSpec.template	v1.PodTemplateSpec	
v1.PodTemplateSpec.metadata.labels	Map of String	app: nginx
v1.PodTemplateSpec.spec	v1.PodSpec	
v1.PodSpec.containers	array[v1.Container]	As same as pod

The following example is the minimal configuration of the replication controller (in the YAML format):

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx-rc
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
```

Services

Item	Type	Example
apiVersion	String	v1
kind	String	Service

Item	Type	Example
metadata.name	String	my-nginx-service
spec	v1.ServiceSpec	
v1.ServiceSpec.selector	Map of String	sel: my-selector
v1.ServiceSpec.ports	array[v1.ServicePort]	
v1.ServicePort.protocol	String	TCP
v1.ServicePort.port	Integer	80

The following example is the minimal configuration of service (in the YAML format):

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx

spec:
  selector:
     sel: my-selector
  ports:
    - protocol: TCP
      port: 80
```

See also

This recipe described how to find and understand a configuration syntax. Kubernetes has some detailed options to define container and components. For more details, the following recipes will describe how to define pods, replication controllers, and services:

- The *Working with pods*, *Working with a replication controller*, and *Working with services* recipes in Chapter 2, *Walking through Kubernetes Concepts*

Chapter 4. Building a High Availability Cluster

In this chapter, we will cover the following topics:

- Clustering etcd
- Building multiple masters

Introduction

Avoiding a single point of failure is a concept we need to always keep in mind. In this chapter, you will learn how to build components in Kubernetes with high availability. We will also go through the steps to build a three-node etcd cluster and masters with multinodes.

Clustering etcd

etcd stores network information and states in Kubernetes. Any data loss could be crucial. Clustering is strongly recommended in etcd. etcd comes with support for clustering; a cluster of *N* members can tolerate up to *(N-1)/2* failures. There are three mechanisms for creating an etcd cluster. They are as follows:

- Static
- etcd discovery
- DNS discovery

In this recipe, we will discuss how to bootstrap an etcd cluster by static and etcd discovery.

Getting ready

Before you start building an etcd cluster, you have to decide how many members you need. How big the etcd cluster should be really depends on the environment you want to create. In the production environment, at least three members are recommended. Then, the cluster can tolerate at least one permanent failure. In this recipe, we will use three members as an example of the development environment:

Name/Hostname	**IP address**
`ip-172-31-0-1`	`172.31.0.1`
`ip-172-31-0-2`	`172.31.0.2`

Name/Hostname	IP address
`ip-172-31-0-3`	`172.31.0.3`

How to do it…

A static mechanism is the easiest way to set up a cluster. However, the IP address of every member should be known beforehand. It means that if you bootstrap an etcd cluster in some cloud provider environment, the static mechanism might not be so practical. Therefore, etcd also provides a discovery mechanism to bootstrap itself from the existing cluster.

Static

With a static mechanism, you have to know the address information of each member:

Parameters	Meaning
`-name`	The name of this member
`-initial-advertise-peer-urls`	Used to peer with other members, should be the same as the one listing in `-initial-cluster`
`-listen-peer-urls`	The URL to accept peer traffic

Parameters	Meaning
`-listen-client-urls`	The URL to accept client traffic
`-advertise-client-urls`	etcd member used to advertise to other members
`-initial-cluster-token`	A unique token for distinguishing different clusters
`-initial-cluster`	Advertised peer URLs of all the members
`-initial-cluster-state`	Specifies the state of the initial cluster

Use the etcd command-line tool to bootstrap a cluster with additional parameters on each member:

```
// on the host ip-172-31-0-1, running etcd command
to make it peer with ip-172-31-0-2 and ip-172-31-
0-3, advertise and listen other members via port
2379, and accept peer traffic via port 2380
# etcd -name ip-172-31-0-1 \
       -initial-advertise-peer-urls
http://172.31.0.1:2380 \
       -listen-peer-urls http://172.31.0.1:2380 \
       -listen-client-urls http://0.0.0.0:2379 \
       -advertise-client-urls
http://172.31.0.1:2379 \
       -initial-cluster-token mytoken \
       -initial-cluster ip-172-31-0-
```

```
1=http://172.31.0.1:2380,ip-172-31-0-
2=http://172.31.0.2:2380,ip-172-31-0-
3=http://172.31.0.3:2380 \
       -initial-cluster-state new
...
2016-05-01 18:57:26.539787 I | etcdserver:
starting member e980eb6ff82d4d42 in cluster
8e620b738845cd7
2016-05-01 18:57:26.551610 I | etcdserver:
starting server... [version: 2.2.5, cluster
version: to_be_decided]
2016-05-01 18:57:26.553100 N | etcdserver: added
member 705d980456f91652 [http://172.31.0.3:2380]
to cluster 8e620b738845cd7
2016-05-01 18:57:26.553192 N | etcdserver: added
member 74627c91d7ab4b54 [http://172.31.0.2:2380]
to cluster 8e620b738845cd7
2016-05-01 18:57:26.553271 N | etcdserver: added
local member e980eb6ff82d4d42
[http://172.31.0.1:2380] to cluster
8e620b738845cd7
2016-05-01 18:57:26.553349 E | rafthttp: failed to
dial 705d980456f91652 on stream MsgApp v2 (dial
tcp 172.31.0.3:2380: getsockopt: connection
refused)
2016-05-01 18:57:26.553392 E | rafthttp: failed to
dial 705d980456f91652 on stream Message (dial tcp
172.31.0.3:2380: getsockopt: connection refused)
2016-05-01 18:57:26.553424 E | rafthttp: failed to
dial 74627c91d7ab4b54 on stream Message (dial tcp
172.31.0.2:2380: getsockopt: connection refused)
2016-05-01 18:57:26.553450 E | rafthttp: failed to
dial 74627c91d7ab4b54 on stream MsgApp v2 (dial
tcp 172.31.0.2:2380: getsockopt: connection
refused)
```

The `etcd` daemon on `ip-172-31-0-1` will then start checking whether all the members are online. The logs show connection refused since `ip-172-31-0-2` and `ip-172-31-0-3` are still offline. Let's go to the next member and run the `etcd` command:

```
// on the host ip-172-31-0-2, running etcd command
to make it peer with ip-172-31-0-1 and ip-172-31-
0-3, advertise and listen other members via port
```

```
2379, and accept peer traffic via port 2380
# etcd -name ip-172-31-0-2 \
       -initial-advertise-peer-urls
http://172.31.0.2:2380 \
       -listen-peer-urls http://172.31.0.2:2380 \
       -listen-client-urls http://0.0.0.0:2379 \
       -advertise-client-urls
http://172.31.0.2:2379 \
       -initial-cluster-token mytoken \
       -initial-cluster ip-172-31-0-
1=http://172.31.0.1:2380,ip-172-31-0-
2=http://172.31.0.2:2380, ip-172-31-0-
3=http://172.31.0.3:2380 -initial-cluster-state
new
...
2016-05-01 22:59:55.696357 I | etcdserver:
starting member 74627c91d7ab4b54 in cluster
8e620b738845cd7
2016-05-01 22:59:55.696397 I | raft:
74627c91d7ab4b54 became follower at term 0
2016-05-01 22:59:55.696407 I | raft: newRaft
74627c91d7ab4b54 [peers: [], term: 0, commit: 0,
applied: 0, lastindex: 0, lastterm: 0]
2016-05-01 22:59:55.696411 I | raft:
74627c91d7ab4b54 became follower at term 1
2016-05-01 22:59:55.706552 I | etcdserver:
starting server... [version: 2.2.5, cluster
version: to_be_decided]
2016-05-01 22:59:55.707627 E | rafthttp: failed to
dial 705d980456f91652 on stream MsgApp v2 (dial
tcp 172.31.0.3:2380: getsockopt: connection
refused)
2016-05-01 22:59:55.707690 N | etcdserver: added
member 705d980456f91652 [http://172.31.0.3:2380]
to cluster 8e620b738845cd7
2016-05-01 22:59:55.707754 N | etcdserver: added
local member 74627c91d7ab4b54
[http://172.31.0.2:2380] to cluster
8e620b738845cd7
2016-05-01 22:59:55.707820 N | etcdserver: added
member e980eb6ff82d4d42 [http://172.31.0.1:2380]
to cluster 8e620b738845cd7
2016-05-01 22:59:55.707873 E | rafthttp: failed to
dial 705d980456f91652 on stream Message (dial tcp
172.31.0.3:2380: getsockopt: connection refused)
2016-05-01 22:59:55.708433 I | rafthttp: the
connection with e980eb6ff82d4d42 became active
2016-05-01 22:59:56.196750 I | raft:
```

```
74627c91d7ab4b54 is starting a new election at
term 1
2016-05-01 22:59:56.196903 I | raft:
74627c91d7ab4b54 became candidate at term 2
2016-05-01 22:59:56.196946 I | raft:
74627c91d7ab4b54 received vote from
74627c91d7ab4b54 at term 2
2016-05-01 22:59:56.949201 I | raft: raft.node:
74627c91d7ab4b54 elected leader e980eb6ff82d4d42
at term 112
2016-05-01 22:59:56.961883 I | etcdserver:
published {Name:ip-172-31-0-2 ClientURLs:
[http://10.0.0.2:2379]} to cluster 8e620b738845cd7
2016-05-01 22:59:56.966981 N | etcdserver: set the
initial cluster version to 2.1
```

After starting member 2, we can see that the current cluster version is `2.1`. The following error message shows the connection to peer `705d980456f91652` is unhealthy. By observing the log, we can find that member `705d980456f91652` is pointing to `http://172.31.0.3:2380`. Let's start up the last member `ip-172-31-0-3`:

```
# etcd -name ip-172-31-0-3 \
       -initial-advertise-peer-urls
http://172.31.0.3:2380 \
       -listen-peer-urls http://172.31.0.3:2380 \
       -listen-client-urls http://0.0.0.0:2379 \
       -advertise-client-urls
http://172.31.0.3:2379 \
       -initial-cluster-token mytoken \
       -initial-cluster ip-172-31-0-
1=http://172.31.0.1:2380,ip-172-31-0-
2=http://172.31.0.2:2380, ip-172-31-0-
3=http://172.31.0.3:2380 -initial-cluster-state
new
2016-05-01 19:02:19.106540 I | etcdserver:
starting member 705d980456f91652 in cluster
8e620b738845cd7
2016-05-01 19:02:19.106590 I | raft:
705d980456f91652 became follower at term 0
2016-05-01 19:02:19.106608 I | raft: newRaft
705d980456f91652 [peers: [], term: 0, commit: 0,
applied: 0, lastindex: 0, lastterm: 0]
2016-05-01 19:02:19.106615 I | raft:
```

```
705d980456f91652 became follower at term 1
2016-05-01 19:02:19.118330 I | etcdserver:
starting server... [version: 2.2.5, cluster
version: to_be_decided]
2016-05-01 19:02:19.120729 N | etcdserver: added
local member 705d980456f91652
[http://10.0.0.75:2380] to cluster 8e620b738845cd7
2016-05-01 19:02:19.120816 N | etcdserver: added
member 74627c91d7ab4b54 [http://10.0.0.204:2380]
to cluster 8e620b738845cd7
2016-05-01 19:02:19.120887 N | etcdserver: added
member e980eb6ff82d4d42 [http://10.0.0.205:2380]
to cluster 8e620b738845cd7
2016-05-01 19:02:19.121566 I | rafthttp: the
connection with 74627c91d7ab4b54 became active
2016-05-01 19:02:19.121690 I | rafthttp: the
connection with e980eb6ff82d4d42 became active
2016-05-01 19:02:19.143351 I | raft:
705d980456f91652 [term: 1] received a MsgHeartbeat
message with higher term from e980eb6ff82d4d42
[term: 112]
2016-05-01 19:02:19.143380 I | raft:
705d980456f91652 became follower at term 112
2016-05-01 19:02:19.143403 I | raft: raft.node:
705d980456f91652 elected leader e980eb6ff82d4d42
at term 112
2016-05-01 19:02:19.146582 N | etcdserver: set the
initial cluster version to 2.1
2016-05-01 19:02:19.151353 I | etcdserver:
published {Name:ip-172-31-0-3 ClientURLs:
[http://10.0.0.75:2379]} to cluster
8e620b738845cd7
2016-05-01 19:02:22.022578 N | etcdserver: updated
the cluster version from 2.1 to 2.2
```

We can see, on member 3, we successfully initiated the etcd cluster without any errors and the current cluster version is `2.2`. How about member 1 now?

```
2016-05-01 19:02:19.118910 I | rafthttp: the
connection with 705d980456f91652 became active
2016-05-01 19:02:22.014958 I | etcdserver:
updating the cluster version from 2.1 to 2.2
```

```
2016-05-01 19:02:22.018530 N | etcdserver: updated
the cluster version from 2.1 to 2.2
```

With member 2 and 3 online, member 1 can now get connected and go online too. When observing the log, we can see the leader election took place in the etcd cluster:

```
ip-172-31-0-1: raft: raft.node: e980eb6ff82d4d42
(ip-172-31-0-1) elected leader e980eb6ff82d4d42
(ip-172-31-0-1) at term 112
ip-172-31-0-2: raft: raft.node: 74627c91d7ab4b54
(ip-172-31-0-2) elected leader e980eb6ff82d4d42
(ip-172-31-0-1) at term 112
ip-172-31-0-3: 2016-05-01 19:02:19.143380 I |
raft: 705d980456f91652 became follower at term 112
```

The etcd cluster will send the heartbeat to the members in the cluster to check the health status. Note that when you need to add or remove any members in the cluster, the preceding `etcd` command needs to be rerun on all the members in order to notify that there are new members joining the cluster. In this way, all the members in the cluster are aware of all the online members; if one node goes offline, the other members will poll the failure member until it refreshes members by the `etcd` command. If we set a message from a member, we could get the same message from the other members too. If one member becomes unhealthy, the other members in the etcd cluster will still be in service and elect for a new leader.

etcd discovery

Before using the etcd discovery, you should have a discovery URL that is used to bootstrap a cluster. If you want to add or remove a member, you should use the `etcdctl` command as the runtime reconfiguration. The command line is pretty much the same as the static mechanism. What we need to do is change `-initial-cluster` to `-discovery`, which is used to specify the discovery service URL. We could use the etcd discovery service (https://discovery.etcd.io) to request a discovery URL:

```
// get size=3 cluster url from etcd discovery
service
# curl -w "\n" 'https://discovery.etcd.io/new?
size=3'
https://discovery.etcd.io/be7c1938bbde83358d8ae978
895908bd
// Init a cluster via requested URL
# etcd -name ip-172-31-0-1 -initial-advertise-
peer-urls http://172.31.43.209:2380 \
 -listen-peer-urls http://172.31.0.1:2380 \
 -listen-client-urls http://0.0.0.0:2379 \
 -advertise-client-urls http://172.31.0.1:2379 \
 -discovery
https://discovery.etcd.io/be7c1938bbde83358d8ae978
895908bd
...
2016-05-02 00:28:08.545651 I | etcdmain: listening
for peers on http://172.31.0.1:2380
2016-05-02 00:28:08.545756 I | etcdmain: listening
for client requests on http://127.0.0.1:2379
2016-05-02 00:28:08.545807 I | etcdmain: listening
for client requests on http://172.31.0.1:2379
2016-05-02 00:28:09.199987 N | discovery: found
self e980eb6ff82d4d42 in the cluster
2016-05-02 00:28:09.200010 N | discovery: found 1
peer(s), waiting for 2 more
```

The first member has joined the cluster; wait for the other two peers. Let's start `etcd` in the second node:

```
# etcd -name ip-172-31-0-2 -initial-advertise-
peer-urls http://172.31.0.2:2380 \
 -listen-peer-urls http://172.31.0.2:2380 \
 -listen-client-urls http://0.0.0.0:2379 \
 -advertise-client-urls http://172.31.0.2:2379 \
 -discovery
https://discovery.etcd.io/be7c1938bbde83358d8ae978
895908bd
...
2016-05-02 00:30:12.919005 I | etcdmain: listening
for peers on http://172.31.0.2:2380
2016-05-02 00:30:12.919074 I | etcdmain: listening
for client requests on http://0.0.0.0:2379
```

```
2016-05-02 00:30:13.018160 N | discovery: found
self 25fc8075ab1ed17e in the cluster
2016-05-02 00:30:13.018235 N | discovery: found 1
peer(s), waiting for 2 more
2016-05-02 00:30:22.985300 N | discovery: found
peer e980eb6ff82d4d42 in the cluster
2016-05-02 00:30:22.985396 N | discovery: found 2
peer(s), waiting for 1 more
```

We know there are two members in etcd already and they are waiting for the last one to join. The following code starts the last node:

```
# etcd -name ip-172-31-0-3 -initial-advertise-
peer-urls http://172.31.0.3:2380 \
 -listen-peer-urls http://172.31.0.3:2380 \
 -listen-client-urls http://0.0.0.0:2379 \
 -advertise-client-urls http://172.31.0.3:2379 \
 -discovery
https://discovery.etcd.io/be7c1938bbde83358d8ae978
895908bd
```

After new nodes join, we can check from the logs that there is a new election taking place:

```
2016-05-02 00:31:01.152215 I | raft:
e980eb6ff82d4d42 is starting a new election at
term 308
2016-05-02 00:31:01.152272 I | raft:
e980eb6ff82d4d42 became candidate at term 309
2016-05-02 00:31:01.152281 I | raft:
e980eb6ff82d4d42 received vote from
e980eb6ff82d4d42 at term 309
2016-05-02 00:31:01.152292 I | raft:
e980eb6ff82d4d42 [logterm: 304, index: 9739] sent
vote request to 705d980456f91652 at term 309
2016-05-02 00:31:01.152302 I | raft:
e980eb6ff82d4d42 [logterm: 304, index: 9739] sent
vote request to 74627c91d7ab4b54 at term 309
2016-05-02 00:31:01.162742 I | rafthttp: the
connection with 74627c91d7ab4b54 became active
2016-05-02 00:31:01.197820 I | raft:
```

```
e980eb6ff82d4d42 received vote from
74627c91d7ab4b54 at term 309
2016-05-02 00:31:01.197852 I | raft:
e980eb6ff82d4d42 [q:2] has received 2 votes and 0
vote rejections
2016-05-02 00:31:01.197882 I | raft:
e980eb6ff82d4d42 became leader at term 309
```

With the discovery method, we can see that the cluster can be launched without knowing the others' IPs beforehand. etcd will start a new election if new nodes join or leave, and always keep the service online with the multi-nodes setting.

See also

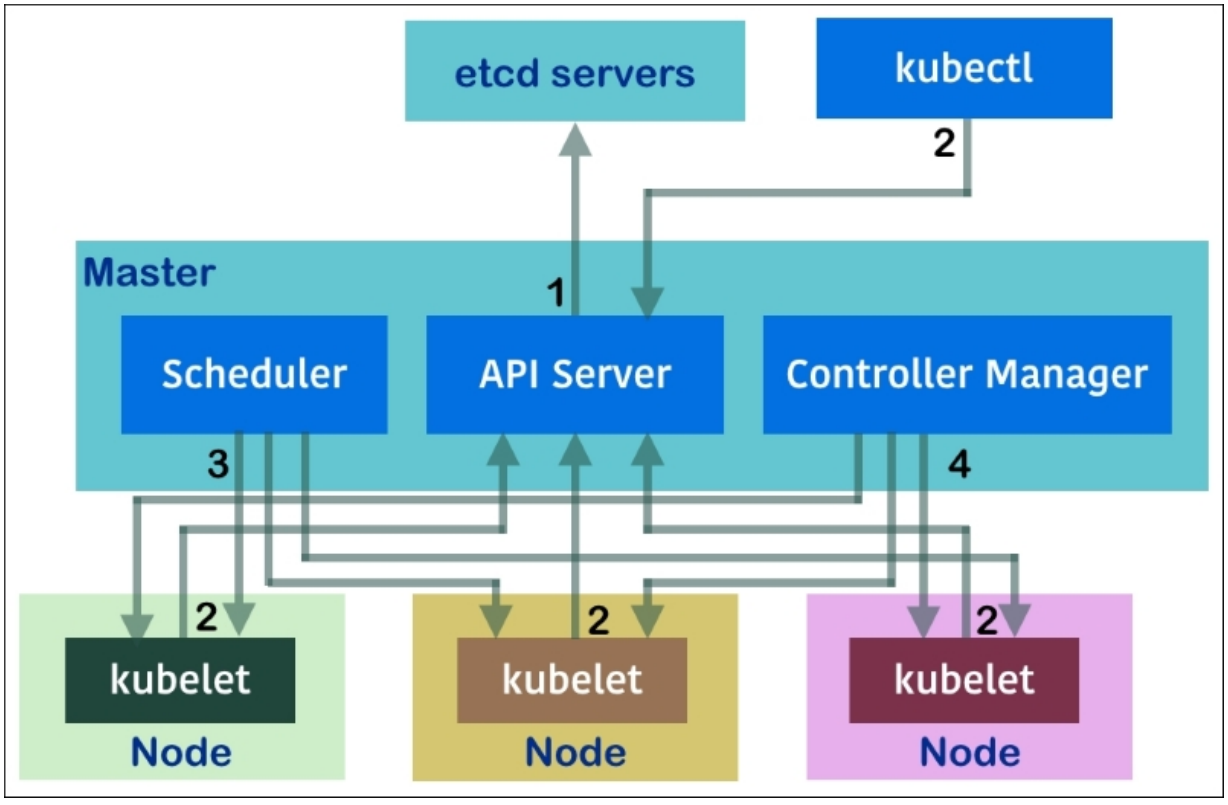
To understand the installation of a single etcd server, refer to the *Building datastore* recipe in Chapter 1, *Building Your Own Kubernetes*.

Building multiple masters

The master node serves as a kernel component in the Kubernetes system. Its duties include the following:

- Pushing and pulling information from the datastore and the etcd servers
- Being the portal for requests
- Assigning tasks to nodes
- Monitoring the running tasks

Three major daemons support the master fulfilling the preceding duties, which are numbered in the following image:



As you can see, the master is the communicator between workers and clients. Therefore, it will be a problem if the master node crashes. A multiple-master Kubernetes system is not only fault tolerant, but also workload-balanced. There will be no longer only

one API server for accessing nodes and clients sending requests. Several API server daemons in separated master nodes would help to solve the tasks simultaneously and shorten the response time.

Getting ready

The brief concepts for building a multiple-master system are listed here:

- Add a load balancer server in front of the masters. The load balancer will become the new endpoint accessed by nodes and clients.
- Every master runs its own API server daemon.
- Only one scheduler and one controller manager are in the system, which can avoid conflict directions from different daemons while managing containers.
- `Pod master` is a new daemon installed in every master. It elects to decide the master node-running daemon scheduler and the master node-running controller manager. It could be the same master that runs both the daemons.
- Make a more flexible way to run a daemon scheduler and a controller manager as a container. Install kubelet in master and manage the daemons as pods by configuring files.

In this recipe, we are going to build a two-master system, which has similar methods while scaling more masters.

How to do it…

Now, we will guide you step by step in building a multiple-master system. Before this, you have to deploy a load balancer server for masters.

> **Note**

To learn about deploying the load balancer and to build the system on AWS, please check the *Building the Kubernetes infrastructure in AWS* recipe in Chapter 6, *Building Kubernetes on AWS* on how to build a master load balancer.

Preparing multiple master nodes

First, install another master node in your previous Kubernetes system, which should be in the same environment as the original master. Then, stop the daemon services of scheduler and controller manager in both the masters:

- For the systemd-controlled system, stop the services directly using the commands `systemctl kube-scheduler stop` and `systemctl kube-controller-manager stop`
- For the init service-controlled system, stop the master service first. Next, delete or comment on the lines about scheduler and controller manager in the initialization script:

```
// Checking current daemon processes on master
server
# service kubernetes-master status
kube-apiserver (pid 3137) is running...
kube-scheduler (pid 3138) is running...
kube-controller-manager (pid 3136) is
running...
# service kubernetes-master stop
Shutting down /usr/local/bin/kube-controller-
manager:       [  OK  ]
Shutting down /usr/local/bin/kube-scheduler:
[  OK  ]
Shutting down /usr/local/bin/kube-apiserver:
[  OK  ]
// Or, for "hypercube" command with init
script, we block out scheduler and controller-
manager. Just leave apiserver daemon in master
node.
// Put comment on the scheduler and controller
manager daemons
// the variable $prog is
/usr/local/bin/hyperkube
```

```
# cat /etc/init.d/kubernetes-master
(ignored above parts)
       # Start daemon.
       echo $"Starting apiserver: "
       daemon $prog apiserver \
       --service-cluster-ip-
range=${CLUSTER_IP_RANGE} \
       --insecure-port=8080 \
       --secure-port=6443 \
       --address=0.0.0.0 \
       --etcd_servers=${ETCD_SERVERS} \
       --cluster_name=${CLUSTER_NAME} \
       > ${logfile}-apiserver.log 2>&1 &

#       echo $"Starting controller-manager: "
#       daemon $prog controller-manager \
#       --master=${MASTER} \
#       > ${logfile}-controller-manager.log
2>&1 &
#
#       echo $"Starting scheduler: "
#       daemon $prog scheduler \
#       --master=${MASTER} \
#       > ${logfile}-scheduler.log 2>&1 &
(ignored below parts)
# service kubernetes-master start
Starting apiserver:
```

At this step, you have two masters serving in the system with two processes of the API server.

Setting up kubelet in master

Because we are going to install the daemons' scheduler and controller manager as pods, a kubelet process is a must-have daemon. Download the latest (version `1.1.4`) kubelet binary file (https://storage.googleapis.com/kubernetes-release/release/v1.1.4/bin/linux/amd64/kubelet) and put it under the directory of the system's binary files:

```
# wget https://storage.googleapis.com/kubernetes-
release/release/v1.1.4/bin/linux/amd64/kubelet
```

```
# chmod 755 kubelet
# mv kubelet /usr/local/bin/
```

Alternatively, for the RHEL system, you can download `kubelet` from the YUM repository:

```
# yum install kubernetes-node
```

Later, we will configure the `kubelet` daemon with specific parameters and values:

Tag Name	Value	Purpose
`--api-servers`	`127.0.0.1:8080`	To communicate with the API server in local.
`--register-node`	`false`	Avoid registering this master, local host, as a node.
`--allow-privileged`	`true`	To allow containers to request the privileged mode, which means containers have the ability to access the host device, especially, the network device in this case.
`--config`	`/etc/kubernetes/manifests`	To manage local containers by the template files under

Tag Name	Value	Purpose
		this specified directory.

If your system is monitored by `systemctl`, put the preceding parameters in the configuration files:

- In `/etc/kubernetes/config`:
    - Modify `KUBE_MASTER` to `--master=127.0.0.1:8080`:

```
KUBE_LOGTOSTDERR="--logtostderr=true"
KUBE_LOG_LEVEL="--v=0"
KUBE_ALLOW_PRIV="--
allow_privileged=false"
KUBE_MASTER="--master=127.0.0.1:8080"
```

- In `/etc/kubernetes/kubelet`:
    - Put the tag `--api-servers` to variable `KUBELET_API_SERVER`.
    - Put the other three tags to variable `KUBELET_ARGS`:

```
KUBELET_ADDRESS="--address=0.0.0.0"
KUBELET_HOSTNAME="--
hostname_override=127.0.0.1"
KUBELET_API_SERVER="--
api_servers=127.0.0.1:8080"
KUBELET_ARGS="--register-node=false --
allow-privileged=true --config
/etc/kubernetes/manifests"
```

On the other hand, modify your script file of `init` service management and append the tags after the daemon `kubelet`. For example, we have the following settings in `/etc/init.d/kubelet`:

```
# cat /etc/init.d/kubelet
```

```
prog=/usr/local/bin/kubelet
lockfile=/var/lock/subsys/`basename $prog`
hostname=`hostname`
logfile=/var/log/kubernetes.log

start() {
        # Start daemon.
        echo $"Starting kubelet: "
        daemon $prog \
                --api-servers=127.0.0.1:8080 \
                --register-node=false \
                --allow-privileged=true \
                --config=/etc/kubernetes/manifests
\
                > ${logfile} 2>&1 &
      (ignored)
```

It is fine to keep your kubelet service in the stopped state, since we will start it after the configuration files of scheduler and the controller manager are ready.

Getting the configuration files ready

We need three templates as configuration files: pod master, scheduler, and controller manager. These files should be put at specified locations.

Pod master handles the elections to decide which master runs the scheduler daemon and which master runs the controller manager daemon. The result will be recorded in the etcd servers. The template of pod master is put in the kubelet config directory, making sure that the pod master is created right after kubelet starts running:

```
# cat /etc/kubernetes/manifests/podmaster.yaml
apiVersion: v1
kind: Pod
metadata:
 name: podmaster
 namespace: kube-system
spec:
 hostNetwork: true
```

```
 containers:
 - name: scheduler-elector
   image: gcr.io/google_containers/podmaster:1.1
   command: ["/podmaster", "--etcd-servers=
<ETCD_ENDPOINT>", "--key=scheduler", "--source-
file=/kubernetes/kube-scheduler.yaml", "--dest-
file=/manifests/kube-scheduler.yaml"]
   volumeMounts:
   - mountPath: /kubernetes
     name: k8s
     readOnly: true
   - mountPath: /manifests
     name: manifests
 - name: controller-manager-elector
   image: gcr.io/google_containers/podmaster:1.1
   command: ["/podmaster", "--etcd-servers=
<ETCD_ENDPOINT>", "--key=controller", "--source-
file=/kubernetes/kube-controller-manager.yaml", "-
-dest-file=/manifests/kube-controller-
manager.yaml"]
   terminationMessagePath: /dev/termination-log
   volumeMounts:
   - mountPath: /kubernetes
     name: k8s
     readOnly: true
   - mountPath: /manifests
     name: manifests
 volumes:
 - hostPath:
     path: /srv/kubernetes
   name: k8s
 - hostPath:
     path: /etc/kubernetes/manifests
   name: manifests
```

In the configuration file of pod master, we will deploy a pod with two containers, the two electors for different daemons. The pod `podmaster` is created in a new namespace called `kube-system` in order to separate pods for daemons and applications. We will need to create a new namespace prior to creating resources using templates. It is also worth mentioning that the path `/srv/kubernetes` is where we put the daemons' configuration files. The content of the files is like the following lines:

```
# cat /srv/kubernetes/kube-scheduler.yaml
apiVersion: v1
kind: Pod
metadata:
 name: kube-scheduler
 namespace: kube-system
spec:
 hostNetwork: true
 containers:
 - name: kube-scheduler
   image: gcr.io/google_containers/kube-
scheduler:34d0b8f8b31e27937327961528739bc9
   command:
   - /bin/sh
   - -c
   - /usr/local/bin/kube-scheduler --
master=127.0.0.1:8080 --v=2 1>>/var/log/kube-
scheduler.log 2>&1
   livenessProbe:
     httpGet:
       path: /healthz
       port: 10251
     initialDelaySeconds: 15
     timeoutSeconds: 1
   volumeMounts:
   - mountPath: /var/log/kube-scheduler.log
     name: logfile
   - mountPath: /usr/local/bin/kube-scheduler
     name: binfile
 volumes:
 - hostPath:
     path: /var/log/kube-scheduler.log
   name: logfile
 - hostPath:
     path: /usr/local/bin/kube-scheduler
   name: binfile
```

There are some special items set in the template, such as namespace and two mounted files. One is a log file; the streaming output can be accessed and saved in the local side. The other one is the execution file. The container can make use of the latest kube-scheduler on the local host:

```
# cat /srv/kubernetes/kube-controller-manager.yaml
apiVersion: v1
kind: Pod
metadata:
 name: kube-controller-manager
 namespace: kube-system
spec:
 containers:
 - command:
   - /bin/sh
   - -c
   - /usr/local/bin/kube-controller-manager --
master=127.0.0.1:8080 --cluster-cidr=
<KUBERNETES_SYSTEM_CIDR> --allocate-node-
cidrs=true --v=2 1>>/var/log/kube-controller-
manager.log 2>&1
   image: gcr.io/google_containers/kube-
controller-
manager:fda24638d51a48baa13c35337fcd4793
   livenessProbe:
     httpGet:
       path: /healthz
       port: 10252
     initialDelaySeconds: 15
     timeoutSeconds: 1
   name: kube-controller-manager
   volumeMounts:
   - mountPath: /srv/kubernetes
     name: srvkube
     readOnly: true
   - mountPath: /var/log/kube-controller-
manager.log
     name: logfile
   - mountPath: /usr/local/bin/kube-controller-
manager
     name: binfile
 hostNetwork: true
 volumes:
 - hostPath:
     path: /srv/kubernetes
   name: srvkube
 - hostPath:
     path: /var/log/kube-controller-manager.log
   name: logfile
 - hostPath:
```

```
      path: /usr/local/bin/kube-controller-manager
    name: binfile
```

The configuration file of the controller manager is similar to the one of the scheduler. Remember to provide the CIDR range of your Kubernetes system in the daemon command.

For the purpose of having your templates work successfully, there are still some preconfigurations required before you start the pod master:

- Create empty log files. Otherwise, instead of the file format, the container will regard the path as a directory and cause the error of pod creation:

```
// execute these commands on each master
# touch /var/log/kube-scheduler.log
# touch /var/log/kube-controller-manager.log
```

- Create the new namespace. The new namespace is separated from the default one. We are going to put the pod for system usage in this namespace:

```
// Just execute this command in a master, and
other masters can share this update.
# kubectl create namespace kube-system
// Or
# curl -XPOST -
d'{"apiVersion":"v1","kind":"Namespace","metad
ata":{"name":"kube-system"}}'
"http://127.0.0.1:8080/api/v1/namespaces"
```

Starting the kubelet service and turning daemons on!

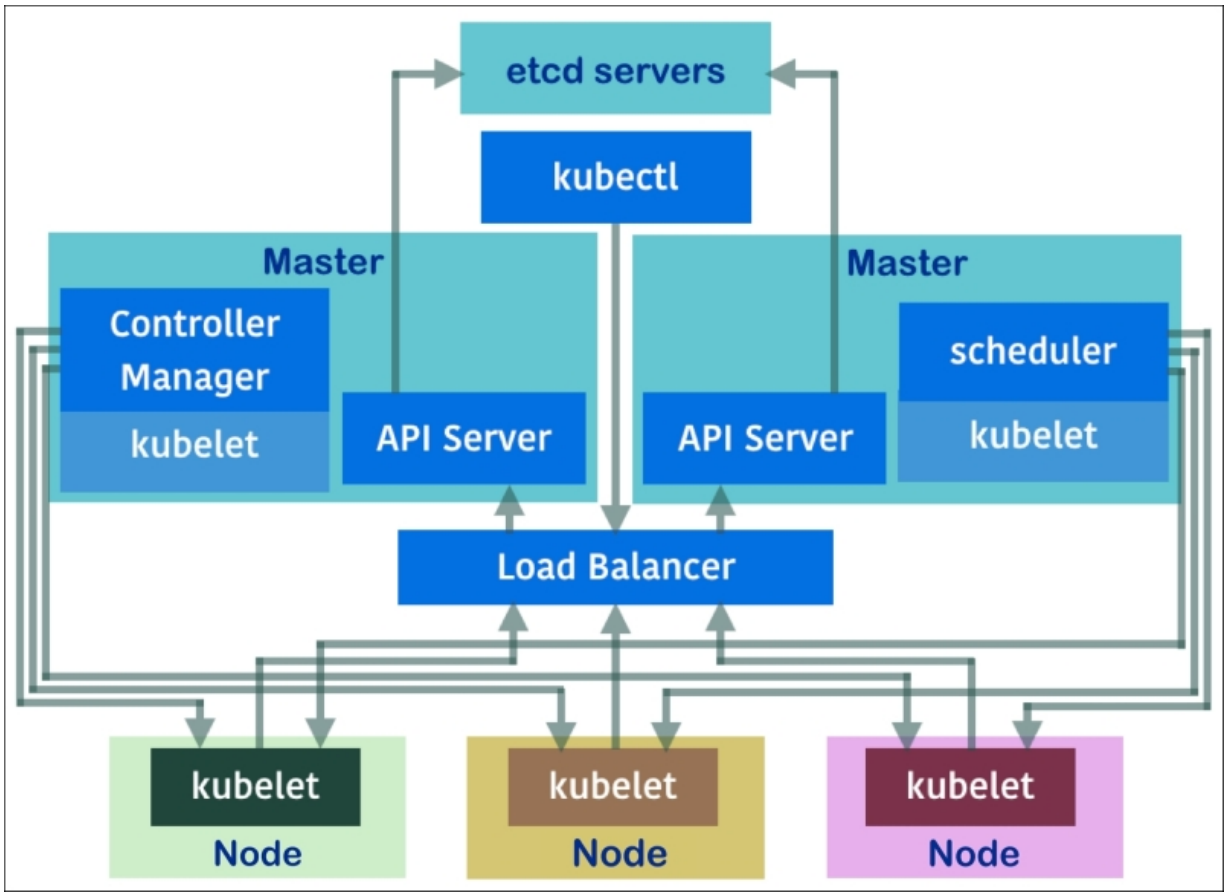
Before starting kubelet for our pod master and two master-owned daemons, please make sure you have Docker and flanneld started first:

```
# Now, it is good to start kubelet on every
masters
# service kubelet start
```

Wait for a while; you will get a pod master running on each master and you will finally get a pair of scheduler and controller manager:

```
# Check pods at namespace "kube-system"
# kubectl get pod --namespace=kube-system
NAME                                        READY
STATUS    RESTARTS   AGE
kube-controller-manager-kube-master1   1/1
Running   0          3m
kube-scheduler-kube-master2            1/1
Running   0          3m
podmaster-kube-master1                 2/2
Running   0          1m
podmaster-kube-master2                 2/2
Running   0          1m
```

Congratulations! You have your multiple-master Kubernetes system built up successfully. And the structure of the machines looks like following image:



You can see that now, a single node does not have to deal with the whole request load. Moreover, the daemons are not crowded in a master; they can be distributed to different masters and every master has the ability to do the recovery. Try to shut down one master; you will find that your scheduler and controller manager are still providing services.

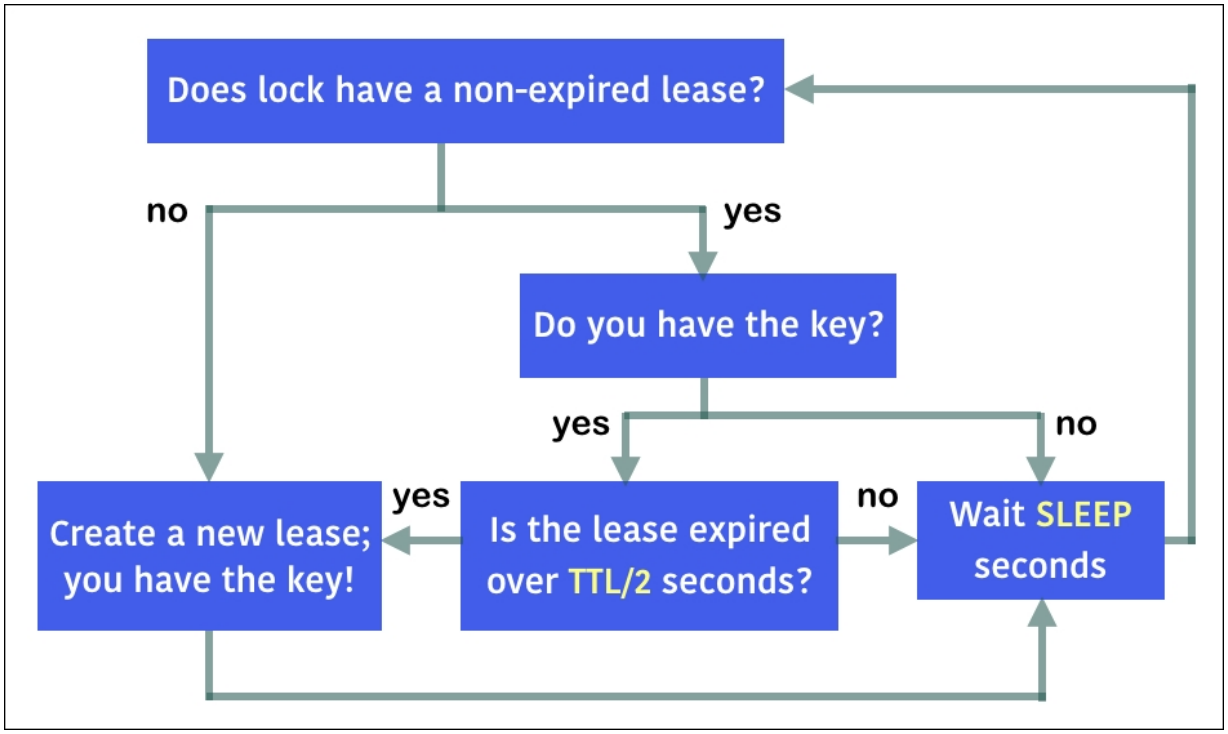
How it works…

Check the log of the container pod master; you will get two kinds of messages, one for who is holding the key and one without a key on hand:

```
// Get the log with specified container name
# kubectl logs podmaster-kube-master1 -c
scheduler-elector --namespace=kube-system
I0211 15:13:46.857372       1 podmaster.go:142] --
```

```
whoami is empty, defaulting to kube-master1
I0211 15:13:47.168724       1 podmaster.go:82] key
already exists, the master is kube-master2,
sleeping.
I0211 15:13:52.506880       1 podmaster.go:82] key
already exists, the master is kube-master2,
sleeping.
(ignored)
# kubectl logs podmaster-kube-master1 -c
controller-manager-elector --namespace=kube-system
I0211 15:13:44.484201       1 podmaster.go:142] --
whoami is empty, defaulting to kube-master1
I0211 15:13:50.078994       1 podmaster.go:73] key
already exists, we are the master (kube-master1)
I0211 15:13:55.185607       1 podmaster.go:73] key
already exists, we are the master (kube-master1)
(ignored)
```

The master with the key should take charge of the specific daemon and the said scheduler or controller manager. This current high-availability solution for the master is realized by the lease-lock method in etcd:



The preceding loop image indicates the progress of the lease-lock method. Two time periods are important in this method: **SLEEP** is the period for checking lock, and **Time to Live** (**TTL**) is the period of lease expiration. We can say that if the daemon-running master crashed, the worst case for the other master taking over its job requires the time **SLEEP** + **TTL**. By default, **SLEEP** is 5 seconds and **TTL** is 30 seconds.

> ## Note
>
> You can still take a look at the source code of pod master for more concepts (`podmaster.go`: https://github.com/kubernetes/contrib/blob/master/pod-master/podmaster.go).

See also

Before you read this recipe, you should have the basic concept of single master installation. Refer to the related recipes mentioned here and get an idea of how to build a multiple-master system automatically:

- The *Configuring master* recipe in Chapter 1, *Building Your Own Kubernetes*
- *Clustering etcd*
- The *Building the Kubernetes infrastructure in AWS* recipe in Chapter 6, *Building Kubernetes on AWS*

Chapter 5. Building a Continuous Delivery Pipeline

In this chapter, we will cover the following topics:

- Moving monolithic to microservices
- Integrating with Jenkins
- Working with the private Docker registry
- Setting up the Continuous Delivery pipeline

Introduction

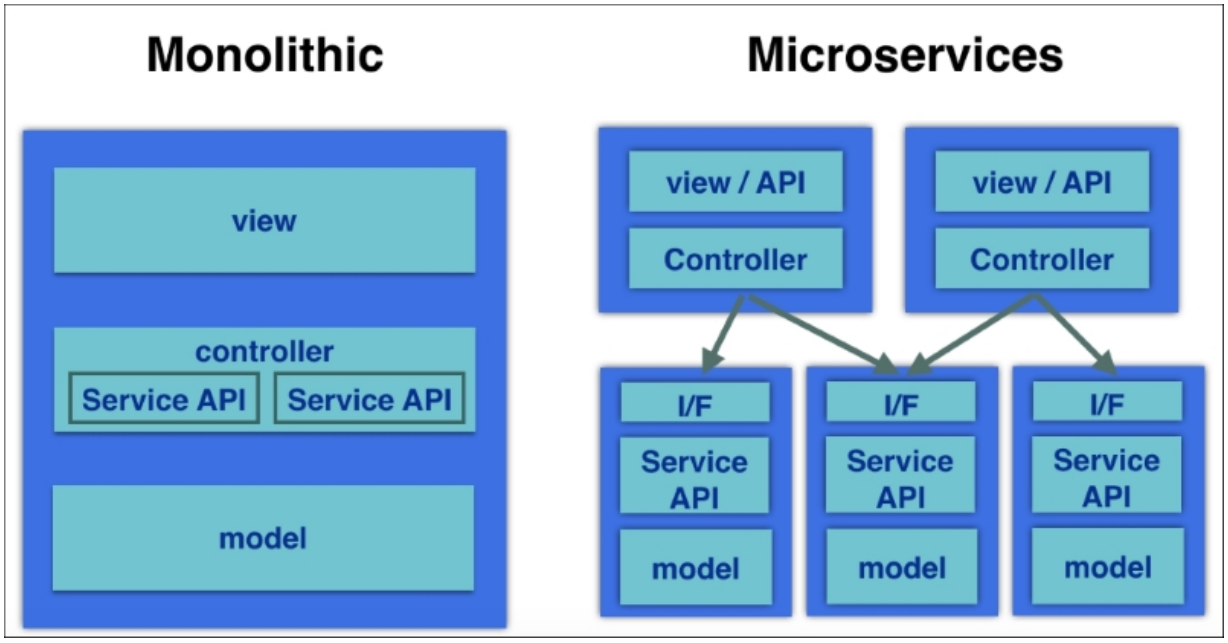
Kubernetes is a perfect match with applications featuring the microservices architecture. However, most of the old applications are all built in monolithic style. We will give you the idea about how to move from monolithic to the microservices world. As for microservices, deployment will become a burden if you are doing it manually. We will then learn how to build up our own Continuous Delivery pipeline by coordinating Jenkins, the Docker registry, and Kubernetes.

Moving monolithic to microservices

Typically, application architecture was the monolithic design that contains **Model-View-Controller** (**MVC**) and every component within a single big binary. Monolithic has some benefits, such as less latency within components, all in one straightforward packaging, and being easy to deploy and test.

However, a monolithic design has some downsides because the binary will be getting bigger and bigger. You always need to take care of the side effects when adding or modifying the code, therefore, making release cycles longer.

Containers and Kubernetes give more flexibility in using microservices for your application. The microservices architecture is very simple that can be divided into some modules or some service classes with MVC together.



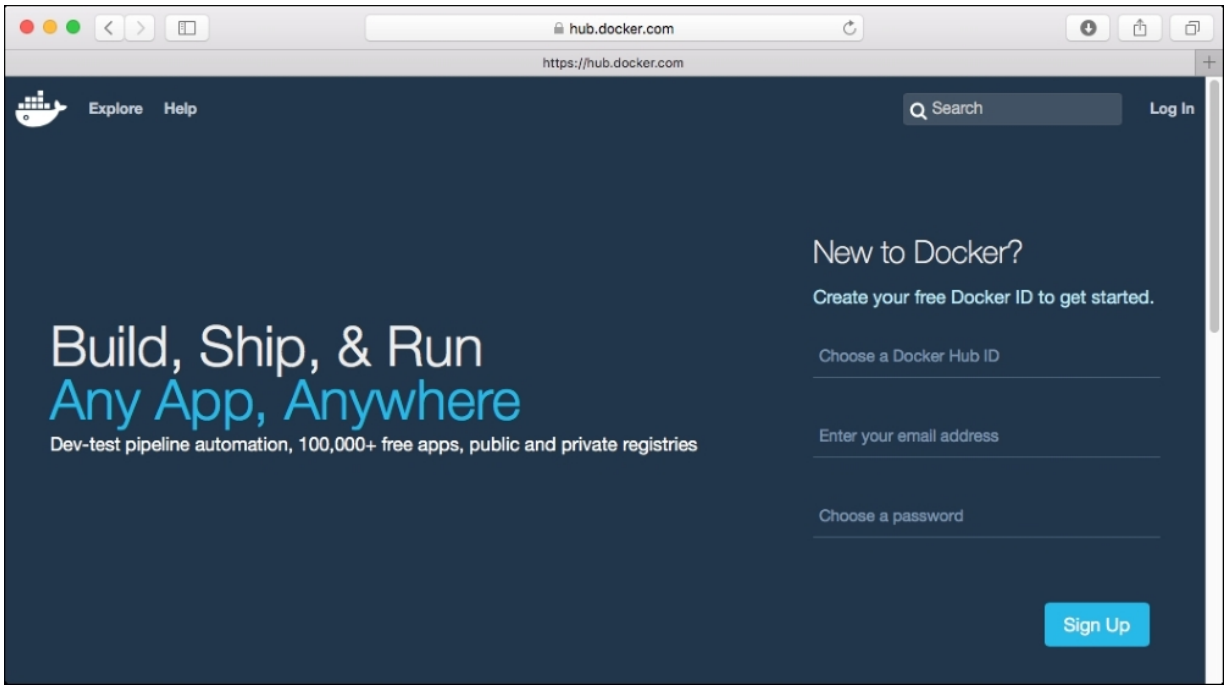
Monolithic and microservices design

Each microservice provides **Remote Procedure Call** (**RPC**) using RESTful or some standard network APIs to other microservices. The benefit is that each microservice is independent. There are minimal side effects when adding or modifying the code. Release the cycle independently, so it perfectly fits with the Agile software development methodology and allows to reuse these microservices to construct another application that builds the microservices ecosystem.

