O'REILLY®

Kubernetes Up & Running

DIVE INTO THE FUTURE OF INFRASTRUCTURE

Kelsey Hightower

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

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.

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 at first may seem arbitrary. However, they allow the Kubernetes cluster to utilize techniques in automatic service discovery and fine-grained application monitoring while avoiding pitfalls such as port collisions.

This chapter covers:

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

A Kubernetes cluster comprises 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 which can be used to store shared configuration and service discovery information. Etcd has a focus on being:

- Secure: optional SSL client cert authentication

- Fast: benchmarked at 1000s of writes per second
- Reliable: properly distributed using the Raft consensus algorithm

The usage of a strongly consistent data store is critical to the proper operation of a Kubernetes cluster. Cluster state *must be consistent* to ensure cluster services have accurate information when making scheduling decisions or enforcing end-user policies and desired end-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 within 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.kubernetes-up-and-running.internal	node0.kuar.io
node1.c.kubernetes-up-and-running.internal	node1.kuar.io
node2.c.kubernetes-up-and-running.internal	node2.kuar.io
node3.c.kubernetes-up-and-running.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 Hard disk 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 any given machine.

The lab used in this book utilizes the Google Cloud Platform (GCP), a set of cloud services from Google.

The Kubernetes Machine Image

The nodes in the lab will be running a customized Debian 8 machine image which has been pre-populated with the necessary software components to bootstrap a Kubernetes cluster. In order to utilize this image it must be imported it into your GCP project.

In the lab I imported the Kubernetes Image into my project using `compute images create` command:

```
$ gcloud compute images create kubernetes-1-0-6-v20151004 \
  --source-uri https://storage.googleapis.com/kuar/kubernetes-1-0-6-v20151004.tar.gz
```

For a complete gcloud command reference type `gcloud compute --help`. The Google gcloud tool can be obtained from *https://cloud.google.com/sdk/gcloud/*

Launching Kubernetes Nodes

Create 4 machines on GCE using the `compute instances create` command:

```
$ for i in {0..3}; do
  gcloud compute instances create node${i} \
  --image kubernetes-1-0-6-v20151004 \
  --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.

Once all 4 nodes have been provisioned verify they are up and running using the `compute instances list` command:

```
$ gcloud compute instances list
```

Output:

```
NAME  ZONE          MACHINE_TYPE  INTERNAL_IP EXTERNAL_IP     STATUS
node0 us-central1-f n1-standard-1 10.240.0.2  146.148.60.178  RUNNING
node1 us-central1-f n1-standard-1 10.240.0.3  173.255.112.106 RUNNING
node2 us-central1-f n1-standard-1 10.240.0.4  173.255.118.37  RUNNING
node3 us-central1-f n1-standard-1 10.240.0.5  104.197.91.189  RUNNING
```

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.

Table 2-2. Docker Bridge Mapping

hostname	bip
node0.c.kubernetes-up-and-running.internal	10.200.0.1/24
node1.c.kubernetes-up-and-running.internal	10.200.1.1/24
node2.c.kubernetes-up-and-running.internal	10.200.2.1/24
node3.c.kubernetes-up-and-running.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.

It is rare but you may have to consider a different CIDR range if you have any VPN rules which could route traffic inadvertently away from the bridge IPs defined above.

Create the Docker systemd unit file by writing the following contents to a file named `docker.service`:

```
# /etc/systemd/system/docker.service

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

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

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

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.

Move the Docker systemd unit file to the local systemd configuration path:

```
$ sudo mv docker.service /etc/systemd/system/
```

Start the Docker service:

```
$ sudo systemctl daemon-reload  # reloads systemd loading any changes
$ sudo systemctl enable docker  # allows startup on boot
$ sudo systemctl start docker   # starts the service
```

Repeat the above steps on each node and be sure each Docker bridge IP is unique.

Configuring the Kubernetes Kubelet

The Kubelet is responsible for managing pods, mounts, node registration, and reporting metrics and health status to the API server. The Kubelet will be used to bootstrap the Kubernetes controller components and worker nodes.

Let start by configuring the Kubelet on node0:

```
$ gcloud compute ssh node0
```

Create the Kubelet systemd unit file by writing the following contents to a file named kubelet.service:

```
# /etc/systemd/system/kubelet.service

[Unit]
Description=Kubernetes Kubelet
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=docker.service
Requires=docker.service

[Service]
ExecStartPre=/bin/mkdir -p /etc/kubernetes/manifests
ExecStart=/usr/local/bin/kubelet \
  --api-servers=http://node0:8080 \
  --allow-privileged=true \
  --cluster-dns=10.200.100.10 \
  --cluster-domain=cluster.local \
  --config=/etc/kubernetes/manifests
Restart=on-failure
RestartSec=5

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

Move the kubelet systemd unit file to the local systemd configuration path:

```
$ sudo mv kubelet.service /etc/systemd/system/
```

Start the kubelet service:

```
$ sudo systemctl daemon-reload   # reloads systemd loading any changes
$ sudo systemctl enable kubelet  # allows startup on boot
$ sudo systemctl start kubelet   # starts the service
```

Repeat the above steps for the other three nodes (node1, node2, and node3).

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 [2] 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.

On *each node* add a NAT rule for outbound container traffic:

```
$ sudo iptables -t nat -A POSTROUTING ! -d 10.0.0.0/8 -o eth0 -j MASQUERADE
```

In the lab, the outbound interface to the public network is eth0, be sure to change this to match your environment.

Validating the Networking Configuration

Using the Docker command line client we can start two containers running on different hosts and validate our network setup. First login to node0:

```
$ gcloud compute ssh node0
```

Start a busybox container using the `docker run` command:

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

2 iptables is service maintaining rules for the Linux kernel firewall

```
$ sudo docker run -t -i --rm busybox /bin/sh
```

docker command

This particular `docker` command starts a container that is capable of receiving standard input from a controlling terminal. The container will also be removed once it is stopped (when you exit the terminal).

sudo command

You will notice that the remainder of this book will often reference the superuser command program `sudo` when interacting with docker. This is because the user (you) which gcloud has ssh'd onto the node does not belong in the docker group. To remedy this execute the following:

```
$ sudo groupadd docker
$ sudo usermod -aG docker $(whoami)
$ sudo systemctl restart docker
$ docker ps  # command should work under the current user
>>
CONTAINER ID   IMAGE     COMMAND     CREATED      STATUS     PORT     NAMES
```

At this point you are now running inside a busybox container. Show the IP address of the container using the `ip` command and interface `eth0`:

```
# 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:

```
$ gcloud compute ssh node1
```

Start a new busybox container on node1:

```
$ sudo 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 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 pods running under the Kubelet.

Start by logging into the controller node:

```
$ gcloud compute ssh node0
```

etcd

Create the etcd pod manifest:

```
# /etc/kubernetes/manifests/etcd-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: etcd
spec:
  hostNetwork: true
  volumes:
    - name: "etcd-datadir"
      hostPath:
        path: "/var/lib/etcd"
  containers:
    - name: "etcd"
      image: "b.gcr.io/kuar/etcd:2.2.0"
      args:
        - "--data-dir=/var/lib/etcd"
        - "--advertise-client-urls=http://127.0.0.1:2379"
        - "--listen-client-urls=http://127.0.0.1:2379"
        - "--listen-peer-urls=http://127.0.0.1:2380"
        - "--name=etcd"
      volumeMounts:
        - mountPath: /var/lib/etcd
          name: "etcd-datadir"
```

Move the etcd pod manifest to the local kubelet pod manifest directory:

```
$ sudo mv etcd-pod.yaml /etc/kubernetes/manifests/
```

The full spec and definition of the POD object can be found at:
http://kubernetes.io/docs/api-reference/v1/definitions/#_v1_pod

API Server

Create the kube-apiserver pod manifest by writing the following contents to a file named kube-apiserver-pod.yaml.

```
# /etc/kubernetes/manifests/kube-apiserver-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: kube-apiserver
spec:
  hostNetwork: true
  containers:
    - name: "kube-apiserver"
      image: "b.gcr.io/kuar/kube-apiserver:1.0.6"
      args:
        - "--allow-privileged=true"
        - "--etcd-servers=http://127.0.0.1:2379"
        - "--insecure-bind-address=0.0.0.0"
        - "--service-cluster-ip-range=10.200.100.0/24"
        - "--service-node-port-range=30000-37000"
```

Move the kube-apiserver pod manifest to the local kubelet pod manifest directory:

```
$ sudo mv kube-apiserver-pod.yaml /etc/kubernetes/manifests/
```

Controller Manager

Create the kube-controller-manager pod manifest by writing the following contents to a file named kube-controller-manager-pod.yaml.

