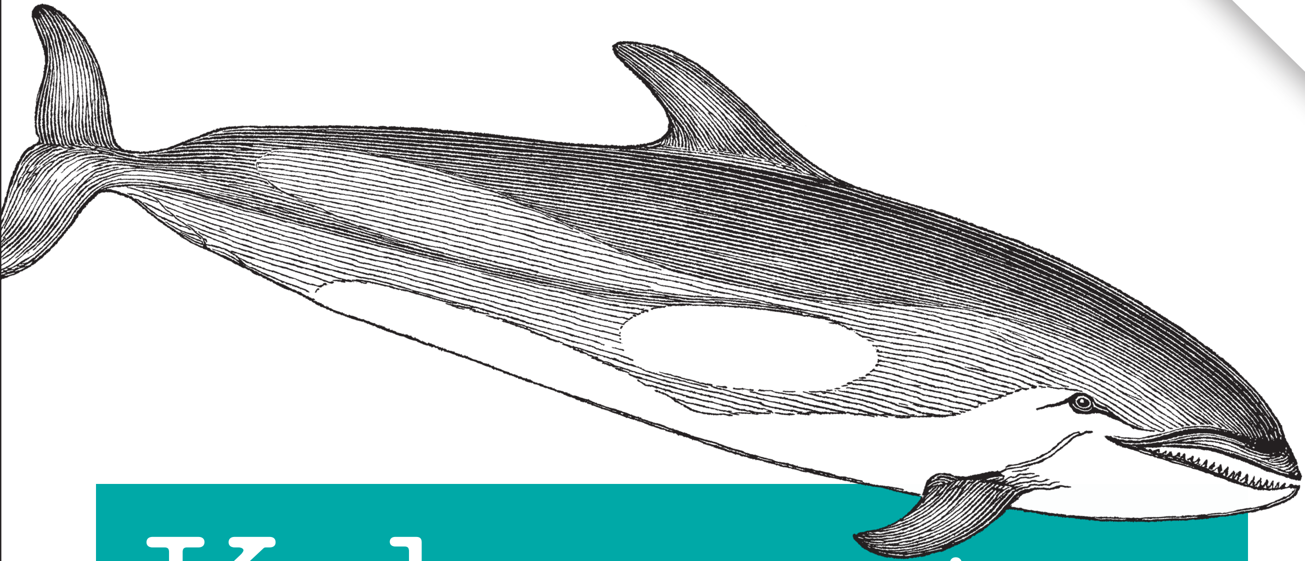
O'REILLY®

Compliments of CoreOS



Kubernetes Up & Running

DIVE INTO THE FUTURE OF INFRASTRUCTURE

PREVIEW EDITION

Kelsey Hightower

This Preview Edition of *Kubernetes: Up and Running*, *Chapters 1 and 2*, is a work in progress. The final book is currently scheduled for release in January 2016 and will be available at *oreilly.com* and other retailers once it is published.

Kubernetes: Up and Running

Dive into the Future of Infrastructure

Kelsey Hightower

Beijing · Boston · Farnham · Sebastopol · Tokyo

O'REILLY®

Kubernetes: Up and Running
by Kelsey Hightower

Copyright © 2015 Kelsey Hightower. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

Editor: Brian Anderson

See *http://www.oreilly.com/catalog/errata.csp?isbn=0636920039426* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Kubernetes: Up and Running*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-92477-8

LSI

For Klarissa and Kelis, who keep me sane. And for my Mom, who taught me a strong work ethic and how to rise above all odds. —Kelsey Hightower

Table of Contents

Preface

Containers have changed the way applications are packaged, distributed, and deployed in the modern datacenter. Containers provide the prefect abstraction for complex applications in the form of an image, which bundle applications, along with their dependencies, into a single artifact that's easy to distribute and run under a container runtime engine such as Docker or rkt.

Containers offer a lighter, more agile alternative to virtual machines for isolation between applications, and raises the bar in terms of performance, resource utilization, and portability between platforms. The ease of building and running containers has led to very high density application deployments, which in turn has driven a need for more robust tools for container management.

This book covers Kubernetes. Kubernetes was written by Google to act as a living document chronicling the lessons Google has learned from the last 10 years of running containers at scale. As a result, Kubernetes is the world's premier container management system, and enables robust patterns to deal with the wonderful problems containers are creating. I wrote this book for Developers and System administrators who are looking to manage containers at scale using proven distributed computing concepts while also leveraging modern advances in datacenter automation.

My goal with this book is to document the Kubernetes platform and provide practical examples of how to deploy and manage applications with Kubernetes.