Getting ready

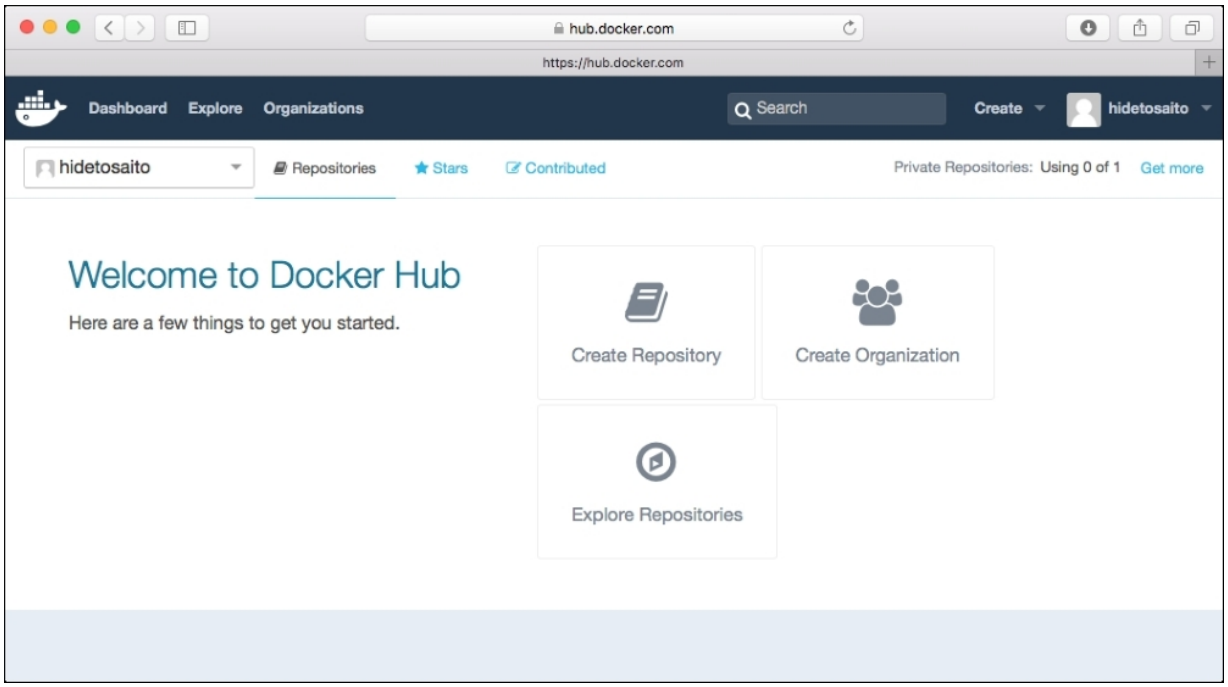
Prepare the simple microservices program. In order to push and pull your microservices, please register to Docker Hub (https://hub.docker.com/) to create your free Docker Hub ID in advance:

> ## Tip
>
> Attention: If you push the Docker image to Docker Hub, it will be public; anyone can pull your image. Therefore, don't put any confidential information into the image.



Docker Hub registration page

Once you successfully log in to your Docker Hub ID, you will be redirected to your **Dashboard** page as follows:



After logging to Docker Hub

How to do it…

Prepare both microservices and the Frontend WebUI as a Docker image. Then, deploy them using the Kubernetes replication controller and service.

Microservices

1. Here is the simple microservice using Python Flask (http://flask.pocoo.org/):

```
$ cat entry.py
from flask import Flask, request

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

@app.route("/power/<int:base>/<int:index>")
def power(base, index):
    return "%d" % (base ** index)

@app.route("/addition/<int:x>/<int:y>")
def add(x, y):
    return "%d" % (x+y)

@app.route("/substraction/<int:x>/<int:y>")
def substract(x, y):
    return "%d" % (x-y)


if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

2. Prepare a `Dockerfile` as follows in order to build the Docker image:

```
$ cat Dockerfile
FROM ubuntu:14.04
```

```
# Update packages
RUN apt-get update -y

# Install Python Setuptools
RUN apt-get install -y python-setuptools git
telnet curl

# Install pip
RUN easy_install pip

# Bundle app source
ADD . /src
WORKDIR /src

# Add and install Python modules
RUN pip install Flask

# Expose
EXPOSE  5000

# Run
CMD ["python", "entry.py"]
```

3. Then, use the `docker build` command to build the Docker image as follows:

> ## Tip
>
> If you publish the Docker image, you should use `Docker Hub ID/image name` as the Docker image name.

```
//name as "your_docker_hub_id/my-calc"
$ sudo docker build -t hidetosaito/my-calc .
Sending build context to Docker daemon 3.072
kB
Step 1 : FROM ubuntu:14.04
 ---> 6cc0fc2a5ee3
Step 2 : RUN apt-get update -y
 ---> Using cache
```

```
(snip)

Step 8 : EXPOSE 5000
 ---> Running in 7c52f4bfe373
 ---> 28f79bb7481f
Removing intermediate container 7c52f4bfe373
Step 9 : CMD python entry.py
 ---> Running in 86b39c727572
 ---> 20ae465bf036
Removing intermediate container 86b39c727572
Successfully built 20ae465bf036

//verity your image
$ sudo docker images
REPOSITORY            TAG
IMAGE ID            CREATED
VIRTUAL SIZE
hidetosaito/my-calc   latest
20ae465bf036        19 seconds ago      284 MB
ubuntu                14.04
6cc0fc2a5ee3        3 weeks ago         187.9
MB
```

4. Then, use the `docker login` command to log in to Docker Hub:

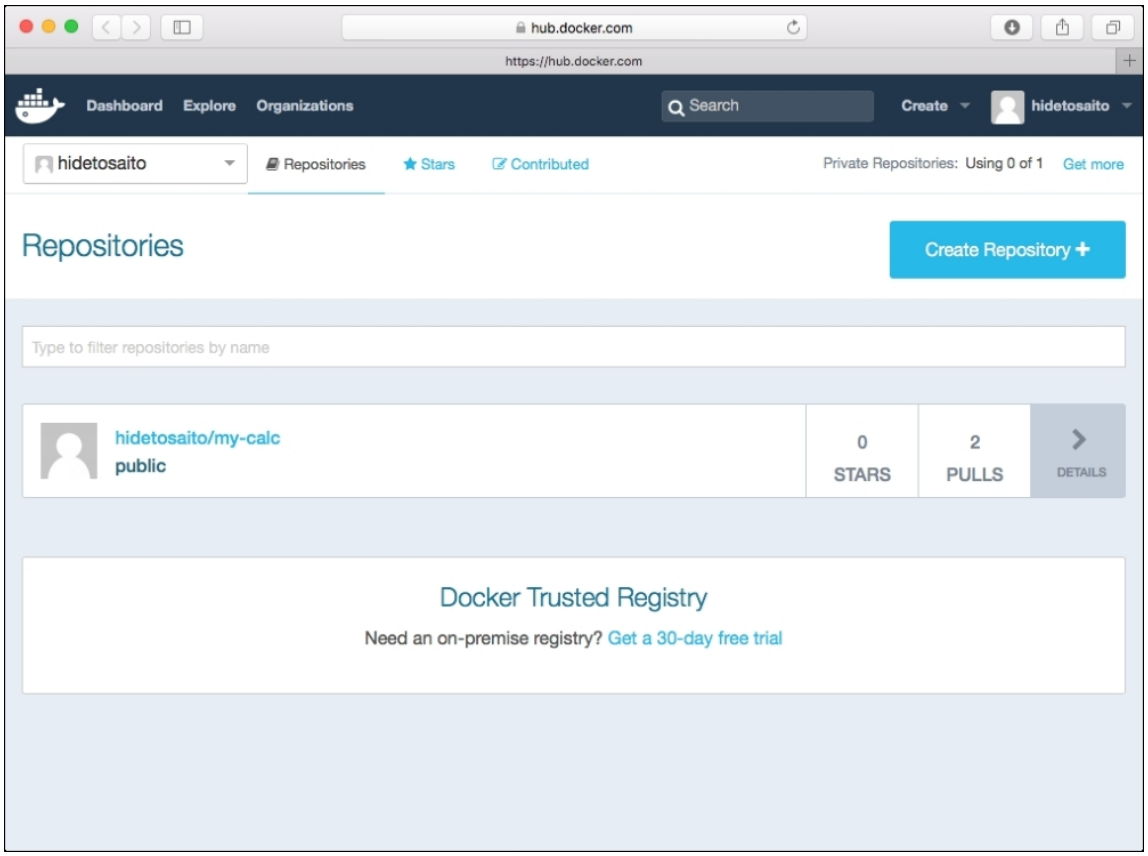
```
//type your username, password and e-mail
address in Docker hub
$ sudo docker login
Username: hidetosaito
Password:
Email: hideto.saito@yahoo.com
WARNING: login credentials saved in /home/ec2-
user/.docker/config.json
Login Succeeded
```

5. Finally, use the `docker push` command to register to your Docker Hub repository as follows:

```
//push to your docker index
$ sudo docker push hidetosaito/my-calc
The push refers to a repository
[docker.io/hidetosaito/my-calc] (len: 1)
20ae465bf036: Pushed
(snip)
92ec6d044cb3: Pushed
latest: digest:
```

```
sha256:203b81c5a238e228c154e0b53a58e60e6eb3d15
63293483ce58f48351031a474 size: 19151
```

On accessing Docker Hub, you can see your microservices in the repository:



Microservice image on Docker Hub

Frontend WebUI

1. Here is the simple Frontend WebUI that is also using Python Flask:

```
import os
import httplib
from flask import Flask, request,
render_template
app = Flask(__name__)
@app.route("/")
def index():
```

```
    return render_template('index.html')

@app.route("/add", methods=['POST'])
def add():
    #
    # from POST parameters
    #
    x = int(request.form['x'])
    y = int(request.form['y'])

    #
    # from Kubernetes Service(environment
variables)
    #
    my_calc_host =
os.environ['MY_CALC_SERVICE_SERVICE_HOST']
    my_calc_port =
os.environ['MY_CALC_SERVICE_SERVICE_PORT']

    #
    # remote procedure call to
MicroServices(my-calc)
    #
    client =
httplib.HTTPConnection(my_calc_host,
my_calc_port)
    client.request("GET", "/addition/%d/%d" %
(x, y))
    response = client.getresponse()
    result = response.read()

    return render_template('index.html',
        add_x=x, add_y=y, add_result=result)

if __name__ == "__main__":
    app.debug = True
    app.run(host='0.0.0.0')
```

Tip

Kubernetes service generates the Kubernetes service name and port number as an environment variable to the other

pods. Therefore, the environment variable's name and the Kubernetes service name must be consistent.

In this scenario, the `my-calc` service name must be `my-calc-service`.

2. Frontend WebUI uses the Flask HTML template; it is similar to PHP and JSP such that `entry.py` will pass the parameter to the template (`index.html`) to render the HTML:

```
<html>
<body>
<div>
    <form method="post" action="/add">
        <input type="text" name="x" size="2"/>
        <input type="text" name="y" size="2"/>
        <input type="submit"
value="addition"/>
    </form>

    {% if add_result %}
    <p>Answer : {{ add_x }} + {{ add_y }} = {{
add_result }}</p>
    {% endif %}
</div>
</body>
</html>
```

3. `Dockerfile` is exactly the same as microservices. So, eventually, the file structure will be as follows. Note that `index.html` is a template file; therefore, put it under the templates directory:

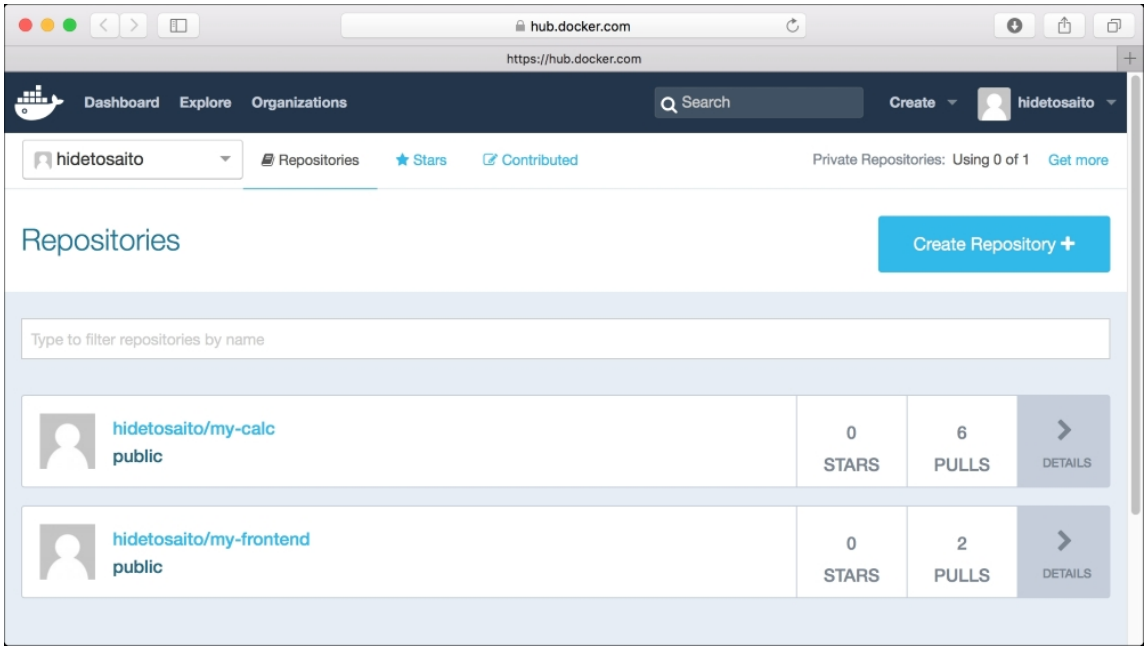
```
/Dockerfile
/entry.py
/templates/index.html
```

4. Then, build a Docker image and push to Docker Hub as follows:

Note

In order to push your image to Docker Hub, you need to log in using the `docker login` command. It is needed only once; the system checks `~/.docker/config.json` to read from there.

```
//build frontend Webui image
$ sudo docker build -t hidetosaito/my-frontend
.

//login to docker hub, if not login yet
$ sudo docker login

//push frontend webui image
$ sudo docker push hidetosaito/my-frontend
```



Microservices and Frontend WebUI image on Docker Hub

How it works…

Launch both microservices and the Frontend WebUI.

Microservices

Microservices (`my-calc`) uses the Kubernetes replication controller and service, but it needs to communicate to other pods only. In other words, there's no need to expose it to the outside Kubernetes network. Therefore, the service type is set as `ClusterIP`:

```
# cat my-calc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-calc-rc
spec:
  replicas: 2
  selector:
        app: my-calc
  template:
    metadata:
      labels:
        app: my-calc
    spec:
      containers:
      - name: my-calc
        image: hidetosaito/my-calc
---
apiVersion: v1
kind: Service
metadata:
  name: my-calc-service

spec:
  ports:
    - protocol: TCP
      port: 5000
  type: ClusterIP
  selector:
     app: my-calc
```

Use the `kubectl` command to load the `my-calc` pods as follows:

```
$ sudo kubectl create -f my-calc.yaml
replicationcontroller "my-calc-rc" created
service "my-calc-service" created
```

Frontend WebUI

Frontend WebUI also uses the replication controller and service, but it exposes the port (TCP port `30080`) in order to access it from an external web browser:

```
$ cat my-frontend.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-frontend-rc
spec:
  replicas: 2
  selector:
        app: my-frontend
  template:
    metadata:
      labels:
        app: my-frontend
    spec:
      containers:
      - name: my-frontend
        image: hidetosaito/my-frontend
---
apiVersion: v1
kind: Service
metadata:
  name: my-frontend-service

spec:
  ports:
    - protocol: TCP
      port: 5000
      nodePort: 30080
  type: NodePort
  selector:
     app: my-frontend

$ sudo kubectl create -f my-frontend.yaml
replicationcontroller "my-frontend-rc" created
service "my-frontend-service" created
```

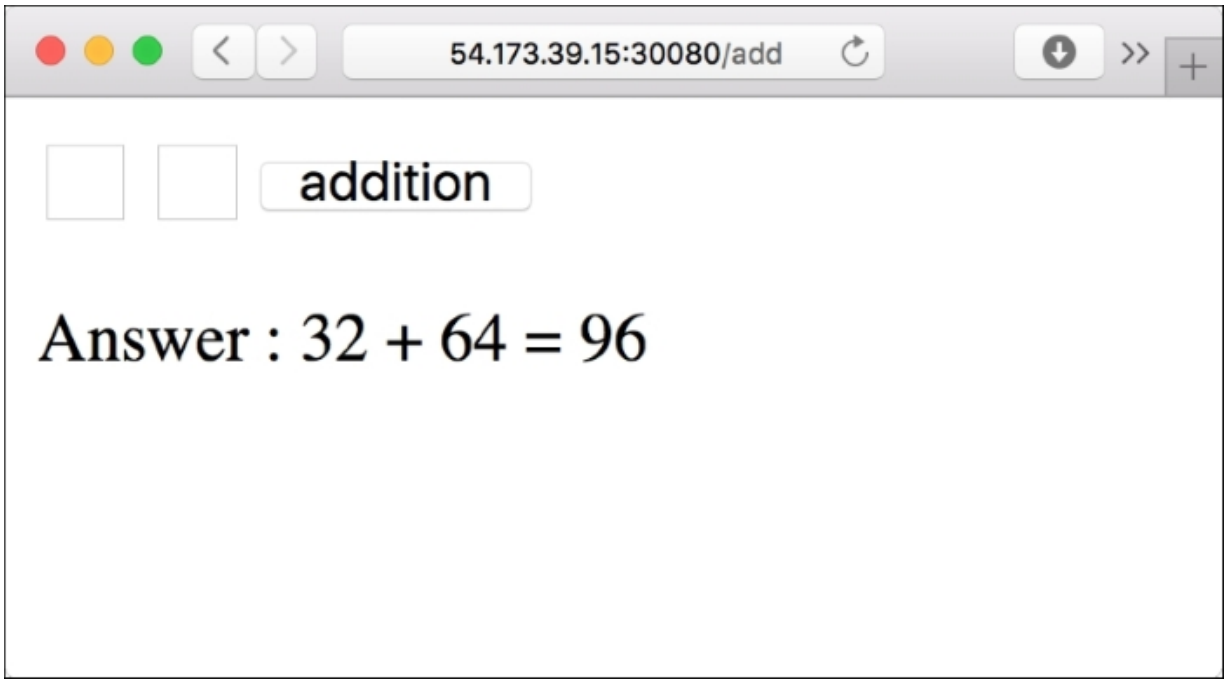
You have exposed your service to an external port on all the nodes in your cluster. If you want to expose this service to the external Internet, you may need to set up firewall rules for the service port(s) (TCP port `30080`) to serve traffic. Refer to http://releases.k8s.io/release-1.1/docs/user-guide/services-firewalls.md for more details.

Let's try to access `my-frontend` using a web browser. You can access any Kubernetes node's IP address; specify the port number `30080` as follows:



Access to the Frontend WebUI

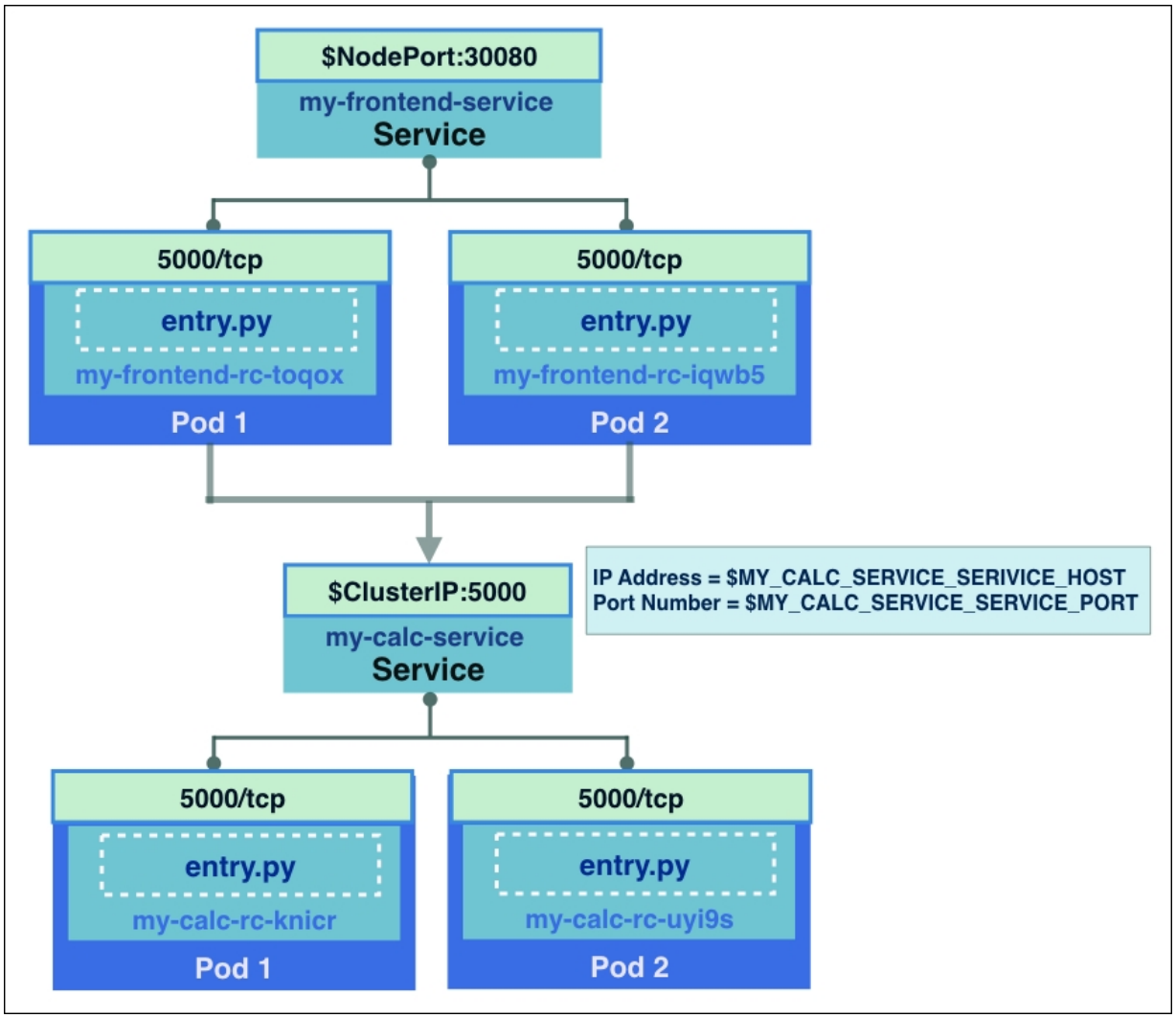
When you click on the **addition** button, it will forward a parameter to microservices (`my-calc`). Microservices compute the addition (yes, just an addition!) and then return the result back to the Frontend WebUI as follows:



Get a result from microservices and render the HTML

So now, it is easy to adjust the number of replicas for the Frontend WebUI and microservices independently. For example, WebUI replicas range from 2 to 8 and microservice replicas range from 2 to 16.

Also, if there's a need to fix some bugs, for example, there's a frontend need to validate the input parameter to check whether it is numeric or string (yes, if you type string and then submit, it will show an error!); it will not affect the build and deploy the cycle against microservices:



The frontend WebUI and microservices diagram

In addition, if you want to add an additional microservice, for example, subtract microservices, you may need to create another Docker image and deploy with another replication controller and service, so it will be independent from the current microservices.

Then, you can keep accumulate your own microservices ecosystem to re-use for another application.

See also

This recipe described how your application aligns to the microservices architecture. The microservices and the Docker

container perfectly fit the concept. The following recipe also helps in managing your microservices' container images:

- *Working with the private Docker registry*

Integrating with Jenkins

In software engineering, **Continuous Integration** (**CI**) (https://en.wikipedia.org/wiki/Continuous_integration) and **Continuous Delivery** (**CD**) (https://en.wikipedia.org/wiki/Continuous_delivery), abbreviated as CI/CD, have the idea to simplify the procedure of the traditional development process with a continuous testing mechanism in order to reduce the panic of serious confliction, namely, to solve small errors immediately once you have found it. Furthermore, through automatic tools, a product running on the CI/CD system can achieve better efficiency for bug fixes or new feature delivery. Jenkins is one of the well-known CI/CD applications. Projects in Jenkins pull codes from the code base server and test or deploy. In this recipe, we will show you how the Kubernetes system joins Jenkins servers as a CI/CD group.

Getting ready

Before you start this recipe, prepare a Docker Hub account (https://hub.docker.com) if you don't have your Docker registry; we will put the images built by Jenkins here. It is also where Kubernetes pulls images from. Next, make sure both the Kubernetes servers and your Jenkins servers are ready. You can set up your own standalone Jenkins server through the Docker image as well; the details are indicated here.

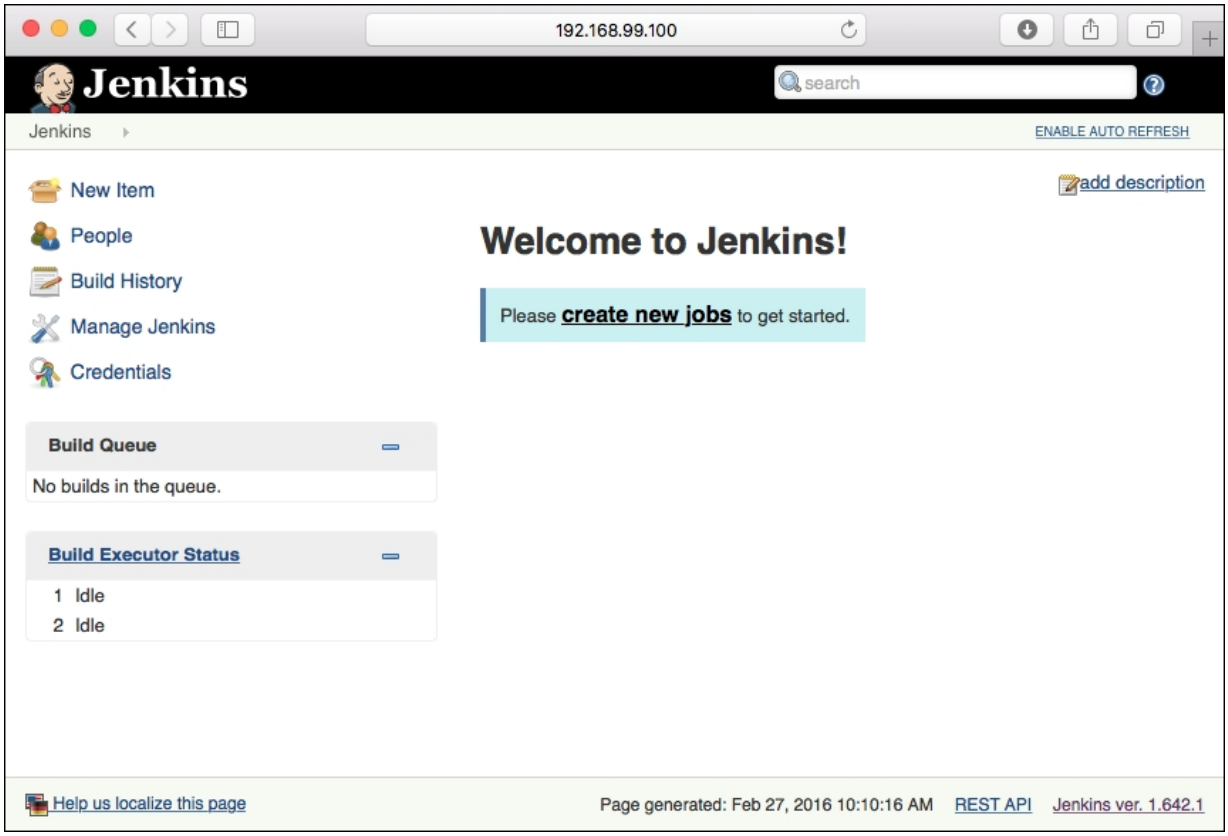
Install a Jenkins server which can build a Docker program

First, you need a Docker-installed machine. Then, you can go ahead to create a Jenkins server using this command:

```
// Pull the Jenkins image from Docker Hub
```

```
$ docker run -p 8080:8080 -v
/your/jenkins_home/:/var/jenkins_home -v $(which
docker):/bin/docker jenkins
```

Port `8080` is the portal of a website. It is recommended to assign a host directory for mounting the Jenkins home directory. Therefore, you can keep the data even with the container shutdown. We will also mount the Docker binary file to our Jenkins container, since this Jenkins official image didn't install the `docker` command. After you push this command, logs for installation will show on your terminal. Once you see this information, **INFO: Jenkins is fully up and running**, it is good for you to check the web console of Jenkins using `DOCKER_MACHINE_IP_ADDRESS:8080`:



After this, you have to install the Git and Docker plugins for Jenkins. Installing plugins is the way to customize your Jenkins server. Here, these two plugins can make you fulfill the workflow from the code in your laptop to the containers on the Kubernetes system. You can refer to the following steps:

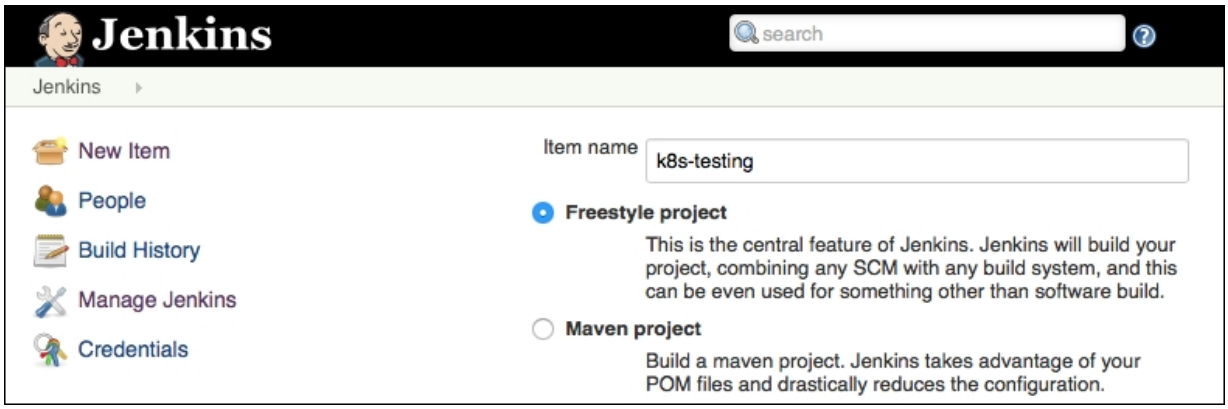
1. Click on **Manage Jenkins** on the left menu.
2. Pick **Manage Plugins**.
3. Choose the tag **Available**.
4. Key in **Git Plugin** and check it; do the same for **CloudBees Docker Build and Publish Plugin**.
5. Start the installation and reboot.

How to do it…

We are going to make Kubernetes the working station for running testing containers or deploying containers officially. Currently, there are no Jenkins plugins for deploying pods on the Kubernetes systems. Therefore, we are going to call the Kubernetes API for our usage. In order to run a program test, we can use the Kubernetes job mentioned in *Ensuring flexible usage of your containers* recipe in Chapter 3, *Playing with Containers*, which will create the job-like pod that could be terminated once finished. It is suitable for testing and verification. On the other hand, we can directly create the replication controller and service for official deployment.

Create your Jenkins project

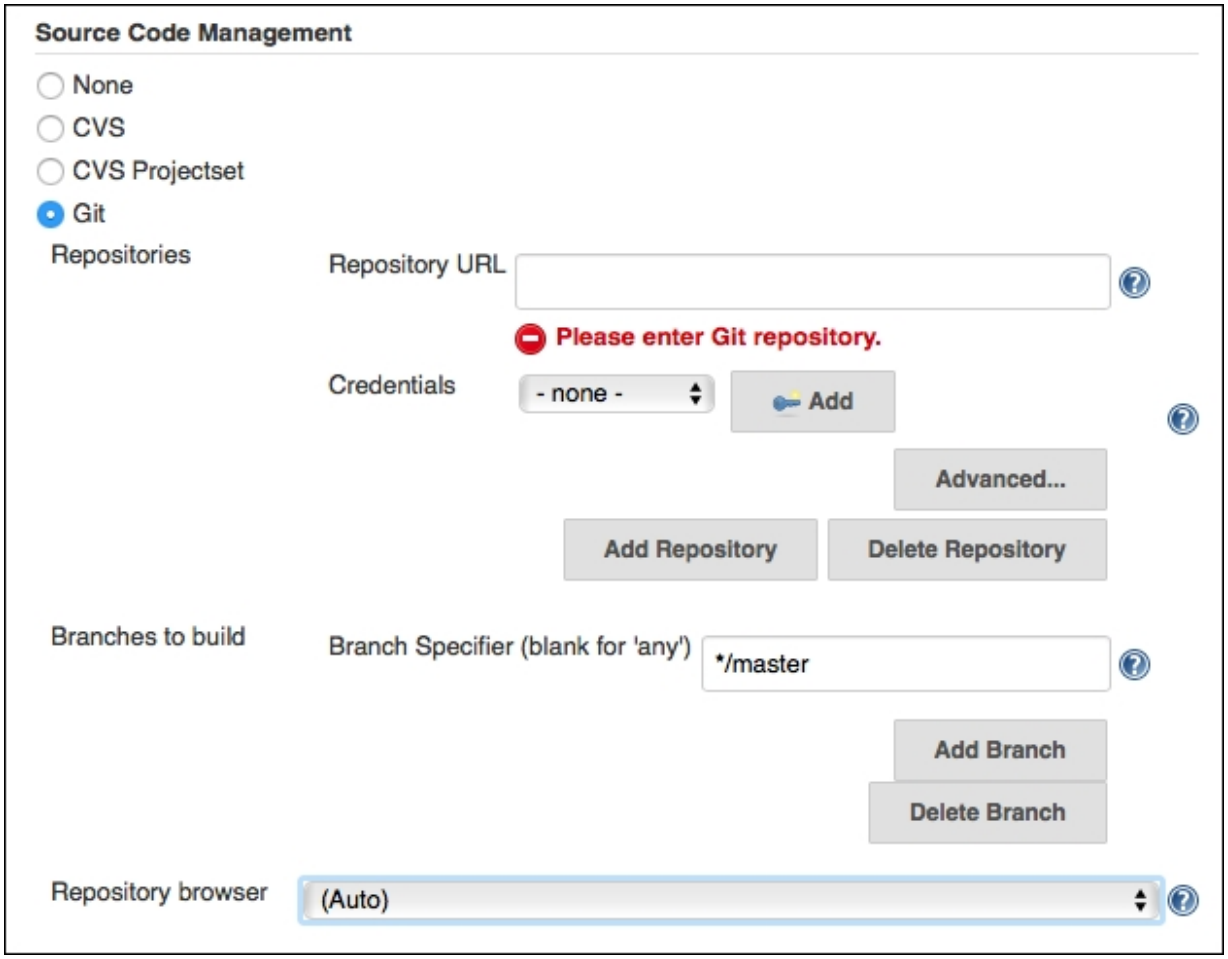
As a program build, we will create a single Jenkins project for each program. Now, you can click on **New Item** on the menu on the left-hand side. In this recipe, choose **Freestyle project**; it is good enough for the building examples later on. Name the project and click on **OK**:



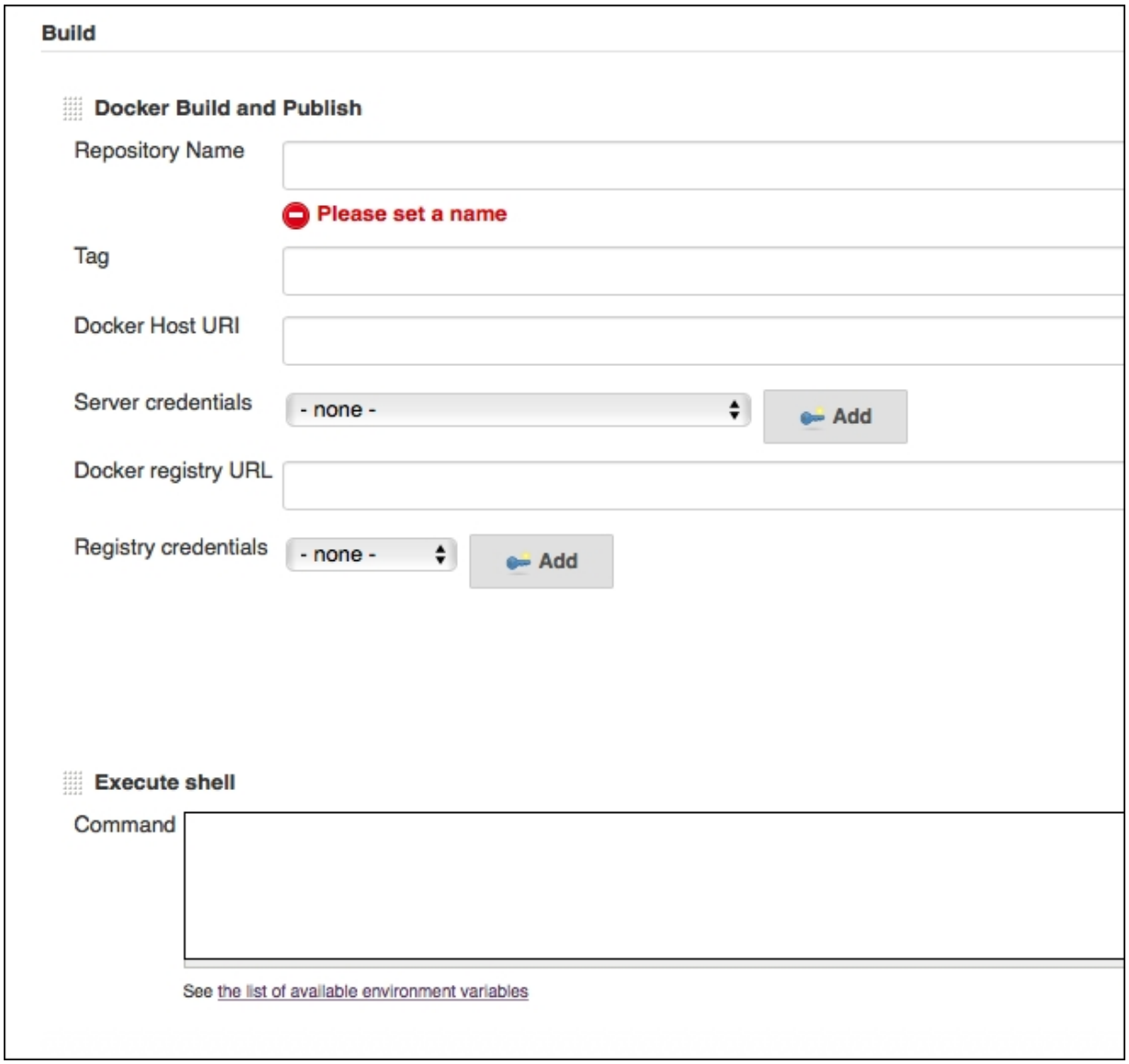
Then, you will see the following blocks of categories on the configuration page:

- **Advanced Project Options**
- **Source Code Management**
- **Build Triggers**
- **Build**
- **Post-build Actions**

These settings can make your project more flexible and more close to the one you need. However, we will only focus on the **Source Code Management** and **Build** parts to meet our requirements. **Source Code Management** is where we will define our codebase location. In the later sections, we will choose **Git** and put the corresponding repository:



In the category **Build**, several steps, such as **Docker Build and Publish** and **Execute shell**, will help us build Docker images, push images to the Docker registry, and trigger the Kubernetes system to run the programs:



The following two scenarios will demonstrate how to set the configurations for running your programs on the Kubernetes systems.

Run a program testing

Kubernetes' job is to handle some programs that have terminal conditions, such as unit testing. Here, we have a short dockerized program, which is suitable for this situation. Feel free to check it on GitHub: https://github.com/kubernetes-cookbook/sleeper. As indicated in `Dockerfile`, this tiny program will be killed after 10 seconds of running and doing nothing. The following steps will guide you set up the project's configurations:

1. At **Source Code Management**, choose **Git**, put the **HTTPS URL** of your GitHub repository at **Repository URL**. For example, https://github.com/kubernetes-cookbook/sleeper.git. And you will find that the **WARNING** message has disappeared for a public repository; or you will have to add a credential.
2. At **Build**, we need the following two build steps:

    1. Add **Docker Build and Publish** first to build your program as an image. A repository name is a required item. In this case, we will use Docker Hub as a registry; please name your repository as `DOCKERHUB_ACCOUNT:YOUR_CUSTOMIZED_NAME`. For example, `nosus:sleeper`. Adding a tag, such as `v$BUILD_NUMBER`, could tag your Docker image with the Jenkins build number. Leave **Docker Host URI** and **Server credentials** empty if your Jenkins servers already installed the Docker package. But if you follow the instructions on the previous page to install the Jenkins server as a container, please check the following tips for detailed settings of these two items. Leave **Docker registry URL** empty, since we used Docker Hub as a Docker registry. However, set a new credential for your Docker Hub accessing permission.
    2. Next, add an **Execute shell** block for calling the Kubernetes API. We put two API calls for our purpose: one is to create a Kubernetes job using the JSON format template and the other is to query whether the job completes successfully or not:

```
#run a k8s job
```

```
curl -XPOST -
d'{"apiVersion":"extensions/v1beta1","kin
d": "Job","metadata":{"name":"sleeper"},
"spec": {"selector": {"matchLabels":
{"image": "ubuntu","test":
"jenkins"}},"template": {"metadata":
{"labels": {"image": "ubuntu","test":
"jenkins"}},"spec": {"containers":
[{"name": "sleeper","image":
"nosus/sleeper"}],"restartPolicy":
"Never"}}}}'
http://YOUR_KUBERNETES_MASTER_ENDPOINT/ap
is/extensions/v1beta1/namespaces/default/
jobs
#check status
count=1
returnValue=-1
while [ $count -lt 60 ]; do
  curl -XGET
http://YOUR_KUBERNETES_MASTER_ENDPOINT/ap
is/extensions/v1beta1/namespaces/default/
jobs/sleeper | grep "\"succeeded\": 1" &&
returnValue=0 && break
  sleep 3
count=$(($count+1))
done

return $returnValue
```

We will also add a `while` loop for a 3-minute timeout limitation. Periodically checking the status of the job is a simple way to know whether it is completed. If you fail to get the message **succeeded: 1**, judge the job as a failure.

Tip

Set Docker Host URI and Server Credential for a containerized Jenkins server

If you use `docker-machine` to build your Docker environment (for example, an OS X user), type this command in your terminal:

```
$ docker-machine env
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export
DOCKER_CERT_PATH="/Users/YOUR_ACCOUNT/.docker/
machine/machines/default"
export DOCKER_MACHINE_NAME="default"
```

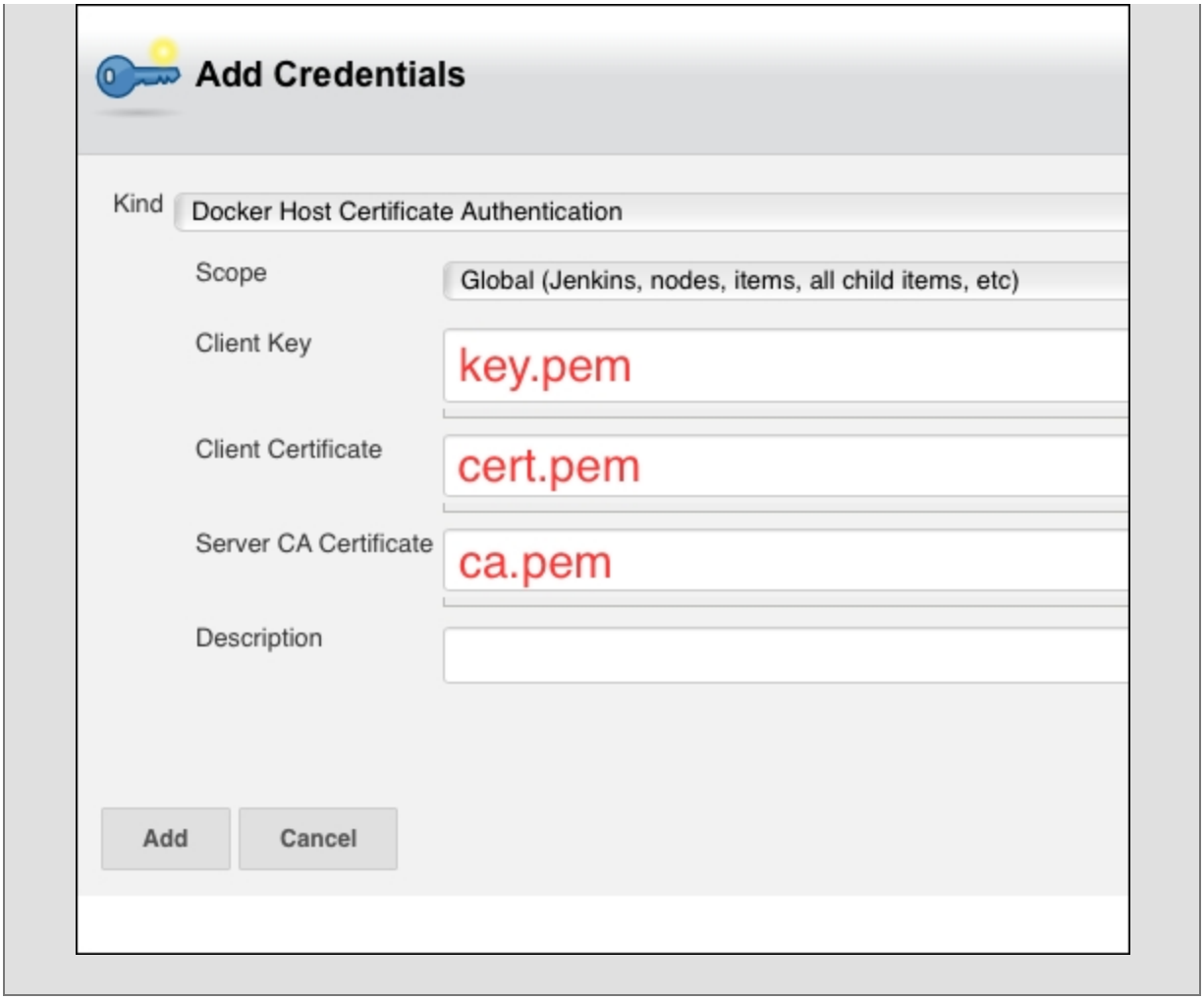
Run this command to configure your shell:

```
# eval $(docker-machine env)
```

By default, you will get the preceding information. Please paste the value of `$DOCKER_HOST` in the item **Docker Host URI**. Then, check your directory at `$DOCKER_CERT_PATH`:

```
$ ls
/Users/carol/.docker/machine/machines/default
boot2docker.iso cert.pem        default
id_rsa          key.pem         server.pem
ca.pem          config.json     disk.vmdk
id_rsa.pub      server-key.pem
```

Certain files listed here are capable for you to get permission. Please click on the **Add** button beside **Registry credentials** and paste the content of a part of the previous files, as shown in the following screenshot:



After you finish project configurations, you can click on **Save** and then click on **Build Now** to check the result. You may find your image is pushed to Docker Hub as well:



Deploying a program

Deploying a dockerized program through the Jenkins server has similar configurations in the project's settings. We will prepare a simple nginx program for you to try on! Open a new Jenkins project for new settings as follows:

1. At **Source Code Management**, choose **Git**; put the value https://github.com/kubernetes-cookbook/nginx-demo.git in **Repository URL**.
2. At **Build**, we will need two build steps as well. They are as follows:

    1. Add a **Docker Build and Publish** first; we are going to put `nosus:nginx-demo` as the repository name. Specify appropriate values at **Docker Host URI** and **Server credentials**.
    2. We need an **Execute shell** block for calling the Kubernetes API. There are two API calls put in the space: first one is for creating a pod and the other one is for creating a service to expose the pod:

```
#create a pod
curl -XPOST -d'{"apiVersion":
"v1","kind": "Pod","metadata": {"labels":
{"app": "nginx"},"name": "nginx-
demo"},"spec": {"containers": [{"image":
"nosus/nginx-demo","imagePullPolicy":
"Always","name": "nginx-demo","ports":
[{"containerPort": 80,"name":
"http"}]}],"restartPolicy": "Always"}}'
http://
YOUR_KUBERNETES_MASTER_ENDPOINT/api/v1/na
mespaces/default/pods
#create a service
curl -XPOST -d'{"apiVersion":
"v1","kind": "Service","metadata":
{"name": "nginx-demo"},"spec": {"ports":
[{"port": 8081,"protocol":
"TCP","targetPort": 80}],"selector":
{"app": "nginx"},"type": "NodePort"}}'
http://YOUR_KUBERNETES_MASTER_ENDPOINT
/api/v1/namespaces/default/services
```

Feel free to check the endpoint of the Kubernetes service you just created using Jenkins! It is even better if you add a Git server webhook with Jenkins. You can directly get the last result after each code is pushed!

How it works…

The Jenkins server communicates with the Kubernetes system through its RESTful API. You can also take a look at more functionalities via the URL: `http://YOUR_KUBERNETES_MASTER_ENDPOINT:KUBE_API_PORT/swagger_ui`. Hey! The API list is just kept inside your server!

On the other hand, it is possible for you to install the plugin called **HTTP Request Plugin** (https://wiki.jenkins-ci.org/display/JENKINS/HTTP+Request+Plugin) to fulfill the Kubernetes API calls. We will not explain this one, since the `POST` function of the current version failed to use the JSON format payload. You can still try the other types of API calls with this plugin.

Currently, there are no plugins that are able to help deploy the Kubernetes tasks. That is why the building procedure is such a headache for the long `curl` commands. It is also an inspiration that you can combine other systems or services with your Kubernetes system through the RESTful API.

There's more…

You may find that in the section *Deploying a program*, if we build the same project again, the Kubernetes API call will return an error response, replying that the pod and service already exist. In this situation, Kubernetes does not have any rolling update API for the live updating of our program.

Still, on the aspect of infrastructure, there are two ideas for Kubernetes integration:

- A Jenkins plugin called **Kubernetes Plugin** (https://wiki.jenkins-ci.org/display/JENKINS/Kubernetes+Plugin) helps to build Jenkins slaves dynamically.

- You can try to make your Kubernetes master as a Jenkins slave! As a result, it is wonderful to create pods without complicated API calls. There is no problem to use a rolling update!

In the version 1.2 of Kubernetes, there is a new resource type deployment that controls pods and replica sets, which should be the new solution for managing your pods. Deployment has the following features:

- Desired state and available state to indicate whether the pods are ready to use
- Rolling update for the pods for any modification
- It is capable to roll back to the previous revision of deployment

Resource deployment is in the API version extensions `/v1beta1`. Kubernetes supports its API call for both update and rollback. Please take the following API calls as a reference:

```
// Create a deployment
curl -XPOST  -d "{\"metadata\":{\"name\":\"nginx-
123\"},\"spec\":{\"replicas\":2,\"template\":
{\"metadata\":{\"labels\":
{\"app\":\nginx\"}},\"spec\":{\"containers\":
[{\"name\":\"nginx-
deployment\",\"image\":\"nosus/nginx-
demo:v$BUILD_NUMBER\",\"ports\":
[{\"containerPort\": 80}]}]}}}}"
YOUR_KUBERNETES_MASTER_ENDPOINT
/v1beta1/namespaces/default/deployments
```

A patch call of API is used for update deployments. Here, we will change the image version. Some details about patch operation can be found at https://github.com/kubernetes/kubernetes/blob/master/docs/devel/api-conventions.md:

```
// Update a deployment
curl -H "Content-Type: application/strategic-
```

```
merge-patch+json" -XPATCH -d '{"spec":{"template":
{"spec":{"containers":[{"name":"nginx-
deployment","image":"nosus/nginx-demo:v1"}]}}}}'
YOUR_KUBERNETES_MASTER_ENDPOINT
/apis/extensions/v1beta1/namespaces/default/deploy
ments/nginx-deployment
// Rollback the deployment
curl -H "Content-Type: application/json" -XPOST -d
'{"name":"nginx-deployment","rollbackTo":
{"revision":0}}' YOUR_KUBERNETES_MASTER_ENDPOINT
/apis/extensions/v1beta1/namespaces/default/deploy
ments/nginx-deployment/rollback
```

In the last API call, we rollback the deployment to the original version, indicated as version `0`. Just try to manage the new resource type deployment by yourself!

See also

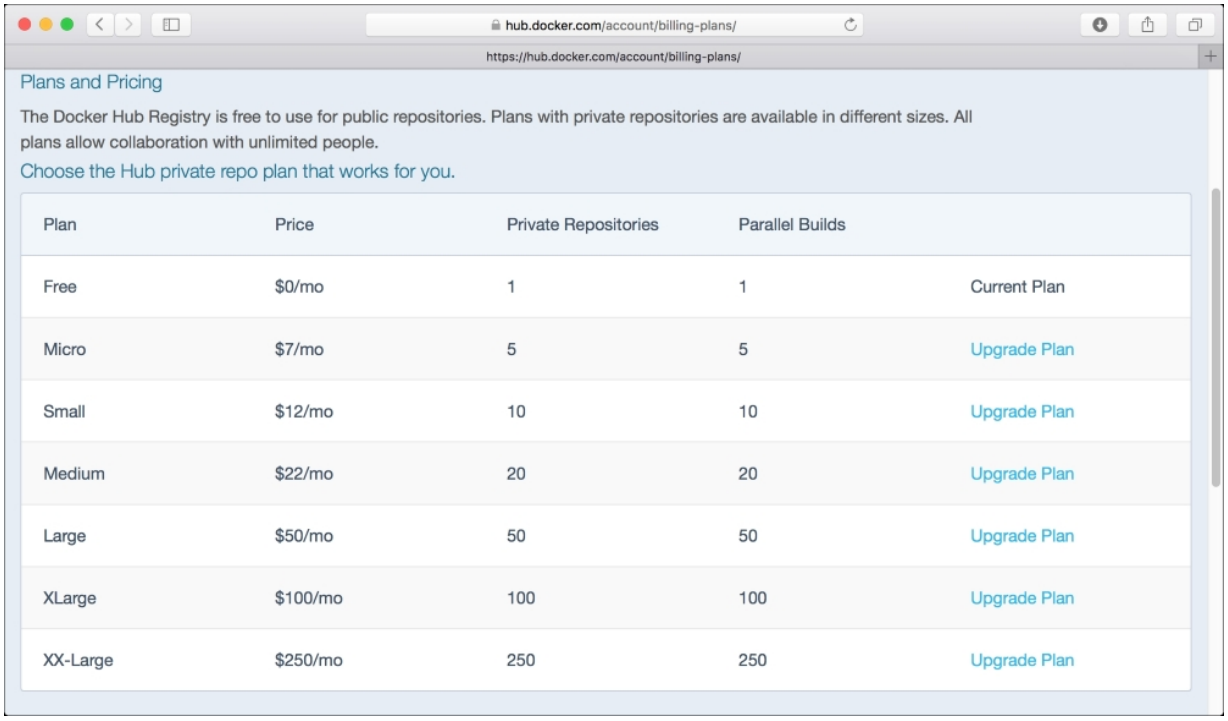
Please continue to read the next recipe for more CI/CD concepts. It will definitely help you while pushing your programs on the Kubernetes system. Also refer to the following recipes:

- *Setting up the Continuous Delivery pipeline*
- The *Working with a RESTful API* and *Authentication and authorization* recipes in Chapter 7, *Advanced Cluster Administration*

Working with the private Docker registry

Once you start maintaining your own Docker image, you might need to have some private Docker registry to put some sensitive information into an image or your organization policy.

Docker Hub offers the private repository, which only the authenticated user can push and pull images, and is not visible to other users. However, there is only one quota for a free Docker Hub account. You may pay to increase the private repositories quota, but if you adopt the microservices architecture, you will need a large number of private repositories:



Docker Hub private repositories price

There are some ways to set up your own private Docker registry that unlimited Docker image quota locates inside of your network.

Getting ready

The easiest way to set up your Docker registry is use an official Docker registry image (https://docs.docker.com/registry/). Run the `docker pull` command to download the Docker registry image as follows:

```
$ docker pull registry:2
2: Pulling from library/registry
f32095d4ba8a: Pull complete
9b607719a62a: Pull complete
973de4038269: Pull complete
2867140211c1: Pull complete
8da16446f5ca: Pull complete
fd8c38b8b68d: Pull complete
136640b01f02: Pull complete
e039ba1c0008: Pull complete
c457c689c328: Pull complete
Digest:
sha256:339d702cf9a4b0aa665269cc36255ee7ce424412d56
bee9ad8a247afe8c49ef1
Status: Downloaded newer image for registry:2

//create Docker image datastore under
/mnt/docker/images
$ sudo mkdir /mnt/docker/images

//launch registry that expose the 5000/tcp to
8888/tcp on host
$ sudo docker run -p 8888:5000 -v
/mnt/docker/images:/var/lib/registry registry:2
```

Note

It will store the images to `/mnt/docker/images` on the host machine. It is highly recommended to consider to mount a network data volume such as NFS or use Docker volume

How to do it…

Let's create your simple Docker image based on `nginx`:

1. Firstly, prepare `index.html` as follows:

```
$ cat index.html
<html>
    <head><title>My Image</title></head>

    <body>
        <h1>Hello Docker !</h1>
    </body>

</html>
```

2. Also, prepare `Dockerfile` as follows to build your Docker image:

```
$ cat Dockerfile
FROM nginx
COPY index.html /usr/share/nginx/html
```

3. Then, build a Docker image name as `<your name>/mynginx` as follows:

```
$ ls
Dockerfile  index.html

$ docker build -t hidetosaito/mynginx .
Sending build context to Docker daemon 3.072
kB
Step 1 : FROM nginx
 ---> 9737f81306ee
Step 2 : COPY index.html /usr/share/nginx/html
 ---> 74dd7902a931
Removing intermediate container eefc2bb17e24
Successfully built 74dd7902a931
```

   At this moment, the `mynginx` image is stored on this host only.

4. Now, it's time to push to your own private registry. First of all, it needs to tag the image name as `<private_registry:port number>/<your name>/mynginx` as follows:

```
$ docker tag hidetosaito/mynginx ip-10-96-219-
25:8888/hidetosaito/mynginx
$ docker images
REPOSITORY                                    TAG
IMAGE ID            CREATED
VIRTUAL SIZE
ip-10-96-219-25:8888/hidetosaito/mynginx
latest              b69b2ab1f31b        7
minutes ago         134.6 MB
hidetosaito/mynginx
latest              b69b2ab1f31b        7
minutes ago         134.6 MB
```

> ## Note
>
> You may see that `IMAGE ID` are the same, because they are the same image.

5. Then, push to the private registry using the `docker push` command as follows:

```
$ docker push ip-10-96-219-
25:8888/hidetosaito/mynginx
The push refers to a repository [ip-10-96-219-
25:8888/hidetosaito/mynginx] (len: 1)
b69b2ab1f31b: Pushed
ae8e1e9c54b3: Pushed
18de280c0e54: Pushed
cd0794b5fd94: Pushed
f32095d4ba8a: Pushed
latest: digest:
sha256:7ac04fdaedad1cbcc8c92fb2ff099a6509f4f29
b0f694ae044a0cffc8ffe58b4 size: 15087
```

Now, your `mynginx` image has been stored in your private registry. Let's deploy this image using Kubernetes.

6. Prepare the YAML file, which contains command to use `nginx` from the private registry and use the Kubernetes service to expose to TCP port `30080`:

```
# cat my-nginx-with-service.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: mynginx
spec:
  replicas: 2
  selector:
        app: mynginx
  template:
    metadata:
      labels:
        app: mynginx
    spec:
      containers:
      - name: mynginx
        image: ip-10-96-219-
25:8888/hidetosaito/mynginx
---
apiVersion: v1
kind: Service
metadata:
  name: mynginx-service

spec:
  ports:
    - protocol: TCP
      port: 80
      nodePort: 30080
  type: NodePort
  selector:
     app: mynginx
```

7. Then, use the `kubectl create` command to load this YAML file:

```
# kubectl create -f my-nginx-with-service.yaml
replicationcontroller "mynginx" created
```

You have exposed your service to an external port on all the nodes in your cluster. If you want to expose this service to the external Internet, you may need to set up firewall rules for the service port(s) (TCP port `30080`) to serve traffic. Refer to http://releases.k8s.io/release-1.1/docs/user-guide/services-firewalls.md for more details:

```
service "mynginx-service" created
```

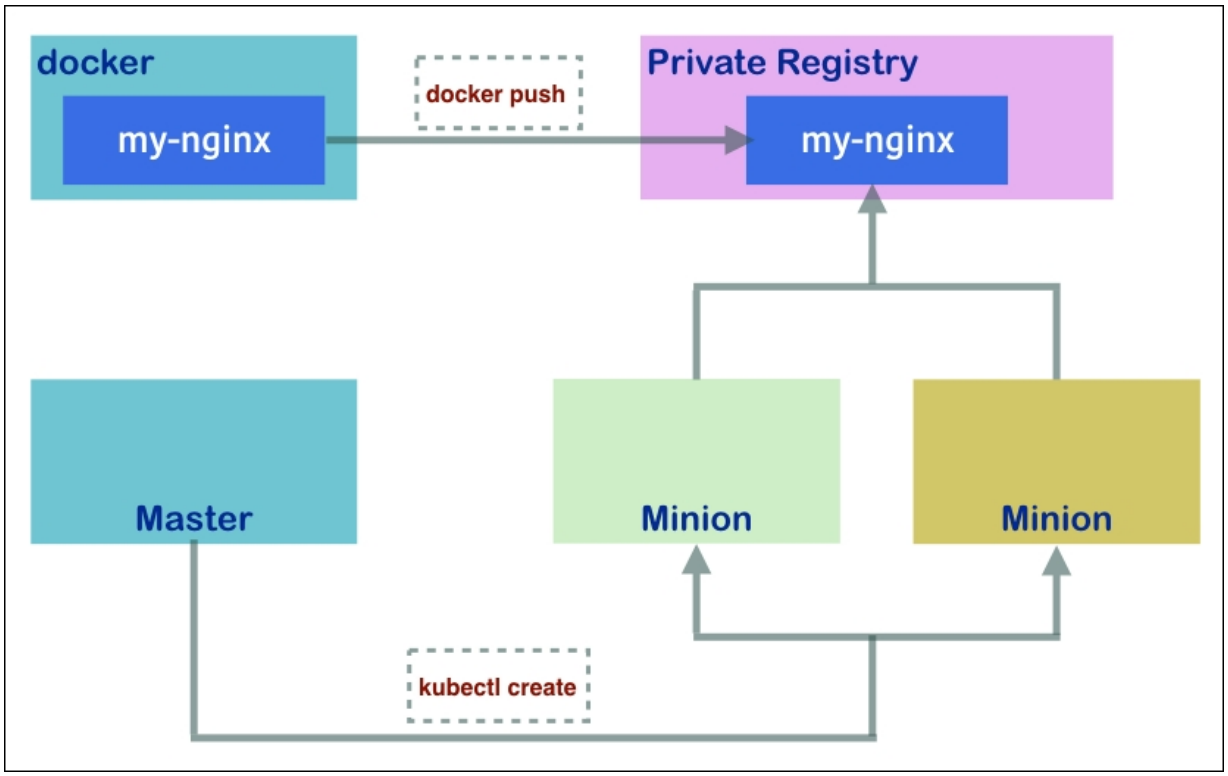
Then, access any Kubernetes node on TCP port `30080`; you may see `index.html` as follows:



How it works…

On running the `docker push` command, it uploads your Docker image to the private registry. Then, when the `kubectl create` command is run, the Kubernetes node performs `docker pull` from the private registry.

Using the private registry is the easiest way to propagate your Docker image to all the Kubernetes nodes:



Alternatives

The official Docker registry image is the standard way to set up your private registry. However, from the management and maintenance points of view, you might need to make more effort. There is an alternative way to construct your own Docker private registry.

Docker Trusted Registry

Docker Trusted Registry is an enterprise version of Docker registry. It comes with Web Console, LDAP integration, and so on. To know more about Docker Trusted Registry, please refer to the following link:

https://www.docker.com/products/docker-trusted-registry

Nexus Repository Manager

Nexus Repository Manager is one of the popular repository managers; it supports Java Maven, Linux apt/yum, Docker, and

more. You can integrate all the software repository into the Nexus Repository Manager. Read more about Nexus Repository here:

http://www.sonatype.com/nexus/solution-overview/nexus-repository

Amazon EC2 Container Registry

Amazon Web Services (**AWS**) also provides a managed Docker registry service. It is integrated with **Identity Access Management** (**IAM**) and the charges are based on storage usage and data transfer usage, instead of the number of images. To know more about it, please refer to following link:

`https://aws.amazon.com/ecr/`

See also

This recipe described how to set up your own Docker registry. The private registry brings you more flexibility and security for your own images. The following recipes help to understand the need for private registry:

- *Moving monolithic to microservices*
- The *Working with volumes* recipe in Chapter 2, *Walking through Kubernetes Concepts*

Setting up the Continuous Delivery pipeline

Continuous Delivery is a concept that was first introduced in the book: *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* written by *Jez Humble* and *David Farley*. By automation of the test, build, and deployment, the pace of software release can be the time to market. It also helps in the collaboration between developers, operations, and testers, reducing communication effort and bugs. CD pipeline aims to be a reliable and repeatable process and tools of delivering software.

Kubernetes is one of the destinations in CD pipeline. This section will describe how to deliver your new release software into Kubernetes by Jenkins and Kubernetes deployment.

Getting ready

Knowing Jenkins is prerequisite to this section. For more details on how to build and setup Jenkins from scratch, please refer to *Integrating with Jenkins* section in this chapter. We will use the sample *Flask* (http://flask.pocoo.org) app `my-calc` mentioned in *Moving monolithic to microservices* section. Before setting up our Continuous Delivery pipeline with Kubernetes, we should know what Kubernetes deployment is. Deployment in Kubernetes could create a certain number of pods and replication controller replicas. When a new software is released, you could then roll updates or recreate the pods that are listed in the deployment configuration file, which can ensure your service is always alive.

Just like Jobs, deployment is a part of the extensions API group and still in the `v1beta` version. To enable a deployment resource, set the following command in the API server configuration when launching. If you have already launched the server, just modify the

`/etc/kubernetes/apiserver` configuration file and restart the `kube-apiserver` service. Please note that, for now, it still supports the `v1beta1` version:

```
--runtime-
config=extensions/v1beta1/deployments=true
```

After the API service starts successfully, we could start building up the service and create the sample `my-calc` app. These steps are required, since the concept of Continuous Delivery is to deliver your software from the source code, build, test and into your desired environment. We have to create the environment first.

Once we have the initial `docker push` command in the Docker registry, let's start creating a deployment named `my-calc-deployment`:

```
# cat my-calc-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-calc-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: my-calc
    spec:
      containers:
      - name: my-calc
        image: msfuko/my-calc:1
        ports:
        - containerPort: 5000

// create deployment resource
# kubectl create -f deployment.yaml
deployment "my-calc-deployment" created
```

Also, create a service to expose the port to the outside world:

```
# cat deployment-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-calc
spec:
  ports:
    - protocol: TCP
      port: 5000
  type: NodePort
  selector:
     app: my-calc
// create service resource
# kubectl create -f deployment-service.yaml
You have exposed your service on an external port
on all nodes in your cluster. If you want to
expose this service to the external internet, you
may need to set up firewall rules for the service
port(s) (tcp:31725) to serve traffic.
service "my-calc" created
```

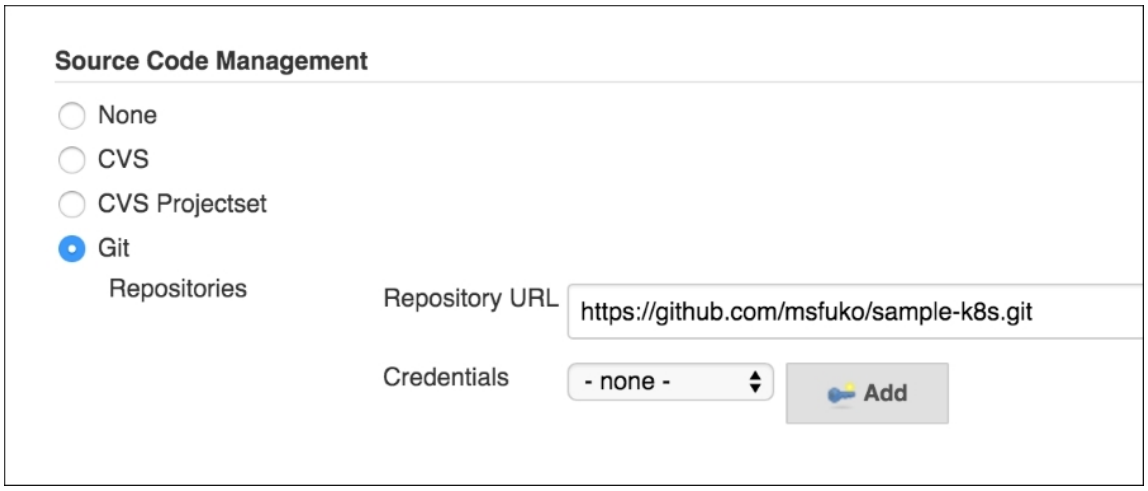
How to do it…

To set up the Continuous Delivery pipeline, perform the following steps:

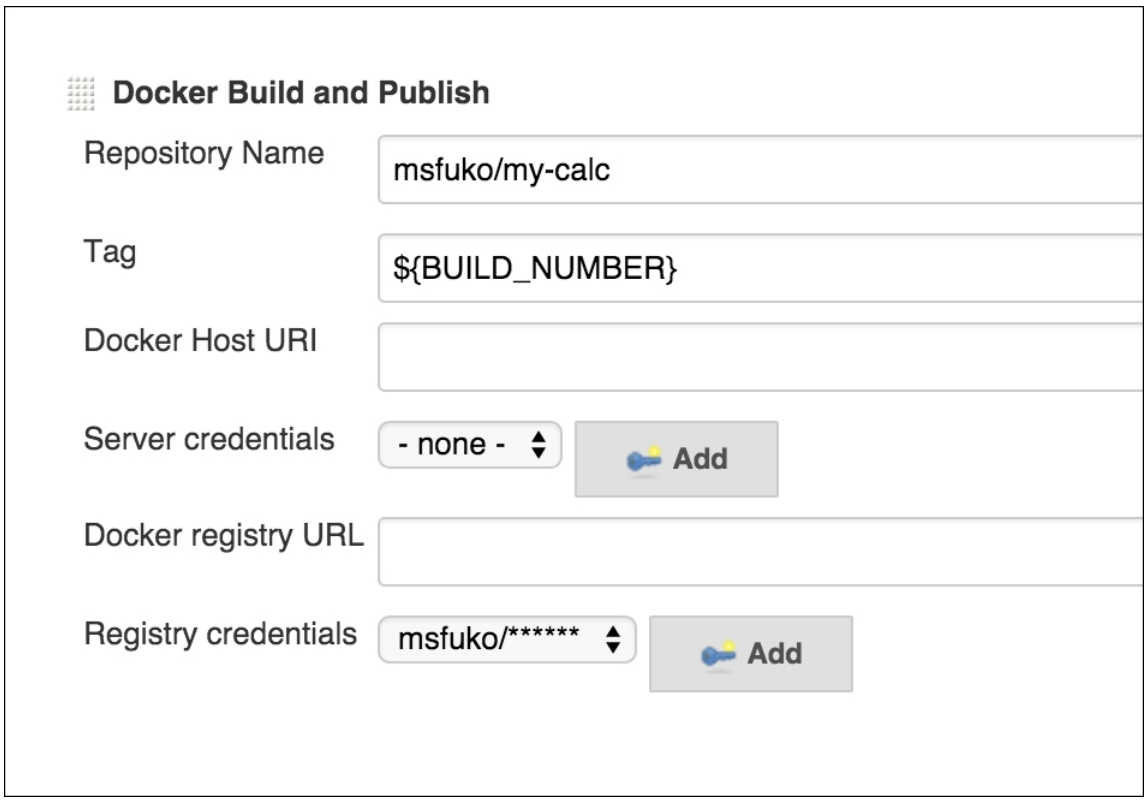
1. At first, we'll start a Jenkins project named `Deploy-My-Calc-K8S` as shown in the following screenshot:



2. Then, import the source code information in the **Source Code Management** section:



3. Next, add the targeted Docker registry information into the **Docker Build and Publish** plugin in the **Build** step:



4. At the end, add the **Execute Shell** section in the **Build** step and set the following command:

```
curl -XPUT -
d'{"apiVersion":"extensions/v1beta1","kind":"D
```

```
eployment","metadata":{"name":"my-calc-
deployment"},"spec":{"replicas":3,"template":
{"metadata":{"labels":{"app":"my-
calc"}},"spec":{"containers":[{"name":"my-
calc","image":"msfuko/my-
calc:${BUILD_NUMBER}","ports":
[{"containerPort":5000}]}]}}}}'
http://54.153.44.46:8080/apis/extensions/v1bet
a1/namespaces/default/deployments/my-calc-
deployment
```

Let's explain the command here; it's actually the same command with the following configuration file, just using a different format and launching method. One is by using the RESTful API, another one is by using the `kubectl` command.

5. The `${BUILD_NUMBER} tag` is an environment variable in Jenkins, which will export as the current build number of the project:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-calc-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: my-calc
    spec:
      containers:
      - name: my-calc
        image: msfuko/my-calc:${BUILD_NUMBER}
        ports:
         containerPort: 5000
```

6. After saving the project and we could start our build. Click on **Build Now**. Then, Jenkins will pull the source code from your Git repository, building and pushing the image. At the end, call the RESTful API of Kubernetes:

```
# showing the log in Jenkins about calling API
of Kubernetes
...
```

```
[workspace] $ /bin/sh -xe
/tmp/hudson3881041045219400676.sh
+ curl -XPUT -
d'{"apiVersion":"extensions/v1beta1","kind":"D
eployment","metadata":{"name":"my-cal-
deployment"},"spec":{"replicas":3,"template":
{"metadata":{"labels":{"app":"my-
cal"}},"spec":{"containers":[{"name":"my-
cal","image":"msfuko/my-cal:1","ports":
[{"containerPort":5000}]}]}}}'
http://54.153.44.46:8080/apis/extensions/v1bet
a1/namespaces/default/deployments/my-cal-
deployment
  % Total    % Received % Xferd  Average Speed
Time    Time     Time  Current
                                 Dload  Upload
Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0
--:--:-- --:--:-- --:--:--     0
100  1670  100  1407  100   263   107k  20534
--:--:-- --:--:-- --:--:--  114k
{
  "kind": "Deployment",
  "apiVersion": "extensions/v1beta1",
  "metadata": {
    "name": "my-calc-deployment",
    "namespace": "default",
    "selfLink":
"/apis/extensions/v1beta1/namespaces/default/d
eployments/my-calc-deployment",
    "uid": "db49f34e-e41c-11e5-aaa9-
061300daf0d1",
    "resourceVersion": "35320",
    "creationTimestamp": "2016-03-
07T04:27:09Z",
    "labels": {
      "app": "my-calc"
    }
  },
  "spec": {
    "replicas": 3,
    "selector": {
      "app": "my-calc"
    },
    "template": {
      "metadata": {
        "creationTimestamp": null,
```

```
            "labels": {
              "app": "my-calc"
            }
          },
          "spec": {
            "containers": [
              {
                "name": "my-calc",
                "image": "msfuko/my-calc:1",
                "ports": [
                  {
                    "containerPort": 5000,
                    "protocol": "TCP"
                  }
                ],
                "resources": {},
                "terminationMessagePath":
"/dev/termination-log",
                "imagePullPolicy": "IfNotPresent"
              }
            ],
            "restartPolicy": "Always",
            "terminationGracePeriodSeconds": 30,
            "dnsPolicy": "ClusterFirst"
          }
        },
        "strategy": {
          "type": "RollingUpdate",
          "rollingUpdate": {
            "maxUnavailable": 1,
            "maxSurge": 1
          }
        },
        "uniqueLabelKey":
"deployment.kubernetes.io/podTemplateHash"
      },
      "status": {}
}
Finished: SUCCESS
```

7. Let's check it using the `kubectl` command line after a few minutes:

```
// check deployment status
# kubectl get deployments
```

```
NAME                 UPDATEDREPLICAS   AGE
my-cal-deployment    3/3               40m
```

We can see that there's a deployment named `my-cal-deployment`.

8. Using `kubectl describe`, you could check the details:

```
// check the details of my-cal-deployment
# kubectl describe deployment my-cal-
deployment
Name:        my-cal-deployment
Namespace:      default
CreationTimestamp:    Mon, 07 Mar 2016
03:20:52 +0000
Labels:        app=my-cal
Selector:      app=my-cal
Replicas:      3 updated / 3 total
StrategyType:      RollingUpdate
RollingUpdateStrategy:    1 max unavailable, 1
max surge, 0 min ready seconds
OldReplicationControllers:  <none>
NewReplicationController:       deploymentrc-
1448558234 (3/3 replicas created)
Events:
  FirstSeen  LastSeen  Count  From
SubobjectPath  Reason  Message
  _________  ________  _____  ____
_____________  ______    _______
  46m    46m    1  {deployment-controller }
ScalingRC  Scaled up rc deploymentrc-
3224387841 to 3
  17m    17m    1  {deployment-controller }
ScalingRC  Scaled up rc deploymentrc-
3085188054 to 3
  9m    9m    1  {deployment-controller }
ScalingRC  Scaled up rc deploymentrc-
1448558234 to 1
  2m    2m    1  {deployment-controller }
ScalingRC  Scaled up rc deploymentrc-
1448558234 to 3
```

We could see one interesting setting named `RollingUpdateStrategy`. We have `1 max unavailable`, `1 max surge`, and `0 min ready seconds`. It means that we could set up our strategy to roll the update.

Currently, it's the default setting; at the most, one pod is unavailable during the deployment, one pod could be recreated, and zero seconds to wait for the newly created pod to be ready. How about replication controller? Will it be created properly?

```
// check ReplicationController
# kubectl get rc
CONTROLLER                CONTAINER(S)   IMAGE(S)
SELECTOR
REPLICAS   AGE
deploymentrc-1448558234   my-cal
msfuko/my-cal:1        app=my-
cal,deployment.kubernetes.io/podTemplateHash=14485
58234    3          1m
```

We could see previously that we have three replicas in this RC with the name `deploymentrc-${id}`. Let's also check the pod:

```
// check Pods
# kubectl get pods
NAME                                READY     STATUS
RESTARTS   AGE
deploymentrc-1448558234-qn45f   1/1       Running
0          4m
deploymentrc-1448558234-4utub   1/1       Running
0          12m
deploymentrc-1448558234-iz9zp   1/1       Running
0          12m
```

We could find out deployment trigger RC creation, and RC trigger pods creation. Let's check the response from our app `my-calc`:

```
# curl http://54.153.44.46:31725/
Hello World!
```

Assume that we have a newly released application. We'll make **Hello world!** to be **Hello Calculator!**. After pushing the code into GitHub,

Jenkins could be either triggered by the SCM webhook, periodically run, or triggered manually:

```
[workspace] $ /bin/sh -xe
/tmp/hudson8771905048970590l3.sh
+ curl -XPUT -
d{"apiVersion":"extensions/v1beta1","kind":"Deploy
ment","metadata":{"name":"my-calc-
deployment"},"spec":{"replicas":3,"template":
{"metadata":{"labels":{"app":"my-calc"}},"spec":
{"containers":[{"name":"my-
calc","image":"msfuko/my-calc:2","ports":
[{"containerPort":5000}]}]}}}}
http://54.153.44.46:8080/apis/extensions/v1beta1/n
amespaces/default/deployments/my-calc-deployment
  % Total    % Received % Xferd  Average Speed
Time    Time     Time  Current
                                 Dload  Upload
Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:-
-:-- --:--:-- --:--:--     0
100  1695  100  1421  100   274  86879  16752 --:-
-:-- --:--:-- --:--:-- 88812
{
  "kind": "Deployment",
  "apiVersion": "extensions/v1beta1",
  "metadata": {
    "name": "my-calc-deployment",
    "namespace": "default",
    "selfLink":
"/apis/extensions/v1beta1/namespaces/default/deplo
yments/my-calc-deployment",
    "uid": "db49f34e-e41c-11e5-aaa9-061300daf0d1",
    "resourceVersion": "35756",
    "creationTimestamp": "2016-03-07T04:27:09Z",
    "labels": {
      "app": "my-calc"
    }
  },
  "spec": {
    "replicas": 3,
    "selector": {
      "app": "my-calc"
    },
```

```
    "template": {
      "metadata": {
        "creationTimestamp": null,
        "labels": {
          "app": "my-calc"
        }
      },
      "spec": {
        "containers": [
          {
            "name": "my-calc",
            "image": "msfuko/my-calc:2",
            "ports": [
              {
                "containerPort": 5000,
                "protocol": "TCP"
              }
            ],
            "resources": {},
            "terminationMessagePath":
"/dev/termination-log",
            "imagePullPolicy": "IfNotPresent"
          }
        ],
        "restartPolicy": "Always",
        "terminationGracePeriodSeconds": 30,
        "dnsPolicy": "ClusterFirst"
      }
    },
    "strategy": {
      "type": "RollingUpdate",
      "rollingUpdate": {
        "maxUnavailable": 1,
        "maxSurge": 1
      }
    },
    "uniqueLabelKey":
"deployment.kubernetes.io/podTemplateHash"
  },
  "status": {}
}
Finished: SUCCESS
```

How it works…

Let's continue the last action. We built a new image with the `$BUILD_NUMBER` tag and triggered Kubernetes to replace a replication controller by a deployment. Let's observe the behavior of the replication controller:

```
# kubectl get rc
CONTROLLER                CONTAINER(S)   IMAGE(S)
SELECTOR
REPLICAS   AGE
deploymentrc-1705197507   my-calc
msfuko/my-calc:1       app=my-
calc,deployment.kubernetes.io/podTemplateHash=1705
197507   3         13m
deploymentrc-1771388868   my-calc
msfuko/my-calc:2       app=my-
calc,deployment.kubernetes.io/podTemplateHash=1771
388868   0         18s
```

We can see deployment create another RC named `deploymentrc-1771388868`, whose pod number is currently 0. Wait a while and let's check it again:

```
# kubectl get rc
CONTROLLER                CONTAINER(S)   IMAGE(S)
SELECTOR
REPLICAS   AGE
deploymentrc-1705197507   my-calc
msfuko/my-calc:1       app=my-
calc,deployment.kubernetes.io/podTemplateHash=1705
197507   1         15m
deploymentrc-1771388868   my-calc
msfuko/my-calc:2       app=my-
calc,deployment.kubernetes.io/podTemplateHash=1771
388868   3         1m
```

The number of pods in RC with the old image `my-calc:1` reduces to 1 and the new image increase to 3:

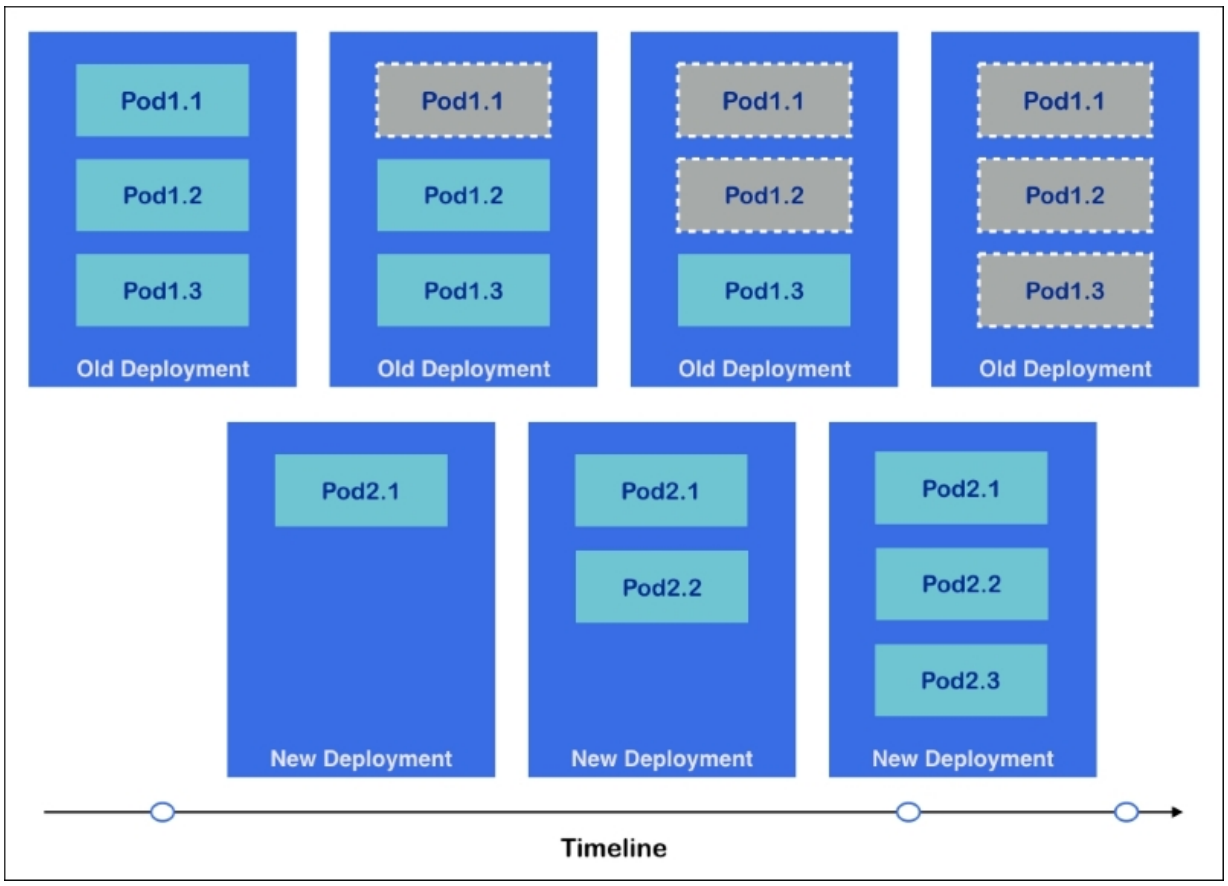
```
# kubectl get rc
```

```
CONTROLLER                CONTAINER(S)    IMAGE(S)
SELECTOR
REPLICAS   AGE
deploymentrc-1705197507   my-calc
msfuko/my-calc:1       app=my-
calc,deployment.kubernetes.io/podTemplateHash=1705
197507   0          15m
deploymentrc-1771388868   my-calc
msfuko/my-calc:2       app=my-
calc,deployment.kubernetes.io/podTemplateHash=1771
388868   3          2m
```

After a few seconds, the old pods are all gone and the new pods replace them to serve users. Let's check the response by the service:

```
# curl http://54.153.44.46:31725/
Hello Calculator!
```

The pods have been rolling updates to the new image individually. Following is the illustration on how it works. Based on `RollingUpdateStrategy`, Kubernetes replaces pods one by one. After the new pod launches successfully, the old pod is destroyed. The bubble in the timeline arrow shows the timing of the logs we got on the previous page. At the end, the new pods will replace all the old pods:



There's more…

Deployment is still in the beta version, while some functions are still under development, for example, deleting a deployment resource and recreating strategy support. However, it gives the chance to Jenkins to make the Continuous Delivery pipeline available. It's pretty easy and makes sure all the services are always online to update. For more details of the RESTful API, please refer to `http://YOUR_KUBERNETES_MASTER_ENDPOINT:KUBE_API_PORT/swagger-ui/#!/v1beta1/listNamespacedDeployment`.

See also

By deployment, we could achieve the goals of rolling the update. However, `kubectl` also provides a `rolling-update` command, which is really useful, too. Check out the following recipes:

- The *Updating live containers* and *Ensuring flexible usage of your containers* recipes in Chapter 3, *Playing with Containers*
- *Moving monolithic to microservices*
- *Integrating with Jenkins*
- The *Working with a RESTful API* and *Authentication and authorization* recipes in Chapter 7, *Advanced Cluster Administration*

Chapter 6. Building Kubernetes on AWS

In this chapter, we will cover the following topics:

- Building the Kubernetes infrastructure in AWS
- Managing applications using AWS OpsWorks
- Auto-deploying Kubernetes through Chef recipes
- Using AWS CloudFormation for fast provisioning

Introduction

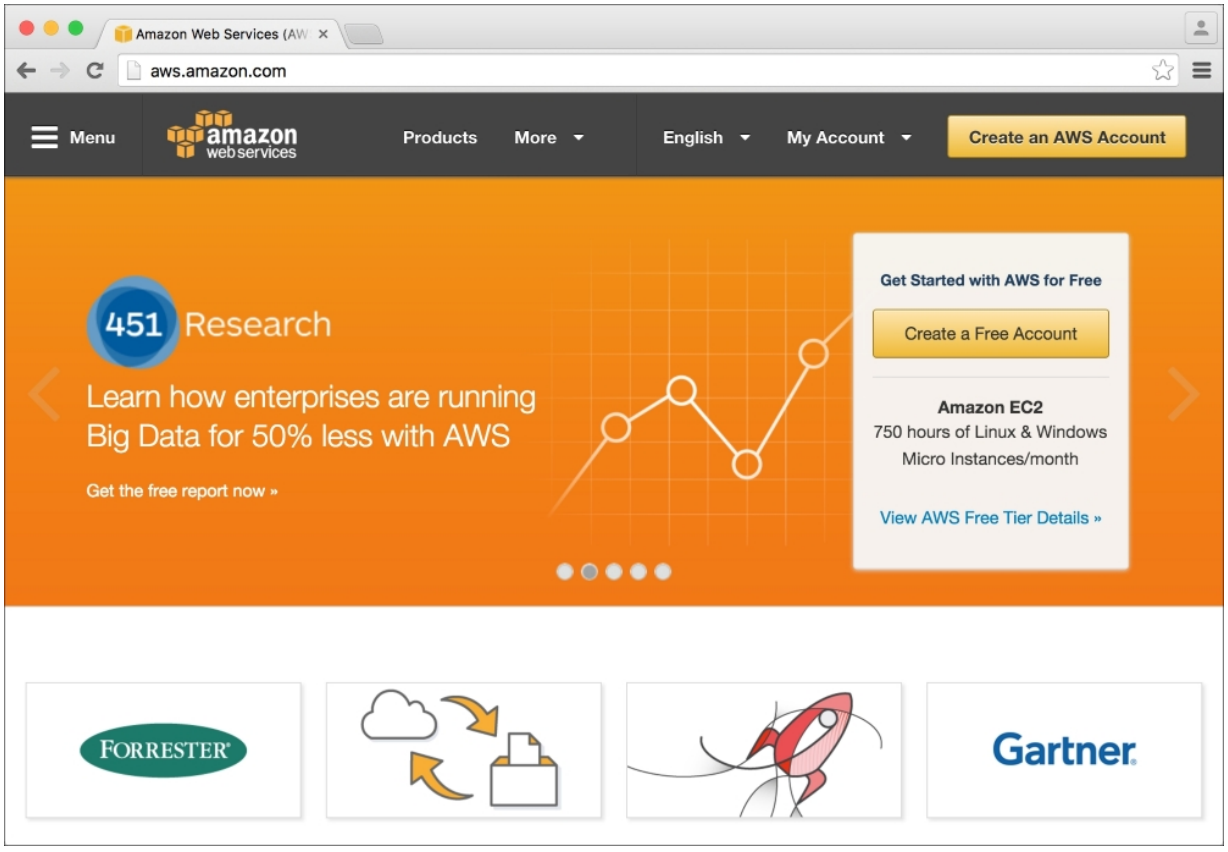
In this chapter, you will learn how to build up all the components on popular public cloud Amazon Web Services. However, we like our infrastructure as code. The infrastructure can be built repeatedly in a reliable way. You will learn how to manage an application's life cycle using AWS OpsWorks, which is powered by Chef. Finally, we will leverage what we learned and build all the infrastructure via a text file in the JSON format!

Building the Kubernetes infrastructure in AWS

Amazon Web Services (AWS) is the most popular cloud service. You can launch several virtual machines on the Amazon datacenter. This section covers sign-up, setting up AWS infrastructure, and launching Kubernetes on AWS.

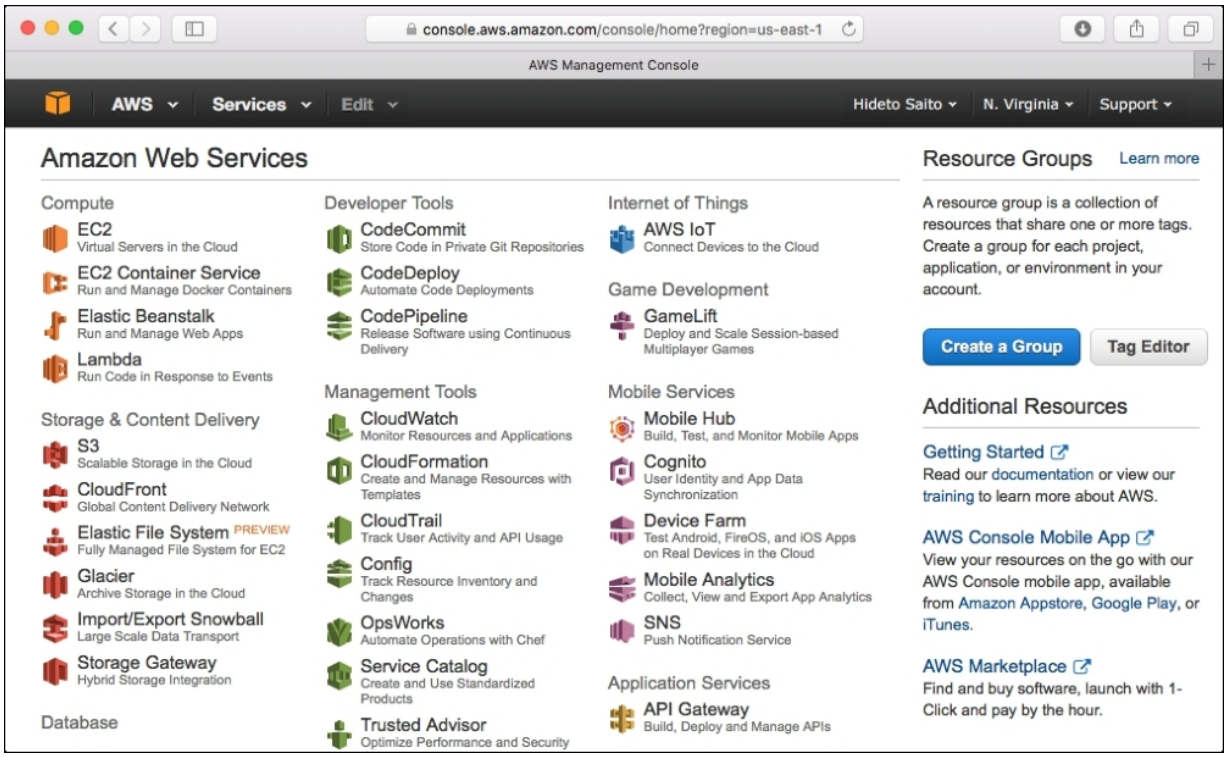
Getting ready

You must sign up to AWS to create an account. Access `http://aws.amazon.com` to put in your information and credit card number:



AWS registration

After registration, you may need to wait up to 24 hours in order to validate your account. After this, you will see the following page after logging on to the AWS console:



AWS console

How to do it…

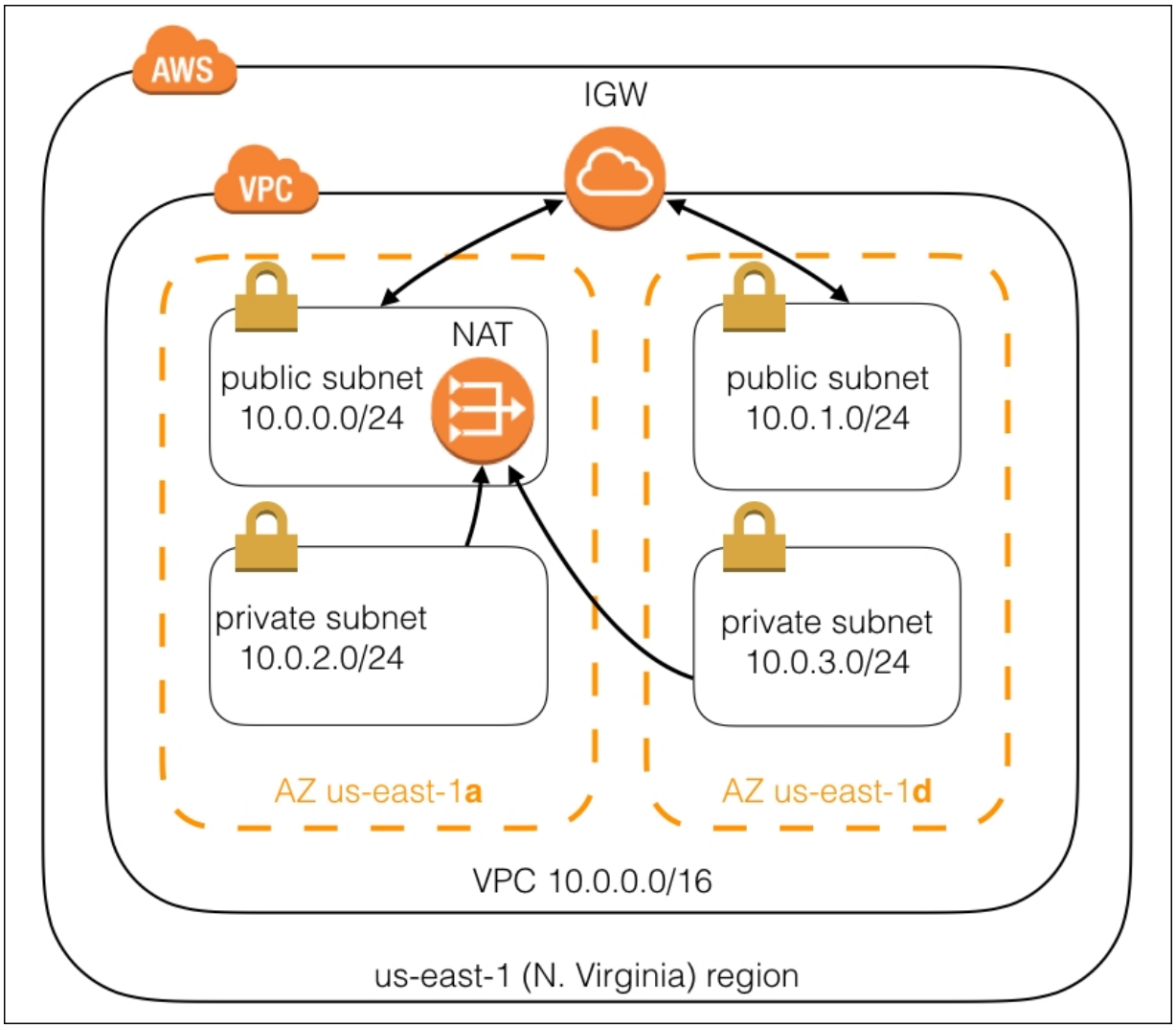
AWS supports multiple region datacenters; you may choose the nearest and cheapest region. Inside the region, there are several **Availability Zones** (**AZ**), which are physically isolated locations that are available.