```
# /etc/kubernetes/manifests/kube-controller-manager-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: kube-controller-manager
spec:
  hostNetwork: true
  containers:
    - name: "kube-controller-manager"
      image: "b.gcr.io/kuar/kube-controller-manager:1.0.6"
      args:
        - "--master=http://127.0.0.1:8080"
```

Move the kube-controller-manager pod manifest to the local kubelet pod manifest directory:

```
$ sudo mv kube-controller-manager-pod.yaml /etc/kubernetes/manifests/
```

Scheduler

Create the kube-scheduler pod manifest by writing the following contents to a file named kube-scheduler-pod.yaml:

```
# /etc/kubernetes/manifests/kube-scheduler-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: kube-scheduler
spec:
  hostNetwork: true
  containers:
    - name: "kube-scheduler"
      image: "b.gcr.io/kuar/kube-scheduler:1.0.6"
      args:
        - "--master=http://127.0.0.1:8080"
```

Move the kube-scheduler pod manifest to the local kubelet manifest directory:

```
$ sudo mv kube-scheduler-pod.yaml /etc/kubernetes/manifests/
```

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

Checking the Health of the Cluster Components

There are 4 major cluster components that make up the Kubernetes controller set. More information about these components will follow in later chapters. For now, consider that these are pods managed by Kubernetes and as such their health status can be checked. The four components are:

- api
- controller-manager
- scheduler
- etcd

The health of each component can be checked using the Kubernetes API. For example, use the curl command below we query the send a get request to the api server for the scheduler. A response is returned in json format which describes the status of the Kubernetes Scheduler along with additional metadata:

```
$ curl http://127.0.0.1:8080/api/v1/namespaces/default/componentstatuses/scheduler
```

Output:

```
{
  "kind": "ComponentStatus",
  "apiVersion": "v1",
  "metadata": {
    "name": "scheduler",
    "selfLink": "/api/v1/namespaces/componentstatuses/scheduler",
    "creationTimestamp": null
  },
  "conditions": [
    {
      "type": "Healthy",
      "status": "True",
      "message": "ok",
      "error": "nil"
    }
  ]
}
```

These particular results indicate the scheduler component is healthy. The status for the other components can be retrieved by replacing the component name in the query path like so:

```
/api/v1/namespaces/default/componentstatuses/{name}
```

There is no component status endpoint for the API server. The ability to send requests and receive a valid response is enough to verify the health of the API server.

To learn more about all the APIs available visit: *http://kubernetes.io/docs/api-reference/v1/operations/*

Bootstrapping Kubernetes Worker Nodes

A Kubernetes worker node runs the following components:

- docker
- kubelet
- kube-proxy

Both docker and the kubelet have already been configured earlier in the chapter. The only component left to configure is the Kubernetes proxy.

Service Proxy

The Kubernetes proxy is a network proxy that is capable of handling various TCP/UDP streaming or round robin forwarding across the cluster. It runs on every node.

Create the kube-proxy pod manifest by writing the following contents to a file named kube-proxy-pod.yaml for the current node you are logged into (node0).

```
# /etc/kubernetes/manifests/kube-proxy-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: kube-proxy
  version: 1.0.6
spec:
  hostNetwork: true
  volumes:
    - name: "lib"
      hostPath:
        path: "/lib"
    - name: "sbin"
      hostPath:
        path: "/sbin"
    - name: "lib64"
      hostPath:
        path: "/lib64"
  containers:
    - name: "kube-proxy"
      image: "b.gcr.io/kuar/kube-proxy:1.0.6"
      args:
        - "--master=http://node0:8080"
      securityContext:
        privileged: true
      volumeMounts:
        - mountPath: /lib
          name: "lib"
        - mountPath: /sbin
          name: "sbin"
        - mountPath: /lib64
          name: "lib64"
```

Move the kube-proxy pod manifest to the local kubelet manifest directory:

```
$ sudo mv kube-proxy-pod.yaml /etc/kubernetes/manifests/
```

Repeat the above steps for the other nodes to complete the deployment of the worker nodes (node1, node2, node3).

The Kubernetes Client

The official Kubernetes client is `kubectl`: a command line tool for interacting with the Kubernetes API. `kubectl` can be used to manage most kubernetes objects such as pods, replication controllers, and services. `kubectl` can also be used to verify the overall health of the cluster.

First log into the controller node if you are not already logged in:

```
$ gcloud compute ssh node0
```

Checking Cluster Component Status

```
$ kubectl get componentstatuses
```

Output:

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

Listing Kubernetes Worker Nodes

```
$ kubectl get nodes
```

Output:

```
NAME      LABELS                           STATUS
node0     kubernetes.io/hostname=node0     Ready
node1     kubernetes.io/hostname=node1     Ready
node2     kubernetes.io/hostname=node2     Ready
node3     kubernetes.io/hostname=node3     Ready
```

Use the `kubectl describe` command to get more information about a specific node such as node0:

```
$ kubectl describe nodes node0
```

Output:

```
Name:                   node0
Labels:                 kubernetes.io/hostname=node0
Addresses:      10.240.0.2
Capacity:
 pods:          40
 cpu:           1
 memory:        3794264Ki
Version:
 Kernel Version:                4.1.0-0.bpo.2-amd64
 OS Image:                      Debian GNU/Linux 8 (jessie)
```

```
 Container Runtime Version:     docker://1.8.2
 Kubelet Version:               v1.0.6
 Kube-Proxy Version:            v1.0.6
ExternalID:                     node0
Pods:                           (5 in total)
  Namespace                     Name
  default                       etcd-node0
  default                       kube-apiserver-node0
  default                       kube-controller-manager-node0
  default                       kube-proxy-node0
  default                       kube-scheduler-node0
```

At this point we have a working Kubernetes cluster.

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 add-on in this chapter and defer DNS, monitoring, and logging until the discussion on cluster administration later in the book.

Kubernetes UI

The Kubernetes UI provides a read-only web console for viewing the current state of the cluster. It provides a view into the monitoring of node resource utilization along with any cluster events. `kubectl` can be used to deploy the `kube-ui` add-on.

Create the Kubernetes UI replication controller by writing the following contents to a file named kube-ui-rc.yaml within `~/kubernetes/addons/`

```
# ~/kubernetes/addons/kube-ui-rc.yaml:

apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-ui-v2
```

```
    namespace: kube-system
    labels:
      k8s-app: kube-ui
      version: v2
      kubernetes.io/cluster-service: "true"
  spec:
    replicas: 1
    selector:
      k8s-app: kube-ui
      version: v2
    template:
      metadata:
        labels:
          k8s-app: kube-ui
          version: v2
          kubernetes.io/cluster-service: "true"
      spec:
        containers:
        - name: kube-ui
          image: gcr.io/google_containers/kube-ui:v2
          resources:
            limits:
              cpu: 100m
              memory: 50Mi
          ports:
          - containerPort: 8080
          livenessProbe:
            httpGet:
              path: /
              port: 8080
            initialDelaySeconds: 30
            timeoutSeconds: 5
```

Launch a Kubernetes UI replication controller:

```
$ kubectl create -f ~/kubernetes/addons/kube-ui-rc.yaml
# or alternatively cd into ~/kubernetes/addons/, and run `kubectl create -f kube-ui-rc.yaml`
```

Next create the Kubernetes UI service by writing the following contents to a file named `kube-ui-svc.yaml` within `~/kubernetes/addons/`.

```
# ~/kubernetes/addons/kube-ui-svc.yaml

apiVersion: v1
kind: Service
metadata:
  name: kube-ui
  namespace: kube-system
  labels:
    k8s-app: kube-ui
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeUI"
```

```
spec:
  selector:
    k8s-app: kube-ui
  ports:
  - port: 80
    targetPort: 8080
```

Create the Kubernetes UI service:

```
$ kubectl create -f ~/kubernetes/addons/kube-ui-svc.yaml
# or alternatively cd into ~/kubernetes/addons/, and run `kubectl create -f 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 accessible remotely due to the lack of security.

Use kubectl

Now is a great time to try and use the kubectl command to retrieve information rather than deploy. Try the command below, you should see entries for the work you've done up to this point.

$ kubectl get pods,services,replicationcontrollers

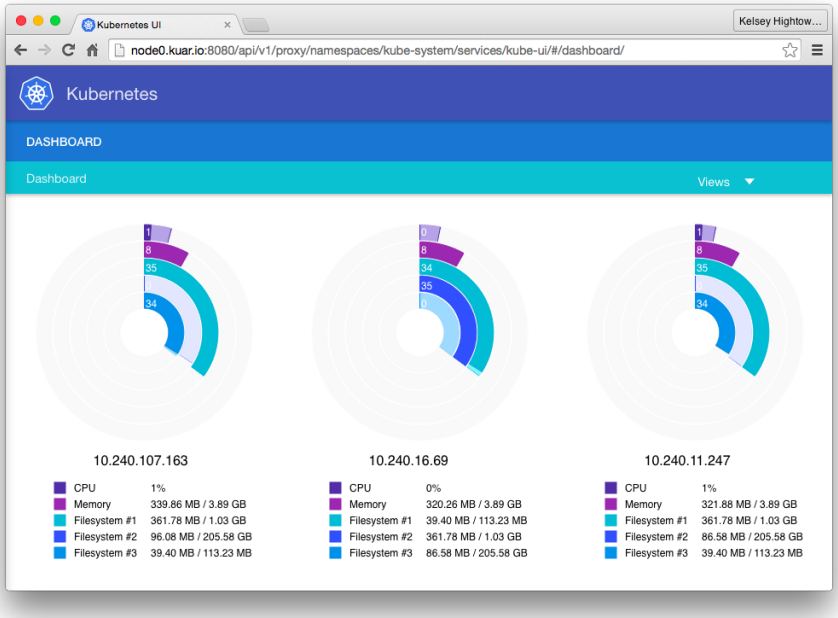
Securely Exposing the API Server

Instead of exposing the API server to the public internet over an insecure port, a SSH tunnel can be used to 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-external-ip>
```

The UI should be available at *http://127.0.0.1:8080/api/v1/proxy/namespaces/kube-system/services/kube-ui/#/* dashboard/ on the client machine.



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 above to add instances 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 these topics 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?

CHAPTER 3
Containers

Containers are self-contained applications composed of application binaries and their dependencies under a root filesystem suitable for running within a chroot environment. These root filesystems often bundle language runtimes such as Ruby, Python, or Java, and any supporting files and libraries. To support reusability and promote smaller container images, base dependencies shared across multiple applications can be distributed in separate container images and layered in at runtime. For example, you may have a Ruby image shared by all your apps, and separate Rails and Sinatra images that inherit from that base Ruby image.

Container Layering

When we talk about layering we are describing that a container can inherit a filesystem from another. To help explain this in detail consider these containers we could build below. Note that for correctness the ordering should be bottom up, but for ease of understanding we take the opposite approach.

```
.
└── container A: a base operating system only, such as Debian
    └── container B: build upon #0, by adding Ruby v2.1.10
    └── container C: build upon #0, by adding Golang v1.6
```

At this point we have three containers. Containers `A`, `B`, and `C`. `B` and `C` are *forked* from A and share nothing besides the base container's files. Taking if further we can build on top of `B` by adding rails. In fact, we may have to support our legacy application which requires another version - 3.2.x. Someday we'll migrate that app to v4.