This book does not make many assumptions regarding your understanding of containers, cluster management, and scheduling applications across a large cluster of machines. Instead, this book will guide you from building your first container to standing up a complete Kubernetes environment. Along this journey you will be introduced to every component in a Kubernetes cluster, and you will be shown how to leverage each of these pieces in designing highly efficient patterns for application deployments.

This book is also meant to serve as a reference guide for Kubernetes including the various APIs and components that make up the platform.

CHAPTER 1

Introduction

Kubernetes is an open source automation framework for deploying, managing, and scaling applications. It is the essence of an internal Google project known as Borg [1], infused with the lessons learned from over a decade of experience managing applications with Borg (and other internal frameworks) at scale.

Google scale.

But it is said that 99% of the world will never reach Google scale, and this raises the question: "Why should I care about Kubernetes?"

One word: Efficiency.

Efficiency can be measured by the ratio of the useful work performed by a machine or process to the total amount of energy spent doing so. When it comes to deploying and managing applications many of the tools and processes available are not what I would call efficient. When discussing efficiency it's often helpful to think of the cost of running a server, and the human cost required to manage it.

Running a server incurs a cost based on power usage, cooling requirements, data center space, and raw compute power. Once a server is racked and powered on(or clicked and spun-up), the meter literally starts running. Any idle CPU time is money wasted. Thus, it becomes part of the system administrator's responsibilities to keep utilization at acceptable (ie high) levels, which requires ongoing management. This is where containers and the Kubernetes workflow come in. Kubernetes provides tools

1 Large-scale cluster management at Google with Borg: *http://research.google.com/pubs/pub43438.html*

which automate the distribution of applications across a cluster of machines, ensuring higher levels of utilization than what is possible with traditional tooling.

Once applications are deployed, humans are often employed to keep an eye on things, and hold the responsibility of responding to failures, managing application configurations, performing updates, and monitoring. Many of these tasks are handled using a collection of unrelated tools that lack synergy thus requiring one-off glue utilities to fill the gaps. Kubernetes provides a common API and self-healing framework which automatically handles machine failures and streamlines application deployments, logging, and monitoring.

Why are things so inefficient?

Think about the foundation on which many automation tools are built. Most tools stem from the days of the runbook. Runbooks held the exact details on how to deploy and configure an application on a target machine. Administrators would follow runbooks blindly and only after costly outages would runbooks be updated in an attempt to prevent future outages. But no matter how large the runbooks grew, the outages never stopped. Turns out the critical flaw in the system was the humans.

See, people make mistakes. We make typos, fall asleep, or flat out skip a step or two. If only there was a way to remove the human element from the deployment process.

Enter deployment scripts.

Oh those were the good old days, we would write scripts for everything, which eventually made runbooks obsolete. If you wanted to deploy an application or update a configuration file, you ran a shell script on a specific machine. If you wanted to get *really* fancy you could leverage SSH in a for loop and deploy an application to multiple systems at a time.

Scripting application deployments started a movement known to some as Infrastructure as Code. This era of automation spawned a new class of management tools that I like to call scripting frameworks, which the industry at large calls configuration management. These configuration management systems provide a common set of reusable code that help people manage machines and the applications deployed to them. Configuration management moved the industry to faster application deployments and fewer mistakes. There was only one problem: software started eating the world.

Even as the ability to deploy applications got faster, the efficiency of doing so did not improve very much. We exchanged runbooks and deployment meetings for Infrastructure as Code where you write software to deploy software. Which also means you need to follow software development processes for application management code.

The other issue that is not as obvious to many is that configuration management, like the runbooks of yore, treat machines as first class citizens. "Applications are things that run on machines", says Config Management. "And machines belong to Applications", it states in redundant affirmation. The strong coupling between applications and machines has caused tools based on imperative scripting models to hit their maximum level of efficiency, especially compared to modern, robust, and scalable decoupled approaches.

Kubernetes Features

Kubernetes centers around a common API for deploying all types of software ranging from web applications, batch jobs, and databases. This common API is based on a declarative set of APIs and cluster configuration objects that allow you to express a desired state for your cluster.

Rather than manually deploying applications to specific servers, you describe the number of application instances that must be running at a given time. Kubernetes will perform the necessary actions to enforce the desired state. For example, if you declare 5 instances of your web application must be running at all times, and one of the nodes running an instance of the web application fails, Kubernetes will automatically reschedule the application on to another node.