Once you choose a region, you can set up the **Virtual Private Cloud** (**VPC**) on your own network, such as `10.0.0.0/16`. Inside VPC, you can also define public and private subnets that will do the following:

- Public subnet : Allows you to assign a public IP address and access from/to public Internet via Internet Gateway
- Private subnet : Assigns a private IP address only; can't access from public Internet, outgoing access to Internet through NAT

- Between public subnet and private subnet are accessible

Each subnet must be located in single AZ. Therefore, it would better to create multiple public subnets and multiple private subnets in order to avoid a **single point of failure** (**SPOF**).



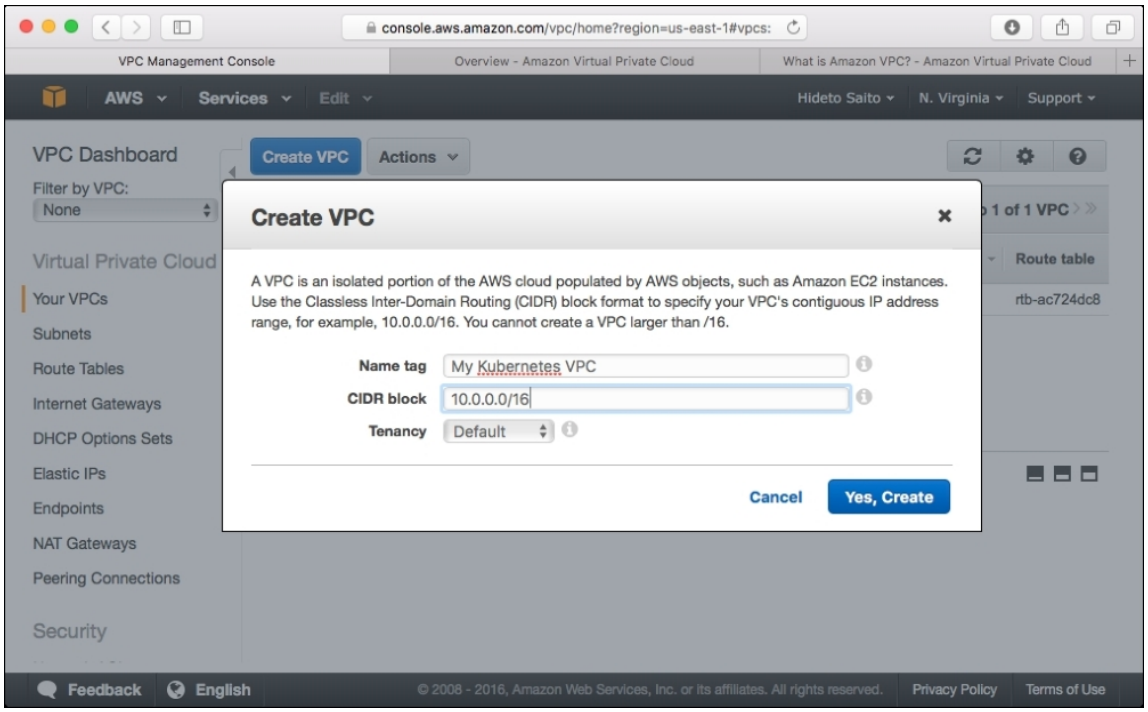
Typical VPC and subnet design

It would better to consider multiAZ for NAT; however, in this cookbook, we will skip it, as it is not necessary. For more details about the NAT gateway, please follow the link http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/vpc-nat-gateway.html.

Let's create this VPC on your AWS.

VPC and subnets

1. On the AWS console, access VPC and click on **Create VPC**. Then, input the name tag as **My Kubernetes VPC** and CIDR as **10.0.0.0/16**:



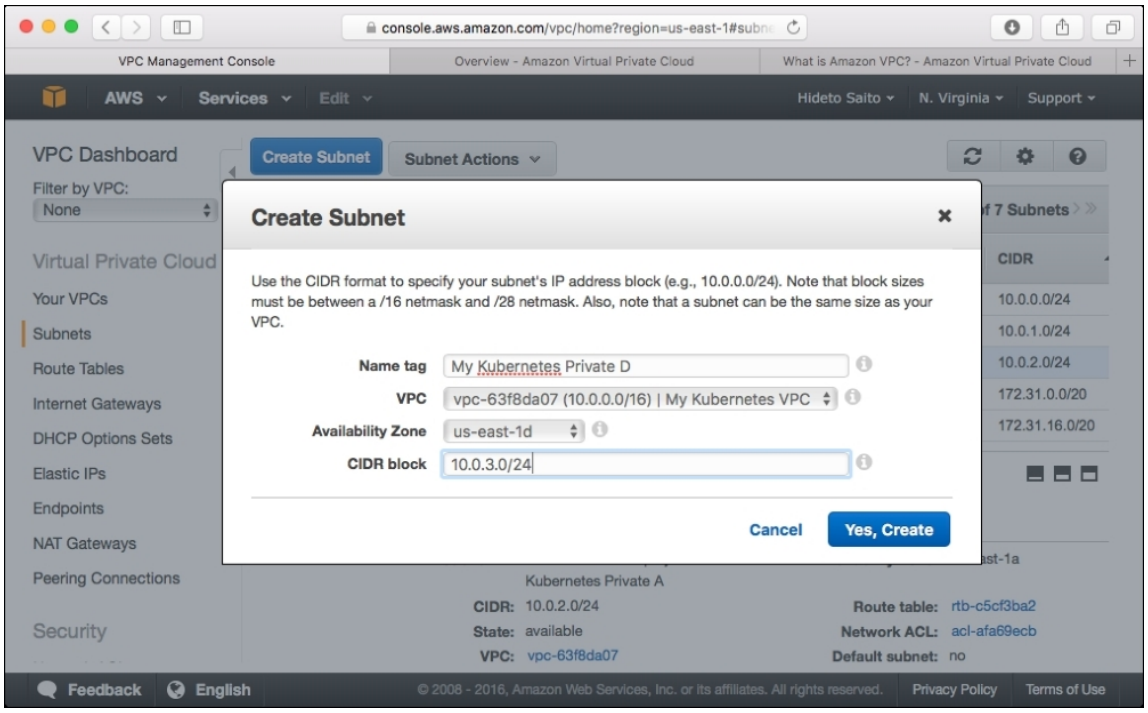
Create VPC window

Under the VPC, subnets for both public and private on multiAZ are mentioned in the following table:

Name Tag	CIDR Block	Availability Zone	Auto-Assign Public IP
My Kubernetes	10.0.0.0/24	us-east-1a	Yes

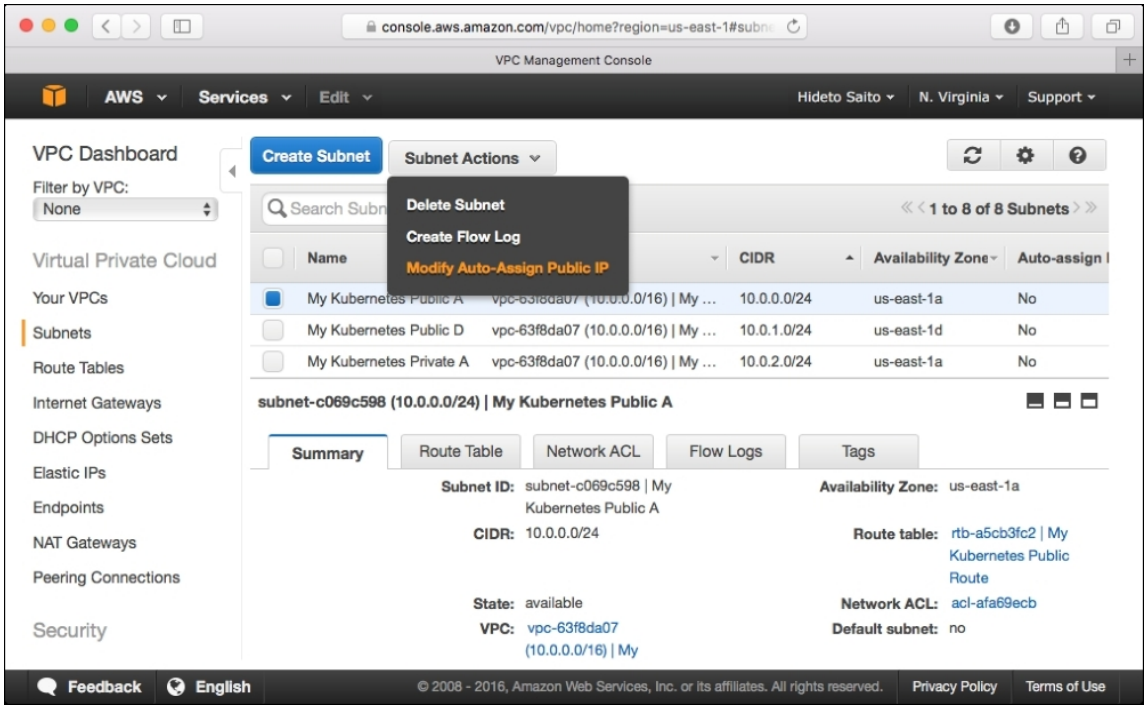
Name Tag	CIDR Block	Availability Zone	Auto-Assign Public IP
Public A			
My Kubernetes Public D	10.0.1.0/24	us-east-1d	Yes
My Kubernetes Private A	10.0.2.0/24	us-east-1a	No (Default)
My Kubernetes Private D	10.0.3.0/24	us-east-1d	No (Default)

2. Click on **Subnets** on the left navigation link. Then, click on the **Create Subnet** button. Fill out the information and choose the **VPC** and **Availability Zone** for each subnet. Repeat this four times to create four subnets:



Creating subnet

3. Select the public subnet and click on the **Subnet Actions** button. Then, choose **Modify Auto-Assign Public IP** to enable public IP auto-assignment.



Set Auto-Assign Public IP

Internet Gateway and NAT

Each subnet should have a gateway that goes to an external network. There are two types of gateway, as follows:

- **Internet Gateway** (**IGW**): It allows you to access from/to the Internet (bidirectional) for a server that has a public IP address
- **Network Address Translation** (**NAT**): It allows you to access the Internet (one direction) for a server that has a private IP address

Public subnets associate with an Internet Gateway; on the other hand, private subnets can go to the Internet through NAT. Let's create IGW and NAT as follows:

Type	**Associate to**
Internet Gateway	VPC
NAT Gateway	Public Subnet A

Route Table

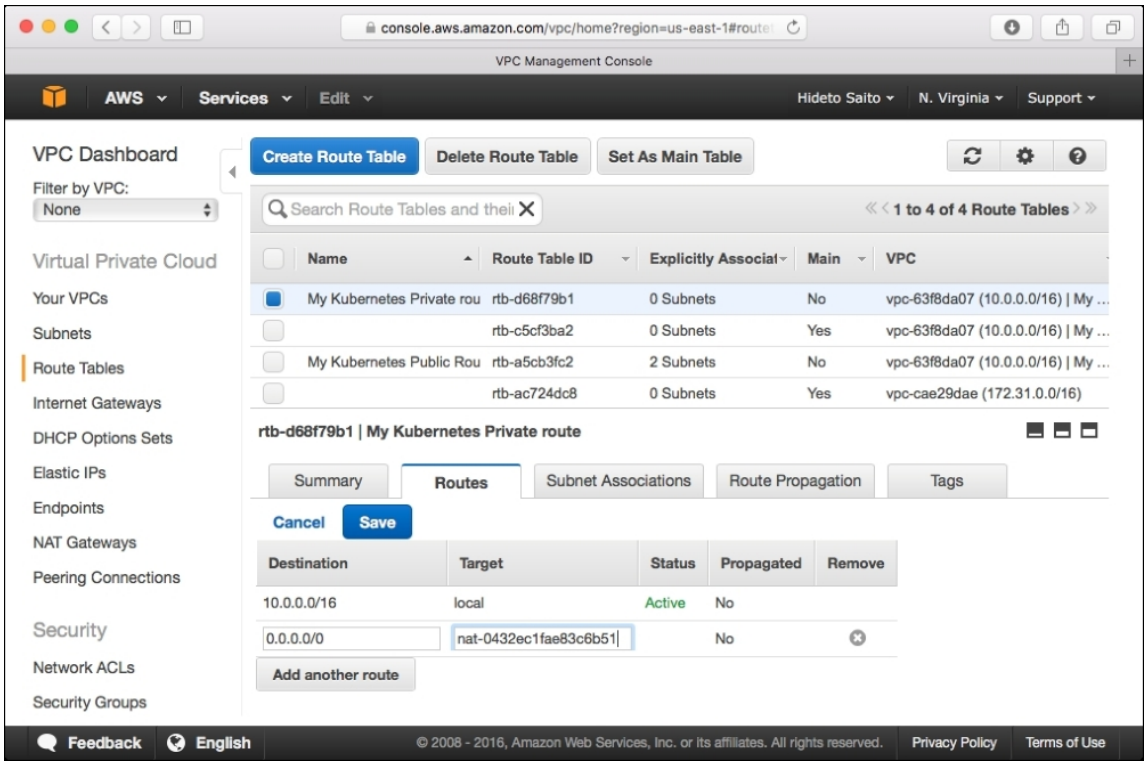
1. After creating IGW and NAT, you need to adjust the route table to set the default gateway to IGW or NAT, as follows:

Route Table Name	Route Destination	Association
`My Kubernetes Public Route`	`10.0.0.0/16 local` `0.0.0.0/0 IGW`	Public Subnet A Public Subnet D
`My Kubernetes Private Route`	`10.0.0.0/16 local` `0.0.0.0/0 NAT`	Private Subnet A Private Subnet D

2. On the AWS console, click on **Route Tables** on the left navigation pane. Then, click on the **Create Route Table** button. Fill out **Table Name** and choose the VPN that you created. Repeat this procedure twice for a public route and private route.
3. After creating routes, you need to add the default route as either Internet Gateway (IGW) or NAT.

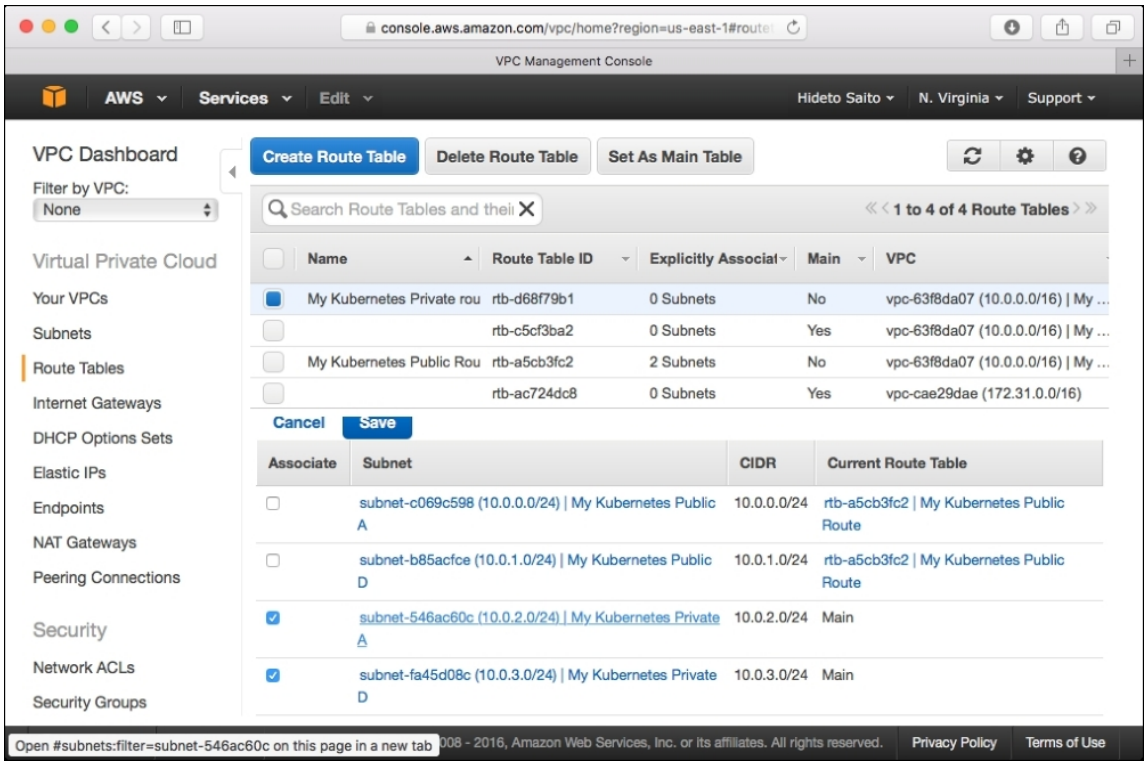
   For a public route, click on the **Routes** tab and click on **Edit**. Then, add the default route as `0.0.0.0/0` and Target as the IGW ID.

   For a private route, click on the **Routes** tab and click on **Edit**. Then, add the default route as `0.0.0.0/0` and Target as the NAT Gateway ID.



Set default route to NAT

4. Finally, click on the **Subnet Associations** tab and then on the **Edit** button. Then, choose public subnets for a public route and private subnets for a private route, as follows:



Associate route table to subnet

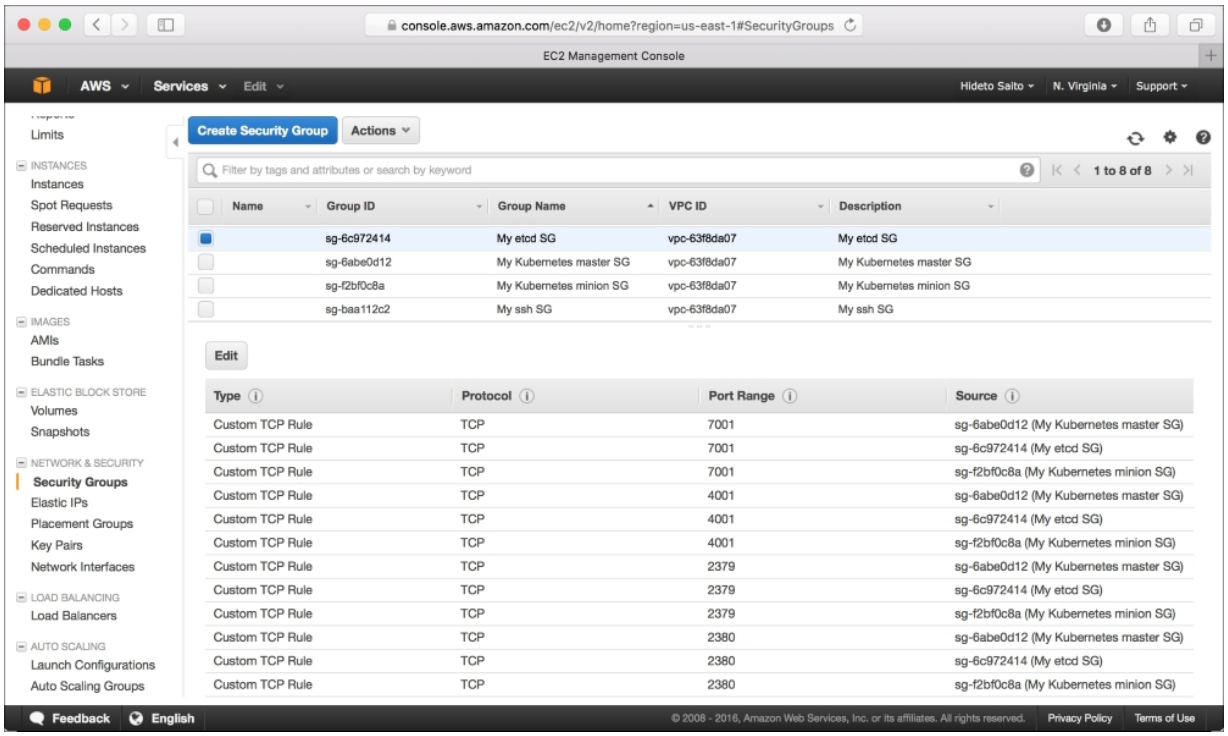
Security group

Security group is a kind of firewall to set up a rule that allows either inbound traffic or outbound traffic. For Kubernetes, there are some known traffic rules that should be set as follows:

Rule name	Inbound Protocol and port number	Source
My Kubernetes master SG	• 8080/tcp	• My Kubernetes node
My Kubernetes node SG	• 30000-32767/tcp (Service)	• 0.0.0.0/0

Rule name	Inbound Protocol and port number	Source
My etcd SG	• 7001/tcp • 4001/tcp • 2379/tcp • 2380/tcp	• My etcd SG • My Kubernetes master SG • My Kubernetes node SG
My flannel SG	• 8285/udp • 8472/udp	• My flannel SG
My ssh SG	• 22/tcp	• 0.0.0.0/0

On the AWS console, click on **Security Groups** on the left navigation pane, create five **Security Groups**, and add Inbound Rules, as follows:



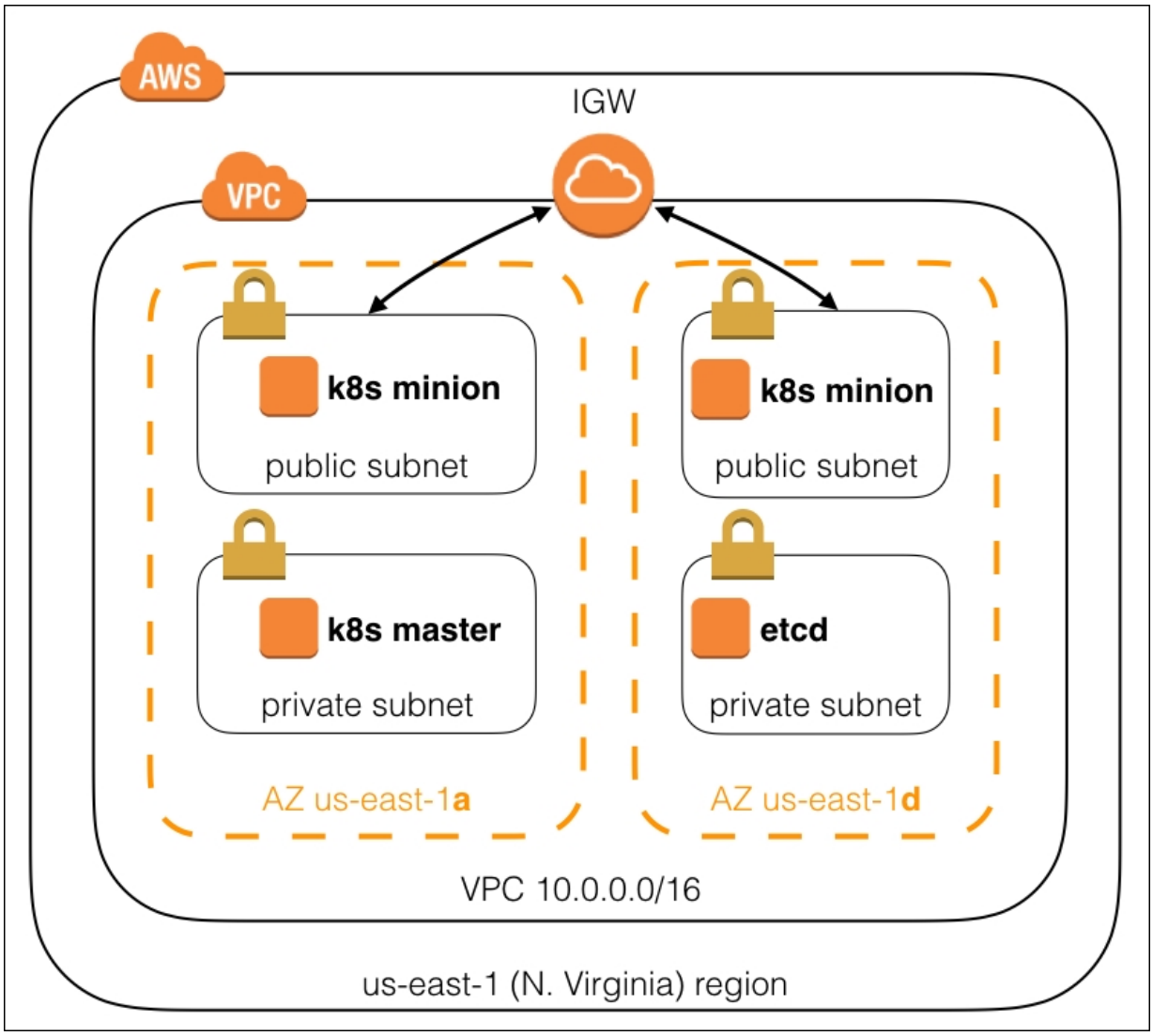
Creating a Security Group

How it works…

Once you create your own VPC and related subnets and security groups, you can launch the EC2 instance to set up your own Kubernetes cluster. Note that the EC2 instances should be launched and associated with subnets and security groups as follows:

Instance	Subnet	Security Group
etcd	Private	• `My etcd SG` • `My ssh SG`
Kubernetes node (with flannel)	Public	• `My flannel SG` • `My Kubernetes node SG` • `My ssh SG`

Instance	Subnet	Security Group
Kubernetes master (with flannel)	Private	• `My flannel SG` • `My Kubernetes master SG` • `My ssh SG`



Minimal configuration of the Kubernetes cluster in the AWS VPC

See also

In this recipe, you learned how to register Amazon Web Services and how to create your own infrastructure. AWS provides a huge number of services; however, it has a lot of documentation and related webinars online. It is recommended that you read and watch to understand the best practices to use the AWS infrastructure. Following is a list of good resources:

- https://www.youtube.com/user/AmazonWebServices
- https://aws.amazon.com/blogs/aws/
- http://www.slideshare.net/AmazonWebServices

Furthermore, look at the following recipes:

- The *Building datastore* and *Creating an overlay network* recipes in Chapter 1, *Building Your Own Kubernetes*
- *Managing applications using AWS OpsWorks*
- *Auto-deploying Kubernetes through Chef recipes*
- *Using AWS CloudFormation for fast provisioning*

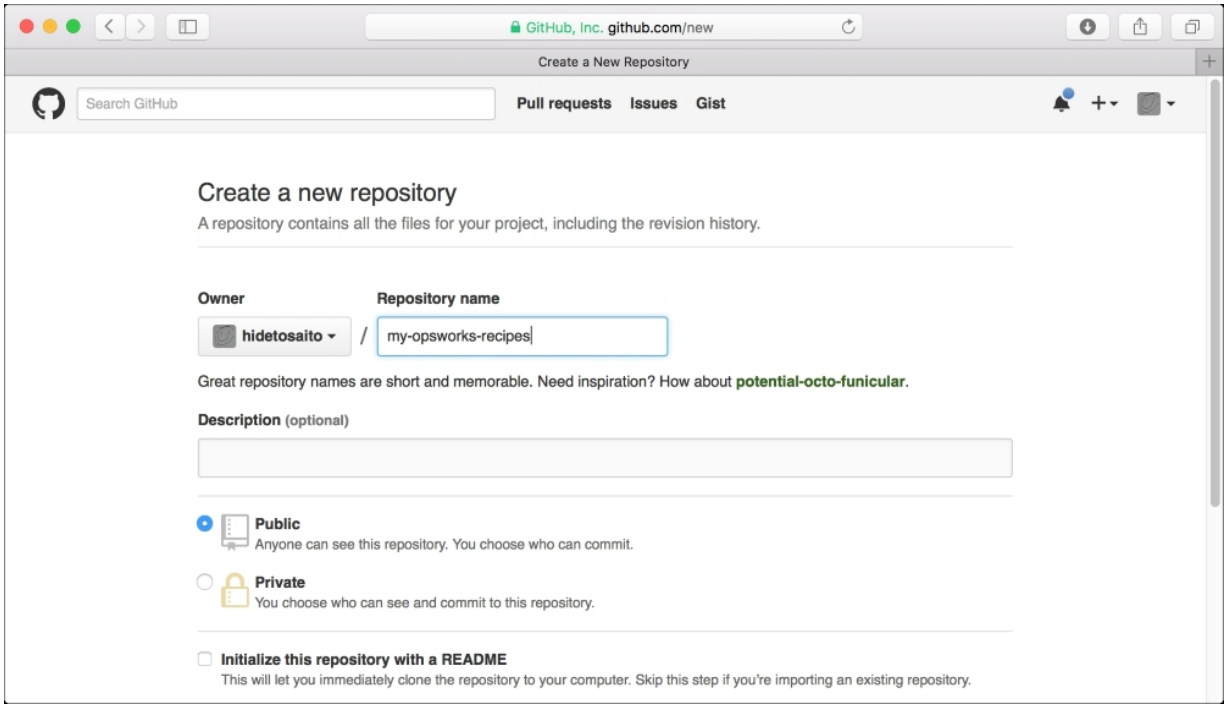
Managing applications using AWS OpsWorks

AWS OpsWorks is a comprehensive AWS EC2 and application deployment framework, which is based on Chef (http://chef.io/). It is easy to associate the Chef recipe and EC2 instance using the OpsWorks UI.

Getting ready

In order to upload and maintain your Chef recipes, it is recommended that you prepare a GitHub (http://github.com) account. After creating an account on GitHub, create a repository `my-opsworks-recipes` for OpsWorks recipes.

Just so you know, a free user on GitHub can only create a public repository.



Creating Git repository

After creating the `my-opsworks-recipes` repository, you can access the Git repository via `http://github.com/<your username>/my-opsworks-recipes.git`.

Let's use the AWS CloudWatchLogs as a sample deployment to put a recipe into your repository, as follows:

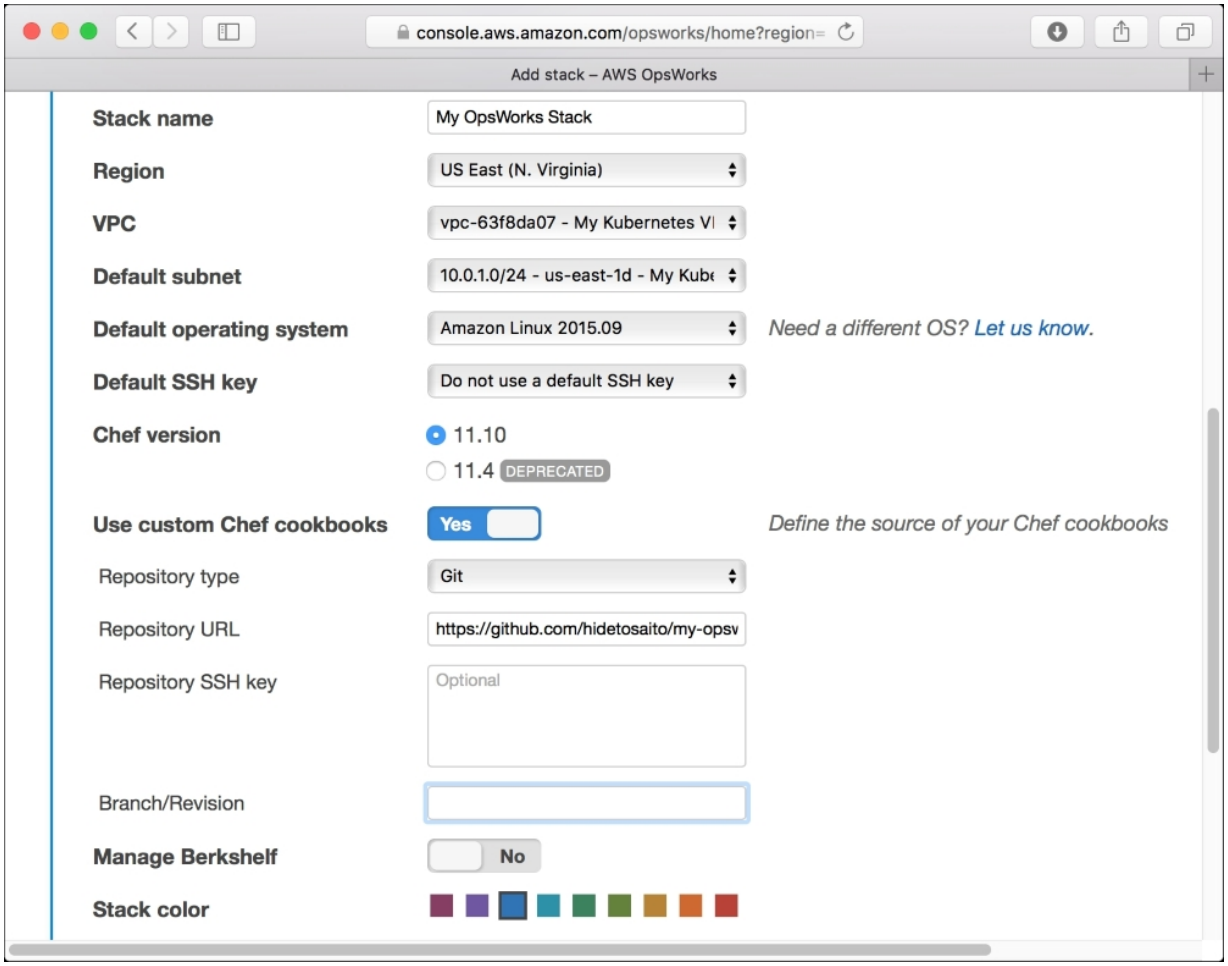
```
//Download CloudWatchLogs Cookbooks
$ curl -L -O https://s3.amazonaws.com/aws-
cloudwatch/downloads/CloudWatchLogs-Cookbooks.zip
//unzip
$ unzip CloudWatchLogs-Cookbooks.zip
//clone your GitHub repository
$ git clone https://github.com/hidetosaito/my-
opsworks-recipes.git
//copy CloudWatchLogs Cookbooks into your Git
repository
$ mv CloudWatchLogs-Cookbooks/logs my-opsworks-
recipes/
$ cd my-opsworks-recipes/
//add recipes to Git
$ git add logs
$ git commit -a -m "initial import"
[master (root-commit) 1d9c16d] initial import
 5 files changed, 59 insertions(+)
 create mode 100755 logs/attributes/default.rb
 create mode 100644 logs/metadata.rb
 create mode 100755 logs/recipes/config.rb
 create mode 100755 logs/recipes/install.rb
 create mode 100755
logs/templates/default/cwlogs.cfg.erb
//push to GitHub.com
$ git push
Username for 'https://github.com': hidetosaito
Password for 'https://hidetosaito@github.com':
Counting objects: 12, done.
```

How to do it…

Access the AWS Web console and navigate to OpsWorks. Then, create an OpsWorks stack first and an OpsWorks layer.

The OpsWorks stack

The OpsWorks stack is the container of the OpsWorks framework. The OpsWorks stack can associate one VPC. You can use your own VPC or the default VPC. Note that, due to compatibility reasons, choose Chef `11.10` to use the `CloudWatchLogs` recipe and don't forget to enable the custom Chef cookbooks and specify your GitHub repository, as follows:



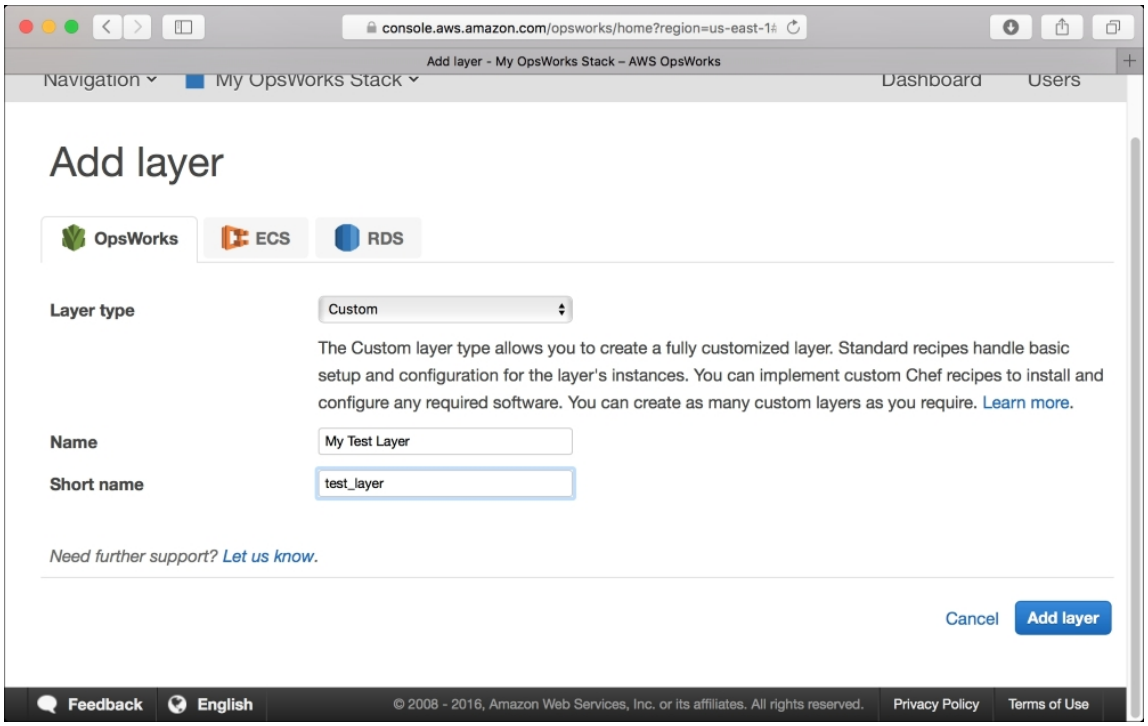
Creating the OpsWorks stack

The OpsWorks layer

Inside of the OpsWorks stack, you can have one or many OpsWorks layers. One layer can associate one or many Chef recipes. Let's

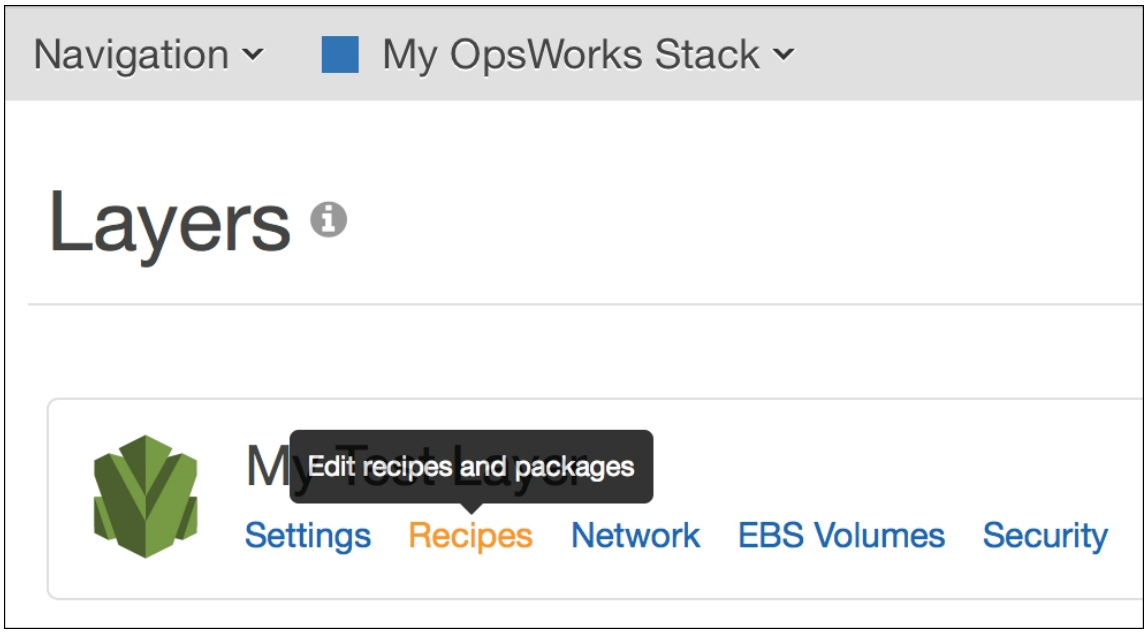
create one custom layer and associate with the `CloudWatchLogs` recipe:

1. Access OpsWorks UI to create a custom layer and put the layer name as follows:



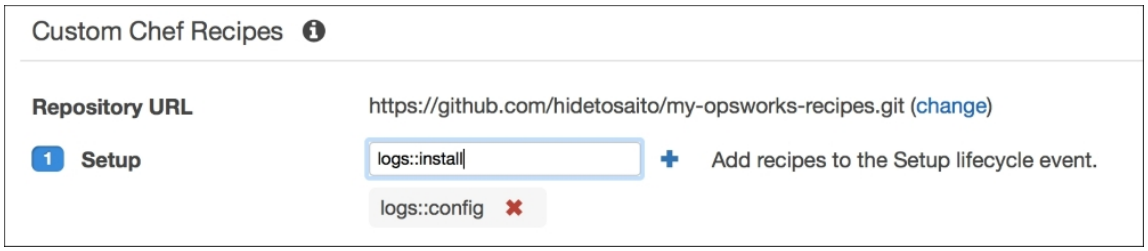
Creating the OpsWorks layer

2. Then, open the recipe page, as shown in the following screenshot:



Recipes settings page

3. Then, add the `logs::config` and `logs::install` recipes to set up a lifecycle event:

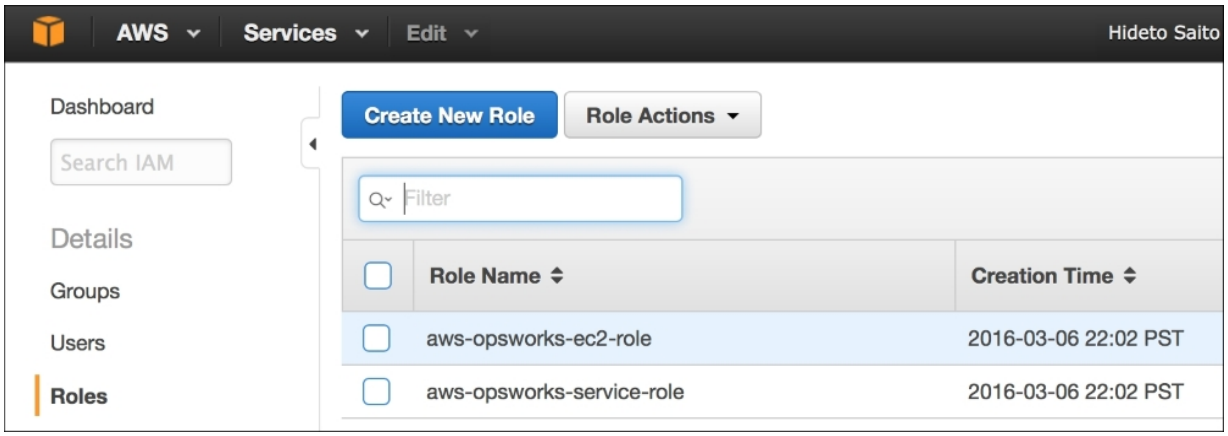


Associate recipes

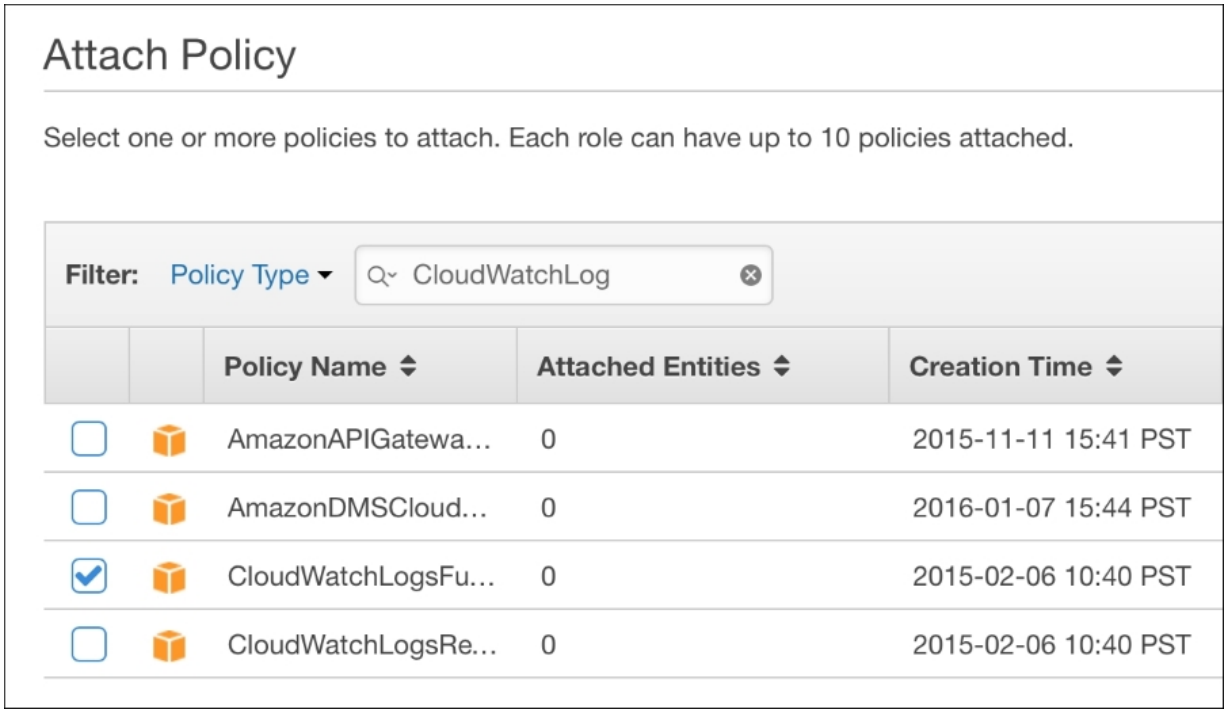
Adjusting the IAM role

In order to send a log to the `CloudWatchLogs` service, you need to grant permission to the IAM role. Access the AWS IAM console and choose `aws-opsworks-ec2-role`.

The `aws-opsworks-ec2-role` role is created with the OpsWorks stack by default. If you use another role, you would need to change the OpsWorks stack setting to choose your role.