```
. (continuing from above)
└── container B: a base operating system only, such as Debian
    └── container D: builds upon #B, by adding Rails v4.2.6
    └── container E: builds upon #B, by adding Rails v3.2.x
```

Conceptually each container builds upon a previous one. Each parent reference is a pointer. While the example here is a simple set of containers, other real world containers can be part of a larger and extensive directed acyclic graph.

Container images are typically combined with a container configuration file, which provides instructions on how to setup the container environment and execute an application `entrypoint`. The container configuration often includes information on how to setup networking, namespace isolation, resource constraints (cgroups), and what syscall restrictions should be placed on a running container instance. The container root filesystem and configuration file are typically bundled using the Docker image format.

Containers fall into two main categories:

- System containers
- Application containers

System Containers

System containers seek to mimic virtual machines and often run a full boot process. They often include a set of system services typically found in a Virtual Machine such as SSH, cron, and syslog.

Application Containers

Application containers differ from system containers in that they commonly run a single application. While running a single application per container might seem like an unnecessary constraint, it provides the perfect level of granularity for composing scalable applications, and is a design philosophy that is leveraged heavily by Pods.

Building Application Images with Docker

When working with application containers it's often helpful to package them in a way that makes it easy to share with others and distribute. Docker, the default container runtime engine, makes it easy to package an application and push to a remote registry where it can later be pulled by others.

In this chapter we are going to work with influxdb to show this workflow in action.

Dockerfiles

A Dockerfile can be used to automate the creation of a Docker container image. The following example describes the steps required to build the influxdb image that is both secure and lightweight in terms of size.

```
FROM scratch ❶
MAINTAINER Kelsey Hightower <kelsey.hightower@kuar.io>
COPY influxd /influxd
ENTRYPOINT ["/influxd"]
```

❶ The scratch base image does not take any space.

The text above can be stored in a text file, typically named Dockerfile, and be used to create a Docker image.

Run the following command to create the influxdb Docker image:

```
$ docker build -t influxdb:0.9.4.2 .
```

The final image should check in at around 12MB, which is drastically smaller than many publicly available images which tend to be built on top of an entire operating system such as Debian.

At this point the influxdb image lives in the local Docker registry where the image was built and is only accessible to a single machine. The true power of Docker comes from the ability to share images across thousands of machines and the broader Docker community.

Optimizing for Size and Security

When it comes to security there are no shortcuts. When building images that will ultimately run in a production Kubernetes cluster, be sure to follow best practices for packaging and distributing applications. For example, don't build containers with passwords baked in. You will be hacked, and you will bring shame to your entire company or department. We all want to be on TV some day, but there are better ways to go about that.

Storing Images in a Remote Registry

What good is a Docker image if it's only available on a single machine?

Kubernetes relies on the fact that images described in a Pod manifest are available across every machine in the cluster. One option would be to export the influxdb image and import it on every other machine in the Kubernetes cluster. I can't think of anything more tedious than managing Docker images this way. The process of man-

ually importing and exporting Docker images has human error written all over it. Just say no!

The standard within the Docker community is to store Docker images in a remote registry. There are tons of options when it comes to Docker Registries and what you choose will be largely based your needs in terms of security requirements and collaboration features.

In the lab I chose to go with the Google Container Registry mainly because I'm running my Kubernetes cluster in GCE, but any Docker registry would work.

I was able to run the following commands to tag and publish the influxdb Docker image to the Google Container Registry:

First I authenticated to the Google Container Registry using gcloud:

```
$ gcloud docker --authorize-only
```

Next I tagged the influxdb image by prepending the target Docker registry:

```
$ docker tag influxdb:0.9.4.2 b.gcr.io/kuar/influxdb:0.9.4.2
```

Then I pushed the influxdb image:

```
$ docker push b.gcr.io/kuar/influxdb:0.9.4.2
```

By default all images pushed to GCR are private, which means all consumers of the image must authenticate. For our example I wanted to make the influxdb image public. Docker images stored on GCR are backed by a Google cloud storage bucket. In my case all images used in this book are stored in the kuar (Kubernetes Up and Running) bucket. To make all images in this bucket public I ran the following command:

```
gsutil -m acl ch -R -u AllUsers:R gs://kuar
```

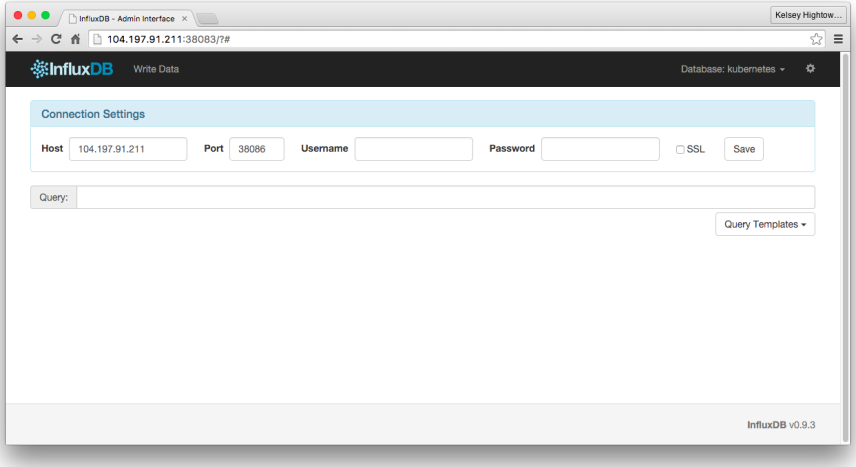
Now that the influxdb image is available on a remote registry, it's time to deploy it using Docker.

Running Containers with Docker

The docker cli tool can be used to deploy containers. To deploy a container from the b.gcr.io/kuar/influxdb:0.9.4.2 image run the following command:

```
$ docker run -d --name influxdb \
  --publish 38083:8083 \
  --publish 38086:8086 \
  b.gcr.io/kuar/influxdb:0.9.4.2
```

The above command starts the influxdb database and maps ports 38083 to 8083 and 38086 to 8086. This will allow us to access influxdb on the host IP address. The ability to map high level ports to containers helps avoid port conflicts. Each container gets its own IP address assigned by Docker.



Exporing the influxdb API

The influxdb container exposes a HTTP API on port 38086, which we can use to create a database to hold some metrics:

```
$ curl -G 'http://localhost:38086/query' \
  --data-urlencode "q=CREATE DATABASE kubernetes"
```

Output:

```
{"results":[{}]}
```

Now lets add some test data to the kubernetes database:

```
$ curl -X POST 'http://localhost:38086/write?db=kubernetes' \
  -d 'pod,host=node0 count=10'

$ curl -X POST 'http://localhost:38086/write?db=kubernetes' \
  -d 'pod,host=node1 count=9'

$ curl -X POST 'http://localhost:38086/write?db=kubernetes' \
  -d 'pod,host=node2 count=12'
```

Run the following command to get the sum of all pod counts stored in the kuberentes test database:

```
$ curl -G http://localhost:38086/query?pretty=true \
  --data-urlencode "db=kubernetes" \
  --data-urlencode "q=SELECT SUM(count) FROM pod"
```

Output:

```
{
    "results": [
        {
            "series": [
                {
                    "name": "pod",
                    "columns": [
                        "time",
                        "sum"
                    ],
                    "values": [
                        [
                            "1970-01-01T00:00:00Z",
                            31
                        ]
                    ]
                }
            ]
        }
    ]
}
```

Notice we get the expected result of 31, which means everything seems to be working.

Limiting Resource Usage

Docker provides the ability to limit the amount of resources used by applications by exposing the underlying cgroup technology provided by the Linux kernel.

Limiting Memory Resources

One of the key benefits to running applications within a container is the ability to restrict resource utilization. This allows multiple applications to co-exist on the same hardware and ensure fair usage.

To limit influxdb to 200 megabytes of memory and 1 gigabyte of swap space, use the `--memory` and `--memory-swap` flags with the docker run command.

Stop and remove the current influxdb container:

```
$ docker stop influxdb
$ docker rm influxdb
```

Then start another influxdb container using the appropriate flags to limit memory usage:

```
$ docker run -d --name influxdb \
  --publish 38083:8083 \
  --publish 38086:8086 \
  --memory 200m \
  --memory-swap 1G \
  b.gcr.io/kuar/influxdb:0.9.4.2
```

Limiting CPU Resources

Another critical resource on a machine is the CPU. Restrict CPU utilization using the `--cpu-shares` flag with the docker run command:

```
$ docker run -d --name influxdb \
  --publish 38083:8083 \
  --publish 38086:8086 \
  --memory 200m \
  --memory-swap 1G \
  --cpu-shares 1024 \
  b.gcr.io/kuar/influxdb:0.9.4.2
```

Persisting Data with Volumes

Up to this point all data created by influxdb would disappear once the container is deleted. This means the data is tied to the life of the container in which it was created. For stateless applications this is what we want, but when it comes to datastores like influxdb we often want our data to persist beyond the life of the container.

Stop and remove the current influxdb container:

```
$ docker stop influxdb

$ docker rm influxdb
```

Start a new instance of the influxdb database container:

```
$ docker run -d --name influxdb \
  --publish 38083:8083 \
  --publish 38086:8086 \
  --memory 200m \
  --memory-swap 1G \
  --cpu-shares 1024 \
  b.gcr.io/kuar/influxdb:0.9.4.2
```

Run the query to sum the pod count in the kubernetes test database again:

```
$ curl -G http://localhost:38086/query?pretty=true \
  --data-urlencode "db=kubernetes" \
  --data-urlencode "q=SELECT SUM(count) FROM pod"
```

Output:

```
{
    "results": [
        {
            "error": "database not found: kubernetes"
        }
    ]
}
```

The query returns an error because the data produced by the previous influxdb container is no longer available. If we want the data to outlive the container we need to use a volume.

Let's create a directory on our container host to house that volume.

```
$ mkdir /var/lib/influxdb/
```

Now we'll use the `--volume` flag to bind mount the `/var/lib/influxdb` host directory into the influxdb container.

By default influxdb stores its data under the .influxdb in the working directory, the current directory is /, so the path will be `/.influxdb`.