In addition to application scheduling, Kubernetes helps automate application configuration in the form of service discovery and secrets. Kubernetes keeps a global view of the entire cluster, which means once applications are deployed Kubernetes has the ability to track them, even in the event they are re-scheduled due to node failure. This service information is exposed to other apps through environment variables and DNS, making it easy for both cluster native and traditional applications to locate and communicate with other services running within the cluster.

Kubernetes also provides a set of APIs that allows for custom deployment workflows such as rolling updates, canary deploys, and blue-green deployments.

Kubernetes Design Overview

Kubernetes aims to decouple applications from machines by leveraging the foundations of distributed computing and application containers. At a high level Kubernetes sits on top of a cluster of machines and provides an abstraction of a single machine.

Concepts

Clusters

Clusters are the set of compute, storage, and network resources where pods are deployed, managed, and scaled. Clusters are made of nodes connected via a “flat” network, in which each node and pod can communicate with each other. A typical Kubernetes cluster size ranges from 1 - 200 nodes, and it’s common to have more than one Kubernetes cluster in a given data center based on node count and service SLAs.

Pods

Pods are a colocated group of application containers that share volumes and a networking stack. Pods are the smallest units that can be deployed within a Kubernetes cluster. They are used for run once jobs, can be deployed individually, but long running applications, such as web services, should be deployed and managed by a replication controller.

Replication Controllers

Replication Controllers ensure a specific number of pods, based on a template, are running at any given time. Replication Controllers manage pods based on labels and status updates.

Services

Services deliver cluster wide service discovery and basic load balancing by providing a persistent name, address, or port for pods with a common set of labels.

Labels

Labels are used to organize and select groups of objects, such as pods, based on key/value pairs.

The Kubernetes Control Plane

The control plane is made up of a collection of components that work together to provide a unified view of the cluster.

etcd

etcd is a distributed, consistent key-value store for shared configuration and service discovery, with a focus on being: simple, secure, fast, and reliable. etcd uses the Raft consensus algorithm to achieve fault-tolerance and high-availability. etcd provides the ability to "watch" for changes, which allows for fast coordination between Kubernetes components. All persistent cluster state is stored in etcd.

Kubernetes API Server

The apiserver is responsible for serving the Kubernetes API and proxying cluster components such as the Kubernetes web UI. The apiserver exposes a REST interface that processes operations such as creating pods and services, and updating the corresponding objects in etcd. The apiserver is the only Kubernetes component that talks directly to etcd.

Scheduler

The scheduler watches the apiserver for unscheduled pods and schedules them onto healthy nodes based on resource requirements.

Controller Manager

There are other cluster-level functions such as managing service end-points, which is handled by the endpoints controller, and node lifecycle management which is handled by the node controller. When it comes to pods, replication controllers provide the ability to scale pods across a fleet of machines, and ensure the desired number of pods are always running.

Each of these controllers currently live in a single process called the Controller Manager.

The Kubernetes Node

The Kubernetes node runs all the components necessary for running application containers and load balancing service end-points. Nodes are also responsible for reporting resource utilization and status information to the API server.

Docker

Docker, the container runtime engine, runs on every node and handles downloading and running containers. Docker is controlled locally via its API by the Kubelet.

Kubelet

Each node runs the Kubelet, which is responsible for node registration, and management of pods. The Kubelet watches the Kubernetes API server for pods to create as scheduled by the Scheduler, and pods to delete based on cluster events. The Kubelet also handles reporting resource utilization, and health status information for a specific node and the pods it's running.

Proxy

Each node also runs a simple network proxy with support for TCP and UDP stream forwarding across a set of pods as defined in the Kubernetes API.

Summary

Clustering is viewed by many as an unapproachable dark art, but hopefully the high level overviews and component breakdowns in this chapter have shone some light on the subject, hopefully the history of deployment automation has shown how far we've come, and hopefully the goals and design of Kubernetes have shown the path forward. In the next chapter we'll take our first step toward that path, and take a detailed look at setting up a multi-node Kubernetes cluster.

CHAPTER 2

Deploying Kubernetes

Overview

This chapter will walk you through provisioning a multi-node Kubernetes cluster capable of running a wide variety of container based workloads. Kubernetes requires a specific network layout where each pod running in a given cluster has a dedicated IP address. The Kubernetes networking model also requires each pod and node have the ability to directly communicate with every other pod and node in the cluster. These requirements may seem arbitrary, but it allows for a cluster that can utilize techniques like automatic service discovery, and fine-grained application monitoring, while avoiding pitfalls like port collisions.

This chapter covers:

- Provisioning cluster nodes
- Configuring a Kubernetes compatible network
- Deploying Kubernetes services and client tools

A Kubernetes cluster consists of a consistent data store, and a collection of controller and worker services.