Then, attach the `CloudWatchLogsFullAccess` policy, as shown in the following screenshot:

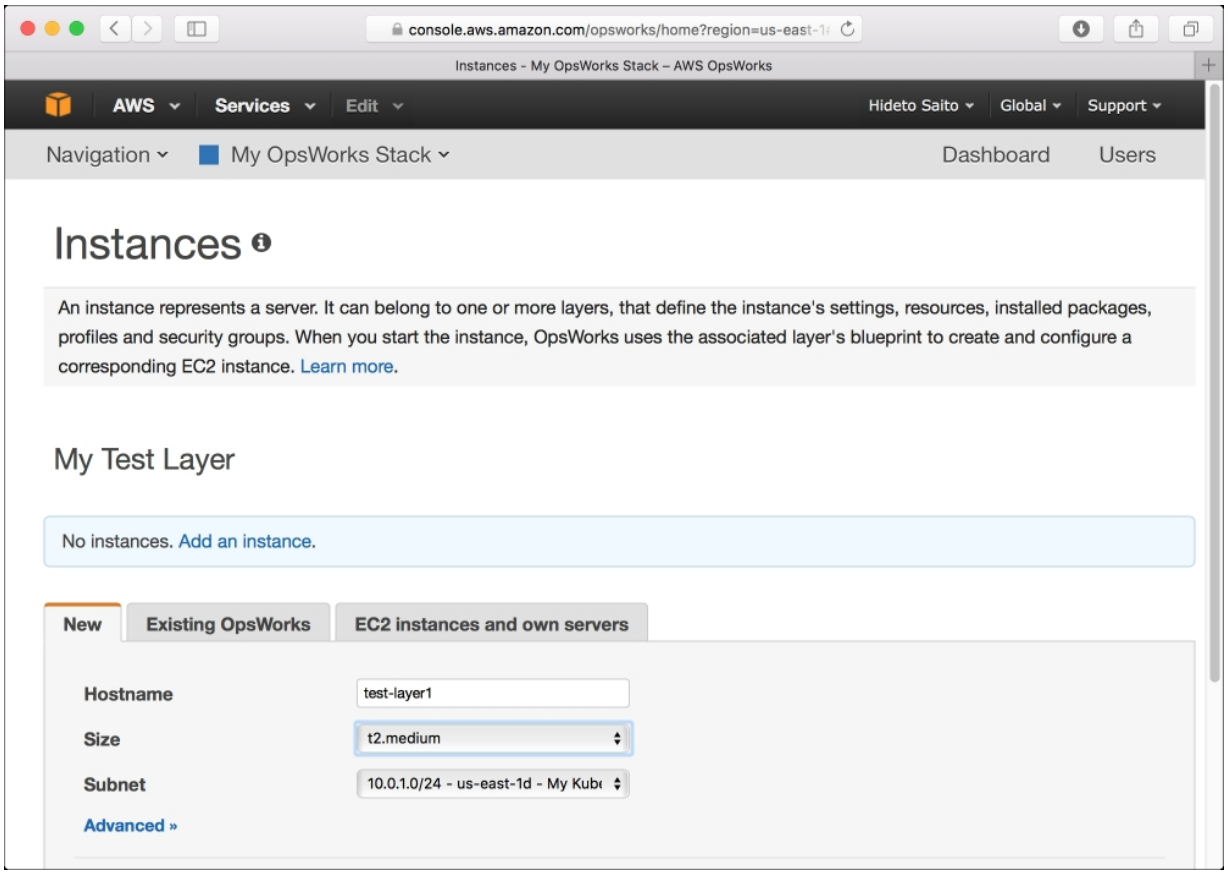


This role will be used when you launch the OpsWorks instance, which uses the `CloudWatchLogs` recipe.

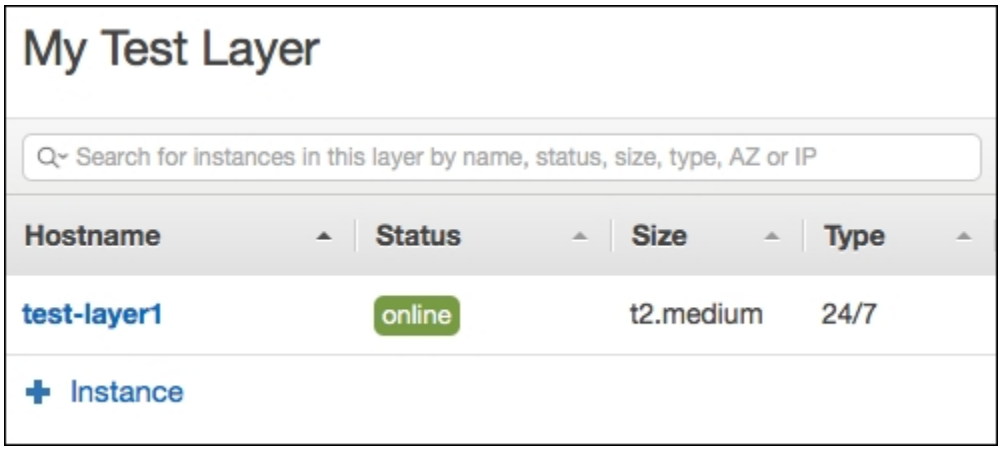
The OpsWorks instance

The OpsWorks instance is a managed EC2 instance, which is associated with the OpsWorks layer. It will automatically set up the

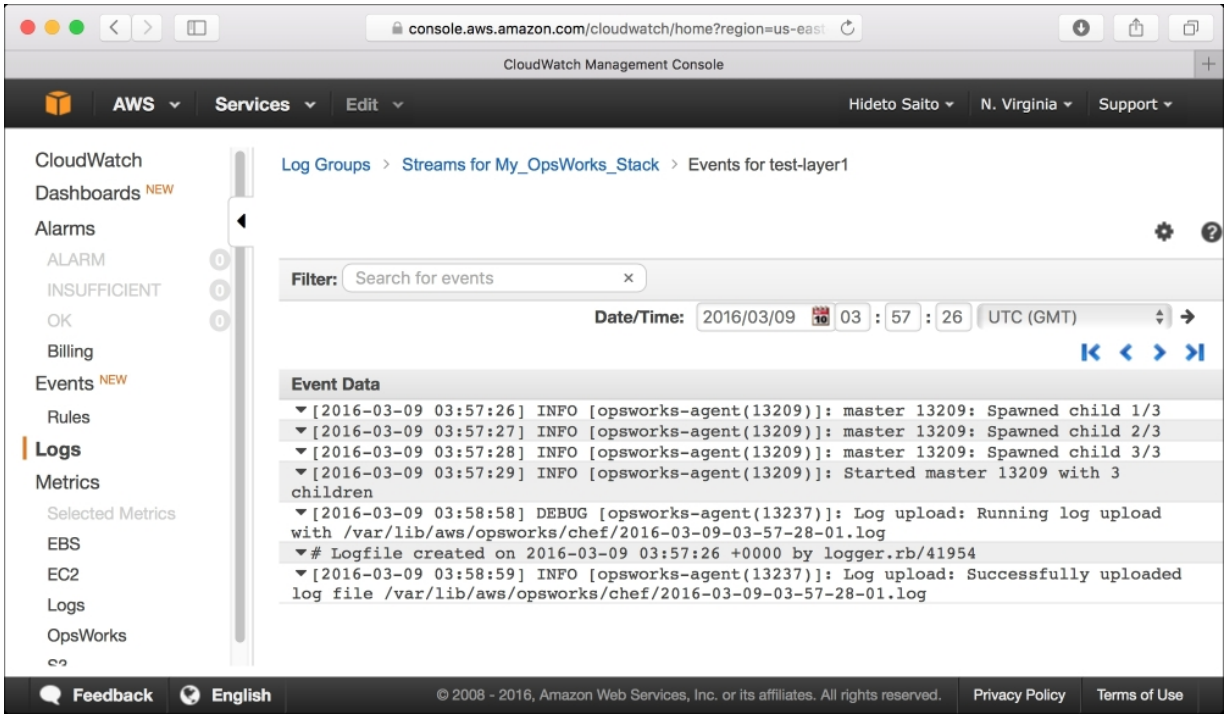
Chef Environment and cook your recipe. Let's access the AWS OpsWorks console again and choose the OpsWorks layer to launch an instance, as follows:



After a few minutes, your instance state will be **online** which means, the launch of an EC2 instance and installation of the `CloudWatchLogs` agent has been completed:



Now, you can see some logs on the AWS CloudWatchLogs console, as shown in the following screenshot:



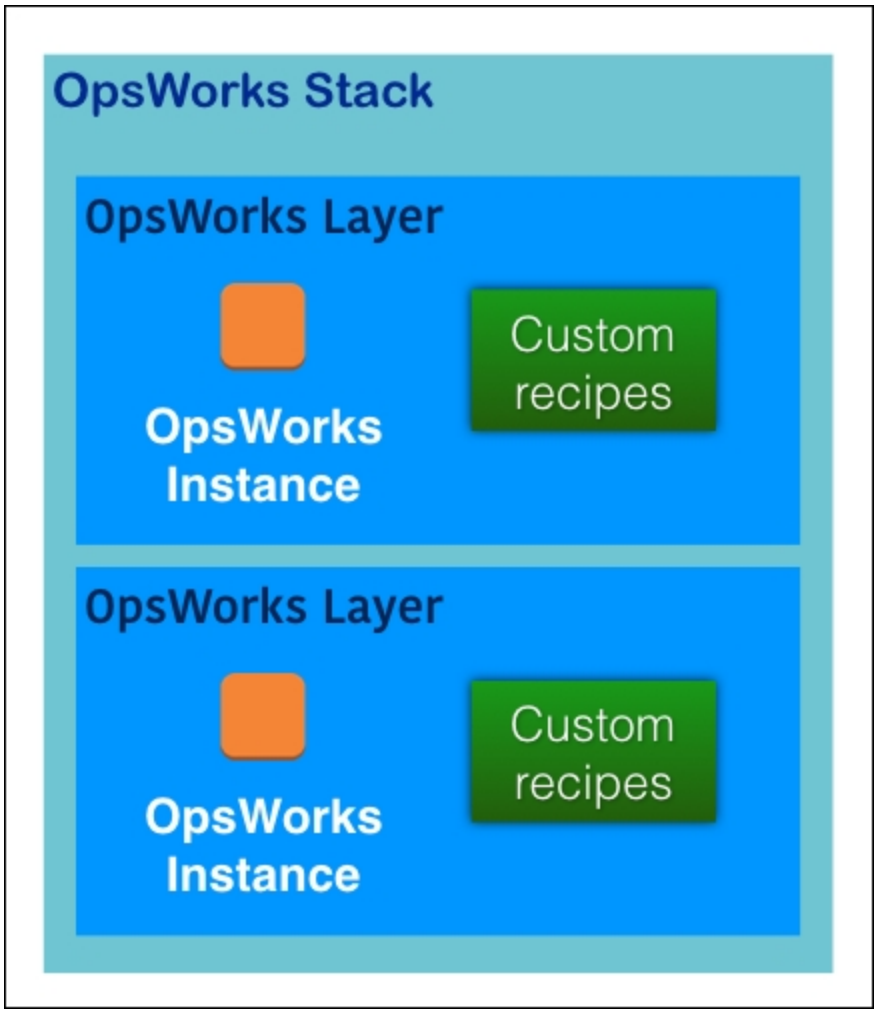
How it works…

Once the OpsWorks instance is launched, it will refer to the associated OpsWorks layer to execute Chef recipes in the particular lifecycle event, as follows:

Lifecycle event	**Timing**
Setup	After instance has finished booting
Configure	On entering or leaving the online state, associating or disassociating an Elastic IP, attaching or detaching from Elastic Load Balancer

Lifecycle event	**Timing**
Deploy	Deploying the application (non custom layer)
Undeploy	When deleting an application (non custom layer)
Shutdown	Before shutdown of an instance

Again, the OpsWorks stack has one or more OpsWorks layers. In addition, each layer can associate one or more custom Chef recipes. Therefore, you should define the layer as an application role, such as frontend, backend, datastore, and so on.



For the Kubernetes setup, it should be defined as follows:

- The Kubernetes master layer
- The Kubernetes node layer
- The etcd layer

These Chef recipes will be described in the next section.

See also

In this recipe, we introduced the OpsWorks service that can define your stack and layers. In this recipe, we used the `CloudWatchLogs` Chef recipe as an example. However, Kubernetes can also be automated to install the agent via the OpsWorks Chef recipe. It is described in the following recipes of this chapter as well:

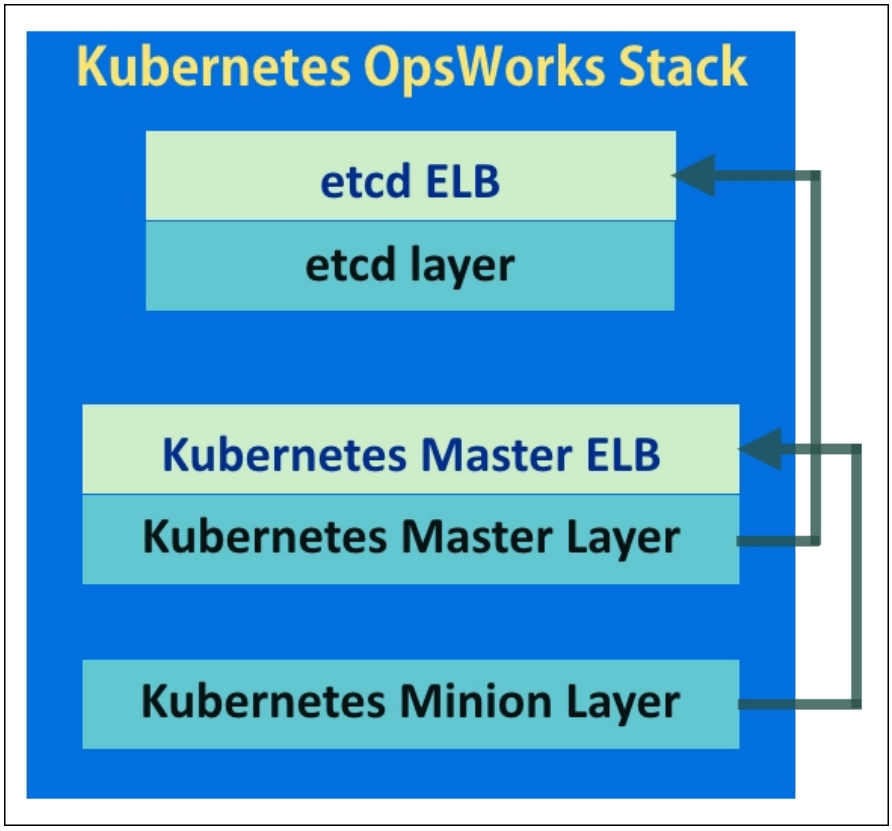
- *Building the Kubernetes infrastructure in AWS*
- *Auto-deploying Kubernetes through Chef recipes*
- *Using AWS CloudFormation for fast provisioning*

Auto-deploying Kubernetes through Chef recipes

To achieve fast deployment in AWS OpsWorks, we can write installation procedures in Chef recipes. Chef is a Ruby-based, auto-deployment managing tool (https://www.chef.io). It can help for program deployment and system configuration. In this recipe, we will show you how Chef works with the AWS OpsWorks.

Getting ready

In the following sections, we will show you how to use Chef recipes with the OpsWorks stack. Therefore, please prepare the OpsWorks environment. Based on the previous recipes in this chapter, we can build a Kubernetes stack with the following structure:



Let's consider that you have the same network settings mentioned in the recipe *Building the Kubernetes infrastructure in AWS*, which means that the VPC, subnets, route tables, and security groups are all ready for use. Then, we can apply the network environment directly in OpsWorks.

Tip

AWS region should be the same for resource utility

Although OpsWorks is a global service, we cannot combine the computing resources across different regions. Be aware that you need to choose the same AWS region of the network environment to create ELB and security groups.

Creating ELB and its security groups

As you can see in the previous stack structure, it is recommended to create ELBs beyond etcd and the Kubernetes master. Because both etcd and the master could be a cluster with multiple nodes, an ELB layer will provide the portal for the other application layers and balance the load to the working nodes. First, let's create the security groups of these two ELBs:

Rule name	Inbound Protocol and port number	Source
`My ELB of etcd SG`	• `80/tcp`	• `My Kubernetes master` • `My Kubernetes node`
`My ELB of Kubernetes master SG`	• `8080/tcp`	• `My Kubernetes node SG`

Next, modify the existing security groups as follows. It will make sure that the network traffic is redirected to ELB first:

Rule name	Inbound Protocol and port number	Source
`My etcd SG`	• `7001/tcp` • `4001/tcp`	• `My etcd SG` • `My ELB of etcd SG`
`My Kubernetes master SG`	• `8080/tcp`	• `My ELB of Kubernetes master SG`

Then, we can create the ELBs with the specified security groups. Go to the EC2 console and click on **Load balancers** on the left-hand side menu. Create new ELBs with the following configurations:

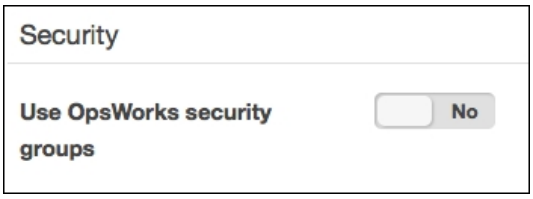
ELB name	VPC	Listener Configuration (ELB Protocol:Port/Instance Protocol: Port)	Subnets	Security Groups	Health check (Ping Protocol:Ping Port/Ping Path)
my-etcd-elb	My Kubernetes VPC (10.0.0.0/16)	HTTP:80/HTTP:4001	My Kubernetes Private A + My Kubernetes Private D	My ELB of etcd SG	HTTP:4001/version
my-k8s-master-elb		HTTP:8080/HTTP:8080		My ELB of Kubernetes master SG	HTTP:8080/version

Except for the previous configurations, it is fine to leave other items with the default ones. You don't have to add any instances in ELBs.

Creating an OpsWorks stack

Defining an application stack in OpsWorks is simple and easy. Refer to the detailed step-by-step approach as follows:

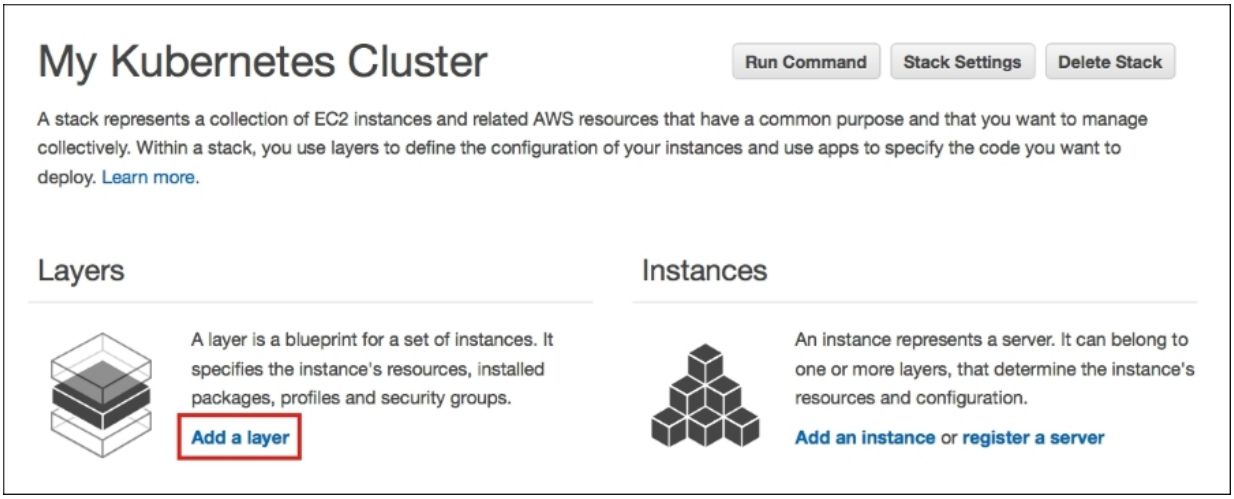
1. Click on the **Add stack** button and you will enter the AWS OpsWorks console.
2. Fill in the following items. It is fine to leave the non-mentioned parts with default values:
    1. Choose **Chef 12 stack**.
    2. Give a stack name. For example, My Kubernetes Cluster.
    3. Assign region and VPC which you just configured for Kubernetes.
    4. For operating systems, a Linux system and the latest Amazon Linux are good for installation later on. For example, Amazon Linux 2015.09.
    5. Click on **Advanced>>** beneath the configurations and disable the block **Use OpsWorks security groups**. Since we have already set up the required security groups, this movement can prevent a lot of unnecessary security groups from being created automatically.



6. Now, go ahead and click on **Add stack** to create a Kubernetes stack.

Creating application layers

After we have the OpsWorks stack, let's create the application layers and attach ELBs:



To create layers in the stack, click on **Add a layer** on the front page of the stack:

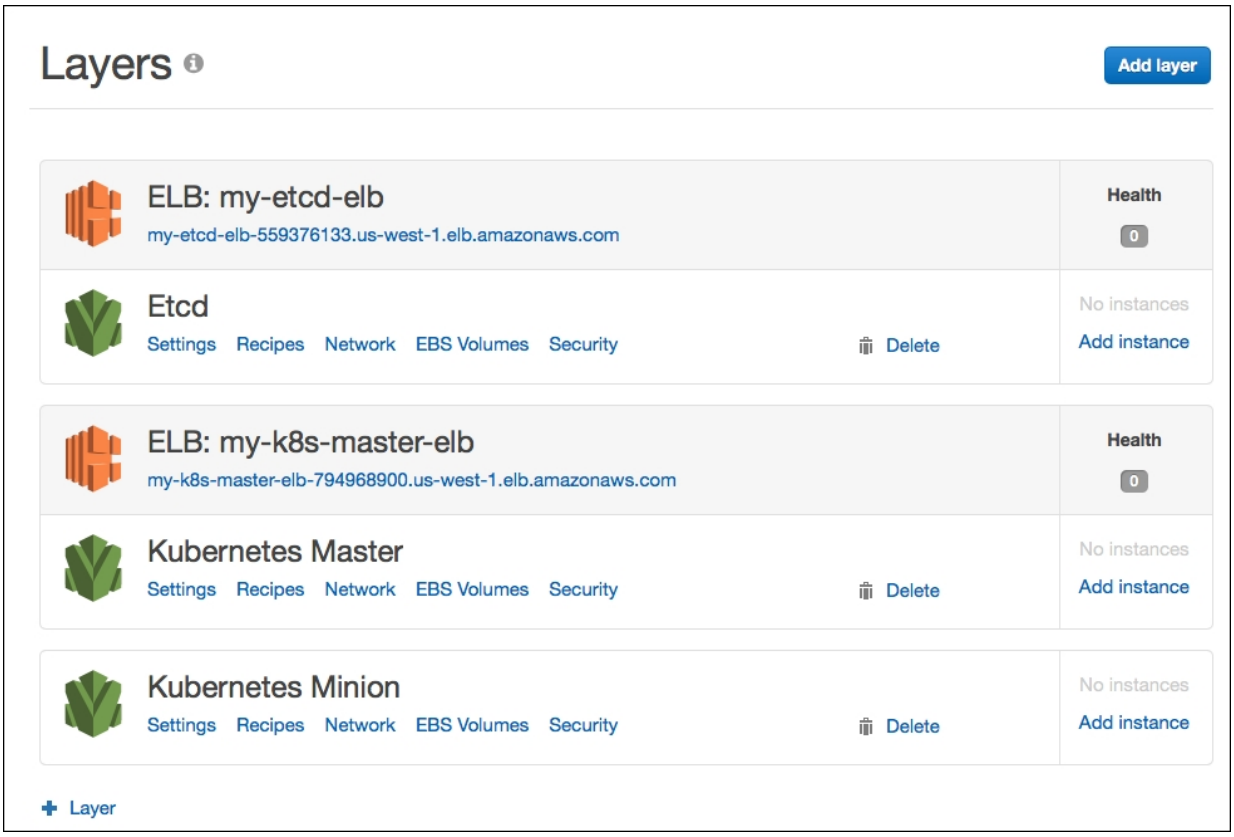


We cannot attach an ELB to a layer at the initial step of creation. Click on **Network** for specified layer modifications after they are created. Help yourself to generate the following layers:

Layer Name	**Short Name (the name as the prefix of instance)**	**Security Group**	**Attached ELB**
Etcd	etcd	My etcd SG	my-etcd-elb
Kubernetes Master	k8s-master	My Kubernetes master SG My flannel SG (optional)	my-k8s-master-elb

Layer Name	Short Name (the name as the prefix of instance)	Security Group	Attached ELB
Kubernetes Node	k8s-node	My Kubernetes node SG My flannel SG	

You will realize that the stack looks as follows, which is the same structure we mentioned at the beginning:



Now, the OpsWorks stack has a basic system structure, but without customized Chef recipes. We will go through the recipe's contents and setup concepts in the next section.

How to do it…

For Kubernetes installation using the Chef recipe, we will prepare a GitHub repository with the following files and relative paths:

```
$ tree .
.
└── kubernetes
```

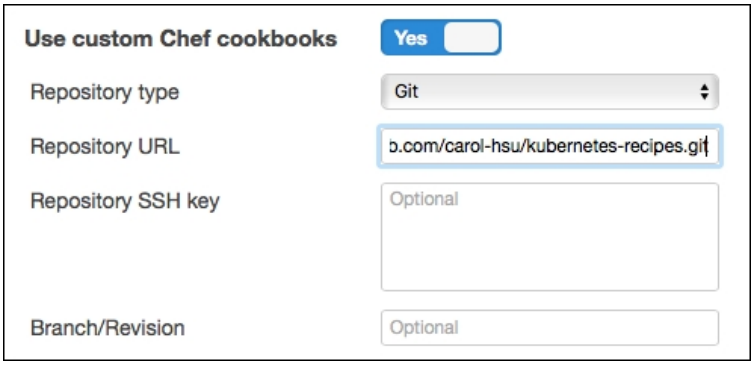
```
├── recipes
│   ├── docker.rb
│   ├── etcd-run.rb
│   ├── etcd.rb
│   ├── flanneld.rb
│   ├── kubernetes-master-run.rb
│   ├── kubernetes-master-setup.rb
│   ├── kubernetes-node-run.rb
│   ├── kubernetes-node-setup.rb
│   └── kubernetes.rb
└── templates
    └── default
        ├── docker.erb
        ├── etcd.erb
        ├── flanneld.erb
        ├── kubernetes-master.erb
        └── kubernetes-node.erb

4 directories, 14 files
```

In this section, different layers will illustrate the recipes and templates separately, but comprehensively. Create the repository and directory on the GitHub server in advance. It will help you set the customized stack configurations.

Stack configuration for custom recipes

In order to run the recipes in the stack, we are supposed to add two items in the configuration of the stack: one is the URL of the GitHub repo, which stored the recipe directory `kubernetes`, the other one is custom JSON. We may put some input parameters for the execution of recipes.

To modify the settings of the current stack, click on **Stack Settings** on the main page and then click on **Edit**. Enable **Use custom Chef Cookbooks**. You will find additional items showing up for codebase configuration. Then, put your GitHub repository URL for reference:



You can also check our GitHub repository for more information via https://github.com/kubernetes-cookbook/opsworks-recipes.git.

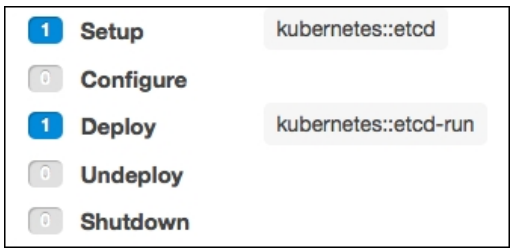
Next, at the block of **Advanced options**, please key in the following information in the item **Custom JSON**:

```
{
  "kubernetes": {
    "version":"1.1.8",
    "cluster_cidr":"192.168.0.0/16",
    "master_url":"<The DNS name of my-k8s-master-elb>"
  },
  "etcd": {
    "elb_url":"<The DNS name of my-etcd-elb>"
  }
}
```

The content of JSON is based on our Chef recipes. Users can define any key-value structure data. Usually, we put the one that may dynamically change by each deployment, for example, the version of the Kubernetes package. It would be better not to have hard code in our recipes, or some data that you don't want to show in the recipes can be made as input parameters. For example, the URL of ELB; it is a changeable value for each deployment. You possibly don't want others to know it. After configure the GitHub repository and custom JSON, we are ready to configuring the recipes in each layer.

Recipes for etcd

The lifecycle event of the etcd layer is as follows:



We will separate the functionality of etcd recipes into two event stages: `kubernetes::etcd` is set at the **Setup** stage for etcd installation and configuration while `kubernetes::etcd-run` is at the **Deploy** stage to start the daemon:

```
$ cat ./kubernetes/recipes/etcd.rb
bash 'install_etcd' do
  user 'root'
  cwd '/tmp'
  code <<-EOH
  if [ ! -f /usr/local/bin/etcd ]; then
    wget --max-redirect 255
https://github.com/coreos/etcd/releases/download/v2.2.5/etcd-v2.2.5-
linux-amd64.tar.gz
    tar zxvf etcd-v2.2.5-linux-amd64.tar.gz
    cd etcd-v2.1.1-linux-amd64
    cp etcd etcdctl /usr/local/bin
  fi
  EOH
end

template "/etc/init.d/etcd" do
  mode "0755"
  owner "root"
```

```
  source "etcd.erb"
end
```

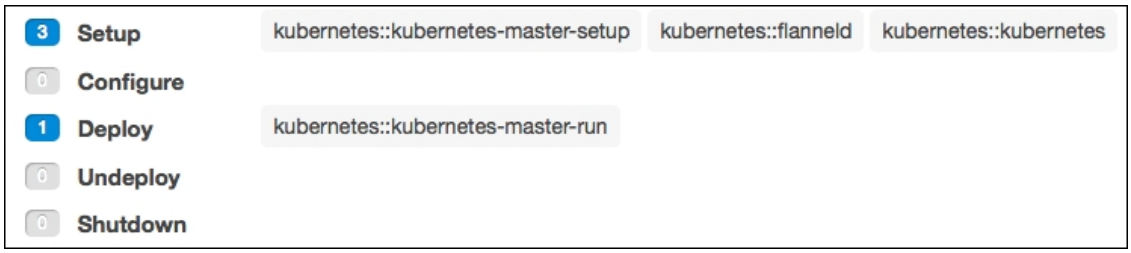
The recipe, `etcd.rb`, does the installation first. It will put the tarball at a temporary location and copy the binary file as a shared local command. To prevent the instance booting up from an already installed environment, we will add an `if` statement to check whether the system has the `etcd` command or not. There is a template `etcd.erb` working as a service configuration file. No dynamic input parameters are needed in this Chef template. It is also the same as we mentioned in the *Building datastore* recipe in Chapter 1, *Building Your Own Kubernetes*:

```
$ cat ./kubernetes/recipes/etcd-run.rb
service 'etcd' do
action [:enable,:start]
end
```

We have a short function in the recipe, `etcd-run.rb`, which enables and starts the etcd service. The stage, **Deploy** will run directly after **Setup**. Therefore, it is confirmed that the installation will finish before starting the service.

Recipes for the Kubernetes master

The recipes for installing the Kubernetes master are configured, as shown in the following screenshot:



Just like the etcd layer, we use the **Setup** stage for installation and configuration file assignment. The recipe `kubernetes::kubernetes` is used for downloading the Kubernetes package. It will be shared to the node layer as well:

```
$ cat ./kubernetes/recipes/kubernetes.rb
bash 'install_kubernetes' do
  user 'root'
  cwd '/tmp'
  code <<-EOH
  if [[ $(ls /usr/local/bin/kubectl) ]]; then
    current_version=$(/usr/local/bin/kubectl version | awk 'NR==1' |
awk -F":\"v" '{ print $2 }' | awk -F"\"," '{ print $1 }')
    if [ "$current_version" -eq "#{node['kubernetes']['version']}" ];
then
        exit
    fi
  fi

  if [[ $(ls /usr/local/bin/kubelet) ]] ; then
```

```
    current_version=$(/usr/local/bin/kubelet --version | awk -
F"Kubernetes v" '{ print $2 }')
    if [ "$current_version" -eq "#{node['kubernetes']['version']}" ];
then
        exit
    fi
  fi
  rm -rf kubernetes/
  wget --max-redirect 255
https://github.com/GoogleCloudPlatform/kubernetes/releases/download/v#
{node['kubernetes']['version']}/kubernetes.tar.gz -O kubernetes-#
{node['kubernetes']['version']}.tar.gz
  tar zxvf kubernetes-#{node['kubernetes']['version']}.tar.gz
  cd kubernetes/server
  tar zxvf kubernetes-server-linux-amd64.tar.gz
  EOH
end
```

In this recipe, the value of the Kubernetes' version will be taken from custom JSON. We may specify the latest version of Kubernetes and enjoy the new features without modifying the recipe. Two nested `if` conditions are used to validate whether the Kubernetes binary file is deployed and updated to the version we requested. One is for the master and the other is for the node; if the condition is satisfied, package downloading will be ignored. The main Kubernetes master installation is written in the recipe `kubernetes-master-setup.rb`:

```
$ cat ./kubernetes/recipes/kubernetes-master-setup.rb
include_recipe 'kubernetes::kubernetes'

bash "master-file-copy" do
    user 'root'
    cwd '/tmp/kubernetes/server/kubernetes/server/bin'
    code <<-EOH
    if [[ $(ls /usr/local/bin/kubectl) ]]; then
        current_version=$(/usr/local/bin/kubectl version | awk 'NR==1'
| awk -F":\"v" '{ print $2 }' | awk -F"\"," '{ print $1 }')
        if [ "$current_version" -eq "#{node['kubernetes']['version']}"
]; then
            exit
        fi
    fi
    cp kubectl kube-apiserver kube-scheduler kube-controller-manager
kube-proxy /usr/local/bin/
    EOH
end

directory '/etc/kubernetes' do
    owner 'root'
    group 'root'
    mode '0755'
    subscribes :create, "bash[master-file-copy]", :immediately
    action :nothing
end

etcd_endpoint="http://#{node['etcd']['elb_url']}:80"

template "/etc/init.d/kubernetes-master" do
  mode "0755"
  owner "root"
  source "kubernetes-master.erb"
```

```
  variables({
    :etcd_server => etcd_endpoint,
    :cluster_cidr => node['kubernetes']['cluster_cidr']
  })
  subscribes :create, "bash[master-file-copy]", :immediately
    action :nothing
end
```

The first line of `kubernetes-master-setup.rb` is the solution to set the dependency within the same event stage. The resource type `include_recipe` requires you to execute the recipe `kubernetes::kubernetes` first. Then, it is able to copy the necessary binary file if the process does not exist in the version verifying condition, and next, prepare the suitable configuration file and directory for service.

Installing flanneld on the master node is an optional deployment. If so, on the Kubernetes master, we can access containers laid on flanneld:

```
$ cat ./kubernetes/recipes/flanneld.rb
bash 'install_flannel' do
  user 'root'
  cwd '/tmp'
  code <<-EOH
  if [ ! -f /usr/local/bin/flanneld ]; then
    wget --max-redirect 255
https://github.com/coreos/flannel/releases/download/v0.5.2/flannel-
0.5.2-linux-amd64.tar.gz
    tar zxvf flannel-0.5.2-linux-amd64.tar.gz
    cd flannel-0.5.2
    cp flanneld /usr/local/bin
    cp mk-docker-opts.sh /opt/
  fi
  EOH
end

template "/etc/init.d/flanneld" do
  mode "0755"
  owner "root"
  source "flanneld.erb"
  variables :elb_url => node['etcd']['elb_url']
  notifies :disable, 'service[flanneld]', :delayed
end

service "flanneld" do
  action :nothing
end
```

Especially, we will move a script file of flanneld to a specific location. This file helps to arrange the flanneld-defined network. Therefore, Docker will be based on the setting and limit its containers in the specific IP range. For the template, the value of etcd endpoints is an input parameter of the recipe. The recipe would inform the template to put the etcd ELB URL as an etcd endpoint:

```
$ cat ./kubernetes/templates/default/flanneld.erb
:
//above lines are ignored
```

```
start() {
  # Start daemon.
  echo -n $"Starting $prog: "
  daemon $prog \
    --etcd-endpoints=http://<%= @elb_url %> -ip-masq=true \
    > /var/log/flanneld.log 2>&1 &
  RETVAL=$?
  echo
  [ $RETVAL -eq 0 ] && touch $lockfile
  return $RETVAL
}
:
```

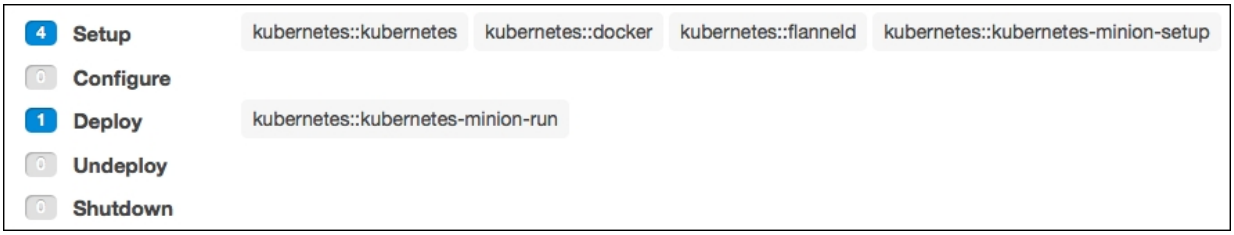
Finally, it is good for us to look at the recipe that starts the service:

```
$ cat ./kubernetes/recipes/kubernetes-master-run.rb
service "flanneld" do
  action :start
end

service "kubernetes-master" do
  action :start
end
```

It is straightforward to start these two independent daemons in this recipe.

Recipes for the Kubernetes node

With the previous experience, you now can easily understand the deployment of custom recipes and Chef functions. Then, we will go further to look at the recipes for the Kubernetes node layer:



Besides flanneld, we have to install Docker for running containers. However, an additional recipe `kubernetes::docker` is put at the **Setup** stage.

```
$ cat ./kubernetes/recipes/docker.rb
package "docker" do
  action :install
end

package "bridge-utils" do
  action :install
end

service "docker" do
  action :disable
```

```
end

template "/etc/sysconfig/docker" do
    mode "0644"
    owner "root"
    source "docker.erb"
end
```

We will install the necessary packages `docker` and `bridge-utils` in this recipe. But keep the Docker service stopped, since there is a service starting dependency:

```
$ cat ./kubernetes/templates/default/docker.erb
# /etc/sysconfig/docker
#
# Other arguments to pass to the docker daemon process
# These will be parsed by the sysv initscript and appended
# to the arguments list passed to docker -d

. /opt/mk-docker-opts.sh
. /run/docker_opts.env

INSECURE_REGISTRY="<YOUR_DOCKER_PRIVATE_REGISTRY>"

other_args="${DOCKER_OPTS} --insecure-registry ${INSECURE_REGISTRY}"
DOCKER_CERT_PATH=/etc/docker

# Location used for temporary files, such as those created by
# docker load and build operations. Default is /var/lib/docker/tmp
# Can be overriden by setting the following environment variable.
# DOCKER_TMPDIR=/var/tmp
```

The preceding template is called using `docker.rb`. Although there are no input parameters, it is worth mentioning that the script from flanneld will be triggered to run. It will generate the network settings for Docker and put it as the file `/run/docker_opts.env`.

Next, you will find that the node setup recipe is similar to the master one. We copy binaries, setup configuration files and directories, and keep the node service stopped:

```
$ cat ./kubernetes/recipes/kubernetes-node-setup.rb
include_recipe 'kubernetes::kubernetes'

bash "node-file-copy" do
    user 'root'
    cwd '/tmp/kubernetes/server/kubernetes/server/bin'
    code <<-EOH
    if [[ $(ls /usr/local/bin/kubelet) ]]; then
        current_version=$(/usr/local/bin/kubelet --version | awk -
F"Kubernetes v" '{ print $2 }')
        if [ "$current_version" -eq "#{node['kubernetes']['version']}"
]; then
            exit
        fi
    fi
    cp kubelet kube-proxy /usr/local/bin/
    EOH
end
```

```
directory '/var/lib/kubelet' do
    owner 'root'
    group 'root'
    mode '0755'
    subscribes :create, "bash[node-file-copy]", :immediately
    action :nothing
end

directory '/etc/kubernetes' do
    owner 'root'
    group 'root'
    mode '0755'
    subscribes :create, "bash[node-file-copy]", :immediately
    action :nothing
end

template "/etc/init.d/kubernetes-node" do
  mode "0755"
  owner "root"
  source "kubernetes-node.erb"
  variables :master_url => node['kubernetes']['master_url']
  subscribes :create, "bash[node-file-copy]", :immediately
  notifies :disable, 'service[kubernetes-node]', :delayed
   action :nothing
end

service "kubernetes-node" do
  action :nothing
end
```

On the other hand, the deploying recipe of node has more functions:

```
$ cat ./kubernetes/recipes/kubernetes-node-run.rb
service "flanneld" do
  action :start
  notifies :run, 'bash[wait_flanneld]', :delayed
end

bash 'wait_flanneld' do
  user 'root'
  cwd '/tmp'
  code <<-EOH
  tries=0
        while [ ! -f /run/flannel/subnet.env -a $tries -lt 10 ]; do
            sleep 1
            tries=$((tries + 1))
        done
  EOH

  action :nothing
  notifies :start, 'service[docker]', :delayed
end
service "docker" do
  action :nothing
  notifies :start, 'service[kubernetes-node]', :delayed
end

service "kubernetes-node" do
  action :nothing
end
```

Because of the dependence, flanneld should be started first and then Docker can be run according to the overlay network. Node service depends on running Docker, so it is the last service that needs to be started.

Starting the instances

Eventually, you have all the recipes ready to deploy a Kubernetes cluster. It is time to boot up some instances! Make sure that etcd is the earliest running instance. At that time, the Kubernetes master layer can run the master, which requires the datastore for resource information. After the master node is ready as well, create as many nodes as you want!

See also

In this recipe, you learned how to create your Kubernetes system automatically. Also, look at the following recipes:

- The *Building datastore*, *Creating an overlay network*, *Configuring master* and *Configuring nodes* recipes in Chapter 1, *Building Your Own Kubernetes*
- The *Clustering etcd* recipe in Chapter 4, *Building a High Availability Cluster*
- *Building the Kubernetes infrastructure in AWS*
- *Managing applications using AWS OpsWorks*
- *Using AWS CloudFormation for fast provisioning*
- The *Authentication and authorization* recipe in Chapter 7, *Advanced Cluster Administration*

Using AWS CloudFormation for fast provisioning

AWS CloudFormation is a service to make AWS resource creation easy. A simple JSON format text file could give you the power to create application infrastructure with just a few clicks. System administrators and developers can create, update, and manage their AWS resources easily without worrying about human error. In this section, we will leverage the content of the previous sections in this chapter and use CloudFormation to create them and launch instances with the Kubernetes setting automatically.

Getting ready

The unit of CloudFormation is a stack. One stack is created by one CloudFormation template, which is a text file listing AWS resources in the JSON format. Before we launch a CloudFormation stack using the CloudFormation template in the AWS console, let's get a deeper understanding of the tab names on the CloudFormation console:

Tab name	Description
`Overview`	Stack profile overview. Name, status and description are listed here
`Output`	The output fields of this stack
`Resources`	The resources listed in this stack

Tab name	Description
Events	The events when doing operations in this stack
Template	Text file in JSON format
Parameters	The input parameters of this stack
Tags	AWS tags for the resources
Stack Policy	Stack policy to use during update. This can prevent you from removing or updating resources accidentally

One CloudFormation template contains many sections; the descriptions are put in the following sample template:

```
{
   "AWSTemplateFormatVersion":"AWS CloudFormation
templateversion date",
   "Description":"stack description",
   "Metadata":{
    # put additional information for this template
   },
   "Parameters":{
    # user-specified the input of your template

   },
   "Mappings":{
    # using for define conditional parameter
values and use it in the template
   },
   "Conditions":{
    # use to define whether certain resources are
created, configured in a certain condition.
```

```
    },
    "Resources":{
     # major section in the template, use to create
and configure AWS resources
    },
    "Outputs":{
     # user-specified output
    }
}
```

We will use these three major sections:

- Parameters
- Resources
- Outputs

`Parameters` are the variable you might want to input when creating the stack, `Resources` are a major section for declaring AWS resource settings, and `Outputs` are the section you might want to expose to the CloudFormation UI so that it's easy to find the output information from a resource when a template is deployed.

Intrinsic Functions are built-in functions of AWS CloudFormation. They give you the power to link your resources together. It is a common use case that you need to link several resources together, but they'll know each other until runtime. In this case, the intrinsic function could be a perfect match to resolve this. Several intrinsic functions are provided in CloudFormation. In this case, we will use `Fn::GetAtt`, `Fn::GetAZs` and `Ref`.

The following table has their descriptions:

Functions	Description	Usage
`Fn::GetAtt`	Retrieve a value of an attribute from a resource	`{"Fn::GetAtt" : [ "logicalNameOfResource", "attributeName" ]}`

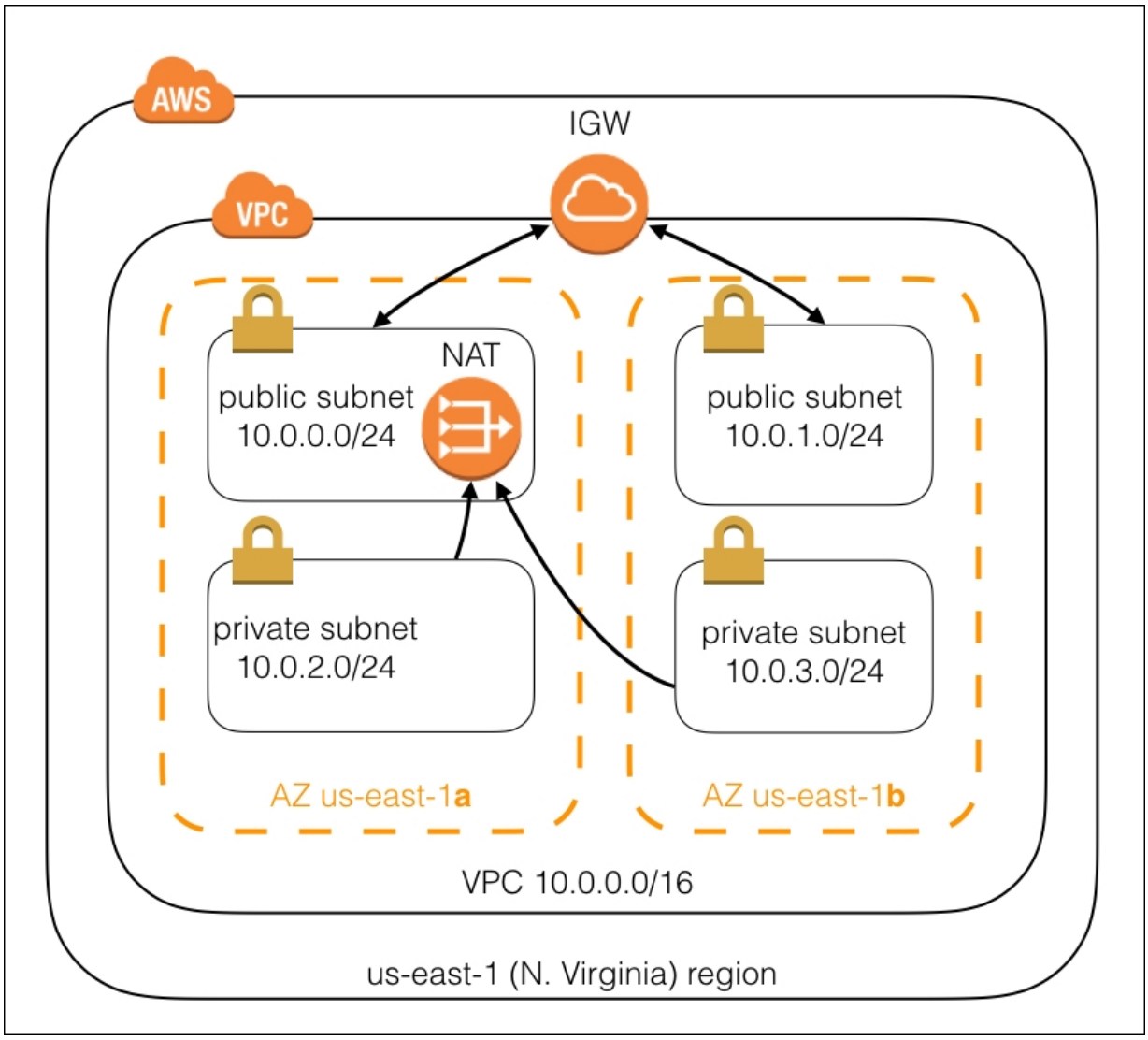
Functions	Description	Usage
`Fn::GetAZs`	Return a list of AZs for the region	`{"Fn::GetAZs" : "us-east-1"}`
`Fn::Select`	Select a value from a list	`{ "Fn::Select" : [ index, listOfObjects ]}`
`Ref`	Return a value from a logical name or parameter	`{"Ref" : "logicalName"}`

How to do it…

Instead of launching a bit template with a thousand lines, we'll split it into two: one is for network resources-only, another one is application-only. Both the templates are available on our GitHub repository via https://github.com/kubernetes-cookbook/cloudformation.