```
$ docker run -d --name influxdb \
  --publish 38083:8083 \
  --publish 38086:8086 \
  --memory 200m \
  --memory-swap 1G \
  --cpu-shares 1024 \
  --volume /var/lib/influxdb:/.influxdb \
  b.gcr.io/kuar/influxdb:0.9.4.2
```

Now any data generated under the `/.influxdb` directory inside the container will be written under the `/var/lib/influxdb` on the host.

Create the kubernetes database:

```
$ curl -G 'http://localhost:38086/query' \
  --data-urlencode "q=CREATE DATABASE kubernetes"
```

Add some data:

```
$ curl -X POST 'http://localhost:38086/write?db=kubernetes' \
  -d 'pod,host=node0 count=10'

$ curl -X POST 'http://localhost:38086/write?db=kubernetes' \
  -d 'pod,host=node1 count=9'

$ curl -X POST 'http://localhost:38086/write?db=kubernetes' \
  -d 'pod,host=node2 count=12'
```

Run the pod sum query:

```
$ curl -G http://localhost:38086/query?pretty=true \
  --data-urlencode "db=kubernetes" \
  --data-urlencode "q=SELECT SUM(count) FROM pod"
```

Output:

```
{
    "results": [
        {
            "series": [
                {
                    "name": "pod",
                    "columns": [
                        "time",
                        "sum"
                    ],
                    "values": [
                        [
                            "1970-01-01T00:00:00Z",
                            31
                        ]
                    ]
                }
            ]
        }
    ]
}
```

We are ready to test data persistence by destroying the influxdb container and recreating it. First stop and remove the current influxdb container:

```
$ docker stop influxdb

$ docker rm influxdb
```

Create the influxdb container again attaching the same host volume:

```
$ docker run -d --name influxdb \
  --publish 38083:8083 \
  --publish 38086:8086 \
  --memory 200m \
  --memory-swap 1G \
  --cpu-shares 1024 \
  --volume /var/lib/influxdb:/.influxdb \
  b.gcr.io/kuar/influxdb:0.9.4.2
```

Run the pod sum query again:

```
$ curl -G http://localhost:38086/query?pretty=true \
  --data-urlencode "db=kubernetes" \
  --data-urlencode "q=SELECT SUM(count) FROM pod"
```

Output:

```
{
    "results": [
        {
            "series": [
                {
                    "name": "pod",
                    "columns": [
                        "time",
                        "sum"
                    ],
                    "values": [
                        [
                            "1970-01-01T00:00:00Z",
                            31
                        ]
                    ]
                }
            ]
        }
    ]
}
```

Notice we get the same value as before. This is because the data produced by the influxdb container lives on the host and will outlive the container.

Summary

Application containers provide a clean abstraction for applications, and when packaged in the Docker image format, applications become easy to build, deploy, and distribute. Containers also provide isolation between applications running on the same machine, which helps avoid dependency conflicts. The ability to bind mount external directories means we can run not only stateless applications in a container, we can also run applications like influxdb which generate lots of data.

CHAPTER 4
Pods

A pod represents a collection of application containers and volumes running in the same execution environment. Pods, not containers, are the smallest deployable artifact in a Kubernetes cluster.

Each container within a pod runs in its own cgroup, but share the following Linux namespaces:

- IPC
- Network
- UTC

Applications running in the same pod share the same IP and port space (Network namespace), have the same hostname (UTC namespace), and can communicate using native interprocess communication channels over SystemV IPC or POSIX message queues (IPC namespace). However other pods are isolated, and containers in separate pods running on the same node might as well be on another server.

Pod Manifest

Pods are described in a pod manifest. Pod manifests are processed by the Kubernetes API service, stored in persistent storage, and then scheduled to a Node by the Kubernetes scheduler. Once scheduled to a Node, Pods don't move and must be explicitly destroyed and rescheduled.

Multiple instances of a Pod can be deployed by repeating the above workflow, however Replication Controllers are better suited for running multiple instances of a Pod. They're also better suited for running a single Pod, but we'll get into that later.

Creating a Pod Manifest

Pod manifests can be written using YAML or JSON, but YAML should be preferred due to the ability to add comments. Pod manifests include a couple of key fields and attributes, mainly a metadata section for describing the pod and its labels, a spec section for describing volumes, and a list of containers that will run in the pod.

In the previous chapter we deployed influxdb using the following docker command:

```
$ docker run -d --name influxdb \
  --publish 38083:8083 \
  --publish 38086:8086 \
  b.gcr.io/kuar/influxdb:0.9.4.2
```

The `b.gcr.io/kuar/influxdb:0.9.4.2` docker image can be deployed to a Kubernetes cluster by writing the following contents to a file named `influxdb-pod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: influxdb
  name: influxdb
spec:
  containers:
    - image: b.gcr.io/kuar/influxdb:0.9.4.2
      name: influxdb
      ports:
        - containerPort: 8083
          name: admin
          protocol: TCP
        - containerPort: 8086
          name: http
          protocol: TCP
```

Running Pods

In the previous section we created a pod manifest which can be used to start a pod running influxdb. Use the `kubectl create` command to launch a single instance of influxdb:

```
$ kubectl create -f influxdb-pod.yaml
```

The pod manifest will be submitted to the Kubernetes API server, then it will be scheduled to run on a healthy node in the cluster, where it will be monitored by the Kubelet. Don't worry if you don't understand all the moving parts of Kubernetes right now, we'll go into more details throughout the book.

Listing Pods

```
$ kubectl get pods
```

Output:

```
NAME       READY     STATUS    RESTARTS   AGE
influxdb   1/1       Running   0          44s
```

Pod Details

Describe the influxdb pod using the `kubectl describe pods` command:

```
$ kubectl describe pods influxdb
```

Output:

```
Name:                           influxdb
Namespace:                      default
Image(s):                       b.gcr.io/kuar/influxdb:0.9.4.2
Node:                           node1/10.240.0.3
Labels:                         name=influxdb
Status:                         Running
Reason:
Message:
IP:                             10.200.1.4
Replication Controllers:        <none>
Containers:
  influxdb:
    Image:              b.gcr.io/kuar/influxdb:0.9.4.2
    State:              Running
      Started:          Mon, 05 Oct 2015 09:38:19 +0000
    Ready:              True
    Restart Count:      0
Conditions:
  Type          Status
  Ready  True
Events:
...
```

Health Checks

Kubernetes has a strong focus on running containers in production and that means we need a way to ensure pods are actually running and healthy after we launch them. To facilitate this, Kubernetes provides a way to declare if a pod is ready using a readiness probe.

ReadinessProbe

Readiness probes allow you to specify checks to determine when a pod is ready for use. There are two methods that can be used to determine readiness. HTTP or Exec.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: influxdb
  name: influxdb
spec:
  containers:
    - image: b.gcr.io/kuar/influxdb:0.9.4.2
      name: influxdb
      readinessProbe:
        httpGet:
          path: /ping
          port: 8086
        initialDelaySeconds: 5
        timeoutSeconds: 1
```

The above pod manifest uses the httpGet readiness probe to perform an HTTP get request against the /ping endpoint on port 8086 of the influxdb container. The readiness probe will not be called until 5 seconds after the all containers in the pod are created. The readiness probe must respond within the 1 second timeout, and the HTTP status code must be equal to or greater than 200 and less than 400 to be considered successful.

LivenessProbe

Once the influxdb pod is up and running we need a way to confirm that it's still healthy and ready for action. Just like the readiness probe, a liveness probe can be used to preform a set of health checks via HTTP or exec.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: influxdb
  name: influxdb
spec:
  containers:
    - image: b.gcr.io/kuar/influxdb:0.9.4.2
      name: influxdb
      livenessProbe:
        httpGet:
          path: /ping
          port: 8086
```

```
        initialDelaySeconds: 15
        timeoutSeconds: 1
```

We are reusing the /ping endpoint to verify the health of the influxdb pod. It's also possible to use another endpoint or leverage a command line utility in the container via the exec probe.

Combining the readiness and liveness probes help ensure only healthy containers are running within the cluster.

Resource Management

Like Docker, Kubernetes provides support for cgroups and placing limits on CPU and memory usage for each container in a pod. To request that the influxdb container lands on a machine with half a CPU free, and gets 128 megabytes of memory allocated to it, set the following resource limits:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: influxdb
  name: influxdb
spec:
  containers:
    - image: b.gcr.io/kuar/influxdb:0.9.4.2
      name: influxdb
      resources:
        limits:
          cpu: "500m"
          memory: "128Mi"
```

The CPU limits directive works as anticipated when requesting resources from the cluster. It will only schedule the pod to a node that has at least .5 CPU unclaimed. However once it lands on a node, it will consume all available CPU until another pod is scheduled to the node. Let's assume our node has 2 cores, if another pod that needs .5 CPU lands on the node, and our pod contains a multi-threaded app, each pod will now consume 50% of each core on the machine. If another similar pod that wants .5 CPU is scheduled, each app will now take 33% of each core. This is because CPU restrictions are implemented via the cpu-shares functionality of Docker.

Since pods are placed on machines by a scheduler, resource limits are critical to avoid resource shortages during high load situations.

Persisting Data with Volumes

Pods provide a way to persist data through the use of volumes. What makes volumes unique in Kubernetes is the fact that volumes are managed by the Kubelet. Which means volumes can be created on the fly.

Using Volumes with Pods

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: influxdb
  name: influxdb
spec:
  volumes:
    - name: "influxdb-data"
      hostPath:
        path: "/var/lib/influxdb"
  containers:
    - image: b.gcr.io/kuar/influxdb:0.9.4.2
      name: influxdb
      volumeMounts:
        - mountPath: ".influxdb"
          name: "influxdb-data"
```

While the above works, there is one glaring problem. What happens when the pod is rescheduled to another machine? That's right, the data will not move. The data will stay on the machine where the data directory lives.

Thanks to volume plugins we have the ability to utilize different storage technologies, some of which can be persisted and paired to our pods no matter what node they land on.