Consistent Data Store

All cluster state is stored in a key-value store called etcd. etcd is a distributed, consistent key-value store for shared configuration and service discovery, with a focus on being:

- Simple: curl'able user facing API (HTTP+JSON)

- Secure: optional SSL client cert authentication
- Fast: benchmarked 1000s of writes per second
- Reliable: properly distributed using Raft

The usage of a strongly consistent data store is critical to proper operation of a Kubernetes cluster. Cluster state must be consistant to ensure cluster services have accurate information when making scheduling decisions and enforcing end-user policies and desired state.

Controller Services

Controller services provide the necessary infrastructure for declaring and enforcing desired cluster state. The controller stack includes the following components:

- API Server
- Controller Manager
- Scheduler

Controller services are deployed to controller nodes. Separating controller and worker services helps protect the SLA of the critical infrastructure services they provide.

Worker Services

Worker services are responsible for managing pods and service endpoints on a given system. The worker stack includes the following components:

- Kubelet
- Service Proxy
- Docker

Docker, the container runtime engine, facilitates the creation and destruction of containers as determined by the Kubelet.

Kubernetes Nodes

Nodes must have network connectivity between them, ideally in the same datacenter or availability zone. Nodes should also have a valid hostname that can be resolved using DNS by other nodes in the cluster. For example, the machines in the lab have the following DNS names:

Table 2-1. Cluster Machines

internal hostname	external hostname
node0.c.kuarlab.internal	node0.kuar.io
node1.c.kuarlab.internal	node1.kuar.io
node2.c.kuarlab.internal	node2.kuar.io
node3.c.kuarlab.internal	node3.kuar.io

System Requirements

System requirements will largely depend on the number of pods running at a given time and their individual resource requirements. In the lab each node has the following system specs:

- 1CPU
- 2GB Ram
- 40GB Hardisk space

Keep in mind these are minimal system requirements, but it will be enough to get you up and running. If you ever run out of cluster resources to run additional pods, just add another node to the cluster or increase the amount of resources on a given machine.

The lab used in this book utilizes the Google Cloud Platform (GCP), a set of cloud services from Google. To meet the requirements mentioned above the following commands where ran to provision 4 virtual machines:

```
$ for i in {0..3}; do
  gcloud compute instances create node${i} \
  --image-project coreos-cloud \
  --image coreos-stable-717-3-0-v20150710 \
  --boot-disk-size 200GB \
  --machine-type n1-standard-1 \
  --can-ip-forward \
  --scopes compute-rw
done
```

IP Forwarding

Most cloud platforms will not allow machines to send packets whose source IP address does not match the IP assigned to the machine. In the case of GCP, instances can be deployed with the `--can-ip-forward` flag to disable this restriction. The ability to do IP forwarding is critical to the network setup recommended later in this chapter.

Configuring the Docker Daemon

The Kubernetes network model requires each pod to have a unique IP address within the cluster. Currently Docker is responsible for allocating pod IPs based on the subnet used by the Docker bridge. To satisfy the pod IP uniqueness constraint we must ensure each Docker host has a unique subnet range. In the lab I used the following mapping to configure each Docker host:

#FIXME what do you think of having the first cluster be c0? so it'd be node0.c0.kuar-lab.internal. and the bip would be 10.0.0.1/24 for that node? It'll make for a smoother example of switching context between clusters later on in the book.

Table 2-2. Docker Bridge Mapping

hostname	bip
node0.c.kuarlab.internal	10.200.0.1/24
node1.c.kuarlab.internal	10.200.1.1/24
node2.c.kuarlab.internal	10.200.2.1/24
node3.c.kuarlab.internal	10.200.3.1/24

The location of the Docker configuration file varies between Linux distributions, but in all cases the `--bip` flag is used to set the Docker bridge IP.

Download the Docker unit file:

```
$ sudo curl https://kuar.io/docker.service \
  -o /etc/systemd/system/docker.service
```

Edit the Docker unit file:

```
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.io
```

```
[Service]
ExecStart=/usr/bin/docker --daemon \
  --bip=10.200.0.1/24 \ ❶
  --iptables=false \
  --ip-masq=false \
  --host=unix:///var/run/docker.sock \
  --storage-driver=overlay
Restart=on-failure
RestartSec=5

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

❶ The bip must be unique across all nodes

In the above configuration Docker will no longer manage iptables or setup the firewall rule necessary for containers to reach the internet. This will be resolved in the Getting Containers Online section.

Start the Docker service:

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable docker
$ sudo systemctl start docker
```

Repeat the above steps on each node.