Creating a network infrastructure

Let's review the following infrastructure listed in the *Building the Kubernetes infrastructure in AWS* section. We will create one VPC with `10.0.0.0/16` with two public subnets and two private subnets in it. Besides these, we will create one Internet Gateway and add related route table rules to a public subnet in order to route the traffic to the outside world. We will also create a NAT Gateway, which is located in the public subnet with one Elastic IP, to ensure a private subnet can get access to the Internet:



How do we do that? At the beginning of the template, we'll define two parameters: one is `Prefix` and another is `CIDRPrefix`. `Prefix` is a prefix used to name the resource we're going to create. `CIDRPrefix` is two sections of an IP address that we'd like to create; the default is `10.0`. We will also set the length constraint to it:

```
"Parameters":{
      "Prefix":{
         "Description":"Prefix of resources",
         "Type":"String",
         "Default":"KubernetesSample",
         "MinLength":"1",
         "MaxLength":"24",
         "ConstraintDescription":"Length is too
```

```
long"
      },
      "CIDRPrefix":{
         "Description":"Network cidr prefix",
         "Type":"String",
         "Default":"10.0",
         "MinLength":"1",
         "MaxLength":"8",
         "ConstraintDescription":"Length is too
long"
      }
   }
```

Then, we will start describing the `Resources` section. For detailed resource types and attributes, we recommend you visit the AWS Documentation via `http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html`:

```
      "VPC":{
         "Type":"AWS::EC2::VPC",
         "Properties":{
            "CidrBlock":{
               "Fn::Join":[
                  ".",
                  [
                     {
                        "Ref":"CIDRPrefix"
                     },
                     "0.0/16"
                  ]
               ]
            },
            "EnableDnsHostnames":"true",
            "Tags":[
               {
                  "Key":"Name",
                  "Value":{
                     "Fn::Join":[
                        ".",
                        [
                           {
                              "Ref":"Prefix"
                           },
                           "vpc"
```

```
                        ]
                    ]
                }
            },
            {
                "Key":"EnvName",
                "Value":{
                    "Ref":"Prefix"
                }
            }
        ]
    }
}
```

Here, we'll create one resource with the logical name `VPC` and type `AWS::EC2::VPC`. Please note that the logical name is important and it cannot be duplicated in one template. You could use `{"Ref": "VPC"}` in any other resource in this template to refer to `VPCId`. The name of VPC will be `$Prefix.vpc` with CIDR `$CIDRPrefix.0.0/16`. The following image is that of a created VPC:



Next, we'll create the first public subnet with CIDR `$CIDRPrefix.0.0/24`. Note that `{Fn::GetAZs:""}` will return a list of all the available AZs. We'll use `Fn::Select` to select the first element with index 0:

```
"SubnetPublicA":{
    "Type":"AWS::EC2::Subnet",
    "Properties":{
        "VpcId":{
            "Ref":"VPC"
        },
        "CidrBlock":{
            "Fn::Join":[
                ".",
                [
                    {
                        "Ref":"CIDRPrefix"
                    },
                    "0.0/24"
                ]
```

```
            ]
         },
         "AvailabilityZone":{
            "Fn::Select":[
               "0",
               {
                  "Fn::GetAZs":""
               }
            ]
         },
         "Tags":[
            {
               "Key":"Name",
               "Value":{
                  "Fn::Join":[
                     ".",
                     [
                        {
                           "Ref":"Prefix"
                        },
                        "public",
                        "subnet",
                        "A"
                     ]
                  ]
               }
            },
            {
               "Key":"EnvName",
               "Value":{
                  "Ref":"Prefix"
               }
            }
         ]
      }
   }
```

The second public subnet and two private subnets are the same as the first one just with a different CIDR `$CIDRPrefix.1.0/24`. The difference between public and private subnets are whether they're Internet reachable or not. Typically, an instance in a public subnet will have a public IP or an Elastic IP with it that is Internet reachable. However, a private subnet cannot be reachable from the Internet, except using a bastion host or via VPN. The difference in the AWS setting is the routes in route tables. In order to let your instances

communicate with the Internet, we should create an Internet Gateway to a public subnet and a NAT Gateway to a private subnet:

```
"InternetGateway":{
   "Type":"AWS::EC2::InternetGateway",
   "Properties":{
      "Tags":[
         {
            "Key":"Stack",
            "Value":{
               "Ref":"AWS::StackId"
            }
         },
         {
            "Key":"Name",
            "Value":{
               "Fn::Join":[
                  ".",
                  [
                     {
                        "Ref":"Prefix"
                     },
                     "vpc",
                     "igw"
                  ]
               ]
            }
         },
         {
            "Key":"EnvName",
            "Value":{
               "Ref":"Prefix"
            }
         }
      ]
   }
},
"GatewayAttachment":{
   "Type":"AWS::EC2::VPCGatewayAttachment",
   "Properties":{
      "VpcId":{
         "Ref":"VPC"
      },
      "InternetGatewayId":{
         "Ref":"InternetGateway"
      }
```

```
    }
}
```

We will declare one Internet Gateway with the name `$Prefix.vpc.igw` and the logical name `InternetGateway`; we will also attach it to VPC. Then, let's create `NatGateway`. `NatGateway` needs one EIP by default, so we'll create it first and use the `DependsOn` function to tell CloudFormation that the `NatGateway` resource must be created after `NatGatewayEIP`. Note that there is `AllocationId` in the properties of `NatGateway` rather than the Gateway ID. We'll use the intrinsic function `Fn::GetAtt` to get the attribute `AllocationId` from the resource `NatGatewayEIP`:

```
"NatGatewayEIP":{
   "Type":"AWS::EC2::EIP",
   "DependsOn":"GatewayAttachment",
   "Properties":{
      "Domain":"vpc"
   }
},
"NatGateway":{
   "Type":"AWS::EC2::NatGateway",
   "DependsOn":"NatGatewayEIP",
   "Properties":{
      "AllocationId":{
         "Fn::GetAtt":[
            "NatGatewayEIP",
            "AllocationId"
         ]
      },
      "SubnetId":{
         "Ref":"SubnetPublicA"
      }
   }
}
```

Time to create a route table for public subnets:

```
"RouteTableInternet":{
   "Type":"AWS::EC2::RouteTable",
   "Properties":{
      "VpcId":{
         "Ref":"VPC"
```

```
            },
            "Tags":[
               {
                  "Key":"Stack",
                  "Value":{
                     "Ref":"AWS::StackId"
                  }
               },
               {
                  "Key":"Name",
                  "Value":{
                     "Fn::Join":[
                        ".",
                        [
                           {
                              "Ref":"Prefix"
                           },
                           "internet",
                           "routetable"
                        ]
                     ]
                  }
               },
               {
                  "Key":"EnvName",
                  "Value":{
                     "Ref":"Prefix"
                  }
               }
            ]
         }
      }
```

What about private subnets? You could use the same declaration; just change the logical name to `RouteTableNat`. After creating a route table, let's create the routes:

```
      "RouteInternet":{
         "Type":"AWS::EC2::Route",
         "DependsOn":"GatewayAttachment",
         "Properties":{
            "RouteTableId":{
               "Ref":"RouteTableInternet"
            },
            "DestinationCidrBlock":"0.0.0.0/0",
            "GatewayId":{
```

```
                "Ref":"InternetGateway"
            }
        }
    }
```

This route is for the route table of a public subnet. It will relocate to the `RouteTableInternet` table and route the packets to `InternetGatway` if the destination CIDR is `0.0.0.0/0`. Let's take a look at a private subnet route:

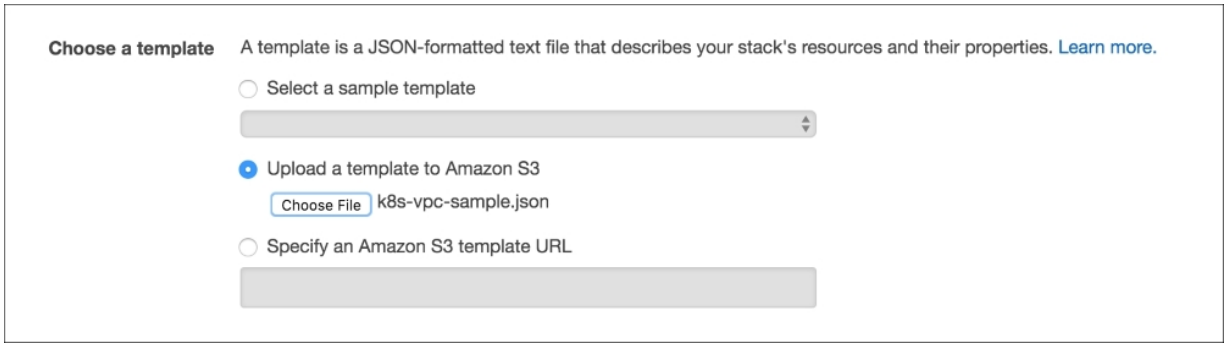
```
    "RouteNat":{
        "Type":"AWS::EC2::Route",
        "DependsOn":"RouteTableNat",
        "Properties":{
            "RouteTableId":{
                "Ref":"RouteTableNat"
            },
            "DestinationCidrBlock":"0.0.0.0/0",
            "NatGatewayId":{
                "Ref":"NatGateway"
            }
        }
    }
```

It is pretty much the same with `RouteInternet` but route the packets to `NatGateway` if there are any, to `0.0.0.0/0`. Wait, what's the relation between subnet and a route table? We didn't see any declaration indicate the rules in a certain subnet. We have to use `SubnetRouteTableAssociation` to define their relation. The following examples define both public subnet and private subnet; you might also add a second public/private subnet by copying them:

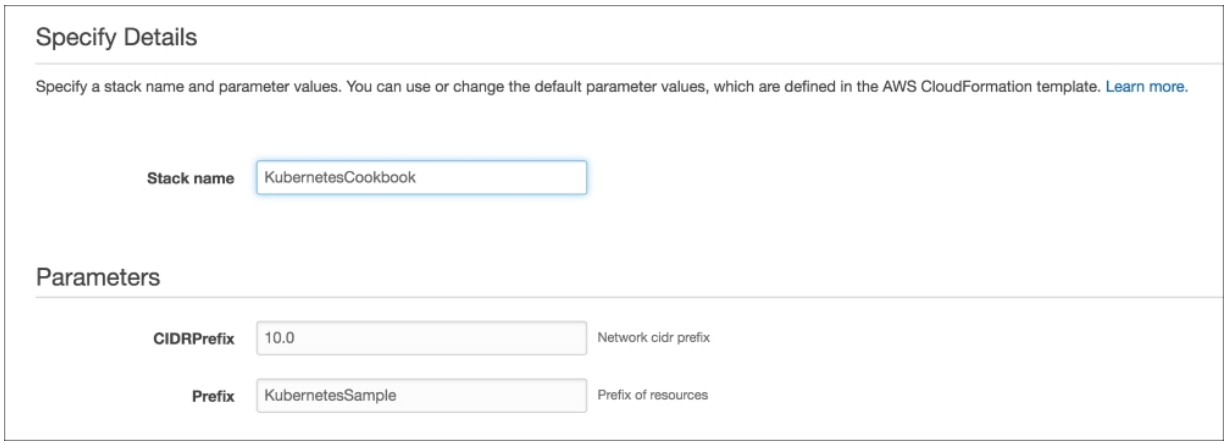
```
        "SubnetRouteTableInternetAssociationA":{

    "Type":"AWS::EC2::SubnetRouteTableAssociation",
            "Properties":{
                "SubnetId":{
                    "Ref":"SubnetPublicA"
                },
                "RouteTableId":{
                    "Ref":"RouteTableInternet"
                }
```

```
            }
        },
        "SubnetRouteTableNatAssociationA":{

"Type":"AWS::EC2::SubnetRouteTableAssociation",
            "Properties":{
                "SubnetId":{
                    "Ref":"SubnetPrivateA"
                },
                "RouteTableId":{
                    "Ref":"RouteTableNat"
                }
            }
        }
```

We're done for the network infrastructure. Then, let's launch it from the AWS console. First, just click and launch a stack and select the VPC sample template.



Click on **next**; you will see the parameters' pages. It has its own default value, but you could change it at the creation/update time of the stack.



After you click on **finish**, CloudFormation will start creating the resources you claim on the template. It will return Status as **CREATE_COMPLETE** after completion.

Creating OpsWorks for application management

For application management, we'll leverage OpsWorks, which is an application lifecycle management in AWS. Please refer to the previous two sections to know more about OpsWorks and Chef. Here, we'll describe how to automate creating the OpsWorks stack and related resources.

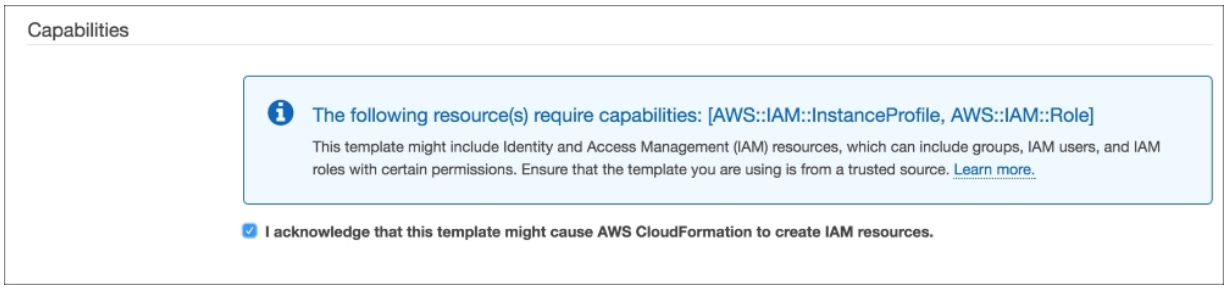
We'll have eight parameters here. Add `K8sMasterBaAccount`, `K8sMasterBaPassword`, and `EtcdBaPassword` as the basic authentication for Kubernetes master and etcd. We will also put the VPC ID and the private subnet ID here as the input, which are created in the previous sample. As parameters, we could use the type `AWS::EC2::VPC::Id` as a drop-down list in the UI. Please refer to the supported type in the AWS Documentation via `http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html`:

```
"Parameters":{
      "Prefix":{
         "Description":"Prefix of resources",
         "Type":"String",
```

```
            "Default":"KubernetesSample",
            "MinLength":"1",
            "MaxLength":"24",
            "ConstraintDescription":"Length is too
long"
         },
         "PrivateNetworkCIDR":{
            "Default":"192.168.0.0/16",
            "Description":"Desired Private Network
CIDR or Flanneld (must not overrap VPC CIDR)",
            "Type":"String",
            "MinLength":"9",

"AllowedPattern":"\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\
.\\d{1,3}/\\d{1,2}",

"ConstraintDescription":"PrivateNetworkCIDR must
be IPv4 format"
         },
         "VPCId":{
            "Description":"VPC Id",
            "Type":"AWS::EC2::VPC::Id"
         },
         "SubnetPrivateIdA":{
            "Description":"Private SubnetA",
            "Type":"AWS::EC2::Subnet::Id"
         },
         "SubnetPrivateIdB":{
            "Description":"Private SubnetB",
            "Default":"subnet-9007ecc9",
            "Type":"AWS::EC2::Subnet::Id"
         },
         "K8sMasterBaAccount":{
            "Default":"admin",
            "Description":"The account of basic
authentication for k8s Master",
            "Type":"String",
            "MinLength":"1",
            "MaxLength":"75",
            "AllowedPattern":"[a-zA-Z0-9]*",
            "ConstraintDescription":"Account and
Password should follow Base64 pattern"
         },
         "K8sMasterBaPassword":{
            "Default":"Passw0rd",
            "Description":"The password of basic
authentication for k8s Master",
            "Type":"String",
```

```
            "MinLength":"1",
            "MaxLength":"75",
            "NoEcho":"true",
            "AllowedPattern":"[a-zA-Z0-9]*",
            "ConstraintDescription":"Account and
Password should follow Base64 pattern"
         },
         "EtcdBaPassword":{
            "Default":"Passw0rd",
            "Description":"The password of basic
authentication for Etcd",
            "Type":"String",
            "MinLength":"1",
            "MaxLength":"71",
            "NoEcho":"true",
            "AllowedPattern":"[a-zA-Z0-9]*",
            "ConstraintDescription":"Password should
follow Base64 pattern"
         }
      }
```

Before we get started with the OpsWorks stack, we need to create two IAM roles for it. One is a service role, which is used to launch instances, attaching ELB, and so on. Another is an instance role, which is to define the permission for what your OpsWorks instances can perform, to access the AWS resources. Here, we won't access any AWS resources from EC2, so we could just create a skeleton. Please note that you'll need to have IAM permission when you launch CloudFormation with IAM creation. Click on the following checkbox when launching the stack:



In the `SecurityGroup` section, we will define each ingress and egress to a set of machines. We'll take the Kubernetes master as an example. Since we put ELB in front of the master and ELB to retain

the flexibility for future HA settings. Using ELB, we'll need to create a security group to ELB and point the ingress of the Kubernetes master that can be in touch with `8080` and `6443` from the ELB security group. Following is the example of the security group for the Kubernetes master; it opens `80` and `8080` to the outside world:

```
"SecurityGroupELBKubMaster":{
     "Type":"AWS::EC2::SecurityGroup",
     "Properties":{
        "GroupDescription":{
           "Ref":"Prefix"
        },
        "SecurityGroupIngress":[
           {
              "IpProtocol":"tcp",
              "FromPort":"80",
              "ToPort":"80",
              "CidrIp":"0.0.0.0/0"
           },
           {
              "IpProtocol":"tcp",
              "FromPort":"8080",
              "ToPort":"8080",
              "SourceSecurityGroupId":{
                 "Ref":"SecurityGroupKubNode"
              }
           }
        ],
        "VpcId":{
           "Ref":"VPCId"
        },
        "Tags":[
           {
              "Key":"Application",
              "Value":{
                 "Ref":"AWS::StackId"
              }
           },
           {
              "Key":"Name",
              "Value":{
                 "Fn::Join":[
                    "-",
                    [
                       {
                          "Ref":"Prefix"
```

```
                        },
                        "SGElbKubMaster"
                    ]
                ]
            }
        }
    ]
}
},
```

Here is the example of the Kubernetes master instance set. It allows you to receive traffic from `8080` and `6443` from its ELB. We will open the SSH port to use the `kubectl` command:

```
"SecurityGroupKubMaster":{
    "Type":"AWS::EC2::SecurityGroup",
    "Properties":{
       "GroupDescription":{
          "Ref":"Prefix"
       },
       "SecurityGroupIngress":[
          {
             "IpProtocol":"tcp",
             "FromPort":"22",
             "ToPort":"22",
             "CidrIp":"0.0.0.0/0"
          },
          {
             "IpProtocol":"tcp",
             "FromPort":"8080",
             "ToPort":"8080",
             "SourceSecurityGroupId":{

"Ref":"SecurityGroupELBKubMaster"
             }
          },
          {
             "IpProtocol":"tcp",
             "FromPort":"6443",
             "ToPort":"6443",
             "SourceSecurityGroupId":{

"Ref":"SecurityGroupELBKubMaster"
             }
          }
       ],
```

```
            "VpcId":{
               "Ref":"VPCId"
            },
            "Tags":[
               {
                  "Key":"Application",
                  "Value":{
                     "Ref":"AWS::StackId"
                  }
               },
               {
                  "Key":"Name",
                  "Value":{
                     "Fn::Join":[
                        "-",
                        [
                           {
                              "Ref":"Prefix"
                           },
                           "SG-KubMaster"
                        ]
                     ]
                  }
               }
            ]
         }
      }
```

Please refer to the examples from the book about the security group setting of etcd and node. Next, we'll start creating the OpsWorks stack. `CustomJson` acts as the input of the Chef recipe. If there is anything that Chef doesn't know at the beginning, you will have to pass the parameters into `CustomJson`:

```
      "OpsWorksStack":{
         "Type":"AWS::OpsWorks::Stack",
         "Properties":{
            "DefaultInstanceProfileArn":{
               "Fn::GetAtt":[
                  "RootInstanceProfile",
                  "Arn"
               ]
            },
            "CustomJson":{
               "kubernetes":{
```

```
                    "cluster_cidr":{
                        "Ref":"PrivateNetworkCIDR"
                    },
                    "version":"1.1.3",
                    "master_url":{
                        "Fn::GetAtt":[
                            "ELBKubMaster",
                            "DNSName"
                        ]
                    }
                },
                "ba":{
                    "account":{
                        "Ref":"K8sMasterBaAccount"
                    },
                    "password":{
                        "Ref":"K8sMasterBaPassword"
                    },
                    "uid":1234
                },
                "etcd":{
                    "password":{
                        "Ref":"EtcdBaPassword"
                    },
                    "elb_url":{
                        "Fn::GetAtt":[
                            "ELBEtcd",
                            "DNSName"
                        ]
                    }
                },
                "opsworks_berkshelf":{
                    "debug":true
                }
            },
            "ConfigurationManager":{
                "Name":"Chef",
                "Version":"11.10"
            },
            "UseCustomCookbooks":"true",
            "UseOpsworksSecurityGroups":"false",
            "CustomCookbooksSource":{
                "Type":"git",

"Url":"https://github.com/kubernetes-
cookbook/opsworks-recipes.git"
            },
            "ChefConfiguration":{
```

```
                "ManageBerkshelf":"true"
             },
             "DefaultOs":"Red Hat Enterprise Linux
7",
             "DefaultSubnetId":{
                "Ref":"SubnetPrivateIdA"
             },
             "Name":{
                "Ref":"Prefix"
             },
             "ServiceRoleArn":{
                "Fn::GetAtt":[
                   "OpsWorksServiceRole",
                   "Arn"
                ]
             },
             "VpcId":{
                "Ref":"VPCId"
             }
          }
       },
```

After creating the stack, we can start creating each layer. Take the Kubernetes master as an example:

```
       "OpsWorksLayerKubMaster":{
          "Type":"AWS::OpsWorks::Layer",
          "Properties":{
             "Name":"Kubernetes Master",
             "Shortname":"kube-master",
             "AutoAssignElasticIps":"false",
             "AutoAssignPublicIps":"false",
             "CustomSecurityGroupIds":[
                {
                   "Ref":"SecurityGroupKubMaster"
                }
             ],
             "EnableAutoHealing":"false",
             "StackId":{
                "Ref":"OpsWorksStack"
             },
             "Type":"custom",
             "CustomRecipes":{
                "Setup":[
                   "kubernetes-rhel::flanneld",
                   "kubernetes-rhel::repo-setup",
```

```
                    "kubernetes-rhel::master-setup"
                ],
                "Deploy":[
                    "kubernetes-rhel::master-run"
                ]
            }
        }
    },
```

The run list of Chef in this layer is `["kubernetes-rhel::flanneld", "kubernetes-rhel::repo-setup", "kubernetes-rhel::master-setup"]` and `["kubernetes-rhel::master-run"]` at the deployment stage. For the run list of etcd, we'll use `["kubernetes-rhel::etcd", "kubernetes-rhel::etcd-auth"]` to perform etcd provisioning and authentication setting. For the Kubernetes nodes, we'll use `["kubernetes-rhel::flanneld", "kubernetes-rhel::docker-engine", "kubernetes-rhel::repo-setup", "kubernetes-rhel::node-setup"]` as a run list at the setup stage and `["kubernetes-rhel::node-run"]` at the deployment stage.

After setting up the layer, we can create ELB and attach it to the stack. The target of health check for the instance is `HTTP:8080/version`. It will then receive traffic from the port `80` and redirect it to the `6443` port in the master instances, and receive traffic from `8080` to the instance port `8080`:

```
    "ELBKubMaster":{
        "DependsOn":"SecurityGroupELBKubMaster",

"Type":"AWS::ElasticLoadBalancing::LoadBalancer",
        "Properties":{
            "LoadBalancerName":{
                "Fn::Join":[
                    "-",
                    [
                        {
                            "Ref":"Prefix"
                        },
                        "Kub"
                    ]
                ]
            },
```

```
            "Scheme":"internal",
            "Listeners":[
               {
                  "LoadBalancerPort":"80",
                  "InstancePort":"6443",
                  "Protocol":"HTTP",
                  "InstanceProtocol":"HTTPS"
               },
               {
                  "LoadBalancerPort":"8080",
                  "InstancePort":"8080",
                  "Protocol":"HTTP",
                  "InstanceProtocol":"HTTP"
               }
            ],
            "HealthCheck":{
               "Target":"HTTP:8080/version",
               "HealthyThreshold":"2",
               "UnhealthyThreshold":"10",
               "Interval":"10",
               "Timeout":"5"
            },
            "Subnets":[
               {
                  "Ref":"SubnetPrivateIdA"
               },
               {
                  "Ref":"SubnetPrivateIdB"
               }
            ],
            "SecurityGroups":[
               {
                  "Fn::GetAtt":[
                     "SecurityGroupELBKubMaster",
                     "GroupId"
                  ]
               }
            ]
         }
      }
```

After creating the master ELB, let's attach it to the OpsWorks stack:

```
      "OpsWorksELBAttachKubMaster":{

   "Type":"AWS::OpsWorks::ElasticLoadBalancerAttachme
```

```
nt",
          "Properties":{
             "ElasticLoadBalancerName":{
                "Ref":"ELBKubMaster"
             },
             "LayerId":{
                "Ref":"OpsWorksLayerKubMaster"
             }
          }
       }
```

That's it! The ELB of etcd is the same setting, but listen to `HTTP:4001/version` as a health check and redirect `80` traffic from the outside to the instance port `4001`. For a detailed example, please refer to our code reference. After launching the second sample template, you should be able to see the OpsWorks stacks, layers, security groups, IAM, and ELBs. If you want to launch by default with CloudFormation, just add the resource type with `AWS::OpsWorks::Instance`, specify the spec, and you are all set.

See also

In this recipe, we got an understanding on how to write and deploy an AWS CloudFormation template. Please check out the following recipes as well:

- The *Exploring architecture* recipe in Chapter 1, *Building Your Own Kubernetes*
- *Building the Kubernetes infrastructure in AWS*
- *Managing applications using AWS OpsWorks*
- *Auto-deploying Kubernetes through Chef recipes*

Chapter 7. Advanced Cluster Administration

In this chapter, we will cover:

- Advanced settings in kubeconfig
- Setting resource in nodes
- Playing with WebUI
- Working with a RESTful API
- Authentication and authorization

Introduction

We will go through advanced topics on administration in this chapter. First, you will learn how to use kubeconfig to manage different clusters. Then, we will work on computing resources in nodes. Kubernetes provides a friendly user interface to illustrate the current status of resources, such as the replication controller, nodes, and pods. You will learn how to build and administrate it.

Next, you will learn how to work with the RESTful API that Kubernetes exposes. It will be a handy way to integrate with other systems. Finally, we want to build a secure cluster; the last section will go through how to set up authentication and authorization in Kubernetes.

Advanced settings in kubeconfig

kubeconfig is a configuration file to manage cluster, context, and authentication settings in Kubernetes. Using the kubeconfig file, we are able to set different cluster credentials, users, and namespaces to switch between clusters or contexts within a cluster. It can be configured via the command line using the `kubectl config` subcommand or a configuration file directly. In this section, we'll describe how to use `kubectl config` to manipulate kubeconfig and how to input a kubeconfig file directly.

Getting ready

Before you start to modify kubeconfig, you should clearly know what your security policies are. Using `kubectl config view`, you can check your current settings:

```
// check current kubeconfig file
# kubectl config view
apiVersion: v1
clusters: []
contexts: []
current-context: ""
kind: Config
preferences: {}
users: []
```

We can see currently we do not have any specific settings in kubeconfig.

How to do it…

Assume we have two clusters, one is under localhost `http://localhost:8080` and another is in the remote `http://remotehost:8080` named `remotehost`. In the example, we'll use localhost as the main console to switch the cluster via context changes. We then run different number of nginx into both the clusters and make sure the pods are all running:

```
// in localhost cluster
# kubectl run localnginx --image=nginx --
replicas=2 --port=80
replicationcontroller "localnginx" created
// check pods are running
# kubectl get pods
NAME                READY     STATUS
RESTARTS   AGE
localnginx-1blru    1/1       Running        0
1m
localnginx-p6cyo    1/1       Running        0
1m

// in remotehost cluster
# kubectl run remotenginx --image=nginx --
replicas=4 --port=80
replicationcontroller "remotenginx" created
// check pods are running
# kubectl get pods
NAME                 READY     STATUS    RESTARTS
AGE
remotenginx-6wz5c    1/1       Running   0
1m
remotenginx-7v5in    1/1       Running   0
1m
remotenginx-c7go6    1/1       Running   0
1m
remotenginx-r1mf6    1/1       Running   0
1m
```

Setting a new credential

First, we will set up two credentials for each cluster. Use `kubectl config set-credentials <nickname>` for adding credential into kubeconfig. There are different authentication methods supported in

Kubernetes. We could use a password, client-certificate, or token. In the example, we'll use HTTP basic authentication for simplifying the scenario. Kubernetes also supports client certificate and token authentications. For more information, please refer to the kubeconfig set-credential page: http://kubernetes.io/docs/user-guide/kubectl/kubectl_config_set-credentials:

```
// in localhost cluster, add a user `userlocal`
with nickname localhost/myself
#  kubectl config set-credentials localhost/myself
--username=userlocal --password=passwordlocal
user "localhost/myself" set.

// in localhost cluster, add a user `userremote`
with nickname remotehost/myself
#  kubectl config set-credentials
remotehost/myself --username=userremote --
password=passwordremote
user "remotehost/myself" set.
```

Let's check out the current view:

```
# kubectl config view
apiVersion: v1
clusters: []
contexts: []
current-context: ""
kind: Config
preferences: {}
users:
- name: localhost/myself
  user:
    password: passwordlocal
    username: userlocal
- name: remotehost/myself
  user:
    password: passwordremote
    username: userremote
```

We can find currently that we have two sets of credentials with nicknames `localhost/myself` and `remotehost/myself`. Next, we'll set

the clusters into the management.

Setting a new cluster

To set a new cluster, we will need the `kubectl config set-cluster <nickname>` command. We will need the `--server` parameter for accessing clusters. Adding `-insecure-skip-tls-verify` will not check the server's certificate. If you are setting up a trusted server with HTTPS, you will need to replace `-insecure-skip-tls-verify` to `--certificate-authority=$PATH_OF_CERT --embed-certs=true`. For more information, check out the kubeconfig set-cluster page: http://kubernetes.io/docs/user-guide/kubectl/kubectl_config_set-cluster:

```
// in localhost cluster: add http://localhost:8080
as localhost
# kubectl config set-cluster localhost --insecure-
skip-tls-verify=true --
server=http://localhost:8080
cluster "localhost" set.

// in localhost cluster: add http://remote:8080 as
localhost
# kubectl config set-cluster remotehost --
insecure-skip-tls-verify=true --
server=http://remotehost:8080
cluster "remotehost" set.
```

Let's check out the current view now. The setting exactly reflects what we've set:

```
// check current view
# kubectl config view
apiVersion: v1
clusters:
- cluster:
    insecure-skip-tls-verify: true
    server: http://localhost:8080
  name: localhost
- cluster:
```

```
    insecure-skip-tls-verify: true
    server: http://remotehost:8080
  name: remotehost
contexts: []
current-context: ""
kind: Config
preferences: {}
users:
- name: localhost/myself
  user:
    password: passwordlocal
    username: userlocal
- name: remotehost/myself
  user:
    password: passwordremote
    username: userremote
```

Note that we do not associate anything between users and clusters yet. We will link them via context.

Setting and changing the current context

One context contains a cluster, namespace, and user. `kubectl` will use the specified user information and namespace to send requests to the cluster. To set up a context, we will use `kubectl config set-context <context nickname> --user=<user nickname> --namespace=<namespace> --cluster=< cluster nickname>` to create it:

```
// in localhost cluster: set a context named
default/localhost/myself for localhost cluster
# kubectl config set-context
default/localhost/myself --user=localhost/myself -
-namespace=default --cluster=localhost
context "default/localhost/myself" set.

// in localhost cluster: set a context named
default/remotehost/myself for remotehost cluster
# kubectl config set-context
default/remotehost/myself --user=remotehost/myself
--namespace=default --cluster=remotehost
context "default/remotehost/myself" set.
```

Let's check out the current view. We can see a list of contexts is in the contexts section now:

```
# kubectl config view
apiVersion: v1
clusters:
- cluster:
    insecure-skip-tls-verify: true
    server: http://localhost:8080
  name: localhost
- cluster:
    insecure-skip-tls-verify: true
    server: http://remotehost:8080
  name: remotehost
contexts:
- context:
    cluster: localhost
    namespace: default
    user: localhost/myself
  name: default/localhost/myself
- context:
    cluster: remotehost
    namespace: default
    user: remotehost/myself
  name: default/remotehost/myself
current-context: ""
kind: Config
preferences: {}
users:
- name: localhost/myself
  user:
    password: passwordlocal
    username: userlocal
- name: remotehost/myself
  user:
    password: passwordremote
    username: userremote
```

After creating contexts, let's start to switch context in order to manage different clusters. Here, we will use the command `kubectl config use-context <context nickname>`. We'll start from the localhost one first:

```
// in localhost cluster: use the context
default/localhost/myself
# kubectl config use-context
default/localhost/myself
switched to context "default/localhost/myself".
```

Let's list pods to see whether it is a localhost cluster:

```
// list the pods
# kubectl get pods
NAME                READY     STATUS
RESTARTS   AGE
localnginx-1blru    1/1       Running       0
1m
localnginx-p6cyo    1/1       Running       0
1m
```

Yes, it looks fine. How about if we switch to the context with the `remotehost` setting?

```
// in localhost cluster: switch to the context
default/remotehost/myself
# kubectl config use-context
default/remotehost/myself
switched to context "default/remotehost/myself".
Let's list the pods to make sure it's under the
remotehost context:
# kubectl get pods
NAME                 READY     STATUS    RESTARTS
AGE
remotenginx-6wz5c    1/1       Running   0
1m
remotenginx-7v5in    1/1       Running   0
1m
remotenginx-c7go6    1/1       Running   0
1m
remotenginx-r1mf6    1/1       Running   0
1m
```

All the operations we have done are in the localhost cluster. kubeconfig makes switching multiple clusters with multiple users easier.

Cleaning up kubeconfig

The kubeconfig file is stored in `$HOME/.kube/config`. If the file is deleted, the configuration is gone; if the file is restored to the directory, the configuration will be restored:

```
// clean up kubeconfig file
# rm -f ~/.kube/config

// check out current view
# kubectl config view
apiVersion: v1
clusters: []
contexts: []
current-context: ""
kind: Config
preferences: {}
users: []
```

See also

kubeconfig manages the setting of clusters, credentials, and namespaces. Check out the following recipes:

- The *Working with namespaces* recipe in Chapter 2, *Walking through Kubernetes Concepts*
- *Authentication and authorization*

Setting resource in nodes

Computing resource management is so important in any infrastructure. We should know our application well and preserve enough CPU and memory capacity in order to prevent running out of resources. In this section, we'll introduce how to manage node capacity in the Kubernetes nodes. Furthermore, we'll also describe how to manage pod computing resources.

Getting ready

Before you start managing computing resources, you should know your applications well in order to know the maximum resources they need. Before we start, check out the current node capacity using the `kubectl` command described in Chapter 1, *Building Your Own Kubernetes*:

```
// check current node capacity
# kubectl get nodes -o json | jq '.items[] |
{name: .metadata.name, capacity:
.status.capacity}'
{
  "name": "kube-node1",
  "capacity": {
    "cpu": "1",
    "memory": "1019428Ki",
    "pods": "40"
  }
}
{
  "name": "kube-node2",
  "capacity": {
    "cpu": "1",
    "memory": "1019428Ki",
    "pods": "40"
  }
}
```

You should know currently, we have two nodes with `1` CPU and `1019428` bytes memory. The node capacity of the pods are `40` for each. Then, we can start planning. How much computing resource capacity is allowed to be used on a node? How much computing resource is used in running our containers?

How to do it…

When the Kubernetes scheduler schedules a pod running on a node, it will always ensure that the total limits of the containers are less than the node capacity. If a node runs out of resources, Kubernetes will not schedule any new containers running on it. If no node is available when you launch a pod, the pod will remain pending, since the Kubernetes scheduler will be unable to find any node that could run your desired pod.

Managing node capacity

Sometimes, we want to explicitly preserve some resources for other processes or future usage on the node. Let's say we want to preserve 200 MB on all my nodes. First, we'll need to create a pod and run the `pause` container in Kubernetes. Pause is a container for each pod for forwarding the traffic. In this scenario, we'll create a resource reserver pod, which is basically doing nothing with a limit of 200 MB:

```
// configuration file for resource reserver
# cat /etc/kubernetes/reserve.yaml
apiVersion: v1
kind: Pod
metadata:
  name: resource-reserver
spec:
  containers:
  - name: resource-reserver
    image: gcr.io/google_containers/pause:0.8.0
    resources:
      limits:
        memory: 200Mi
```

Since it's a pod infra container, we will not use kubectl to launch it. Note that we put it in the `/etc/kubernetes/` folder. You could put it under different paths; write down the path and we'll need to add it into the kubelet config file to launch it. Find the kubelet config file you specified in the *Configuring nodes* recipe in Chapter 1, *Building Your Own Kubernetes* and add the following argument when launching kubelet: `--config=/etc/kubernetes/reserve.yaml`. Restart kubelet. After we restart kubelet, we will see the kubelet log in the node:

```
I0325 20:44:22.937067   21306 kubelet.go:1960]
Starting kubelet main sync loop.
I0325 20:44:22.937078   21306 kubelet.go:2012]
SyncLoop (ADD): "resource-reserver-kube-
node1_default"
I0325 20:44:22.937138   21306 kubelet.go:2012]
SyncLoop (ADD): "mynginx-e09bu_default"
I0325 20:44:22.963425   21306 kubelet.go:2012]
SyncLoop (ADD): "resource-reserver-kube-
node1_default"
I0325 20:44:22.964484   21306 manager.go:1707]
Need to restart pod infra container for "resource-
reserver-kube-node1_default" because it is not
found
```

kubelet will check whether the infra container exists and create it accordingly:

```
// check pods list
# kubectl get pods
NAME                                READY     STATUS
RESTARTS   AGE
resource-reserver-kube-node1   1/1       Running
0          3m
```

The Kubernetes master is aware that the resource reserver pod has been created. Let's describe the details to get deeper insight:

```
# kubectl describe pods resource-reserver-kube-
node1
```

```
Name:         resource-reserver-kube-node1
Namespace:      default
Image(s):
gcr.io/google_containers/pause:0.8.0
Node:         kube-node1/10.0.0.224
Start Time:       Fri, 25 Mar 2016 20:44:24 +0000
Labels:         <none>
Status:         Running
IP:         192.168.99.3
Replication Controllers:  <none>
Containers:
  resource-reserver:
    ...
    QoS Tier:
      memory:  Guaranteed
    Limits:
      memory:  200Mi
    Requests:
      memory:    200Mi
    State:    Running
      Started:    Fri, 25 Mar 2016 20:44:24 +0000
    Ready:    True
```

We can find the limits and requests that are all set as `200Mi`; it means that this container has been reserved a minimum of 200 MB and a maximum of 200 MB. Repeat the same steps in your other nodes and check the status via the `kubectl` command:

```
# kubectl get pods
NAME                                READY     STATUS
RESTARTS   AGE
resource-reserver-kube-node1   1/1       Running
0          11m
resource-reserver-kube-node2   1/1       Running
0          42m
```

Note

Limits or requests?

The Kubernetes scheduler schedules a pod running on a node by checking the remaining computing resources. We could

specify the limits or requests for each pod we launch. Limit means the maximum resources this pod can occupy. Request means the minimum resources this pod needs. We could use the following inequality to represent their relation: 0 <= request <= the resource this pod occupies <= limit <= node capacity.

Managing computing resources in a pod

The concept for managing the capacity in a pod or node is similar. They both specify the requests or limits under the container resource spec.

Let's create an nginx pod with certain requests and limits using `kubectl create -f nginx-resources.yaml` to launch it:

```
# cat nginx-resources.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    resources:
      requests:
        cpu: 250m
        memory: 32Mi
      limits:
        cpu: 500m
        memory: 64Mi

// create the pod
# kubectl create -f nginx-resources.yaml
pod "nginx" created
```

Following are the available resources for this pod:

- CPU: 250 milli core ~ 500 milli core
- Memory: 32MB ~ 64 MB

Please note that the minimum CPU limit is set to 10 millicore. You cannot specify a value less than the minimum limit. Let's get more details via kubectl:

```
# kubectl describe pod nginx
Name:        nginx
Namespace:      default
Image(s):      nginx
Node:        kube-node1/10.0.0.224
Start Time:      Fri, 25 Mar 2016 21:12:43 +0000
Labels:        name=nginx
Status:        Running
Reason:
Message:
IP:        192.168.99.4
Replication Controllers:  <none>
Containers:
  nginx:
    ...
    QoS Tier:
      cpu:  Burstable
      memory:  Burstable
    Limits:
      memory:  64Mi
      cpu:  500m
    Requests:
      cpu:    250m
      memory:    32Mi
    State:    Running
      Started:    Fri, 25 Mar 2016 21:12:44 +0000
    Ready:    True
    Restart Count:  0
```

Everything is expected. QoS Tier is `Burstable`. Compared with `Guaranteed`, `Burstable` has a buffer to burst to the limits; however, `Guaranteed` will always reserve certain resources for the pod. Please note that if you specify too many pods with `Guaranteed`, cluster utilization would be poor, since it wastes the resources if the containers are not reaching the limits all the time.

See also

In this section, you learned how to constrain computing resources in Kubernetes. We give more control to our containers. Check out the following recipes:

- The *Preparing your environment* and *Configuring nodes* recipes in Chapter 1, *Building Your Own Kubernetes*
- The *Working with namespaces* recipe in Chapter 2, *Walking through Kubernetes Concepts*
- The *Working with configuration files* recipe in Chapter 3, *Playing with Containers*
- The *Monitoring master and node* recipe in Chapter 8, *Logging and Monitoring*

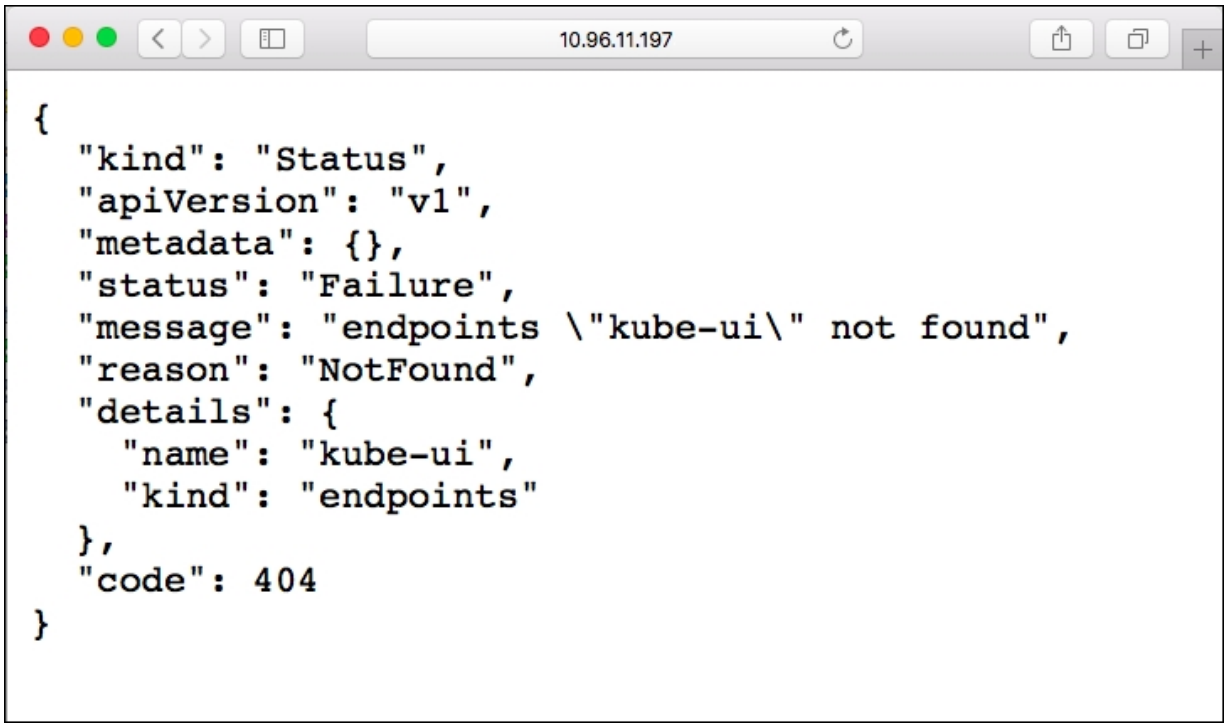
Playing with WebUI

Kubernetes has the WebUI add-on that visualizes Kubernetes' status, such as pod, replication controller, and service.

Getting ready

The Kubernetes WebUI is assigned as `http://<kubernetes master>/ui`. However, it is not launched by default, instead there are YAML files in the release binary.

> **Note**
>
> Kubernetes 1.2 introduces the dashboard. For more details, please refer to http://kubernetes.io/docs/user-guide/ui/.



Access to Kubernetes master/ui page

Let's download a release binary and launch the WebUI:

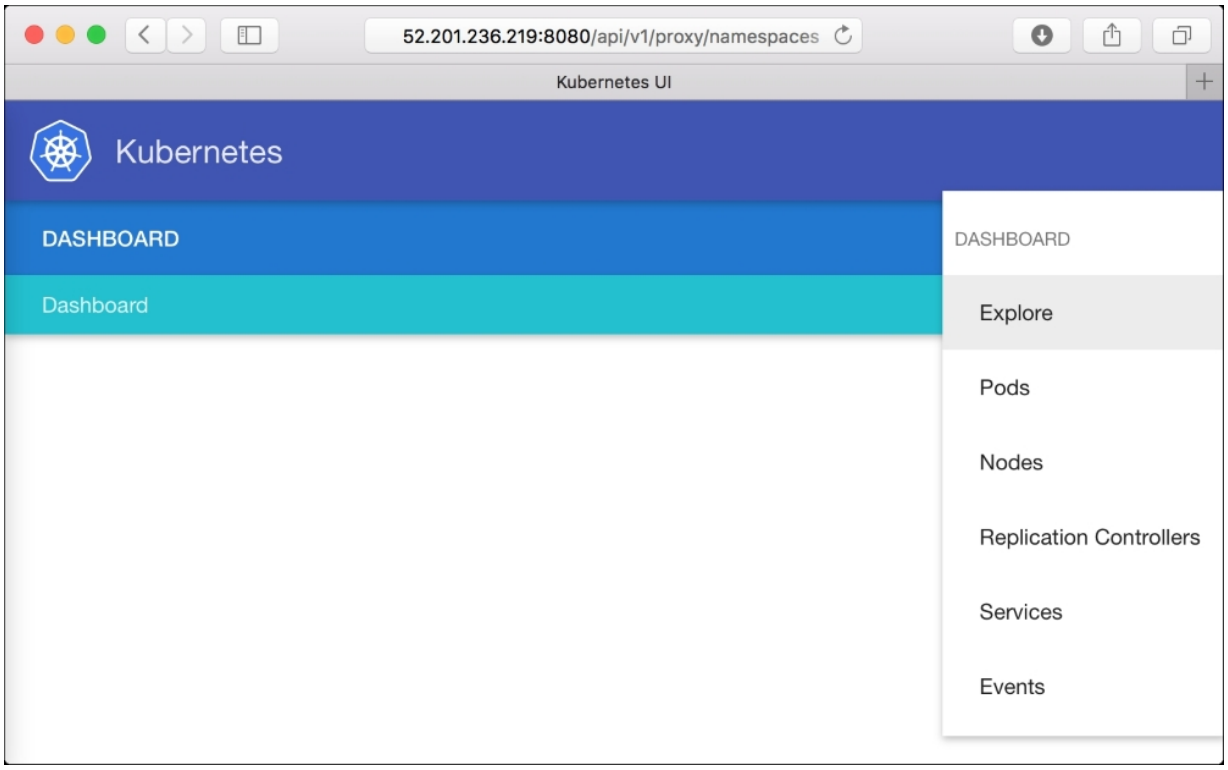
```
//Download a release binary
$ curl -L -O
https://github.com/kubernetes/kubernetes/releases/
download/v1.1.4/kubernetes.tar.gz

//extract the binary
$ tar zxf kubernetes.tar.gz
//WebUI YAML file is under the
cluster/addons/kube-ui directory
$ cd kubernetes/cluster/addons/kube-ui/
$ ls
kube-ui-rc.yaml    kube-ui-svc.yaml
```

How to do it…

Let's launch the kube-ui replication controller and service:

```
# kubectl create -f kube-ui-rc.yaml
replicationcontroller "kube-ui-v2" created

# kubectl create -f kube-ui-svc.yaml
service "kube-ui" created
```

Note that `kube-ui-svc` is a type of ClusterIP service; however, it is associated with Kubernetes master (/ui). You can access, from the outside, the Kubernetes network at `http://<kubernetes master>/ui`.