Volume Plugins

Kubernetes supports various volume plugins including the following:

- emptyDir
- hostPath
- gcePersistentDisk
- awsElasticBlockStore
- nfs

There are several other volume plugins supported by Kubernetes, which will be covered in detail in the Volumes chapter.

Persist data using a GCE Persistent Disk

First create a GCE disk using the gcloud command line tool:

```
$ gcloud compute disks create influxdb --size 20GB
```

Next, edit the influxdb pod manifest to reference the influxdb GCE disk:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: influxdb
  name: influxdb
spec:
  volumes:
    - name: "influxdb-data"
      gcePersistentDisk:
        pdName: influxdb
        fsType: ext4
  containers:
    - image: b.gcr.io/kuar/influxdb:0.9.4.2
      name: influxdb
      volumeMounts:
        - mountPath: ".influxdb"
          name: "influxdb-data"
```

Launch the influxdb pod:

```
$ kubectl create -f influxdb-pod.yaml
```

View the results:

```
$ kubectl describe pods influxdb
```

Output:

```
Name:                           influxdb
Namespace:                      default
Image(s):                       b.gcr.io/kuar/influxdb:0.9.4.2
Node:                           node0/10.240.0.5
Labels:                         name=influxdb
Status:                         Running
...
```

Putting it all together

The stateful nature of influxdb means we must perserve any data written to the database and ensure access to the underlying storage volume regardless of what machine influxdb runs on. As we saw earlier this can be achieved using a persistent volume backed by network attached storage. We also want to ensure a healthy instance of

influxdb is running at all times, which means we want to make sure the container running influxdb is ready before we expose it to clients.

Through a combonation of persistent volumes, readiness and liveness probes, and resource restrictions Kubernetes provides everything require to run stateful services like influxdb.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: influxdb
  name: influxdb
spec:
  volumes:
    - name: "influxdb-data"
      gcePersistentDisk:
        pdName: influxdb
        fsType: ext4
  containers:
    - image: b.gcr.io/kuar/influxdb:0.9.4.2
      name: influxdb
      resources:
        limits:
          cpu: "500m"
          memory: "128Mi"
      volumeMounts:
        - mountPath: ".influxdb"
          name: "influxdb-data"
      livenessProbe:
        httpGet:
          path: /ping
          port: 8086
        initialDelaySeconds: 15
        timeoutSeconds: 1
      readinessProbe:
        httpGet:
          path: /ping
          port: 8086
        initialDelaySeconds: 5
        timeoutSeconds: 1
```

Summary

Pods represent the smallest unit of work in a Kubernetes cluster. Pods are composed of one or more containers, which have hopefully been optimized for size and security. Pods are described in a Pod manifest and these manifests get submitted to the Kubernetes API by either making direct HTTP calls to the Kubernetes API server or indirectly by using the kubectl CLI tool. Pod manifests are stored in persistent storage by

the Kubernetes API server and a cluster event is fired off which triggers the scheduling process.

Once a Pod is scheduled to a node, no rescheduling occurs in case that node goes offline, and to spin up multiple pods we have to create and name them manually. To accomplish those things in a scalable fashion, we need to use a replication controller.

CHAPTER 5

Labels and Annotations

Labels are key/value pairs which can be attached to Kubernetes objects such as pods and replication controllers. They can be arbitrary, and are useful for attaching identifying information to Kubernetes objects. Labels also provide the foundation for grouping objects.

Annotations on the other hand provide a storage mechanism that resembles labels. Annotations are key/value pairs designed to hold non identifying information which can be leveraged by tools and libraries.

Labels

Labels are key/value pairs, where both the key and value are represented by strings. Label keys can be broken down into two parts: an optional prefix and a name separated by a slash. The prefix, if specified, must be a DNS subdomain with a 253 character limit. The key name is required and must be shorter than 63 characters. Names must also start and end with a alphanumeric character and permits the use of dashes(-), underscores(_), and dots(.) between characters.

Label values are strings with a maximum length of 63 characters. The contents of the label value follow the same rules as label keys.

The following chart shows valid label keys and values.

Table 5-1. A Table

key	value
acme.com/app-version	1.0.0

key	value
appVersion	1.0.0
app.version	1.0.0
kubernetes.io/cluster-service	true

Label Selectors

Label selectors are used to filter Kubernetes objects based on a set of labels. Labels are the only way to group objects in Kubernetes.

Filtering Objects

In this section we will explore filtering pods using kubectl and label selectors. First create two replication controllers using the `kubectl run` command.

Create the `nginx-prod` replication controller and set the ver, app, and env labels:

```
$ kubectl run nginx-prod \
  --image=nginx:1.7.12 \
  --replicas=3 \
  --labels="ver=1.7.12,app=nginx,env=prod"
```

Create the `nginx-test` replication controller and set the ver, app, and env labels:

```
$ kubectl run nginx-test\
  --image=nginx:1.9.10 \
  --replicas=2 \
  --labels="ver=1.9.10,app=nginx,env=test"
```

At this point you should have two replications controllers: nginx-prod and nginx-test

```
$ kubectl get rc

CONTROLLER  CONTAINER(S)  IMAGE(S)      SELECTOR
nginx-prod  nginx-prod    nginx:1.7.12  app=nginx,env=prod,ver=1.7.12
nginx-test  nginx-test    nginx:1.9.10  app=nginx,env=test,ver=1.9.10
```

Each replication controller will create a set of pods using the labels specified at creation time. Running the `kubectl get pods` command should return all the pods currently running in the cluster. You should have a total of 5 nginx pods.

```
$ kubectl get pods

NAME               READY     STATUS    RESTARTS   AGE
nginx-prod-32s90   1/1       Running   0          1m
nginx-prod-ht98o   1/1       Running   0          1m
nginx-prod-yflos   1/1       Running   0          1m
```

```
nginx-test-cpw8l   1/1       Running   0          56s
nginx-test-qp9fv   1/1       Running   0          56s
```

If we only wanted to list pods that had a label set to `ver=1.7.12` we could use the `--selector` flag:

```
$ kubectl get pods --selector="ver=1.7.12"

NAME               READY     STATUS    RESTARTS   AGE
nginx-prod-32s90   1/1       Running   0          2m
nginx-prod-ht98o   1/1       Running   0          2m
nginx-prod-yflos   1/1       Running   0          2m
```

One or more labels can be used with the `--selector` flag to perform label queries.

Annotations

Annotations provide a place to store additional metadata for Kubernetes objects with the sole purpose of assisting tools and libraries. Common uses for annotation data include automated deployments, security policies, and supporting multiple schedulers in a single Kubernetes cluster. Think of them as "hidden" labels that you use when you don't want to clutter object descriptions with too many labels.

Annotations are used in various places in Kubernetes with the primary use case being rolling deployments. During rolling deployments, Annotations are used to track rollout status and provide the necessary information required to rollback a deployment to a previous state.

Defining Annotations

Summary

Labels are used to identify and optionally group objects in a Kubernetes cluster. Labels are used in selector queries to provide flexible runtime grouping of object such as pods. Annotations provide object scoped key/value storage of metadata that can be used by automation tooling and client libraries. Annotations can also be used to hold configuration data for external tools such as third party schedulers and monitoring tools.

Labels and annotations are the key to understanding how key components in a Kubernetes cluster work together to ensure the desired cluster state. Using labels and annotations properly unlocks the true power of Kubernetes flexibility and provides the starting point for building automation tools and deployment workflows.

CHAPTER 6

Services

One of the challenges of running a distributed system is keeping track of applications and the services they provide. When a Pod is scheduled in Kubernetes, it is assigned to a random node, and given a random IP. You may be tempted to use this IP, but it is ephemeral. It will change the next time you deploy the Pod, it will change if Kubernetes (in its constant pursuit of high availability) reschedules your Pod, and it will change because it's Tuesday and there's no more waffles. If you're still tempted to use the Pod IP, think about what happens when you need to scale your application horizontally. Now you have anywhere from from 2 to 2000 Pod IPs, and no matter which ones you hardcode into your application, you're going to be sad.

So how do consumers of your service locate applications running in a Kubernetes cluster? The solution to this problem is a concept called Service Discovery. Service discovery is a mechanism that enables clients of a service to locate service endpoints without manual intervention. DNS is the most common implementation of service discovery in use today.

Kubernetes provides service discovery out of the box in the form of an Endpoints API, DNS, and environment variables which can be exposed to Pods. Each service discovery mechanism is backed by Kubernetes Services and Endpoints.

Endpoints

Endpoints hold a collection of IP addresses and ports that back a Kubernetes Service. Endpoints are defined using the Kubernetes Endpoints API object. The following Kubernetes configuration represents an Endpoint for the "my-service" Service with a single service endpoint at 10.52.0.2 on port 80:

```
apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
  namespace: default
subsets:
  - addresses:
      - ip: 10.52.0.2
    ports:
      - port: 80
```

Under most circumstances the IP addresses listed on the Endpoint map to Pods that back the service. In other cases endpoints may map to external services such as a hosted database or caching service. While Endpoint objects can be created directly they are typically created and managed by the Endpoint Controller.

Endpoints are normally create and managed by the Endpoints Controller and Service objects. Only in rare cases would you need to manage endpoints yourself.

Endpoint Controller

The Endpoint Controller is built into the Kubernetes Controller Manager and is responsible for managing Endpoints based on Services. Whenever a Service is created the Endpoint Controller creates a corresponding Endpoint with the same name, and populates the Endpoint's list of backends via a label query based on the label selector defined on the Service.

The Endpoint Controller continuously monitors the Kubernetes API for new Services and modifications to existing Services. If the label query for a specific Service is modified, then the Endpoint Controller recalculates the backends for the corresponding Endpoint.

Virtual IPs and Service Proxies

Kubernetes Services are typically backed by virtual IP addresses, which provide a stable endpoint for Pods identified by a common set of labels. When consumers of a Service target a virtual IP, those consumers are transparently proxied to a Pod in the service group. Services backed by more than one Pods will ensure consumers are distributed across each Pod using a simple round-robin load balancing algorithm.

When Services are created via the Kubernetes API, a virtual IP address is allocated to the Service from the Kubernetes Service address range. One thing to keep in mind Virtual IP addresses are not assigned to any network interfaces on the hosts or Pods.