Configuring the Network

Now that each node has a Docker daemon configured with a unique bridge IP, routing must be setup between the nodes. Routing is an advanced concept, but at a high level each bridge IP requires a route entry. There are many options for setting up routes between nodes including the following:

- static routes on each node
- static routes on a central router or default gateway
- use an overlay network

In this chapter we will leverage static routes on a central router. See Appendix X for more details on setting up other routing configurations including overlay networks.

In the lab I ran the following commands to establish routes between each node.

```
$ gcloud compute routes create default-route-10-200-0-0-24 \
    --destination-range 10.200.0.0/24 \
    --next-hop-instance node0

$ gcloud compute routes create default-route-10-200-1-0-24 \
    --destination-range 10.200.1.0/24 \
    --next-hop-instance node1

$ gcloud compute routes create default-route-10-200-2-0-24 \
    --destination-range 10.200.2.0/24 \
    --next-hop-instance node2

$ gcloud compute routes create default-route-10-200-3-0-24 \
    --destination-range 10.200.3.0/24 \
    --next-hop-instance node3
```

The `gcloud compute routes` command configures the routing table in the lab to route the Docker bridge IP (bip) to the correct node as defined in the Docker Bridge Mapping table.

Getting Containers Online

Network Address Translation[1] is used to ensure containers have access to the internet. This is often necessary because many hosting providers will not route outbound traffic for IPs that don't originate from the host IP. To work around this containers must "masquerade" under the host IP, but we only want to do this for traffic not destined to other containers or nodes in the cluster.

In the lab we have disabled the Docker daemon's ability to manage iptables in favor of the Kubernetes proxy doing all the heavy lifting. Since the proxy does not know anything about the Docker bridge IP or the topology of our network, it does not attempt to setup NAT rules.

Add a NAT rule for outbound container traffic:

```
$ sudo iptables -t nat -A POSTROUTING ! -d 10.0.0.0/8 -o ens4v1 -j MASQUERADE
```

In the lab the outbound interface to the public network is ens4v1, be sure to change this to match your environment.

1 Network Address Translation (NAT) is a way to map an entire network to a single IP address.

Validating the Networking Configuration

Using the Docker client we can start two containers running on different hosts and validate our network setup. Open a new terminal and launch a busybox container:

```
$ ssh core@node0.kuar.io
$ docker run -t -i --rm busybox /bin/sh
```

At this point you are now running inside the busybox container. Print the IP address of the container using the `ip` command:

```
# ip -f inet addr show eth0

4: eth0: <BROADCAST,UP,LOWER_UP> mtu 1460 qdisc noqueue state UP group default
    inet 10.200.0.2/24 scope global eth0
       valid_lft forever preferred_lft forever
```

Open another terminal and launch a busybox container on a different node:

```
$ ssh core@node1.kuar.io
$ docker run -t -i --rm busybox /bin/sh
```

At the command prompt ping the IP address of the first busybox container:

```
# ping -c 3 10.200.0.2

PING 10.200.0.2 (10.200.0.2): 56 data bytes
64 bytes from 10.200.0.2: seq=0 ttl=62 time=0.914 ms
64 bytes from 10.200.0.2: seq=1 ttl=62 time=0.678 ms
64 bytes from 10.200.0.2: seq=2 ttl=62 time=0.667 ms

--- 10.200.0.2 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.667/0.753/0.914 ms
```

If you get simliar output it means you've successfully setup routes between two Docker hosts. Type the `exit` command at both busybox command prompts to exit the containers.

Bootstrapping the Kubernetes Controller Node

In the lab node0.kuar.io has been marked as the controller node for the Kubernetes cluster, and will host the controller services in addition to etcd. All controller services will be managed using systemd.

Start by logging into the controller node:

```
$ ssh kelseyhightower@node0.kuar.io
```

Bootstrapping etcd

etcd will be provisioned on the controller node and provide the storage backend for the Kubernetes API server.

Download the etcd unit file:

```
$ sudo curl https://kuar.io/etcd.service \
  -o /etc/systemd/system/etcd.service
```

Review the etcd unit file:

```
[Unit]
Description=etcd
Documentation=https://github.com/coreos/etcd

[Service]
ExecStartPre=/usr/bin/mkdir -p /opt/bin
ExecStartPre=/usr/bin/curl https://kuar.io/etcd \
  -o /opt/bin/etcd \
  -z /opt/bin/etcd
ExecStartPre=/usr/bin/chmod +x /opt/bin/etcd
ExecStart=/opt/bin/etcd \
  --advertise-client-urls http://127.0.0.1:2379 \
  --data-dir /var/lib/etcd \
  --listen-client-urls http://127.0.0.1:2379 \
  --listen-peer-urls http://127.0.0.1:2380 \
  --name etcd0
Restart=on-failure
RestartSec=5

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

Start the etcd service:

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable etcd
$ sudo systemctl start etcd
```

The etcd service is only available to local services running on node0.kuar.io. This was done intentionally as this etcd instance is dedicated to the Kubernetes API server.