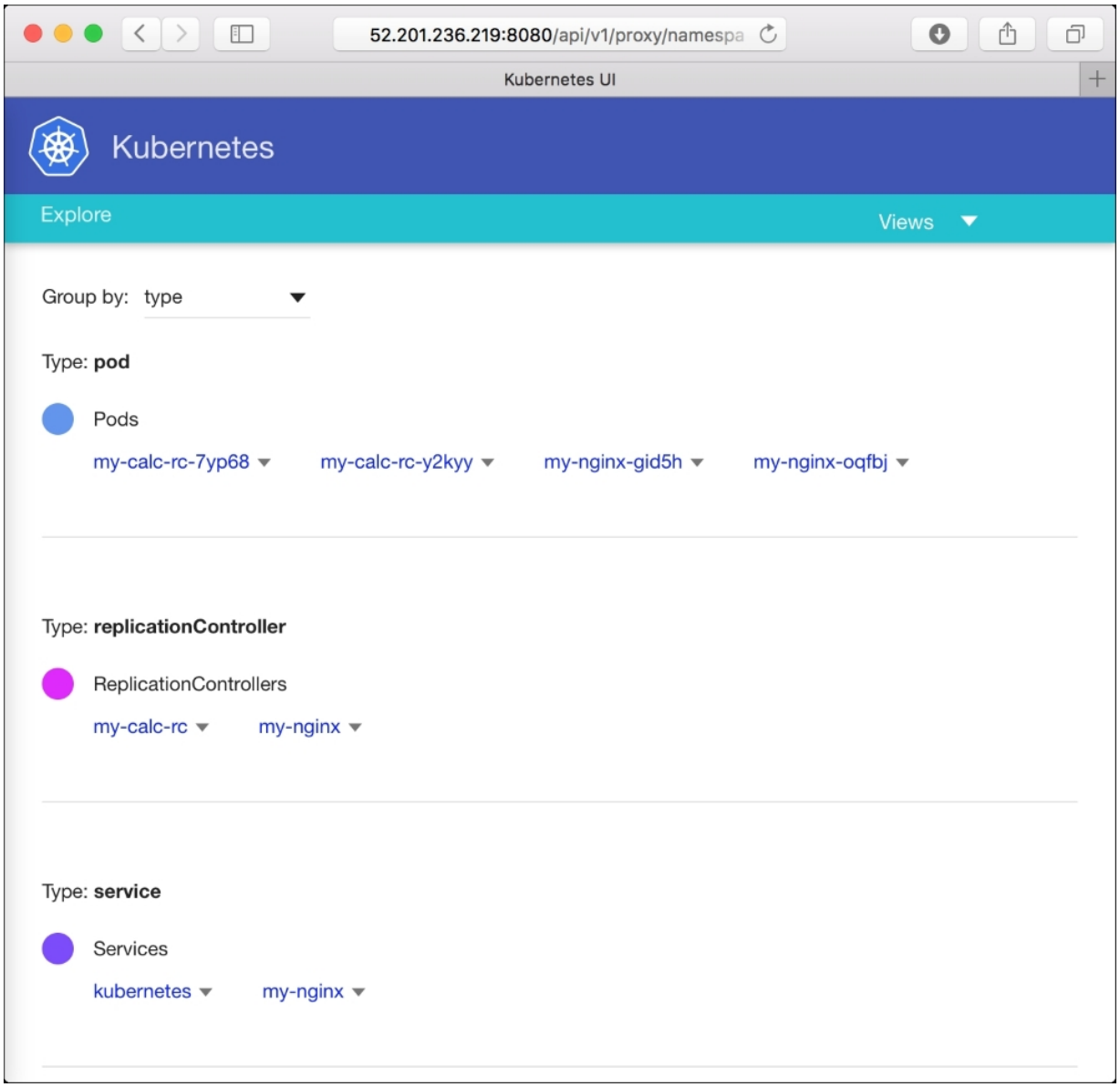


Launching the kube-ui/ui shows the dashboard screen

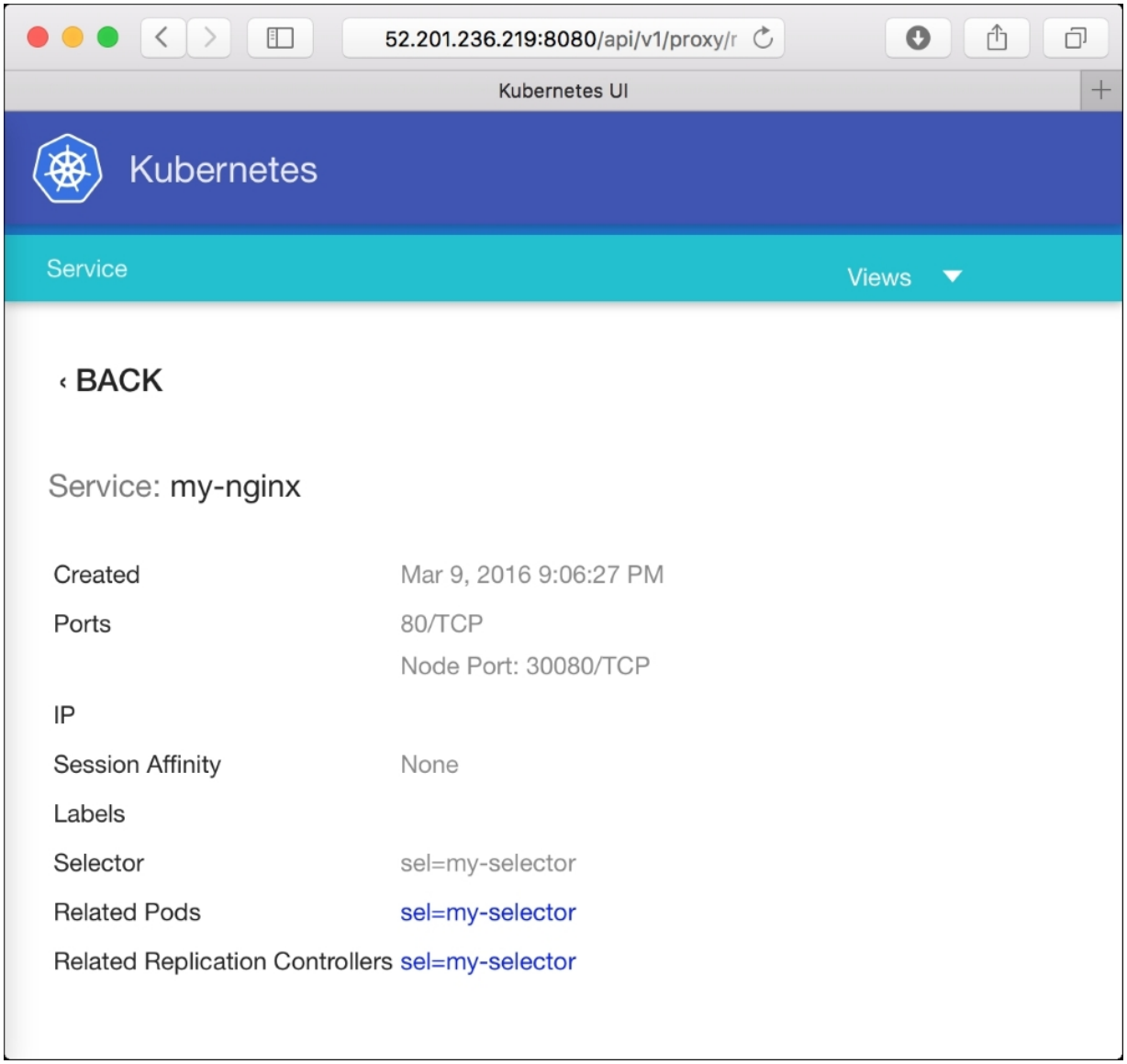
How it works…

The kube-ui replication controller accesses the API Server to get the Kubernetes cluster information same as the `kubectl` command, though read-only. However, it is useful for navigating Kubernetes' status and is easier to explore than the `kubectl` command.

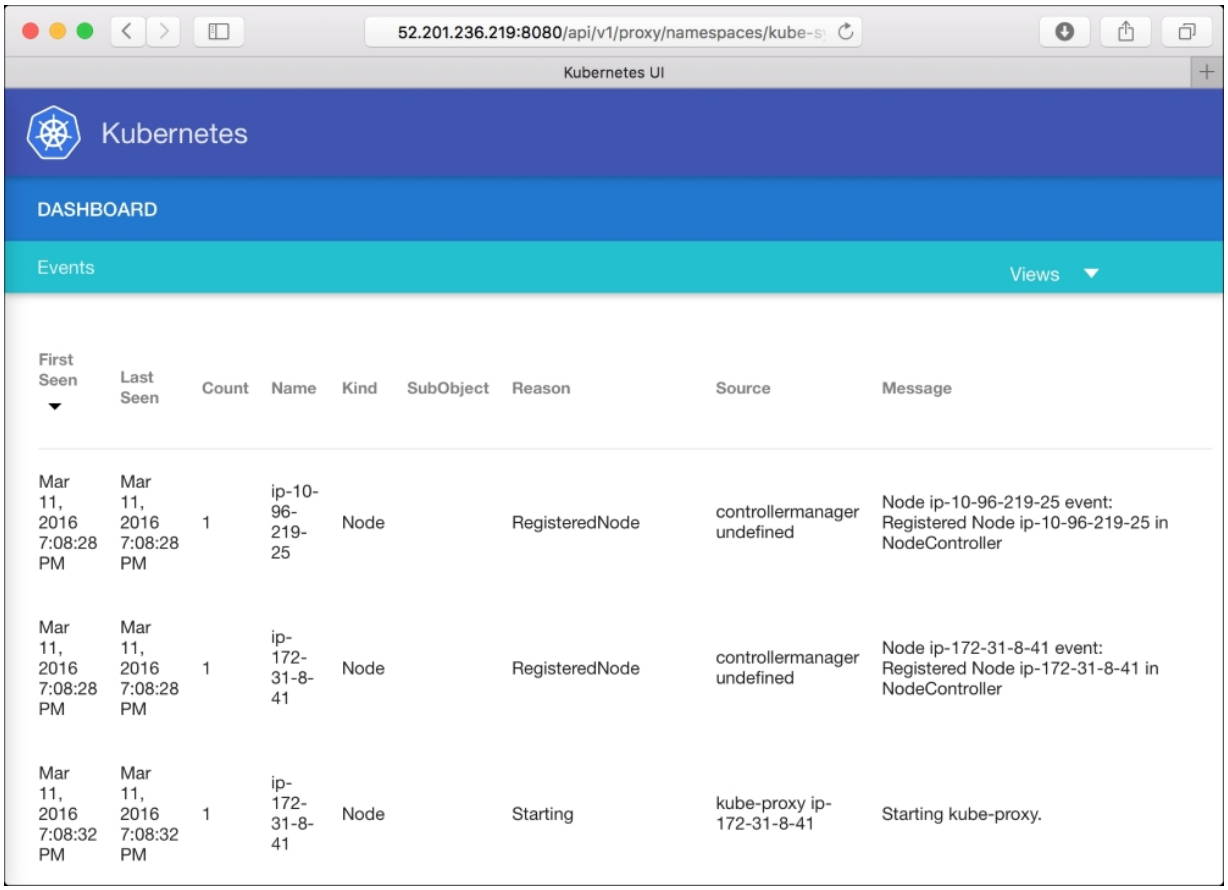
The following screenshot is an **Explore** page that shows pod, replication controller, and service instances:



On clicking on one of the instances, it shows detailed information, as shown in the following screenshot. It shows a service, which indicates port, node port, and selectors. It is easy to find an associated replication controller and pods:



Additionally, UI can also show events, as follows:



See also

This recipe described how to launch a web interface that will help in easily exploring Kubernetes instances, such as pods, replication controllers, and services without the `kubectl` command. Please refer to following recipes on how to get detailed information via the `kubectl` command.

- The *Working with pods*, *Working with a replication controller*, and *Working with services* recipes in Chapter 2, *Walking through Kubernetes Concepts*

Working with a RESTful API

The Kubernetes administrator can control the Kubernetes cluster via the `kubectl` command; it supports local and remote execution. However, some of the administrators or operators may need to integrate a program to control the Kubernetes cluster.

Kubernetes has a RESTful API that allows controlling the Kubernetes cluster via API similar to the `kubectl` command.

Getting ready

The RESTful API is open by default when we launch the API Server; you may access the RESTful API via the `curl` command, as follows:

```
//assume API server is running at localhost port number
8080
# curl http://localhost:8080/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ]
}
```

How to do it…

Let's create a replication controller using the following JSON format:

```
# cat nginx-rc.json
{
    "apiVersion": "v1",
    "kind": "ReplicationController",
    "metadata": {
        "name": "my-first-rc"
    },
    "spec": {
        "replicas": 2,
        "template": {
            "spec": {
                "containers": [
                    {
                        "image": "nginx",
```

```
                        "name": "my-nginx"
                    }
                ]
            },
            "metadata": {
                "labels": {
                    "app": "nginx"
                }
            }
        },
        "selector": {
            "app": "nginx"
        }
    }
}
```

Submit a request to create a replication controller, as follows:

```
# curl -XPOST -H "Content-type: application/json" -d
@nginx-rc.json
http://localhost:8080/api/v1/namespaces/default/replicatio
ncontrollers
```

Then, `kubectl get rc` command should be as follows:

```
# kubectl get rc
CONTROLLER    CONTAINER(S)    IMAGE(S)    SELECTOR
REPLICAS   AGE
my-first-rc   my-nginx        nginx       app=nginx   2
10s
```

Of course, use the `curl` command to check via a RESTful API, as follows:

```
# curl -XGET
http://localhost:8080/api/v1/namespaces/default/replicatio
ncontrollers
```

Deletion can also be done via a RESTful API, as follows:

```
# curl -XDELETE
http://localhost:8080/api/v1/namespaces/default/replicatio
ncontrollers/my-first-rc
```

Let's write the program that performs the same process. Following is the Python 2.7 code that creates the same replication controller:

```
# cat nginx-rc.py

import httplib
import json
k8s_master_url = "localhost"
k8s_master_port = 8080
namespace="default"

headers = {"Content-type": "applicaiton/json"}

rc = {}
rc["apiVersion"] = "v1"
rc["kind"] = "ReplicationController"
rc["metadata"] = {"name" : "my-second-rc"}
rc["spec"] = {
    "replicas": 2,
    "selector": {"app": "nginx"},
    "template": {
        "metadata": {"labels": {"app": "nginx"}},
        "spec": {
            "containers" :[
                {"name": "my-nginx", "image": "nginx"}
            ]
        }
    }
}

h1 = httplib.HTTPConnection(k8s_master_url,
k8s_master_port)
h1.request("POST",
"/api/v1/namespaces/%s/replicationcontrollers" %
namespace, json.dumps(rc), headers)
res = h1.getresponse()

print "return code = %d" % res.status
```

You can run this code using the Python interpreter, as follows:

```
# python nginx-rc.py
return code = 201

//HTTP return code 201 meant "Created"
```

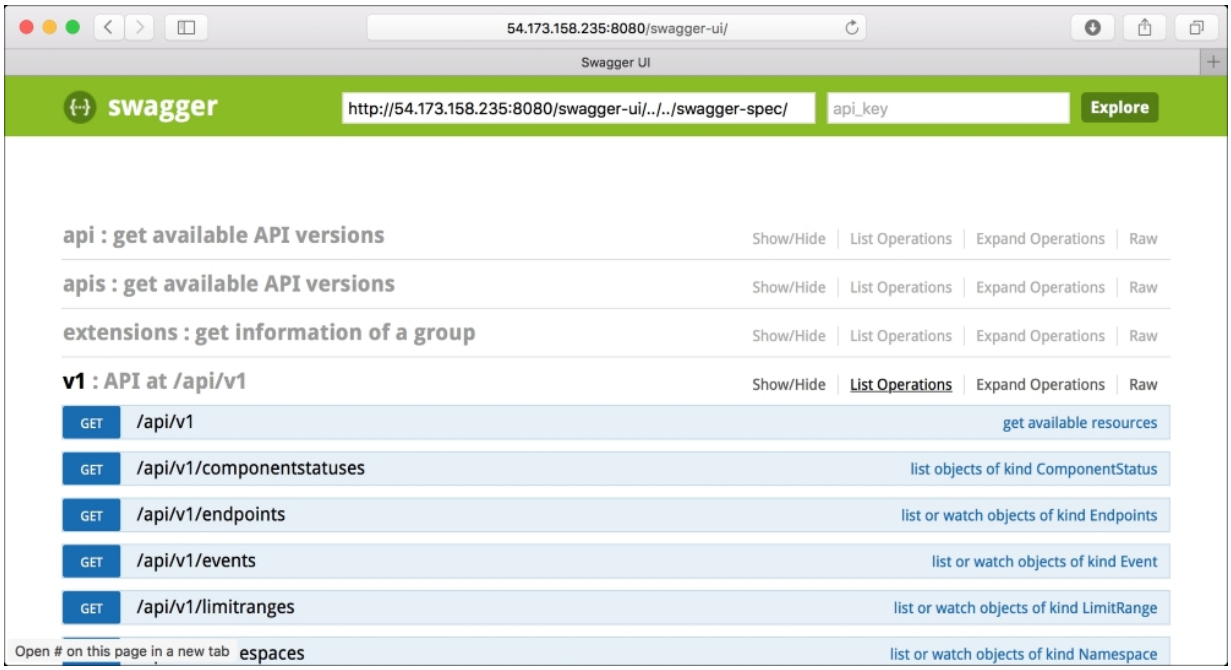
How it works…

The RESTful API allows the CRUD (Create, Read, Update, and Delete) operations, which are the same concepts behind every modern web application. For more details, please refer to https://en.wikipedia.org/wiki/Create,_read,_update_and_delete.

Kubernetes RESTful API examples and related HTTP methods are as follows:

Operation	**HTTP Method**	**Example**
Create	`POST`	`POST /api/v1/namespaces/default/services`
Read	`GET`	`GET /api/v1/componentstatuses`
Update	`PUT`	`PUT /api/v1/namespaces/default/replicationcontrollers/my-first-rc`
Delete	`DELETE`	`DELETE /api/v1/namespaces/default/pods/my-nginx`

The entire Kubernetes RESTful APIs are defined by Swagger (http://swagger.io/). You can see a detailed description via `http://<API Server IP Address>:<API Server port>/swagger-ui`.



See also

This recipe described how to use the Kubernetes RESTful API via a program. It is important to integrate with your automation program remotely. For detailed parameter and security enhancement, please refer to the following recipes:

- The *Working with configuration files* recipe in Chapter 3, *Playing with Containers*
- The *Authentication and authorization* recipe in Chapter 7, *Advanced Cluster administration*

Authentication and authorization

In order to use more advanced management, we can add permission rules to the Kubernetes system. Two permission types could be generated in our cluster: one is between the machines. Nodes having authentication can contact the controlling node. For example, the master who owns certification with the etcd server can store data in etcd. The other permission rule is inside the Kubernetes master. Users can be given authorization for checking and creating the resources. Applying authentication and authorization is a secure solution to prevent your data or status being accessed by others.

Getting ready

Before you start configuring your cluster with some permissions, please have your cluster installed. Nevertheless, stop every service in the system. They will be started later with the authentication enabled.

How to do it…

In this recipe, we will have a discussion on both authentication and authorization. For authentication, etcd and the Kubernetes server need to do identity verification before they respond to the requests. On the other hand, authorization restricts users by different resource access permissions. All of these contacts are based on the API connection. Later sections show you how to complete the configuration and follow the authentication.

Enabling authentication for an API call

There are several methods to block the unauthenticated communication in the Kubernetes system. We are going to introduce basic authentication mechanism. It is easier to set it on not only the Kubernetes masters, but etcd servers.

Basic authentication of etcd

First, let's try to send API requests on the etcd host. You will find that anyone can access the data by default:

```
// Create a key-value pair in etcd
# curl -X PUT -d value="Happy coding"
http://localhost:4001/v2/keys/message
{"action":"set","node":
{"key":"/message","value":"Happy
coding","modifiedIndex":4,"createdIndex":4}}
// Check the value you just push
# curl http://localhost:4001/v2/keys/message
{"action":"get","node":
{"key":"/message","value":"Happy
coding","modifiedIndex":4,"createdIndex":4}}
// Remove the value
# curl -X DELETE
http://localhost:4001/v2/keys/message
{"action":"delete","node":
{"key":"/message","modifiedIndex":5,"createdIndex"
:4},"prevNode":{"key":"/message","value":"Happy
coding","modifiedIndex":4,"createdIndex":4}}
```

Without authentication, neither reading nor writing functions can be protected. The way to enable basic authentication of etcd is through the RESTful API as well. The procedure is as follows:

- Add a password for the admin account `root`
- Enable basic authentication
- Stop both read and write permissions of the guest account

Make sure the etcd service is running. We will transfer the preceding logics into the following commands:

```
// Send the API request for setup root account
# curl -X PUT -d "
{\"user\":\"root\",\"password\":\"
<YOUR_ETCD_PASSWD>\",\"roles\":[\"root\"]}"
http://localhost:4001/v2/auth/users/root
{"user":"root","roles":["root"]}
// Enable authentication
# curl -X PUT http://localhost:4001/v2/auth/enable
// Encode "USERACCOUNT:PASSWORD" string in base64
format, and record in a value
# AUTHSTR=$(echo -n "root:<YOUR_ETCD_PASSWD>" |
base64)
// Remove all permission of guest account. Since
we already enable authentication, use the
authenticated root.
# curl -H "Authorization: Basic $AUTHSTR" -X PUT -
d "{\"role\":\"guest\",\"revoke\":{\"kv\":
{\"read\":[\"*\"],\"write\":[\"*\"]}}}"
http://localhost:4001/v2/auth/roles/guest
{"role":"guest","permissions":{"kv":{"read":
[],"write":[]}}}
```

Now, for validation, try to check anything in etcd through the API:

```
# curl http://localhost:4001/v2/keys
{"message":"Insufficient credentials"}
```

Because we didn't specify any identity for this API call, it is regarded as a request from a guest. No authorization for even viewing data.

For long-term usage, we would put user profiles of etcd in the Kubernetes master's configuration. Check your configuration file of the Kubernetes API server. In the RHEL server, it is the file `/etc/kubernetes/apiserver`; or for the other Linux server, just the one for service, `/etc/init.d/kubernetes-master`. You can find a flag for the etcd server called `--etcd-servers`. Based on the previous settings, the value we attached is a simple URL with a port. It could be `http://ETCD_ELB_URL:80`. Add an account root and its password in a plain text format to the value, which is the authorization header for the HTTP request. The new value for flag `--etcd-servers` would

become `http://root:YOUR_ETCD_PASSWD@ETCD_ELB_URL:80`. Afterwards, your Kubernetes API server daemon will work well with an authentication-enabled etcd endpoint.

Basic authentication of the Kubernetes master

Before we set up authentication in the Kubernetes master, let's check the endpoint of the master first:

```
# curl https://K8S_MASTER_HOST_IP:SECURED_PORT --
insecure
```

or

```
# curl http://K8S_MASTER_ELB_URL:80
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/extensions",
    "/apis/extensions/v1beta1",
    "/healthz",
    "/healthz/ping",
    "/logs/",
    "/metrics",
    "/resetMetrics",
    "/swagger-ui/",
    "/swaggerapi/",
    "/ui/",
    "/version"
  ]
}
```

We don't want the previous message to be exposed to everyone, do we? Same as the etcd host, Kubernetes master can apply basic authentication as a security mechanism. In this section, we are going to limit the permission of the API server. However, different from etcd, the information of authentication is defined in a file:

```
// Create a file for basic authentication with
content: PASSWORD,USERNAME,UID
# cat /root/k8s-bafile
<APISERVER_BA_PASSWORD>,
<APISERVER_BA_USERACCOUNT>,1
```

Then, we have to specify this file in the configuration file. According to your daemon management tool, the configuration file could be located at `/etc/init.d/kubernetes-master` or `/etc/kubernetes/apiserver`. A new flag named `--basic-auth-file` should be added to the configuration file:

- For the file `kubernetes-master`, append flag `--basic-auth-file` after the `kube-apiserver` command or the `hyperkube apiserver` command. The value for this tag should be the full path of the basic authentication file. For instance, `--basic-auth-file=/root/k8s-bafile`.
- For the file `apiserver`, add the tag to the variable `KUBE_API_ARGS`. For example, `KUBE_API_ARGS=--basic-auth-file=/root/k8s-bafile`.

Most important of all, ensure the user who starts the services of Kubernetes, either root or kubelet, has the permission to access the file you attached to the tag. After you add the new tag, it is necessary to restart the service for making the authentication effective. Next, it is good for you to try the `curl` command at the beginning of this section. It will return `Unauthorized` without providing the username and password.

Our nodes communicate with API server through the insecure port `8080`. Although we didn't have to specify any role for authorizing permission, be aware of configuring the firewall of master, which only allows nodes to go through port `8080`. On AWS, a security group can help for this part.

There are still some methods for the Kubernetes master's authentication. Please check the official website for other ideas

(http://kubernetes.io/docs/admin/authentication/).

Making use of user authorization

We can also add different user permissions for the Kubernetes master's API server daemon. There are three flags required to set user authorization. They are as follows:

- `--authorization-mode=ABAC`: The value, ABAC, is the abbreviation of Attribute-Based Access Control. By enabling this mode, we can set up customized user permissions.
- `--token-auth-file=<FULL_PATH_OF_YOUR_TOKEN_FILE>`: This is the file we used to announce the qualified users for API access. It is possible to provide more accounts and token pairs.
- `--authorization-policy-file=<FULL_PATH_OF_YOUR_POLICY_FILE>`: We would need this policy file to generate separated rules for different users.

These special tags are going to be appended after the command `kube-apiserver` or `hyperkube apiserver`. You can refer to the following example:

```
// The daemon configuration file of Kubernetes
master for init.d service
# cat /etc/init.d/kubernetes-master
(above lines are ignored)
:
# Start daemon.
    echo $"Starting apiserver: "
    daemon $apiserver_prog \
    --service-cluster-ip-range=${CLUSTER_IP_RANGE}
\
    --insecure-port=8080 \
    --secure-port=6443 \
    --authorization-mode=ABAC \
    --token-auth-file=/root/k8s-tokenfile \
    --authorization-policy-file=/root/k8s-
policyfile \
    --address=0.0.0.0 \
    --etcd-servers=${ETCD_SERVERS} \
    --cluster-name=${CLUSTER_NAME} \
```

```
    > ${logfile}-apiserver.log 2>&1 &
:
(below lines are ignored)
```

or

```
// The kubernetes apiserver's configuration file
for systemd service in RHEL
# cat /etc/kubernetes/apiserver
(above lines are ignored)
:
KUBE_API_ARGS="--authorization-mode=ABAC --token-
auth-file=/root/k8s-tokenfile --authorization-
policy-file=/root/k8s-policyfile"
```

You still need to configure the file for account and the file for policy. To demonstrate the usage of customizing user permission, the following content of files show you how to create an admin account with full access and a read-only account:

```
# cat /root/k8s-tokenfile
k8s2016,admin,1
happy123,amy,2
```

The format of user definition is similar to the basic authentication file we mentioned before. Each line has these items in sequence: token, username, and UID. Other than admin, we create another user account called `amy`, which will only have read-only permission:

```
# cat /root/k8s-policyfile
{"apiVersion":
"abac.authorization.kubernetes.io/v1beta1",
"kind": "Policy", "spec": {"user": "admin",
"namespace": "*", "resource": "*", "apiGroup":
"*"}}
{"apiVersion":
"abac.authorization.kubernetes.io/v1beta1",
"kind": "Policy", "spec": {"user": "amy",
```

```
"namespace": "*", "resource": "*", "readonly":
true}}
```

For the policy file, each line will be a policy in the JSON format. Every policy should indicate the user who obeys its rule. The first policy for admin allows control permission on every namespace, resource, and API group. The key `apiGroup` specifies different API categories. For instance, the resource `job` is defined in the `extensions` API group. In order to access `job`, the `extensions` type should be included in `apiGroup`. The second policy is defined with read-only permission, which means the role `amy` can only view resources, but not create, remove, and edit actions.

Later, restart all the daemons of the Kubernetes master after you make both configuration files and service files ready:

```
// For init.d service management
# service kubernetes-master restart
// Or, you can restart the individually with
dependency
# systemctl stop kube-scheduler
# systemctl stop kube-controller-manager
# systemctl stop kube-apiserver
# systemctl start kube-apiserver
# systemctl start kube-controller-manager
# systemctl start kube-scheduler
```

See also

It is recommended to read some of the previous recipes on installing the Kubernetes cluster:

- The *Building datastore*, *Configuring master* and *Configuring nodes* recipe in Chapter 1, *Building Your Own Kubernetes*
- The *Auto-deploying Kubernetes through Chef recipes* recipe in Chapter 6, *Building Kubernetes on AWS*

Chapter 8. Logging and Monitoring

In this chapter, we will cover the following topics:

- Collecting application logs
- Working with Kubernetes logs
- Working with etcd log
- Monitoring master and node

Introduction

As DevOps, logging and monitoring are what we always keep in mind. These tell us the stability and the status of our systems. For taking care of logs, you will learn how to collect the application logs inside Kubernetes. You will also learn how to collect and inspect the logs for Kubernetes. Finally, we will go through setting up monitoring systems for Kubernetes.

Collecting application logs

When you start managing the application, the log collection and analysis are two of the important routines to keep tracking the application's status.

However, there are some difficulties when the application is managed by Docker/Kubernetes; because the log files are inside the container, it is not easy to access them from outside the container. In addition, if the application has many pods by the replication controller, it will also be difficult to trace or identify in which pod the issue that has happened.

One way to overcome this difficulty is to prepare a centralized log collection platform that accumulates and preserves the application log. This recipe describes one of the popular log collection platforms **ELK** (**Elasticsearch**, **Logstash**, and **Kibana**).

Getting ready

First, we need to prepare the Elasticsearch server at the beginning. Then, the application will send a log to Elasticsearch using Logstash. We will visualize the analysis result using Kibana.

Elasticsearch

Elasticsearch (https://www.elastic.co/products/elasticsearch) is one of the popular text indexes and analytic engines. There are some examples YAML files that are provided by the Kubernetes source file; let's download it using the `curl` command to set up Elasticsearch:

> **Note**

An example YAML file is located on GitHub at https://github.com/kubernetes/kubernetes/tree/master/examples/elasticsearch.

```
# curl -L -O
https://github.com/kubernetes/kubernetes/releases/
download/v1.1.4/kubernetes.tar.gz
  % Total    % Received % Xferd  Average Speed
Time    Time     Time  Current
                                 Dload  Upload
Total   Spent    Left  Speed
100   593    0   593    0     0   1798      0 --:-
-:-- --:--:-- --:--:--  1802
100  181M  100  181M    0     0  64.4M      0
0:00:02  0:00:02 --:--:-- 75.5M
# tar zxf kubernetes.tar.gz
# cd kubernetes/examples/elasticsearch/
# ls
es-rc.yaml  es-svc.yaml  production_cluster
README.md  service-account.yaml
```

Create ServiceAccount (`service-account.yaml`) and then create the Elasticsearch replication controller (`es-rc.yaml`) and service (`es-svc.yaml`) as follows:

```
# kubectl create -f service-account.yaml
serviceaccount "elasticsearch" created

//As of Kubernetes 1.1.4, it causes validation
error
//therefore append --validate=false option
# kubectl create -f es-rc.yaml --validate=false
replicationcontroller "es" created

# kubectl create -f es-svc.yaml
service elasticsearch" created
```

Then, you can access the Elasticsearch interface via the Kubernetes service as follows:

```
//Elasticsearch is open by 192.168.45.152 in this
example
# kubectl get service
NAME            CLUSTER_IP       EXTERNAL_IP
PORT(S)            SELECTOR                  AGE
elasticsearch   192.168.45.152
9200/TCP,9300/TCP   component=elasticsearch   9s
kubernetes      192.168.0.1      <none>
443/TCP            <none>                    110d

//access to TCP port 9200
# curl http://192.168.45.152:9200/
{
  "status" : 200,
  "name" : "Wallflower",
  "cluster_name" : "myesdb",
  "version" : {
    "number" : "1.7.1",
    "build_hash" :
"b88f43fc40b0bcd7f173a1f9ee2e97816de80b19",
    "build_timestamp" : "2015-07-29T09:54:16Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.4"
  },
  "tagline" : "You Know, for Search"
}
```

Now, get ready to send an application log to Elasticsearch.

How to do it…

Let's use a sample application, which was introduced in the *Moving monolithic to microservices* recipe in Chapter 5, *Building a Continuous Delivery Pipeline*. Prepare a Python Flask program as follows:

```
# cat entry.py

from flask import Flask, request
app = Flask(__name__)

@app.route("/")
```

```
def hello():
    return "Hello World!"

@app.route("/addition/<int:x>/<int:y>")
def add(x, y):
    return "%d" % (x+y)

if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

Use this application to send a log to Elasticsearch.

Logstash

Send an application log to Elasticsearch; using Logstash (https://www.elastic.co/products/logstash) is the easiest way, because it converts from a plain text format to the Elasticsearch (JSON) format.

Logstash needs a configuration file that specifies the Elasticsearch IP address and port number. In this recipe, Elasticsearch is managed by Kubernetes service; therefore, the IP address and port number can be found using the environment variable as follows:

Item	Environment Variable	Example
Elasticsearch IP address	`ELASTICSEARCH_SERVICE_HOST`	`192.168.45.152`
Elasticsearch port number	`ELASTICSEARCH_SERVICE_PORT`	`9200`

However, the Logstash configuration file doesn't support an environment variable directly. Therefore, the Logstash configuration file uses the placeholder as `_ES_IP_` and `_ES_PORT_` as follows:

```
# cat logstash.conf.temp

input {
    stdin {}
}

filter {
  grok {
    match => {
        "message" => "%{IPORHOST:clientip} %
{HTTPDUSER:ident} %{USER:auth} \[%
{DATA:timestamp}\] \"(?:%{WORD:verb} %
{NOTSPACE:request}(?: HTTP/%{NUMBER:httpversion})?
|%{DATA:rawrequest})\" %{NUMBER:response} (?:%
{NUMBER:bytes}|-)"
    }
  }
}

output {
  elasticsearch {
    hosts => ["_ES_IP_:_ES_PORT_"]
    index => "mycalc-access"
}

stdout { codec => rubydebug }
}
```

Startup script

The startup script will read an environment variable, and then replace the placeholder to set the real IP and port number, as follows:

```
#!/bin/sh

TEMPLATE="logstash.conf.temp"
LOGSTASH="logstash-2.2.2/bin/logstash"
```

```
cat $TEMPLATE | sed
"s/_ES_IP_/$ELASTICSEARCH_SERVICE_HOST/g" | sed
"s/_ES_PORT_/$ELASTICSEARCH_SERVICE_PORT/g" >
logstash.conf

python entry.py 2>&1 | $LOGSTASH -f logstash.conf
```

Dockerfile

Finally, prepare Dockerfile as follows to build a sample application:

```
FROM ubuntu:14.04

# Update packages
RUN apt-get update -y

# Install Python Setuptools
RUN apt-get install -y python-setuptools git
telnet curl openjdk-7-jre

# Install pip
RUN easy_install pip

# Bundle app source
ADD . /src
WORKDIR /src

# Download LogStash
RUN curl -L -O
https://download.elastic.co/logstash/logstash/logs
tash-2.2.2.tar.gz

RUN tar -zxf logstash-2.2.2.tar.gz

# Add and install Python modules
RUN pip install Flask

# Expose
EXPOSE  5000

# Run
CMD ["./startup.sh"]
```

Docker build

Let's build a sample application using the `docker build` command:

```
# ls
Dockerfile  entry.py  logstash.conf.temp
startup.sh

# docker build -t hidetosaito/my-calc-elk .
Sending build context to Docker daemon  5.12 kB
Step 1 : FROM ubuntu:14.04
 ---> 1a094f2972de
Step 2 : RUN apt-get update -y
 ---> Using cache
 ---> 40ff7cc39c20
Step 3 : RUN apt-get install -y python-setuptools
git telnet curl openjdk-7-jre
 ---> Running in 72df97dcbb9a

(skip…)

Step 11 : CMD ./startup.sh
 ---> Running in 642de424ee7b
 ---> 09f693436005
Removing intermediate container 642de424ee7b
Successfully built 09f693436005

//upload to Docker Hub using your Docker account
# docker login
Username: hidetosaito
Password:
Email: hideto.saito@yahoo.com
WARNING: login credentials saved in
/root/.docker/config.json
Login Succeeded

//push to Docker Hub
# docker push hidetosaito/my-calc-elk
The push refers to a repository
[docker.io/hidetosaito/my-calc-elk] (len: 1)
09f693436005: Pushed
b4ea761f068a: Pushed

(skip…)

c3eb196f68a8: Image already exists
latest: digest:
```

```
sha256:45c203d6c40398a988d250357f85f1b5ba7b14ae73d
449b3ca64b562544cf1d2 size: 22268
```

Kubernetes replication controller and service

Now, use this application by Kubernetes to send a log to Elasticsearch. First, prepare the YAML file to load this application using the replication controller and service as follows:

```
# cat my-calc-elk.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-calc-elk-rc
spec:
  replicas: 2
  selector:
        app: my-calc-elk
  template:
    metadata:
      labels:
        app: my-calc-elk
    spec:
      containers:
      - name: my-calc-elk
        image: hidetosaito/my-calc-elk
---
apiVersion: v1
kind: Service
metadata:
  name: my-calc-elk-service

spec:
  ports:
    - protocol: TCP
      port: 5000
  type: ClusterIP
  selector:
     app: my-calc-elk
```

Use the `kubectl` command to create the replication controller and service as follows:

```
# kubectl create -f my-calc-elk.yaml
replicationcontroller "my-calc-elk-rc" created
service "my-calc-elk-service" created
```

Check the Kubernetes service to find an IP address for this application as follows. It indicates `192.168.121.63`:

```
# kubectl get service
NAME                  CLUSTER_IP
EXTERNAL_IP   PORT(S)             SELECTOR
AGE
elasticsearch         192.168.101.143
9200/TCP,9300/TCP   component=elasticsearch   15h
kubernetes            192.168.0.1       <none>
443/TCP             <none>                    19h
my-calc-elk-service   192.168.121.63    <none>
5000/TCP            app=my-calc-elk           39s
```

Let's access this application using the `curl` command as follows:

```
# curl http://192.168.121.63:5000/
Hello World!

# curl http://192.168.121.63:5000/addition/3/5
8
```

Kibana

Kibana (https://www.elastic.co/products/kibana) is a visualization tool for Elasticsearch. Download Kibana, and specify the Elasticsearch IP address, and port number, to launch Kibana:

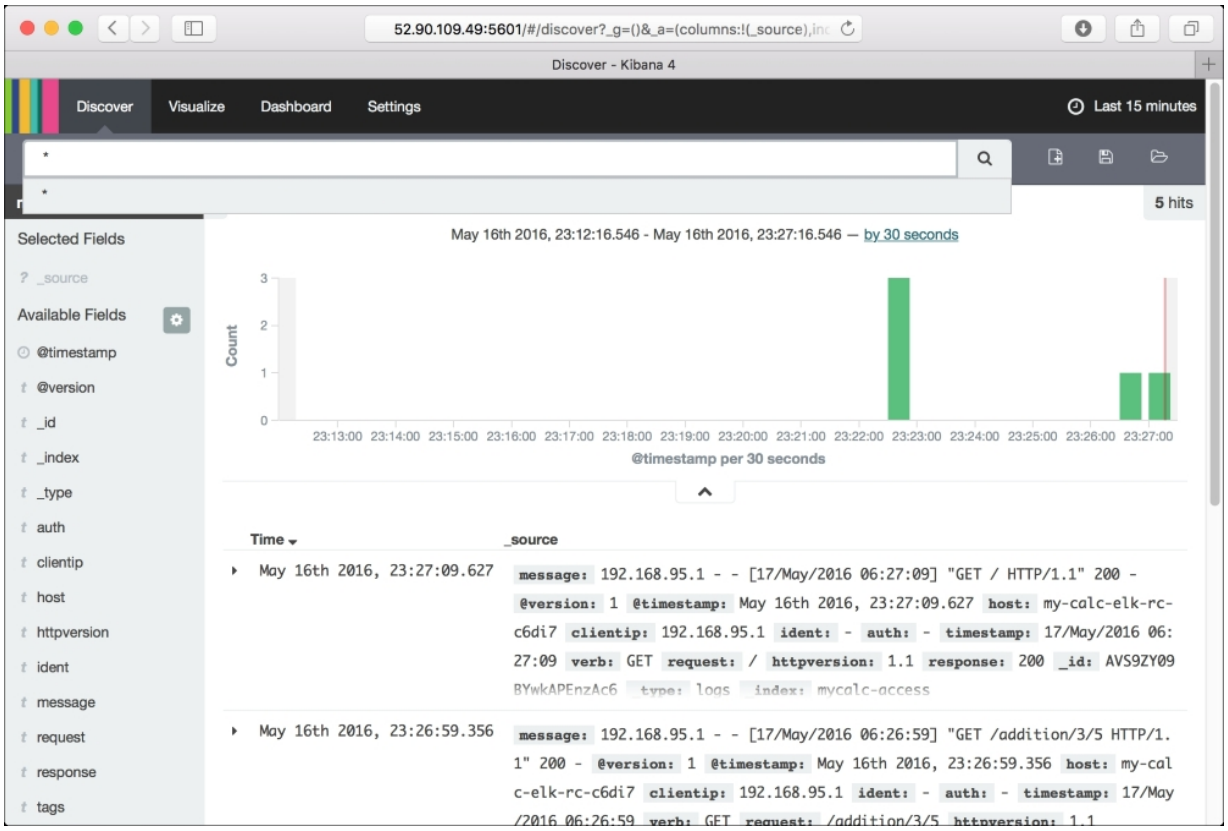
```
//Download Kibana 4.1.6
# curl -O
https://download.elastic.co/kibana/kibana/kibana-
4.1.6-linux-x64.tar.gz
  % Total    % Received % Xferd  Average Speed
```

```
Time    Time     Time  Current
                                 Dload  Upload
Total   Spent    Left  Speed
100 17.7M  100 17.7M    0     0  21.1M      0 --:-
-:-- --:--:-- --:--:-- 21.1M

//unarchive
# tar -zxf kibana-4.1.6-linux-x64.tar.gz

//Find Elasticsearch IP address
# kubectl get services
NAME                    CLUSTER_IP
EXTERNAL_IP   PORT(S)              SELECTOR
AGE
elasticsearch          192.168.101.143
9200/TCP,9300/TCP   component=elasticsearch   19h
kubernetes             192.168.0.1       <none>
443/TCP             <none>                    23h

//specify Elasticsearch IP address
# sed -i -e "s/localhost/192.168.101.143/g"
kibana-4.1.6-linux-x64/config/kibana.yml

//launch Kibana
# kibana-4.1.6-linux-x64/bin/kibana
```

Then, you will see the application log. Create a chart as follows:

Note

This cookbook doesn't cover how to configure Kibana; please visit the official page to learn about the Kibana configuration via https://www.elastic.co/products/kibana.

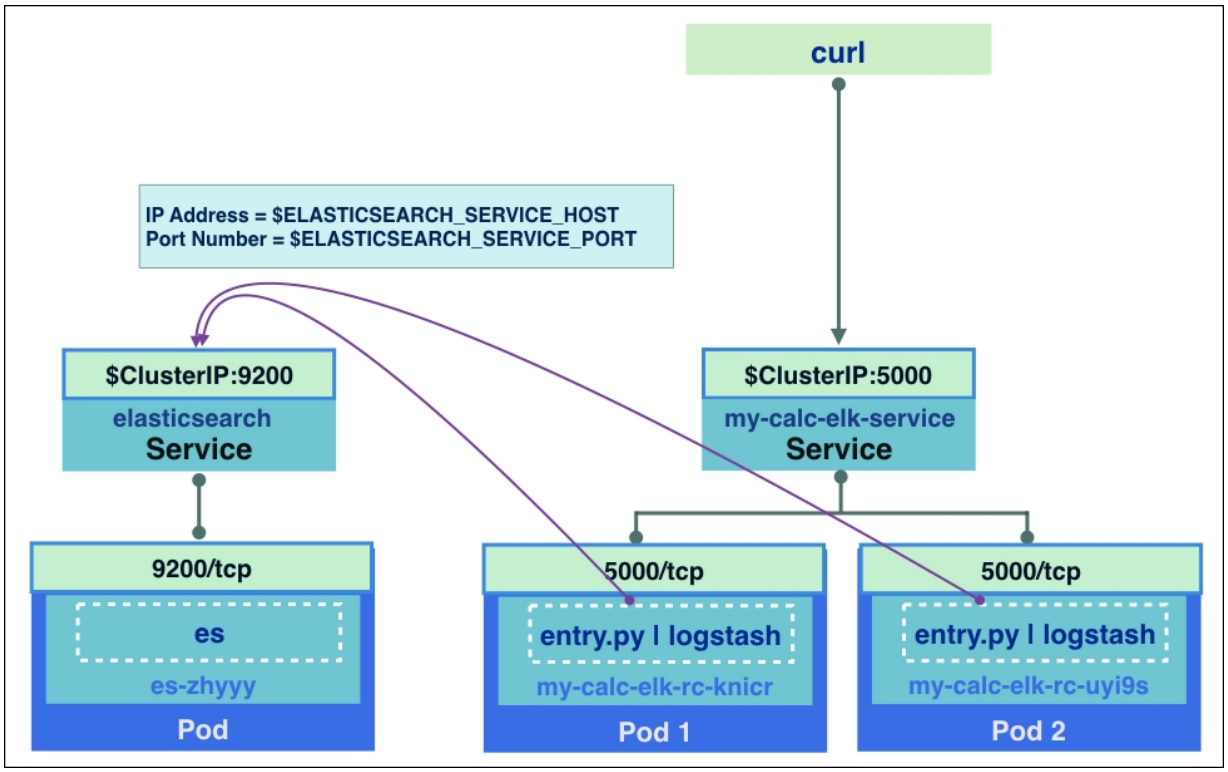


How it works…

Now, the application log is captured by Logstash; it is converted into the JSON format and then sent to Elasticsearch.

Since Logstash is bundled with an application container, there are no problems when the replication controller increases the number of replicas (pods). It captures all the application logs with no configuration changes.

All the logs will be stored in Elasticsearch as follows:



See also

This recipe covers how to integrate to the ELK stack. The centralized log collection platform is important for the Kubernetes environment. The container is easy to launch, and is destroyed by the scheduler, but it is not easy to know which node runs which pod. Check out the following recipes:

- *Working with Kubernetes logs*
- *Working with etcd log*
- The *Moving monolithic to microservices* recipe in Chapter 5, *Building a Continuous Delivery Pipeline*

Working with Kubernetes logs

Kubernetes comes with three daemon processes on master: API server, scheduler, and controller manager. Under the `/var/log` folder, there are three corresponding log files recording the logs of these processes:

Daemon on master	**Log file**	**Description**
API server	`apiserver.log`	Logs for API calls.
Scheduler	`k8s-scheduler.log`	Logs of scheduler data for any containers scheduling events
Controller manager	`controller-manager.log`	Logs for showing any events or issues relate to controller manager

On nodes, we have a `kubelet` process to handle container operations and report to the master:

Daemon on node	Log file	Description
`kubelet`	`kubelet.log`	Logs for any issues happening in container

On both masters and nodes, there is another log file named `kube-proxy.log` to record any network connection issues.

Getting ready

We will use the log collection platform ELK, which was introduced in the previous section, to collect Kubernetes logs as a centralized log platform. For the setting of ELK, we'd suggest you to review the collecting application logs section again. Before we start collecting the Kubernetes logs, knowing the data structure in the logs is important. The preceding logs are this format:

```
<log level><date> <timestamp> <indicator> <source file>:<line number>] <logs>
```

The following is an example:

```
E0328 00:46:50.870875    3189 reflector.go:227] pkg/proxy/config/api.go:60: Failed to watch *api.Endpoints: too old resource version: 45128 (45135)
```

By the heading character of the lines in the log file, we are able to know the log severity of this line:

- D: DEBUG
- I: INFO

- W: WARN
- E: ERROR
- F: FATAL

How to do it…

We will still use the `grok` filter in the `logstash` setting, as discussed in the previous section, but we might need to write our custom pattern for the `<log level><date>` pattern, which is listed at the beginning of the log line. We will create a pattern file under the current directory:

```
// list custom patterns
# cat ./patterns/k8s
LOGLEVEL    [DEFIW]
DATE        [0-9]{4}
K8SLOGLEVEL %{LOGLEVEL:level}%{DATE}
```

The preceding setting is used to split the E0328 pattern into level=E and DATE=0328. The following is an example of how to send `k8s-apiserver.log` into the ElasticSearch cluster:

```
// list config file for k8s-apiserver.log in
logstash
# cat apiserver.conf
input {
  file {
    path => "/var/log/k8s-apiserver.log"
  }
}

filter {
  grok {
    patterns_dir => ["./patterns"]
    match => { "message" => "%{K8SLOGLEVEL} %
{TIME}    %{NUMBER} %{PROG:program}:%
{POSINT:line}] %{GREEDYDATA:message}" }
  }
}

output {
  elasticsearch {
```

```
    hosts => ["_ES_IP_:_ES_PORT_"]
    index => "k8s-apiserver"
  }

  stdout { codec => rubydebug }
}
```

For the input, we will use the file plugin (https://www.elastic.co/guide/en/logstash/current/plugins-inputs-file.html), which adds the path of the `k8s-apiserver.log`. We will use `patterns_dir` in `grok` to specify the definition of our custom patterns `K8SLOGLEVEL`. The hosts' configuration in the output `elasticsearch` section should be specified to your Elasticsearch IP and port number. The following is a sample output:

```
// start logstash with config apiserver.conf
# bin/logstash -f apiserver.conf
Settings: Default pipeline workers: 1
Pipeline main started
{
       "message" => [
        [0] "E0403 15:55:24.706498    2979
errors.go:62] apiserver received an error that is
not an unversioned.Status: too old resource
version: 47419 (47437)",
        [1] "apiserver received an error that is
not an unversioned.Status: too old resource
version: 47419 (47437)"
    ],
 "@timestamp" => 2016-04-03T15:55:25.709Z,
         "level" => "E",
          "host" => "kube-master1",
       "program" => "errors.go",
          "path" => "/var/log/k8s-apiserver.log",
          "line" => "62",
     "@version" => "1"
}
{
       "message" => [
        [0] "E0403 15:55:24.706784    2979
errors.go:62] apiserver received an error that is
not an unversioned.Status: too old resource
version: 47419 (47437)",
```

```
        [1] "apiserver received an error that is
not an unversioned.Status: too old resource
version: 47419 (47437)"
    ],
    "@timestamp" => 2016-04-03T15:55:25.711Z,
         "level" => "E",
          "host" => "kube-master1",
       "program" => "errors.go",
          "path" => "/var/log/k8s-apiserver.log",
          "line" => "62",
      "@version" => "1"
}
```

It shows the current host, the log path, log level, the triggered program, and the total message. The other logs are all in the same format, so it is easy to replicate the settings. Just specify different indexes from `k8s-apiserver` to the others. Then, you are free to search the logs via Kibana, or get the other tools integrated with Elasticsearch to get notifications or so on.

See also

Check out the following recipes:

- The *Configuring master* and *Configuring nodes* recipes in Chapter 1, *Building Your Own Kubernetes*
- *Collecting application logs*
- *Monitoring master and node*

Working with etcd log

The datastore, etcd, works for saving the information of Kubernetes resources. The Kubernetes system won't be stable without robust etcd servers. If the information of a pod is lost, we will not be able to recognize it in the system, mention how to access it through the Kubernetes service, or manage it through the replication controller. In this recipe, you will learn what kind of message you may get from the etcd log and how to collect them with ELK.

Getting ready

Before we start collecting the log of etcd, we should prepare the servers of ELK. Please go back to the *Collecting application logs* recipe in Chapter 8, *Logging and Monitoring* to review how to set up ELK and study its basic usage.

On the other hand, please expose your Kubernetes service of Elasticsearch if your etcd servers are individual machines beyond the Kubernetes cluster. You can modify the service template for Elasticsearch as follows:

```
# cat es-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
  labels:
    component: elasticsearch
spec:
  type: LoadBalancer
  selector:
    component: elasticsearch
  ports:
  - name: http
    port: 9200
    nodePort: 30000
    protocol: TCP
```

```
  - name: transport
    port: 9300
    protocol: TCP
```

Then, your Logstash process on the etcd server can access it using the URL `<NODE_ENDPOINT>:30000`. Port `30000` is exposed on every node, which means that it is possible to be contacted by every nodes' endpoints.

How to do it…

On the etcd server, we have recorded the log from the daemon `etcd` at `/var/log/etcd.log`. The message is line by line in the following format:

```
<date> <time> <subpackage>: <logs>
```

It is quite straightforward to show the timestamp and the information. We can also see where the logs come from, which means we know which kind of subpackages handle this issue. Here is an example of an etcd log:

```
2016/04/4 08:43:51 etcdserver: starting server...
[version: 2.1.1, cluster version: to_be_decided]
```

After you understand the style of the message, it is time to create the Logstash configuration file:

```
# cat etcd.conf
input {
  file {
    path => "/var/log/etcd.log"
  }
}

filter {
  grok {
    match => {
        "message" => "%{DATA:date} %{TIME:time} %
{PROG:subpackage}: %{GREEDYDATA:message}"
```

```
    }
  }
}

output {
  elasticsearch {
    hosts => ["<ELASTIC_SERVICE_IP>:
<EXPOSE_PORT>"]
    index => "etcd-log"
  }

  stdout { codec => rubydebug }
}
```

In the file, we will assign the location of the etcd logfile as input data. The pattern defined in the `grok` filter simply separates the log into three parts: timestamp, the subpackage, and the message. Of course, we not only show the output on screen, but also send the data to the Elasticsearch server for further analysis:

```
// Under the directory of $LOGSTASH_HOME
# ./bin/logstash -f etcd.conf
Settings: Default pipeline workers: 1
Pipeline main started
{
    "subpackage" => "raft",
       "message" => [
        [0] "2016/04/4 08:43:53 raft: raft.node:
ce2a822cea30bfca elected leader ce2a822cea30bfca
at term 2",
        [1] "raft.node: ce2a822cea30bfca elected
leader ce2a822cea30bfca at term 2"
    ],
    "@timestamp" => 2016-04-04T11:23:
27.571Z,
          "time" => "08:43:53",
          "host" => "etcd1",
          "path" => "/var/log/etcd.log",
          "date" => "2016/04/4",
      "@version" => "1"
}
{
    "subpackage" => "etcdserver",
       "message" => [
```

```
        [0] "2016/04/4 08:43:53 etcdserver:
setting up the initial cluster version to 2.1.0",
        [1] "setting up the initial cluster
version to 2.1.0"
    ],
    "@timestamp" => 2016-04-04T11:24:09.603Z,
          "time" => "08:43:53",
          "host" => "etcd1",
          "path" => "/var/log/etcd.log",
          "date" => "2016/04/4",
      "@version" => "1"
}
```

As you can see, through Logstash, we will parse the log in different subpackage issues and at a particular time. It is a good time for you now to access the Kibana dashboard, and work with the etcd logs.

See also

Before you trace the log file of etcd, you must have your own etcd system. Take a look at the previous chapters; you will understand how to build a single node or a cluster-like etcd system. Check out the following recipes:

- The *Building datastore* recipe in Chapter 1, *Building Your Own Kubernetes*
- The *Clustering etcd* recipe in Chapter 4, *Building a High Availability Cluster*
- *Collecting application logs*
- *Working with Kubernetes logs*

Monitoring master and node

During the journey of the previous recipes, you learned how to build your own cluster, run various resources, enjoy different scenarios of usages, and even enhance cluster administration. Now comes a new level of view for your Kubernetes cluster. In this recipe, we are going to talk about monitoring. Through the monitoring tool, users not only learn about the resource consumption of workers and nodes, but also the pods. It will help us have a better efficiency in resource utilization.

Getting ready

Before we set up our monitoring cluster in the Kubernetes system, there are two main prerequisites:

- One is to update the last version of binary files, which makes sure your cluster has stable and capable functionality
- The other one is to set up the DNS server

A Kubernetes DNS server can reduce some steps and dependency for installing cluster-like pods. In here, it is easier to deploy a monitoring system in Kubernetes with a DNS server.

Tip

In Kubernetes, how does the DNS server gives assistance in large-system deployment?

The DNS server can support resolving the name of the Kubernetes service for every container. Therefore, while running a pod, we don't have to set a specific service IP for connecting to other pods. Containers in a pod just need to know the service name.

The daemon of the node `kubelet` assigns containers to the DNS server by modifying the file `/etc/resolv.conf`. Try to check the file or use the command `nslookup` for verification after you have installed the DNS server:

```
# kubectl exec <POD_NAME> [-c
<CONTAINER_NAME>] -- cat /etc/resolv.conf
// Check where the service "kubernetes" served
# kubectl exec <POD_NAME> [-c
<CONTAINER_NAME>] -- nslookup kubernetes
```

Updating Kubernetes to the latest version: 1.2.1

Updating the version of a running Kubernetes system is not troublesome. You can simply follow these steps. The procedure is similar for both master and node:

- Since we are going to upgrade every Kubernetes' binary file, stop all of the Kubernetes services before you upgrade. For instance, `service <KUBERNETES_DAEMON> stop`
- Download the latest tarball file version `1.2.1`:

```
# cd /tmp && wget
https://storage.googleapis.com/kubernetes-
release/release/v1.2.1/kubernetes.tar.gz
```

- Decompress the file in a permanent directory. We are going to use the add-on templates provided in the official source files. These templates can help to create both the DNS server and the monitoring system:

```
// Open the tarball under /opt
# tar -xvf /tmp/kubernetes.tar.gz -C /opt/
// Go further decompression for binary files
# cd /opt && tar -xvf
```

```
/opt/kubernetes/server/kubernetes-server-
linux-amd64.tar.gz
```

- Copy the new files and overwrite the old ones:

```
# cd /opt/kubernetes/server/bin/
// For master, you should copy following files
and confirm to overwrite
# cp kubectl hypercube kube-apiserver kube-
controller-manager kube-scheduler kube-proxy
/usr/local/bin
// For nodes, copy the below files
# cp kubelet kube-proxy /usr/local/bin
```

- Finally, you can start the system services. It is good to verify the version through the command line:

```
# kubectl version
Client Version: version.Info{Major:"1",
Minor:"2", GitVersion:"v1.2.1",
GitCommit:"50809107cd47a1f62da362bccefdd9e6f70
76145", GitTreeState:"clean"}
Server Version: version.Info{Major:"1",
Minor:"2", GitVersion:"v1.2.1",
GitCommit:"50809107cd47a1f62da362bccefdd9e6f70
76145", GitTreeState:"clean"}
```

As a reminder, you should update both the master and node at the same time.

Setting up the DNS server

As mentioned, we will use the official template to build up the DNS server in our Kubernetes system. There are just two steps. First, modify the templates and create the resources. Then, we need to restart the `kubelet` daemon with DNS information.

Start the server using templates

The add-on files of Kubernetes are located at `<KUBERNETES_HOME>/cluster/addons/`. According to the last step, we

can access the add-on files for DNS at `/opt/kubernetes/cluster/addons/dns`. Two template files are going to be modified and executed. Feel free to depend on the following steps:

- Copy the file from the format `.yaml.in` to the YAML file and we will edit the copied ones later:

```
# cp skydns-rc.yaml.in skydns-rc.yaml
```

Input variable	Substitute value	Example
`{{ pillar['dns_domain'] }}`	The domain of this cluster	`k8s.local`
`{{ pillar['dns_replicas'] }}`	The number of replica for this replication controller	`1`
`{{ pillar['dns_server'] }}`	The private IP of DNS server. Must also be in the CIDR of cluster	`192.168.0.2`

```
# cp skydns-svc.yaml.in skydns-svc.yaml
```

- In these two templates, replace the `pillar` variable, which is covered by double big parentheses, with the items in this table. As you know, the default service `kubernetes` will occupy the first

IP in CIDR. That's why we will use IP `192.168.0.2` for our DNS server:

```
# kubectl get svc
NAME         CLUSTER-IP    EXTERNAL-IP
PORT(S)   AGE
kubernetes   192.168.0.1   <none>
443/TCP   4d
```

- In the template for the replication controller, the file named `skydns-rc.yaml` specifies the master URL in the container `kube2sky`:

```
# cat skydns-rc.yaml
(Ignore above lines)
:
- name: kube2sky
  image:
gcr.io/google_containers/kube2sky:1.14
  resources:
    limits:
      cpu: 100m
      memory: 200Mi
    requests:
      cpu: 100m
      memory: 50Mi
  livenessProbe:
    httpGet:
      path: /healthz
      port: 8080
      scheme: HTTP
    initialDelaySeconds: 60
    timeoutSeconds: 5
    successThreshold: 1
    failureThreshold: 5
  readinessProbe:
    httpGet:
      path: /readiness
      port: 8081
      scheme: HTTP
    initialDelaySeconds: 30
    timeoutSeconds: 5
  args:
  # command = "/kube2sky"
  - --domain=k8s.local
```

```
      - --kube-master-url=<MASTER_ENDPOINT_URL>:
    <EXPOSED_PORT>
    :
    (Ignore below lines)
```

After you finish the preceding steps for modification, you can just start them using the subcommand `create`:

```
# kubectl create -f skydns-svc.yaml
service "kube-dns" created
# kubectl create -f skydns-rc.yaml
replicationcontroller "kube-dns-v11" created
```

Enable Kubernetes DNS in kubelet

Next, we have access to each node and add DNS information to the daemon `kubelet`. The tags we used for the cluster DNS are `--cluster-dns`, for assign the IP of DNS server, and `--cluster-domain`, which define the domain of Kubernetes services:

```
// For init service daemon
# cat /etc/init.d/kubernetes-node
(Ignore above lines)
:
# Start daemon.
echo $"Starting kubelet: "
        daemon $kubelet_prog \
                --api_servers=
<MASTER_ENDPOINT_URL>:<EXPOSED_PORT> \
                --v=2 \
                --cluster-dns=192.168.0.2 \
                --cluster-domain=k8s.local \
                --address=0.0.0.0 \
                --enable_server \
                --hostname_override=${hostname} \
                > ${logfile}-kubelet.log 2>&1 &
:
(Ignore below lines)
// Or, for systemd service
# cat /etc/kubernetes/kubelet
(Ignore above lines)
:
```

```
# Add your own!
KUBELET_ARGS="--cluster-dns=192.168.0.2 --cluster-
domain=k8s.local"
```

Now, it is good for you to restart either the service `kubernetes-node` or just `kubelet`! And you can enjoy the cluster with a DNS server.

How to do it…

In this section, we will work on installing a monitoring system and introducing its dashboard. This monitoring system is based on **Heapster** (https://github.com/kubernetes/heapster), a resource usage collecting and analyzing tool. Heapster communicates with kubelet to get the resource usage of both machine and container. Along with Heapster, we have **influxDB** (https://influxdata.com) for storage and **Grafana** (http://grafana.org) as the frontend dashboard, which visualizes the status of resources in several user-friendly plots.

Installing a monitoring cluster

If you have gone through the preceding section about the prerequisite DNS server, you must be very familiar with deploying the system with official add-on templates.

1. Let's check the directory `cluster-monitoring` under `<KUBERNETES_HOME>/cluster/addons`. There are different environments provided for deploying the monitoring cluster. We choose `influxdb` in this recipe for demonstration:

```
# cd /opt/kubernetes/cluster/addons/cluster-
monitoring/influxdb && ls
grafana-service.yaml      heapster-
service.yaml             influxdb-service.yaml
heapster-controller.yaml  influxdb-grafana-
controller.yaml
```

Under this directory, you can see three templates for services and two for replication controllers.

2. We will retain most of the service templates as the original ones. Because these templates define the network configurations, it is fine to use the default settings but expose Grafana service:

```
# cat heapster-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: monitoring-grafana
  namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "Grafana"
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30000
      targetPort: 3000
  selector:
    k8s-app: influxGrafana
```

As you can see, we expose Grafana service with port `30000`. This revision will allow us to access the dashboard of monitoring from browser.

3. On the other hand, the replication controller of Heapster and the one combining influxDB and Grafana require more additional editing to meet our Kubernetes system:

```
# cat influxdb-grafana-controller.yaml
(Ignored above lines)
:
- image:
gcr.io/google_containers/heapster_grafana:v2.6
.0-2
          name: grafana
          env:
          resources:
            # keep request = limit to keep
```

```
this container in guaranteed class
            limits:
              cpu: 100m
              memory: 100Mi
            requests:
              cpu: 100m
              memory: 100Mi
          env:
            # This variable is required to
setup templates in Grafana.
            - name: INFLUXDB_SERVICE_URL
              value: http://monitoring-
influxdb.kube-system:8086
            - name: GF_AUTH_BASIC_ENABLED
              value: "false"
            - name: GF_AUTH_ANONYMOUS_ENABLED
              value: "true"
            - name: GF_AUTH_ANONYMOUS_ORG_ROLE
              value: Admin
            - name: GF_SERVER_ROOT_URL
              value: /
:
(Ignored below lines)
```

For the container of Grafana, please change some environment variables. The first one is the URL of influxDB service. Since we set up the DNS server, we don't have to specify the particular IP address. But an extra-postfix domain should be added. It is because the service is created in the namespace `kube-system`. Without adding this postfix domain, DNS server cannot resolve `monitoring-influxdb` in the default namespace. Furthermore, the Grafana root URL should be changed to a single slash. Instead of the default URL, the root (`/`) makes Grafana transfer the correct webpage in the current system.

4. In the template of Heapster, we run two Heapster containers in a pod. These two containers use the same image and have similar settings, but actually, they take to different roles. We just take a look at one of them as an example of modification:

```
# cat heapster-controller.yaml
(Ignore above lines)
:
```

```
      containers:
        - image:
gcr.io/google_containers/heapster:v1.0.2
          name: heapster
          resources:
            limits:
              cpu: 100m
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          command:
            - /heapster
            - --source=kubernetes:
<MASTER_ENDPOINT_URL>:<EXPOSED_PORT>?
inClusterConfig=false
            - --
sink=influxdb:http://monitoring-influxdb.kube-
system:8086
            - --metric_resolution=60s
:
(Ignore below lines)
```

At the beginning, remove all double-big-parentheses lines. These lines will cause creation error, since they cannot be parsed or considered in the YAML format. Still, there are two input variables that need to be replaced to possible values. Replace `{{ metrics_memory }}` and `{{ eventer_memory }}` to `200Mi`. The value 200MiB is a guaranteed amount of memory that the container could have. And please change the usage for Kubernetes source. We specify the full access URL and port, and disable `ClusterConfig` for refraining authentication. Remember to adjust on both the `heapster` and `eventer` containers.

5. Now you can create these items with simple commands:

```
# kubectl create -f influxdb-service.yaml
service "monitoring-influxdb" created
# kubectl create -f grafana-service.yaml
You have exposed your service on an external
port on all nodes in your
cluster.  If you want to expose this service
to the external internet, you may
```

```
need to set up firewall rules for the service
port(s) (tcp:30000) to serve traffic.

See http://releases.k8s.io/release-
1.2/docs/user-guide/services-firewalls.md for
more details.
service "monitoring-grafana" created
# kubectl create -f heapster-service.yaml
service "heapster" created
# kubectl create -f influxdb-grafana-
controller.yaml
replicationcontroller "monitoring-influxdb-
grafana-v3" created
// Because heapster requires the DB server and
service to be ready, schedule it as the last
one to be created.
# kubectl create -f heapster-controller.yaml
replicationcontroller "heapster-v1.0.2"
created
```

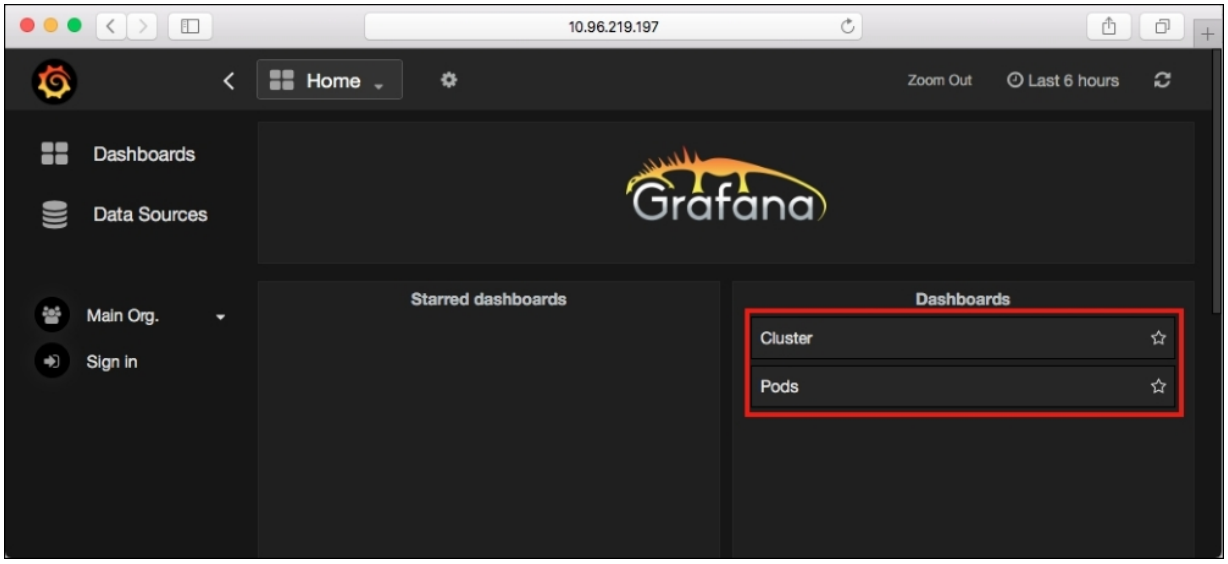
6. Check your Kubernetes resources at namespace `kube-system`:

```
# kubectl get svc --namespace=kube-system
NAME                    CLUSTER-IP
EXTERNAL-IP   PORT(S)              AGE
heapster                192.168.135.85    <none>
80/TCP              12m
kube-dns                192.168.0.2       <none>
53/UDP,53/TCP       15h
monitoring-grafana      192.168.84.223    nodes
80/TCP              12m
monitoring-influxdb     192.168.116.162   <none>
8083/TCP,8086/TCP   13m
# kubectl get pod --namespace=kube-system
NAME                                    READY
STATUS    RESTARTS   AGE
heapster-v1.0.2-r6oc8                   2/2
Running   0          4m
kube-dns-v11-k81cm                      4/4
Running   0          15h
monitoring-influxdb-grafana-v3-d6pcb    2/2
Running   0          12m
```

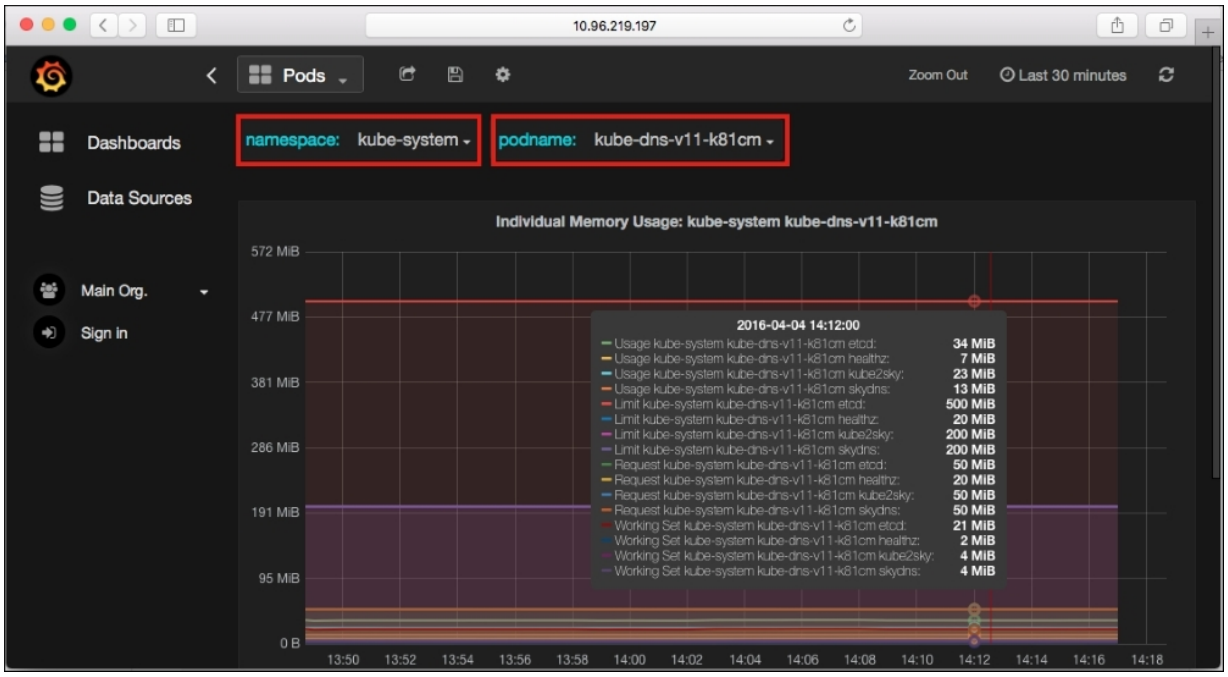
Congratulations! Once you have all the pods in a ready state, let's check the monitoring dashboard.

Introducing the Grafana dashboard

At this moment, the Grafana dashboard is available through nodes' endpoints. Please make sure the node's firewall or security group on AWS has opened port `30000` to your local subnet. Take a look at the dashboard using a browser. Type `<NODE_ENDPOINT>:30000` in your URL search bar:



In the default settings, we have two dashboards **Cluster** and **Pods**. The **Cluster** board covers nodes' resource utilization, such as CPU, memory, network transaction, and storage. The **Pods** dashboard has similar plots for each pod and you can watch each container in a pod.

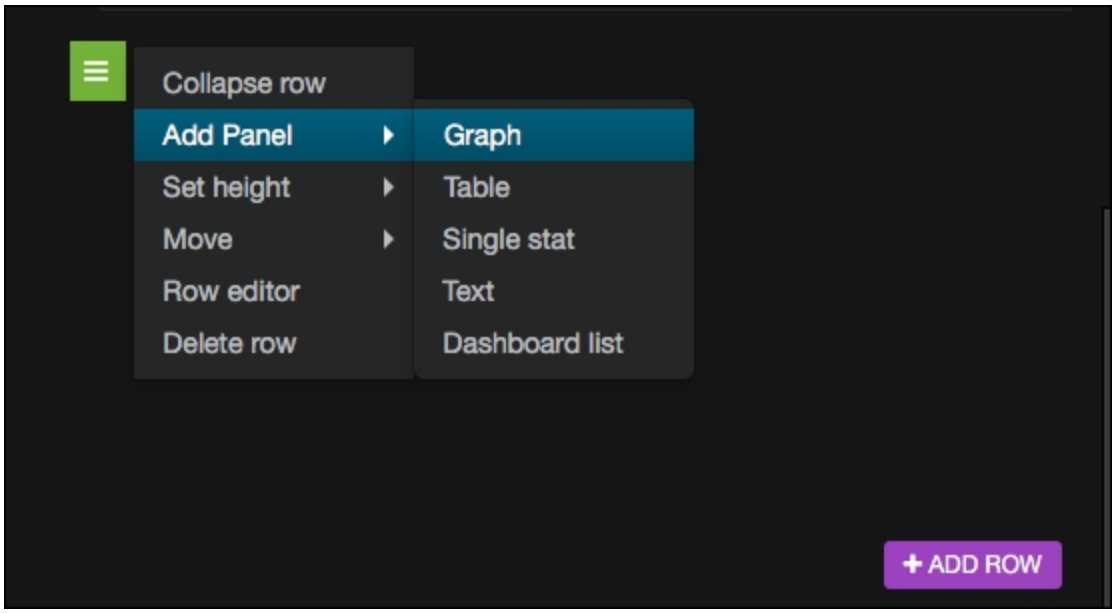


As the preceding images show, for example, we can observe the memory utilization of individual containers in the pod `kube-dns-v11`, which is the cluster of the DNS server. The purple lines in the middle just indicate the limitation we set to the containers `skydns` and `kube2sky`.

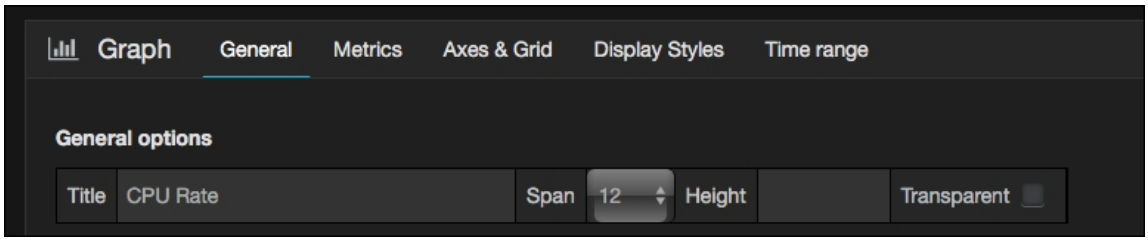
Creating a new metric to monitor pods

There are several metrics for monitoring offered by Heapster (https://github.com/kubernetes/heapster/blob/master/docs/storage-schema.md). We are going to show you how to create a customized panel by yourself. Please take the following steps as a reference:

1. Go to the **Pods** dashboard and click on **ADD ROW** at the bottom of the webpage. A green button will show up on the left-hand side. Choose to add a graph panel.



2. First, give your panel a name. For example, CPU Rate. We would like to create one showing the rate of CPU's utility:

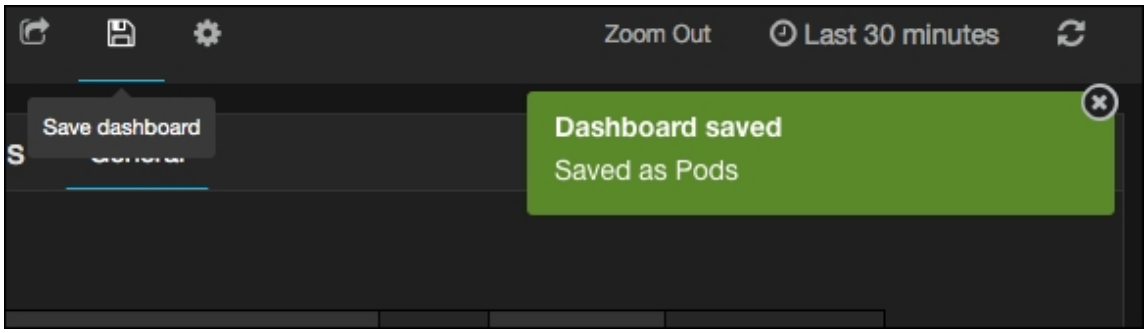


3. Set up the parameters in the query as shown in the following screenshot:



- `FROM`: For this parameter input `cpu/usage_rate`
- `WHERE`: For this parameter set `type = pod_container`
- `AND`: Set this parameter with the `namespace_name=$namespace`, `pod_name= $podname` value
- `GROUP BY`: Enter `tag(container_name)` for this parameter
- `ALIAS BY`: For this parameter input `$tag_container_name`

4. Good job! You can now save the pod by clicking on the icon at the top:



Just try to discover more functionality of the Grafana dashboard and the Heapster monitoring tool. You will get more details about your system, services, and containers through the information from the monitoring system.

See also

This recipe informs you how to monitor your master node and nodes in the Kubernetes system. However, it is wise to study the recipes about the main components and daemons. You can get more of an idea about the view of working processes and resource usage. Moreover, since we have worked with several services to build our monitoring system, reviewing the recipe about the Kubernetes services again will give you a clear idea about how you can build up this monitoring system.

- The *Creating an overlay network*, *Configuring master*, and *Configuring nodes* recipes in Chapter 1, *Building Your Own Kubernetes*
- The *Working with services* recipe in Chapter 2, *Walking through Kubernetes Concepts*

Kubernetes is a project which keeps moving forward and upgrading at a fast speed. The recommended way for catching up is to check out new features on its official website: http://kubernetes.io. Also, you can always get new Kubernetes on GitHub: https://github.com/kubernetes/kubernetes/releases. Keeping your Kubernetes system up to date, and learning new features practically, is the best method to access the Kubernetes technology continuously.

Index

A

- alternatives, private Docker registry
    - Docker Trusted Registry / Docker Trusted Registry
    - Nexus Repository Manager / Nexus Repository Manager
    - Amazon EC2 Container Registry / Amazon EC2 Container Registry
- Amazon EC2 Container Registry
    - about / Amazon EC2 Container Registry
    - reference / Amazon EC2 Container Registry
- Amazon Web Services (AWS) / Amazon EC2 Container Registry
- application
    - managing, AWS OpsWorks used / Managing applications using AWS OpsWorks, Getting ready, How to do it…
- application layers
    - creating / Creating application layers, How to do it…
- application logs
    - collecting / Collecting application logs, Getting ready
    - working / How it works…, See also
- application management
    - OpsWorks, creating for / Creating OpsWorks for application management
- architecture, Kubernetes

    - exploring / Exploring architecture, Getting ready
    - Kubernetes master / Kubernetes master
    - Kubernetes node / Kubernetes node
- authentication
    - about / Authentication and authorization
    - enabling, for API call / Enabling authentication for an API call
- authentication, enabling for API call
    - API requests, sending on etcd host / Basic authentication of etcd
    - in Kubernetes master / Basic authentication of the Kubernetes master
- authorization
    - about / Authentication and authorization
    - using / Making use of user authorization
- auto startup script, datastore
    - about / Auto startup script
    - startup script (systemd) / Startup script (systemd)
    - startup script (init) / Startup script (init)
- Availability Zones (AZ) / How to do it…
- AWS
    - Kubernetes infrastructure, building / Building the Kubernetes infrastructure in AWS, Getting ready, How to do it…
    - URL / Getting ready
    - VPC, creating / VPC and subnets
    - subnets, creating / VPC and subnets
- AWS CloudFormation

    - using, for fast provisioning / Using AWS CloudFormation for fast provisioning, Getting ready, How to do it…
    - network infrastructure, creating / Creating a network infrastructure
- awsElasticBlockStore volume
    - about / awsElasticBlockStore
- AWS OpsWorks
    - used, for managing application / Managing applications using AWS OpsWorks, Getting ready, How to do it…

C

- Ceph RADOS Block Device
    - reference / rbd
- Chef
    - URL / Managing applications using AWS OpsWorks, Auto-deploying Kubernetes through Chef recipes
- Classless Inter-Domain Routing (CIDR) / Getting ready
- ClusterIP service
    - about / Creating a service in a different type
- Command Line Interface (CLI) / An overview of Kubernetes control
- configuration files
    - working with / Working with configuration files
    - YAML / YAML
    - JSON / JSON, How to do it…
    - pods / Pods

- container, running in Kubernetes
    - about / Run your first container in Kubernetes, Getting ready
    - HTTP server (nginx), running / Running an HTTP server (nginx)
    - port, exposing for external access / Exposing the port for external access
    - application, stopping / Stopping the application
    - working / How it works…
- container nginx
    - URL / Run your first container in Kubernetes
- container ports
    - forwarding / Forwarding container ports, Getting ready
    - container-to-container communications / Container-to-container communications
    - pod-to-pod communications / Pod-to-pod communications
    - pod-to-service communications / Pod-to-service communications
    - external-to-internal communications / External-to-internal communications, See also
- containers
    - scaling / Scaling your containers, Getting ready, How to do it…, How it works…, See also
    - references / Getting ready
    - using / Ensuring flexible usage of your containers, Getting ready
    - pod, as job / Pod as a job
    - pod, as daemon set / Pod as a daemon set
- Continuous Delivery (CD)

    - reference / Integrating with Jenkins
    - about / Setting up the Continuous Delivery pipeline
- Continuous Delivery pipeline
    - setting up / Setting up the Continuous Delivery pipeline, Getting ready, How to do it…, How it works…, There's more…
- Continuous Integration (CI)
    - reference / Integrating with Jenkins

D

- daemon set
    - running, on specific nodes / Running the daemon set only on specific nodes, See also
- datastore, building
    - about / Building datastore
    - Red Hat Enterprise Linux 7 / Red Hat Enterprise Linux 7 or CentOS 7
    - CentOS 7 / Red Hat Enterprise Linux 7 or CentOS 7
    - Ubuntu Linux 15.10 Wily Werewolf / Ubuntu Linux 15.10 Wily Werewolf
    - Linux / Other Linux
    - etcd, running / How it works…
    - auto startup script / Auto startup script
    - configuration / Configuration
- DNS server, Kubernetes system
    - setting up / Setting up the DNS server
    - starting, templates used / Start the server using templates
    - enabling, in kubelet / Enable Kubernetes DNS in kubelet

- Docker
  - reference / Kubernetes node
  - URL / Preparing your environment
- docker build command
  - about / Docker build
- Dockerfile
  - about / Dockerfile
- Docker Hub
  - URL / Getting ready, Getting ready
- Docker installation
  - reference / Kubernetes nodes
- Docker Trusted Registry
  - about / Docker Trusted Registry
  - reference / Docker Trusted Registry
- downwardAPI volume
  - about / downwardAPI

E

- Elasticsearch
  - reference / Elasticsearch
  - about / How to do it…
- ELB

    - creating / Creating ELB and its security groups
    - security groups / Creating ELB and its security groups
- ELK (Elasticsearch, Logstash, and Kibana) / Collecting application logs
- emptyDir volume
    - about / emptyDir
- etcd
    - about / etcd, Clustering etcd
    - reference / etcd
    - storing / Clustering etcd, Getting ready
    - static mechanism / Static
- etcd discovery
    - using / etcd discovery
- etcd discovery service
    - reference / etcd discovery
- etcd log
    - working with / Working with etcd log, Getting ready, How to do it…

F

- Flask
    - reference / Getting ready
- Flocker
    - about / flocker
    - reference / flocker

- flocker volume
  - about / flocker
- Frontend WebUI
  - about / Frontend WebUI, How it works…
  - working / Frontend WebUI

G

- gcePersistentDisk volume
  - about / gcePersistentDisk
- GitHub
  - URL / Getting ready
- gitRepo volume
  - about / gitRepo
- GlusterFS
  - about / glusterfs
  - URL / glusterfs
- glusterfs volume
  - about / glusterfs
- Google Container Engine
  - about / Preparing your environment
- Grafana
  - URL / How to do it…

- Grafana dashboard
  - about / Introducing the Grafana dashboard
  - new metric, creating to monitor pods / Creating a new metric to monitor pods

H

- Heapster
  - URL / How to do it…
- hostPath volume
  - about / hostPath
- HTTP Request Plugin
  - reference / How it works…

I

- IAM role
  - adjusting / Adjusting the IAM role
- Identity Access Management (IAM) / Amazon EC2 Container Registry
- influxDB
  - URL / How to do it…
- installation, master
  - about / Installation
  - Red Hat Enterprise Linux 7 / CentOS 7 or Red Hat Enterprise Linux 7

    - CentOS 7 / CentOS 7 or Red Hat Enterprise Linux 7
    - daemon dependency, adding / Adding daemon dependency
    - other Linux options / Other Linux options
    - verification / Verification
- installation, nodes
    - CentOS 7 / CentOS 7 or Red Hat Enterprise Linux 7
    - Red Hat Enterprise Linux 7 / CentOS 7 or Red Hat Enterprise Linux 7
    - other Linux options / Other Linux options
    - verification / Verification
- instances
    - starting / Starting the instances
- Intrinsic Functions
    - about / Getting ready
- iscsi volume
    - about / iscsi

J

- Jenkins integration
    - about / Integrating with Jenkins
    - Jenkins server, installing / Install a Jenkins server which can build a Docker program, How to do it…
    - Jenkins project, creating / Create your Jenkins project
    - program testing, running / Run a program testing
    - program, deploying / Deploying a program, How it works…, There's more…
- job

    - creating, with multiple pods / Creating a job with multiple pods running
- JSON
    - URL / JSON

K

- kernel version
    - reference / Operating system
- Kibana
    - about / Kibana
    - URL / Kibana
- kubeconfig
    - advanced setting / Advanced settings in kubeconfig, Getting ready
    - about / Advanced settings in kubeconfig, Getting ready
    - new credential, setting up / Setting a new credential
    - reference link / Setting a new credential
    - new cluster, setting up / Setting a new cluster
    - new credential, reference link / Setting a new cluster
    - current context, setting up / Setting and changing the current context
    - current context, changing / Setting and changing the current context
    - cleaning up / Cleaning up kubeconfig
- kubectl command / How it works…
    - reference / How it works…
- kubelet binary file

    - reference / Setting up kubelet in master
- Kubernetes
    - about / Introduction, Getting ready
    - architecture / Exploring architecture, Getting ready
    - environment, preparing / Preparing your environment, Getting ready
    - auto-deploying, through Chef recipes / Auto-deploying Kubernetes through Chef recipes
    - reference link / Basic authentication of the Kubernetes master
    - replication controller / Kubernetes replication controller and service
    - service / Kubernetes replication controller and service
    - updating, to latest versionTopicn 1.2.1 / Updating Kubernetes to the latest version: 1.2.1
- Kubernetes 1.2
    - reference link / Getting ready
- Kubernetes control
    - overview / An overview of Kubernetes control, Getting ready, How it works…
- Kubernetes environment
    - hardware resource / Hardware resource
    - operating system / Operating system, How to do it…
    - Kubernetes master / Kubernetes master
    - Kubernetes nodes / Kubernetes nodes
    - etcd / etcd
- Kubernetes infrastructure
    - building, in AWS / Building the Kubernetes infrastructure in AWS, Getting ready, How to do it…

- Kubernetes logs
  - working with / Working with Kubernetes logs, How to do it…
- Kubernetes master
  - about / Kubernetes master
  - functionalities / Kubernetes master
  - kube-apiserver / Kubernetes master
  - kube-controller-manager / Kubernetes master
  - kube-scheduler / Kubernetes master
  - API server (kube-apiserver) / API server (kube-apiserver)
  - scheduler (kube-scheduler) / Scheduler (kube-scheduler)
  - controller manager (kube-controller-manager) / Controller manager (kube-controller-manager)
  - Command Line Interface (kubectl) / Command Line Interface (kubectl)
- Kubernetes node
  - about / Kubernetes node
  - kubelet / kubelet
  - Proxy (kube-proxy) / Proxy (kube-proxy), How it works…
- Kubernetes Plugin
  - reference / There's more…
- Kubernetes RPMs
  - reference / CentOS 7 or Red Hat Enterprise Linux 7

L

- labels
  - about / Working with labels and selectors

    - working with / Working with labels and selectors, Getting ready, How to do it…
- label selectors
    - about / Working with labels and selectors
    - equality-based label selector / Working with labels and selectors
    - set-based label selector / Working with labels and selectors
    - empty label selectors / Working with labels and selectors
    - null label selector / Working with labels and selectors
    - used, for linking service with replication controller / Linking service with a replication controller by using label selectors, There's more…
- Linux
    - about / Other Linux
    - binary, downloading / Download a binary
    - user, creating / Creating a user
    - etcd, installing / Install etcd
- live containers
    - updating / Updating live containers, How to do it…, How it works…, There's more…
- LoadBalancer service
    - about / Creating a service in a different type
- Logstash
    - about / Logstash
    - reference / Logstash

M

- master

    - configuring / Configuring master, How to do it…
- master and node
    - monitoring / Monitoring master and node
- microservices
    - about / Microservices
    - working / Microservices
- Model-View-Controller (MVC) / Moving monolithic to microservices
- monitoring cluster
    - installing / Installing a monitoring cluster
- monolithic
    - moving, to microservices / Moving monolithic to microservices, Getting ready, How to do it…
- multiple masters
    - building / Building multiple masters, Getting ready
    - multiple master nodes, preparing / Preparing multiple master nodes
    - kubelet, setting up / Setting up kubelet in master
    - configuration files, preparing / Getting the configuration files ready
    - kubelet service, starting / Starting the kubelet service and turning daemons on!, How it works…
    - daemons, enabling / Starting the kubelet service and turning daemons on!, How it works…

N

- names

    - working with / Working with names, How to do it…, How it works…
- namespaces
    - working with / Working with namespaces, Getting ready, How to do it…
    - default namespace, changing / Changing the default namespace
    - deleting / Deleting a namespace, There's more…
    - LimitRange, deleting / Deleting LimitRange
- NAT gateway
    - reference / How to do it…
- Network File System(NFS) / nfs
- Nexus Repository Manager
    - about / Nexus Repository Manager
    - reference / Nexus Repository Manager
- nfs volume
    - about / nfs
- nginx
    - URL / Getting ready
- NodePort service
    - about / Creating a service in a different type
- nodes
    - about / Configuring nodes
    - configuring / Configuring nodes, Getting ready
    - installation / Installation

    - resources, setting / Setting resource in nodes, Getting ready
    - capacity, managing / Managing node capacity
    - computing resources, managing in pod / Managing computing resources in a pod, See also

O

- OpsWorks
    - creating, for application management / Creating OpsWorks for application management
- OpsWorks instance
    - about / The OpsWorks instance, How it works…
- OpsWorks layer
    - about / The OpsWorks layer
    - creating / The OpsWorks layer
- OpsWorks stack
    - about / The OpsWorks stack
    - creating / Creating an OpsWorks stack
- overlay network
    - about / Overlay network
    - flannel / Flannel
    - creating / Creating an overlay network
    - installing / Installation
    - CentOS 7 / CentOS 7 or Red Hat Enterprise Linux 7
    - Red Hat Enterprise Linux 7 / CentOS 7 or Red Hat Enterprise Linux 7
    - Linux options / Other Linux options

    - flannel networking configuration / Flannel networking configuration
    - integrating, with Docker / Integrating with Docker, How it works…

P

- PersistentVolume (PV)
    - about / There's more…, PersistentVolume
- pods
    - about / Working with pods
    - working with / Working with pods, Getting ready, How to do it…, How it works…, See also
- private Docker registry
    - working with / Working with the private Docker registry, How to do it…
    - reference / Getting ready
    - alternatives / Alternatives

Q

- qperf
    - reference / Creating an overlay network

R

- rbd volume
    - about / rbd

- recipes, for etcd
  - about / Recipes for etcd
- recipes, for Kubernetes master
  - about / Recipes for the Kubernetes master
- recipes, for Kubernetes node
  - about / Recipes for the Kubernetes node
- Remote Procedure Call (RPC) / Moving monolithic to microservices
- replication controller
  - working with / Working with a replication controller, Getting ready
  - creating / Creating a replication controller
  - information, obtaining / Getting information of a replication controller
  - configuration, changing / Changing the configuration of a replication controller
  - removing / Removing a replication controller, How it works…
- replication controllers
  - about / Replication controllers
- resource
  - setting, in nodes / Setting resource in nodes, Getting ready
- RESTful API
  - working with / Working with a RESTful API, Getting ready, How to do it…, See also

- rkt
    - reference / Kubernetes node

S

- secrets
    - about / Working with secrets
    - working with / Working with secrets
    - types / Working with secrets
    - Opaque / Working with secrets
    - service account token / Working with secrets
    - Docker authentication / Working with secrets
    - creating / Creating a secret
    - picking up, in container / Picking up secret in the container
    - deleting / Deleting a secret, There's more…
- selectors
    - working with / Working with labels and selectors, Getting ready, How to do it…
    - about / Working with labels and selectors
- services
    - working with / Working with services, How to do it…
    - creating, for different resources / Creating services for different resources
    - creating, for pod / Creating a service for a pod
    - creating, for replication controller / Creating a service for the replication controller and adding an external IP
    - external IP, adding / Creating a service for the replication controller and adding an external IP
    - no-selector service, creating for endpoint / Creating a no-selector service for an endpoint

    - creating with session affinity, based on another service / Creating a service with session affinity based on another service
    - creating, in different type / Creating a service in a different type
    - creating, as ClusterIP type / Creating a service in a different type
    - creating, as NodePort type / Creating a service in a different type
    - creating, as LoadBalancer type / Creating a service in a different type
    - creating, in NodePort type / Creating a service in NodePort type
    - deleting / Deleting a service
    - about / Replication controllers, Services
- single point of failure (SPOF) / How to do it…
- stack configuration, for custom recipe
    - about / Stack configuration for custom recipes
- startup script
    - about / Startup script
- subnets
    - creating / VPC and subnets
    - Internet Gateway (IGW) / Internet Gateway and NAT
    - Network Address Translation (NAT) / Internet Gateway and NAT
    - route table, associating / Route Table
    - security group, creating / Security group, How it works…
- Swagger
    - reference link / How it works…
- swagger

    - URL / How to do it…

T

- Time to Live (TTL) / How it works…

U

- user authorization
    - flags / Making use of user authorization

V

- Vagrant
    - URL / Preparing your environment
- Virtual Private Cloud (VPC) / How to do it…
- volumes
    - working with / Working with volumes, How to do it…
    - emptyDir / emptyDir
    - hostPath / hostPath
    - nfs / nfs
    - glusterfs / glusterfs
    - iscsi / iscsi
    - flocker / flocker
    - rbd / rbd
    - gitRepo / gitRepo
    - awsElasticBlockStore / awsElasticBlockStore
    - gcePersistentDisk / gcePersistentDisk
    - downwardAPI / downwardAPI
    - PersistentVolume (PV) / PersistentVolume

W

- WebUI
    - exploring / How to do it…, How it works…, See also

Y

- YAML
    - URL / YAML