Virtual IP addresses must be routed in order to work. There are a few ways to go about this. One way would be to create a router entry for each Virtual IP address with the next-hop set to one of the worker nodes in the Kubernetes cluster. While this will work, this approach has some major drawbacks in terms of management overhead.

The Kubernetes Service address range is configured using the `--service-cluster-ip-range` flag on the kube-apiserver binary. The Service address range should not overlap with the IP subnets and ranges assigned to each Docker bridge or Kubernetes node.

Service Proxies

A local proxy service running on every node creates a set of iptables rules that load balances traffic coming in via the virtual IP across all Pods using a simple round-robin load balancing algorithm.

The service proxy aims to provide a simple load balancing solution out of the box. Cluster administrators can also use external load balancers such as Nginx, HAProxy, or even hardware solutions like an F5. This bypasses the Kubernetes proxy altogether. In these types of deployments endpoints must be programmatically synced from the Kubernetes API to a format consumable by your load balancer of choice.

Creating Services

Kubernetes Services can be created by submitting a Service definition to the Kubernetes API. In this section we will define a Service using a Service configuration file, and then submit it to the Kubernetes API using the `kubectl create`. Before we start creating Services it's critical to understand the different types of services that can be created.

Kubernetes supports 4 categories of services.

Table 6-1. A Table

Service Type	Description
External	External services live outside of the Kubernetes cluster or in a different namespace.
Headless	Kubernetes does not provide a cluster IP or load-balancing.
Internal	A cluster IP address is allocated and load-balancing is setup.
Exposed	Exposed services provide external access to cluster services via external load-balancers or ports on cluster nodes.

Internal services are the most common service type, and provide everything you need for inter-cluster communication. Internal services make it easy for one group of Pods to communication with another set using a stable endpoint. For services such as web frontends that must be exposed to external clients, an exposed service should be used. Exposed services serve as the integration point between external load balancers such as the Google Cloud Load Balancer or Amazon's Elastic Load Balancer (ELB).

External Load Balancer integration depends heavily on Kubernetes' support for your cloud provider. Most popular cloud platforms are supported out of the box. Use the `--cloud-provider` flag on the kube-apiserver binary to select the cloud provider Kubernetes should interact with when creating exposed services.

Service Specification

Services are decoupled from Pods and can be created before or after Pods they expose. Let's start by creating a service without any Pods. Save the following contents to a file named `helloworld.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: helloworld
  name: helloworld
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: helloworld
```

The spec section is where we define our service and it's helpful to walk through it. Using the ports key we define one or more port mappings between the service and Pod endpoints. The targetPort attribute is required to ensure incoming traffic to the Service is routed to the correct container in the Pods making up the service. Services also provide the configuration for the Kubernetes proxy service which provides L4 load balancing across Pods in a service. Both TCP and UDP protocols are supported.

The selector key is where labels are leveraged to identify which Pods should receive traffic from a Service. Labels allow Pods to be discovered dynamically within a Kubernetes cluster and will be covered in more detail in the Label Queries section.

Submit the helloworld service to the Kubernetes API using the `kubectl create` command:

```
$ kubectl create -f helloworld.yaml
service "helloworld" created
```

Managing Services

Services are managed using the Kubernetes API, and the easiest way to do that is via the kubectl client. To list all services in the current namespace use the `get services` command:

```
$ kubectl get services

NAME         CLUSTER_IP       EXTERNAL_IP   PORT(S)   SELECTOR         AGE
helloworld   10.223.244.100   <none>        80/TCP    app=helloworld   20s
kubernetes   10.223.240.1     <none>        443/TCP   <none>           9m
```

Notice there is a Kubernetes service in the output. How did it get there? By default kubernetes creates a service entry for the Kubernetes API server to streamline its discovery by other applications in the cluster.

We can get more details about the helloworld service using the `describe services` command:

```
$ kubectl describe services helloworld

Name:             helloworld
Namespace:        default
Labels:           app=helloworld
Selector:         app=helloworld
Type:             ClusterIP
IP:               10.223.244.100
Port:             <unnamed>       80/TCP
Endpoints:        <none>
Session Affinity: None
No events.
```

Notice that the service has an IP address allocated to it. This IP address was assigned by the Kubernetes control plane when the service was created. When a Kubernetes cluster is bootstrapped an IP range is set using the `--service-cluster-ip-range` flag. Kubernetes keeps track of these IP addresses to ensure each Service has a unique IP address. In this example we allowed Kubernetes to generate a cluster IP for us, but you have the option to assign a fixed address.

When allocating a fixed cluster IP address it must fall within range of the address blocked assigned using the `--service-cluster-ip-range` flag.

Use the `spec.clusterIP` key to assign a fixed cluster IP to the helloworld service.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: helloworld
  name: helloworld
spec:
  clusterIP: 10.223.244.110
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: helloworld
```

The `spec.clusterIP` field is immutable, so the Service must be deleted and recreated in order to change the IP. Use the `kubectl delete services` command to delete the helloworld service:

```
$ kubectl delete services helloworld
```

Use the `kubectl create` command to recreate the helloworld service using the updated service configuration file:

```
$ kubectl create -f helloworld.yaml
```

Verify the helloworld service was created with the fixed cluster IP address of 10.223.244.110:

```
Name:              helloworld
Namespace:         default
Labels:            app=helloworld
Selector:          app=helloworld
Type:              ClusterIP
IP:                10.223.244.110
Port:              <unnamed>       80/TCP
Endpoints:         <none>
Session Affinity: None
No events.
```

Service Discovery

Environment variables

Environment variables are created for every service defined. These variables are made available to every pod in the namespace.

```
the variables made available
```

These variables can be consumed directly by your applications, or used to generate configuration files.

Environment variables are not dynamically updated. For example, if you recreate a service with a new cluster ip, those changes will not be reflected until the Pod is restarted. Similarly if a Pod is started before a Service is defined, the Pod will never receive the environment variables.

Due to these shortcomings, DNS is the preferred way to communicate information about Services.

DNS

In most Kubernetes deployments Services will be mapped to DNS names using the skyDNS cluster add-on. The skyDNS add-on creates DNS entries for each Service defined in Kubernetes, allowing Pods to reach services by referring to the Service name. Unlike environment variables, DNS entries will always be up to date. You can re-create a service with a new IP, and DNS will be updated accordingly. Similarly, Pods can be created before a Service is actually defined.

Exposing Services

NodePort

LoadBalancer

External IPs

Summary

Services give Pods the flexibility to scale and be rescheduled without affecting the consumers of those Pods. Label queries provide the foundation for "service discovery", which allows Pods to be dynamically added and removed from Services without manual configuration. The Services API also provides the key integration point for cluster DNS, external load balancers, and the Kubernetes proxy server.

CHAPTER 7

ReplicaSets

In this chapter we introduce one of the most powerful constructs in Kubernetes, the ReplicaSet. ReplicaSets act as a cluster-wide Pod manager, ensuring that the right type and amount of Pods are running at all times.

ReplicaSets are the building blocks used to describe common application deployment patterns and provide the underpinnings of self-healing for our applications at the infrastructure level. Pods managed by ReplicaSets are automatically rescheduled under certain failure conditions such as node failures and network partitions.

This chapter will also explore the ReplicaSet specification and how to manage ReplicaSets using the kubectl command line tool.

ReplicaSet Spec

Like all concepts in Kubernetes, ReplicaSets are defined using a specification. All ReplicaSets must have a unique name (defined using the metadata.name field), a spec section that describes the number of Pods (replicas) that should be running cluster-wide at a given time, and a Pod template that describes the Pod to be created when the defined amount of replicas are not met.

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: helloworld-v1
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: helloworld
```

```
      version: v1
  template:
    metadata:
      labels:
        app: helloworld
        version: v1
    spec:
      containers:
        - name: helloworld
          image: kelseyhightower/helloworld:v1
          ports:
            - containerPort: 80
```

Pod Template

ReplicaSets create new Pods from a template. When the number of running Pods is less then the number of replicas desired, the ReplicaSet controller submits new Pods to the Kubernetes API, which will result in an unbound Pod. Unbound Pods will be scheduled to nodes by the scheduler.

```
template:
  metadata:
    labels:
      app: helloworld
      version: v1
  spec:
    containers:
      - name: helloworld
        image: kelseyhightower/helloworld:v1
        ports:
          - containerPort: 80
```

Labels

ReplicaSets monitor cluster state using a set of Pod labels. Labels are used to filter Pod listings and track Pods running within a cluster. When ReplicaSets are initially created, the Replication Controller fetches a Pod listing from the Kubernetes API and filters the results by labels. Based on the number of Pods returned by the query, the Replication Controller deletes or creates Pods to meet the desired number of replicas. The labels used for filtering are defined in the ReplicaSet spec section and are the key to understanding how ReplicaSets work.

The selector in the ReplicaSet spec should be a significant subset of the labels in the Pod template.

Creating a ReplicaSet

ReplicaSets are created by submitting a ReplicaSet object to the Kubernetes API. In this section we will create a ReplicaSet using a configuration file and the `kubectl create` command.

The following ReplicaSet configuration file will ensure 1 copy of the `nginx:1.9.15` container is running at a given time.

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: nginx
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
        version: "1.9.15"
    spec:
      containers:
        - name: nginx
          image: "nginx:1.9.15"
```

Use the `kubectl create` command to submit the nginx ReplicaSet to the Kubernetes API:

```
$ kubectl create -f nginx-rs.yaml
replicaset "nginx" created
```

Once the nginx ReplicaSet has been accepted the Replication Controller will detect there are no nginx Pods running that match the desired state, and a new nginx Pod will be created based on the contents of the Pod template.

```
$ kubectl get pods
NAME          READY     STATUS    RESTARTS   AGE
nginx-yvzgd   1/1       Running   0          11s
```

Inspecting a ReplicaSet

```
$ kubectl describe rs nginx
Name:         nginx
Namespace:    default
Image(s):     nginx:1.9.15
Selector:     app=nginx,version=1.9.15
Labels:       app=nginx,version=1.9.15
Replicas:     1 current / 1 desired
Pods Status:  1 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
```

Scaling ReplicaSets

ReplicaSets are scaled up or down by updating the `spec.replicas` key on the ReplicaSet object stored in Kubernetes. When a ReplicaSet is scaled up new Pods are submitted to the Kubernetes API using the Pod template defined on the ReplicaSet.