Deploying the API Server

The API server will run on the controller node and expose the Kubernetes API to other services within the cluster.

Download the kube-apiserver unit file:

```
$ sudo curl https://kuar.io/kube-apiserver.service \
  -o /etc/systemd/system/kube-apiserver.service
```

Review the kube-apiserver unit file:

```
[Unit]
Description=Kubernetes API Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStartPre=/usr/bin/mkdir -p /opt/bin
ExecStartPre=/usr/bin/curl https://kuar.io/kube-apiserver \
  -o /opt/bin/kube-apiserver \
  -z /opt/bin/kube-apiserver
ExecStartPre=/usr/bin/chmod +x /opt/bin/kube-apiserver
ExecStart=/opt/bin/kube-apiserver \
  --insecure-bind-address=0.0.0.0 \
  --etcd-servers=http://127.0.0.1:2379 \
  --service-cluster-ip-range=10.200.20.0/24
Restart=on-failure
RestartSec=5

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

Start the kube-apiserver service:

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable kube-apiserver
$ sudo systemctl start kube-apiserver
```

The Kubernetes API is now exposed to all nodes within the cluster. Before exposing the API server to the outside world additional configuration is required to secure the API endpoint. Later in this chapter we will explore the usage of SSH tunnels to provide secure remote access to the API.

Chapter 10, "Securing a Kubernetes Cluster", will cover more advanced configurations for securing the Kubernetes API via TLS certificates.

Deploying the Controller Manager

Download the kube-controller-manager unit file:

```
$ sudo curl https://kuar.io/kube-controller-manager.service \
  -o /etc/systemd/system/kube-controller-manager.service
```

Review the kube-controller-manager unit file:

```
[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStartPre=/usr/bin/mkdir -p /opt/bin
ExecStartPre=/usr/bin/curl https://kuar.io/kube-controller-manager \
  -o /opt/bin/kube-controller-manager \
  -z /opt/bin/kube-controller-manager
ExecStartPre=/usr/bin/chmod +x /opt/bin/kube-controller-manager
ExecStart=/opt/bin/kube-controller-manager \
  --master=http://127.0.0.1:8080 \
  --v=2
Restart=on-failure
RestartSec=5

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

Start the kube-controller-manager service:

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable kube-controller-manager
$ sudo systemctl start kube-controller-manager
```

Deploying the Scheduler

Download the kube-scheduler unit file:

```
$ sudo curl https://kuar.io/kube-scheduler.service \
  -o /etc/systemd/system/kube-scheduler.service
```

Review the kube-scheduler unit file:

```
[Unit]
Description=Kubernetes Scheduler
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStartPre=/usr/bin/mkdir -p /opt/bin
ExecStartPre=/usr/bin/curl https://kuar.io/kube-scheduler \
  -o /opt/bin/kube-scheduler \
  -z /opt/bin/kube-scheduler
ExecStartPre=/usr/bin/chmod +x /opt/bin/kube-scheduler
ExecStart=/opt/bin/kube-scheduler \
  --master=http://127.0.0.1:8080 \
  --v=2
Restart=on-failure
RestartSec=5

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

Start the kube-scheduler service:

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable kube-scheduler
$ sudo systemctl start kube-scheduler
```

At this point the controller services are up and running on node0.kuar.io.

The Kubernetes Client

The official Kubernetes client is kubectl: a command line tool for interacting with the Kubernetes API. kubectl can be used to verify the overall health of the cluster. First login to the controller node:

```
$ ssh kelseyhightower@node0.kuar.io
```

Download and install kubectl:

```
$ sudo curl -o /opt/bin/kubectl https://kuar.io/linux/kubectl
$ sudo chmod +x /opt/bin/kubectl
```

Next, check the health status of the cluster components:

```
$ /opt/bin/kubectl get cs

NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok                   nil
scheduler            Healthy   ok                   nil
etcd-0               Healthy   {"health": "true"}   nil
```

At this point we have a working Kubernetes control plane and can move on to adding worker nodes.

Bootstrapping Kubernetes Worker Nodes

Deploying the Kubelet

The kubelet is responsible for managing pods, mounts, node registration, and reporting metrics and health status to the API server. The kubelet must be deployed to each worker node.

```
$ ssh kelseyhightower@node1.kuar.io
```

Download the kubelet unit file:

```
$ sudo curl https://kuar.io/kubelet.service \
  -o /etc/systemd/system/kubelet.service
```

Edit the kubelet unit file:

```
[Unit]
Description=Kubernetes Kubelet
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStartPre=/usr/bin/mkdir -p /opt/bin
ExecStartPre=/usr/bin/curl https://kuar.io/kubelet \
  -o /opt/bin/kubelet \
  -z /opt/bin/kubelet
ExecStartPre=/usr/bin/chmod +x /opt/bin/kubelet
ExecStart=/opt/bin/kubelet \
  --api-servers=http://node0.c.kuarlab.internal:8080 \ ❶
  --cluster-dns=10.200.20.10 \ ❷
  --cluster-domain=cluster.local \
  --v=2
Restart=on-failure
RestartSec=5

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

❶ The `api-servers` flag should point to your API server.

❷ The `cluster-dns` and `cluster-domain` flags enable support for the DNS cluster add-on service.

Start the kubelet service:

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable kubelet
$ sudo systemctl start kubelet
```

Repeat the above steps for the other two nodes, then verify the nodes are online by running the following kubectl command on the controller node.

```
$ /opt/bin/kubectl get nodes

NAME                        LABELS                                             STATUS
node1.c.kuarlab.internal    kubernetes.io/hostname=node1.c.kuarlab.internal    Ready
node2.c.kuarlab.internal    kubernetes.io/hostname=node2.c.kuarlab.internal    Ready
node3.c.kuarlab.internal    kubernetes.io/hostname=node3.c.kuarlab.internal    Ready
```

Deploying the Service Proxy

The service proxy must be deployed to each worker node.

Download the kube-proxy unit file:

```
$ sudo curl https://kuar.io/kube-proxy.service \
  -o /etc/systemd/system/kube-proxy.service
```

Edit the kube-proxy unit file:

```
[Unit]
Description=Kubernetes Proxy
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStartPre=/usr/bin/mkdir -p /opt/bin
ExecStartPre=/usr/bin/curl https://kuar.io/kube-proxy \
  -o /opt/bin/kube-proxy \
  -z /opt/bin/kube-proxy
ExecStartPre=/usr/bin/chmod +x /opt/bin/kube-proxy
ExecStart=/opt/bin/kube-proxy \
  --master=http://node0.c.kuarlab.internal:8080 \ ❶
  --v=2
Restart=on-failure
RestartSec=5

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

❶ The `master` flag should point to your API server.

Start the kube-proxy service:

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable kube-proxy
$ sudo systemctl start kube-proxy
```

Repeat the above steps for the other two nodes to complete the deployment of the worker nodes.

Cluster Add-ons

Kubernetes ships with additional functionality through cluster add-ons, which are a collection of Services and Replication Controllers (with pods) that extend the utility of your cluster. While cluster add-ons are not strictly required, they are considered an inherent part of a Kubernetes cluster.

There are four primary cluster add-ons:

- Cluster monitoring
- DNS
- Kubernetes UI
- Logging

We'll cover the Kubernetes UI and DNS add-ons in this chapter and defer monitoring and logging until the discussion on cluster administration later in the book.

DNS

Kubernetes offers a DNS cluster add-on that provides DNS A and SRV records for Kubernetes services. The heavy lifting is done by SkyDNS, an etcd backed DNS server that supports dynamic updates from the Kubernetes API.

Download the SkyDNS replication controller configuration:

```
$ wget https://kuar.io/skydns-rc.yaml
```

Edit the SkyDNS rc config:

```
$ vim skydns-rc.yaml
```

```
- name: kube2sky
  image: gcr.io/google_containers/kube2sky:1.11
  resources:
    limits:
      cpu: 100m
      memory: 50Mi
  args:
  # command = "/kube2sky"
  - -domain=cluster.local
  - -kube_master_url=http://node0.c.kuarlab.internal:8080 ❶
```

❶ The `kube_master_url` flag should point to your API server.

Create the SkyDNS replication controller:

```
$ /opt/bin/kubectl create -f skydns-rc.yaml
```

Next create the SkyDNS service:

```
$ /opt/bin/kubectl create -f https://kuar.io/skydns-svc.yaml
```

Services can now be looked up by service name from pods deployed by the Kubelet:

```
$ service-name.default.svc.cluster.local
```

Cluster configs

The DNS add-on was setup using cluster configs hosted on a remote site. This should be considered a potential security risk. Consider downloading untrusted cluster configs and reviewing them before deploying to your cluster.