To scale the nginx ReplicaSet edit the nginx.yaml configuration file and increase the replicas count to 3.

```
spec:
  replicas: 3
```

Use the `kubectl apply` command to submit the updated nginx ReplicaSet to the API server:

```
$ kubectl apply -f nginx.yaml
replicaset "nginx" configured
```

Now the updated nginx ReplicaSet is in place, the Replication Controller will detect that 2 additional nginx Pods must be created to match the desired state, and will submit 2 new Pods to the Kubernetes API using the Pod template defined on the nginx ReplicaSet. Use the `kubectl get pods` command to list the running nginx Pods.

```
$ kubectl get pods
NAME          READY     STATUS    RESTARTS   AGE
nginx-3a2sb   1/1       Running   0          26s
nginx-wuq9v   1/1       Running   0          26s
nginx-yvzgd   1/1       Running   0          2m
```

Deleting ReplicaSets

When a ReplicaSet set is no longer required it can be deleted using the `kubectl delete` command.

```
$ kubectl delete rs nginx
replicaset "nginx" deleted
```

Deleting a ReplicaSet will also delete all Pods that match the label selector defined on the ReplicaSet. Set the `--cascade` flag to false to ensure only the ReplicaSet object is deleted and not the Pods.

Running the `kubectl get pods` command shows that all the nginx Pods created by the nginx ReplicaSet have also been deleted.

```
$ kubectl get pods
```

If you want to keep Pods managed by a ReplicaSet after the ReplicaSet is deleted used the `--cascade` flag when deleting the ReplicaSet:

```
$ kubectl delete rs --cascade=false nginx
```

Summary

Composing Pods with ReplicaSets provides the foundation for building robust applications with automatic failover, and makes deploying those applications a breeze by enabling scalable and sane deployment patterns. ReplicaSets should be used for any Pod you care about, even if it is a single Pod! Some people even default to using ReplicaSets instead of Pods. A typical cluster will have many ReplicaSets, so please apply liberally to the affected area.

CHAPTER 8

DaemonSets

A DaemonSet ensures a copy of a Pod is running across one or more nodes in a Kubernetes cluster. DaemonSets are used to deploy system daemons such as log collectors and monitoring agents, which typically must run on every node. DaemonSets share similar functionality with ReplicaSets; both create pods which are expected to be long running services and ensure Pods are rescheduled in the event of node failures.

Given the similarities between DaemonSets and ReplicaSets it's important to understand when to use one over the other. ReplicaSets should be used when your application is completely decoupled from the node and can run multiple copies on a given node without special consideration. DaemonSets should be used when a single copy of your application must run on all, or a subset of, nodes in the cluster.

DaemonSet Scheduler

By default a DaemonSet will create a copy of a Pod on every node unless a node selector is used, which would limit eligible nodes to those with a matching set of labels. DaemonSets determine which node a Pod will run on during Pod creation time by specifying the `nodeName` field on the Pod spec. As a result Pods created by DaemonSets are ignored by the Kubernetes Scheduler.

Creating DaemonSets

DaemonSets are created by submitting a DaemonSet configuration to the Kubernetes API server. The following DaemonSet will create a Nginx Pod on every node in the target cluster.

```
apiVersion: extensions/v1beta1
kind: "DaemonSet"
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.10.0
```

DaemonSets require a unique name across all DaemonSets in a given Kubernetes namespace. Each DaemonSet must include a Pod template spec, which will be used to create Pods when necessary. This is where the similarities between ReplicaSets and DaemonSets ends. Unlike ReplicaSets, DaemonSets will create Pods on every node in the cluster by default unless a node selector is used.

Once you have a valid DaemonSet configuration in place, you can use the `kubectl create` command to submit the DaemonSet to the Kubernetes API. In this section we will create a DaemonSet to ensure the Nginx HTTP server is running on every node in our cluster.

```
$ kubectl create -f nginx.yaml
daemonset "nginx" created
```

Once the nginx DaemonSet has been successfully submitted to the Kubernetes API, you can query its current state using the `kubectl describe` command.

```
$ kubectl describe daemonset nginx
Name:           nginx
Image(s):       nginx:1.10.0
Selector:       app=nginx
Node-Selector:  <none>
Labels:         app=nginx
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
```

Based on the above output a nginx Pod was successfully deployed to all 3 nodes in our cluster. We can verify this using the `kubect get pods` command with the `-o` flag to print the nodes where each nginx Pod was assigned.

```
$ kubectl get pods -o wide
```

```
NAME          AGE    NODE
nginx-1q6c6   13m    gke-k0-default-pool-35609c18-z7tb
nginx-mwi7h   13m    gke-k0-default-pool-35609c18-ydae
nginx-zr6l7   13m    gke-k0-default-pool-35609c18-pol3
```

With the nginx DaemonSet in place adding a new node to the cluster will result in a nginx Pod deployed to that node automatically. This is exactly the behavior you want when managing logging daemons and other cluster wide services.

```
$ kubectl get pods -o wide
NAME          AGE    NODE
nginx-1q6c6   13m    gke-k0-default-pool-35609c18-z7tb
nginx-mwi7h   13m    gke-k0-default-pool-35609c18-ydae
nginx-oipmq   43s    gke-k0-default-pool-35609c18-0xnl
nginx-zr6l7   13m    gke-k0-default-pool-35609c18-pol3
```

After adding a new node to our cluster you can see from the above output a new nginx Pod was created by the nginx DaemonSet. No action was required from our end; this is how Kubernetes enforces our desired state in action.

Limiting DaemonSets to Specific Nodes

The most common use case for DaemonSets is to run a Pod across every node in a Kubernetes cluster. However, there are some cases where you want to deploy a Pod to only a subset of nodes. For example, maybe you have a workload that requires a GPU or access to fast storage only available on a subset of nodes in your cluster. In cases like these node labels can be used to tag specific nodes that meet workload requirements.

Adding Labels to Nodes

The first step in limiting DaemonSets to specific nodes is to add the desired set of labels to a subset of nodes. This can be achieved using the `kubectl label` command.

The following command adds the `ssd=true` label to a single node.

```
$ kubectl label nodes gke-k0-default-pool-35609c18-z7tb ssd=true
node "gke-k0-default-pool-35609c18-z7tb" labeled
```

Just like other Kubernetes resources, listing nodes without a label selector returns all nodes in the cluster.

```
$ kubectl get nodes
NAME                                STATUS    AGE
gke-k0-default-pool-35609c18-0xnl   Ready     23m
gke-k0-default-pool-35609c18-pol3   Ready     1d
gke-k0-default-pool-35609c18-ydae   Ready     1d
gke-k0-default-pool-35609c18-z7tb   Ready     1d
```

Using a label selector we can filter nodes based on labels. To list only the nodes that have the ssd label set to true, use the `kubectl get nodes` command with the `--selector` flag:

```
$ kubectl get nodes --selector ssd=true
NAME                                STATUS    AGE
gke-k0-default-pool-35609c18-z7tb   Ready     1d
```

Node Selectors

Node selectors can be used to limit what nodes a Pod can run on in a given Kubernetes cluster. Node selectors are defined as part of the Pod spec when creating a DaemonSet. The following DaemonSet configuration limits nginx to running only on nodes with the `ssd=true` label set.

```
apiVersion: extensions/v1beta1
kind: "DaemonSet"
metadata:
  labels:
    app: nginx
    ssd: "true"
  name: nginx-fast-storage
spec:
  template:
    metadata:
      labels:
        app: nginx
        ssd: "true"
    spec:
      nodeSelector:
        ssd: "true"
      containers:
        - name: nginx
          image: nginx:1.10.0
```

Lets see what happens when we submit the `nginx-fast-storage` DaemonSet to the Kubernetes API.

```
kubectl create -f nginx-fast-storage.yaml
daemonset "nginx-fast-storage" created
```

Since there is only one node with the `ssd=true` label, the `nginx-fast-storage` Pod will only run on that node.

```
$ kubectl get pods -o wide
NAME                       STATUS    NODE
nginx-fast-storage-7b90t   Running   gke-k0-default-pool-35609c18-z7tb
```

Adding the `ssd=true` label to additional nodes will case the `nginx-fast-storage` Pod to be deployment on those nodes. The inverse is also true, if a required label is removed from a node, then the Pod will be removed by the DaemonSet controller.

Removing labels from a node that are required by a DaemonSets node selector will cause the Pod being managed by that DaemonSet to be removed from the node.

Updating a DaemonSet

DaemonSets are great for deploying services across an entire cluster, but what about upgrades? Currently the only way to update Pods managed by a DaemonSet is to first delete the DaemonSet and recreate it. This approach has a major drawback, and that's downtime. When a DaemonSet is deleted all Pods managed by that DaemonSet will also be deleted. Depending on the size of your container images, recreating a DaemonSet may push you outside of your SLA thresholds, so it might be worth considering pulling updated container images across your cluster before updating a DaemonSet.

Deleting a DaemonSet

Deleting a DaemonSet is pretty straightforward using the `kubectl delete daemon sets` command. Just be sure to supply the correct name of the DaemonSet you would like to delete. Keep in mind that deleting a DaemonSet also means deleting the Pods being managed by that DaemonSet.

Deleting a DaemonSet will also delete all the Pods being managed by that DaemonSet. Set the `--cascade` flag to false to ensure only the DaemonSet is deleted and not the Pods.

Summary

DaemonSets provide an easy to use abstraction for running a set of Pods on every node in a Kubernetes cluster, or if the case requires it, on a subset of nodes based on labels. DaemonSet provides its own controller and scheduler to ensure key services like monitoring agents are always up and running on the right nodes in your cluster.