Kubernetes UI

The Kubernetes UI provides a read-only web console for viewing cluster state and monitoring node resource utilization and cluster events. kubectl can be used to deploy the kube-ui add-on.

Launch a Kubernetes UI replication controller:

```
$ /opt/bin/kubectl create -f https://kuar.io/kube-ui-rc.yaml
```

Next create the service for the UI:

```
$ /opt/bin/kubectl create -f https://kuar.io/kube-ui-svc.yaml
```

At this point the Kubernetes UI add-on should be up and running. The Kubernetes API server provides access to the UI via the /ui endpoint. However the Kubernetes API is not accessable remotely due to the lack of security.

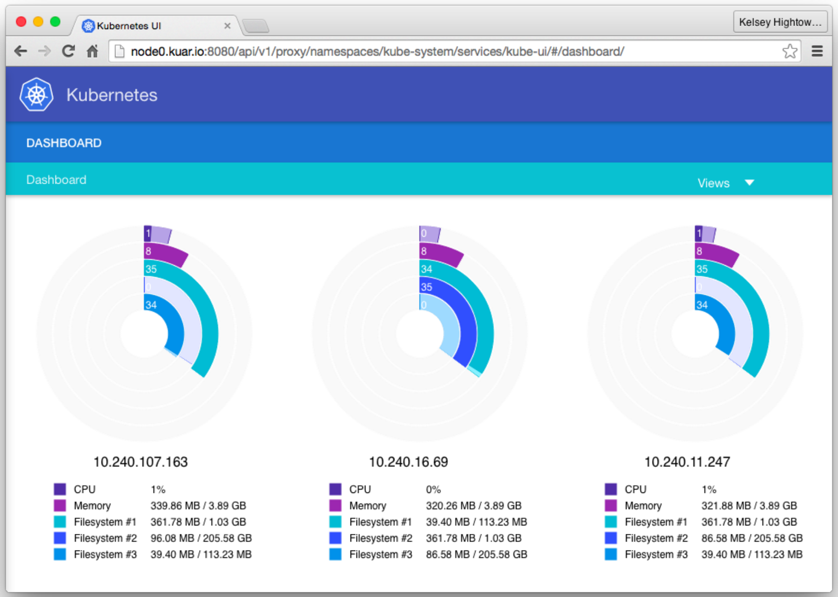
Securely Exposing the API Server

Instead of exposing the API server to the public internet over an insecure port, a SSH tunnel can be can be create between the remote client and API server.

Create a SSH tunnel between a remote client machine and the controller node:

```
$ ssh -f -nNT -L 8080:127.0.0.1:8080 kelseyhightower@node0.kuar.io
```

The UI is available at *http://127.0.0.1:8080/ui* on the client machine.


Kubernetes UI
Kelsey Hightow…
node0.kuar.io:8080/api/v1/proxy/namespaces/kube-system/services/kube-ui/#/dashboard/
Kubernetes
DASHBOARD
Dashboard
Views

10.240.107.163

CPU	1%
Memory	339.86 MB / 3.89 GB
Filesystem #1	361.78 MB / 1.03 GB
Filesystem #2	96.08 MB / 205.58 GB
Filesystem #3	39.40 MB / 113.23 MB

10.240.16.69

CPU	0%
Memory	320.26 MB / 3.89 GB
Filesystem #1	39.40 MB / 113.23 MB
Filesystem #2	361.78 MB / 1.03 GB
Filesystem #3	86.58 MB / 205.58 GB

10.240.11.247

CPU	1%
Memory	321.88 MB / 3.89 GB
Filesystem #1	361.78 MB / 1.03 GB
Filesystem #2	86.58 MB / 205.58 GB
Filesystem #3	39.40 MB / 113.23 MB

Summary

Now that's how you bootstrap a Kubernetes cluster! At this point you have a multi-node Kubernetes cluster with a single controller node and three workers. Adding additional workers is a matter of provisioning new machines and repeating the steps to add the machine to the cluster network, then deploying an instance of the kubelet and proxy services. It should also be noted that the cluster setup in this chapter lacks proper security and high-availability for the controller components, both will be addressed in later chapters. In the meanwhile don't expose the Kubernetes API or Kubelet endpoints on the public internet.

By manually setting up your cluster you now have a better understanding of the components and details of Kubernetes. However, you should consider automating the bootstrap process, especially in large scale environments. What's the point of a cluster if you have to hand-roll each node?