CHAPTER 9

Jobs

So far we have focused on long running processes such as databases and web applications. These types of workloads run until they are either upgraded or the service is no longer needed. While long running processes make up the large majority of workloads that run on a Kubernetes cluster, there is often a need to run short lived one-off tasks. The Jobs object is made for handling these types of tasks

A Job creates Pods that run until succesful termination (i.e. exits with 0), as opposed to a regular Pod that will continually restart regardless of exit code. This pattern is useful for things you only want to do once, such such as database migrations or batch jobs. If run as a regular Pod, your database migration task would run in a loop, continually re-populating the database after every exit.

In this chapter we will explore the the most common job patterns afforded by Kubernetes, and will leverage these patterns in real-life scenarios.

The Job Controller

The Job Controller is responsible for creating and managing pods defined in a job specification. The Job controller submits Pods to the API server where they are picked up by the Kubernetes scheduler. The Scheduler binds the Pod to a node with enough resources to run the Pod. After the Pod is scheduled the Job controller monitors the Pod status until a successful termination occurs.

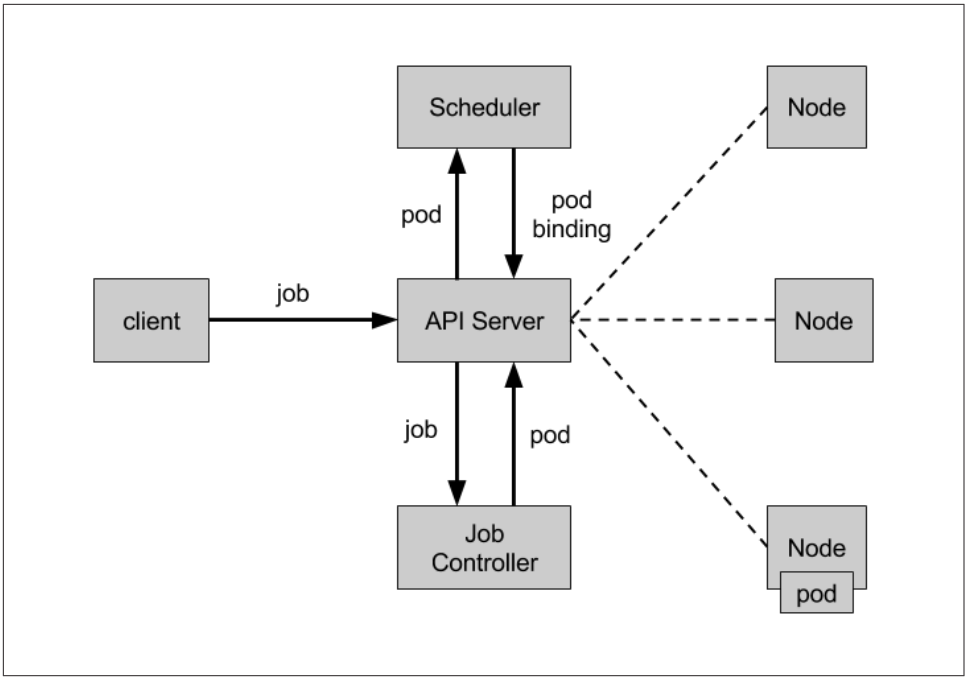


Figure 9-1. Job Controller

If the Pod was to fail before a successful termination, then the Job controller will create a new Pod based on the Pod template in the Job specification, and the cycle repeats itself. Given that Pods have to be scheduled, there is a chance that your Job will not execute if the required resources are not found by the scheduler.

Job Patterns

Jobs are designed to manage batch-like workloads where work items are processed by one or more Pods. By default each job runs a single Pod once until successful termination. This Job pattern is defined by two primary attributes of a Job, mainly the number of Job completions and the number of Pods to run in parallel. In the case of the "Run once until completion" pattern, the number of completions and parallelism is set to one.

The following chart highlights Job patterns based on the combination of completions and parallelism for a Job configuration.

Table 9-1. Job Patterns

Type	Use Case	Behavior	Completions	Parallelism
One Shot	Database migrations	A single pod running once until successful termination	1	1
Work Queue: Fixed Completions	A single pod processing a work queue	One Pod running one or more times until reaching a fixed completion count	2+	1
Work Queue: Parallel Fixed Completions	Multiple pods processing a work queue	One or more Pods running one or more times until reaching a fixed completion count	2+	2+
Work Queue: Parallel Jobs	Multiple pods processing a work queue	One or more Pods running once until successful termination	1	2+

As you can see Kubernetes supports many job patterns. This chapter will cover the two most common and robust patterns.

- One Shot
- Work Queue: Parallel Jobs

One Shot

One shot jobs provide a way to run a single Pod once until successful termination. While this may sound like an easy task, there is a some work involved in pulling this off. First a Pod must be created and submitted to the Kubernetes API. This is done using a Pod template defined in the Job configuration. Once a Job is up and running the Pod backing the job must be monitored for a successful termination. A job can fail for any number of reasons including an application error, uncaught exception during runtime, or a node failure before the job has a chance to complete. In all cases the Job controller is responsible for recreating the Pod until a successful termination occurs.

Creating a One Shot Job

There are multiple ways to create a one shot job in Kubernetes. The first is to use the kubectl command line tool.

```
$ kubectl run -i fizzbuzz \
  --image=kelseyhightower/fizzbuzz \
  --restart=OnFailure
```

Output:

```
1
2
Fizz
...
98
Fizz
Buzz
```

The other option for creating a one shot job is using a job configuration file.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: fizzbuzz
spec:
  template:
    metadata:
      name: fizzbuzz
    spec:
      containers:
      - name: fizzbuzz
        image: kelseyhightower/fizzbuzz
      restartPolicy: OnFailure
```

Submit the job using the `kubectl create` command:

```
$ kubectl create -f jobs/fizzbuzz.yaml
job "fizzbuzz" created
```

Describe the fizzbuzz job:

```
$ kubectl describe jobs fizzbuzz

Name:           fizzbuzz
Namespace:      default
Image(s):       kelseyhightower/fizzbuzz
Selector:       controller-uid=507d3402-2b5d-11e6-91b4-42010a800098
Parallelism:    1
Completions:    1
Start Time:     Sun, 05 Jun 2016 13:37:26 -0700
Labels:         controller-uid=507d3402-2b5d-11e6-91b4-42010a800098,job-name=fizzbuzz
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
No volumes.
Events:
  Message
  -------
  SuccessfulCreate Created pod: fizzbuzz-9bqgy
```

View the results of the fizzbuzz job:

```
$ kubectl logs fizzbuzz-9bqgy
```

Output:

```
1
2
Fizz
...
98
Fizz
Buzz
```

Congratulations, your Job has run successfully!

Work Queue: Parallel Jobs

A common use case for jobs is to process work from a message queue. In this scenario one job creates a number of work items and publishes them to a work queue. Another job can be run to process each work item until the work queue is empty. Each job is associated with a unique job ID, which can be used to coordinate work producers and consumers.

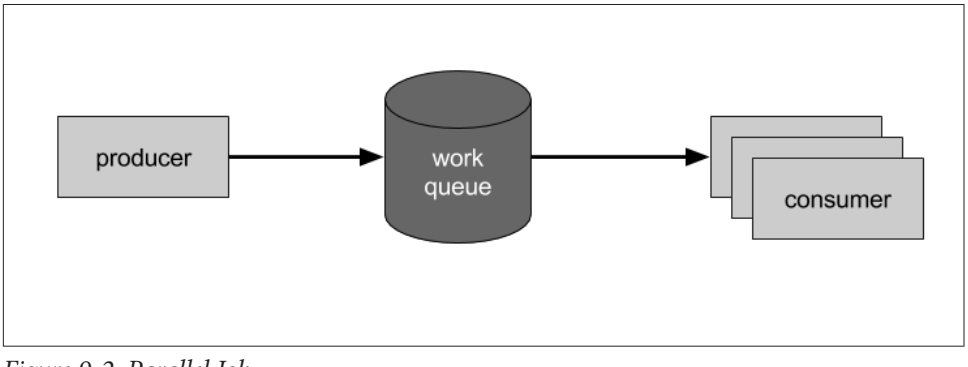


Figure 9-2. Parallel Job

By default all Jobs in Kubernetes run serially, even Jobs where the replica count is greater than one. To get a Job to run in parallel you must set the parallelism key on the Job configuration to a value greater than 1.

Start a Message Broker

In this section we are going to use an in-memory message broker called memq to host the work queues for our parallel jobs.

Save the memq pod:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
```

```
    app: memq
  name: memq
spec:
  containers:
    - name: memq
      image: "kelseyhightower/memq:1.0.0"
      args:
        - "-http=0.0.0.0:80"
```

Create the memq pod.

```
$ kubectl create -f memq.yaml
```

At this point the memq pod should be up and running. Let's use port forwarding to connect to the Pod.

```
$ kubectl port-forward memq 8000:80
Forwarding from 127.0.0.1:8000 -> 80
Forwarding from [::1]:8000 -> 80
```

We can now query the broker and verify that it is up and running. Note that there are no work queues.

```
$ curl 127.0.0.1:8000/stats
{
  "kind": "stats",
  "queues": []
}
```

With the message broker in place, we should expose it using a Service. This will make it easy for producers and consumers to locate the memq broker via DNS.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: memq
  name: memq
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: memq
```

Create the memq service with kubectl:

```
$ kubectl create -f memq-service.yaml
service "memq" created
```

Creating the Producer Job

We are now ready to create the producer job.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: "producer"
spec:
  template:
    metadata:
      name: "producer"
    spec:
      containers:
        - name: producer
          image: kelseyhightower/producer:1.0.0
          args:
            - "-memq=http://memq"
            - "-id=d674925d-2e64-702c-54d0-0164908af570"
            - "-count=100"
      restartPolicy: Never
```

```
$ kubectl create -f jobs/producer.yaml
job "producer" created
```

The job should insert 100 items into the job queue.

```
$ kubectl get jobs
NAME       DESIRED   SUCCESSFUL   AGE
producer   1         1            55s
```

```
$ curl 127.0.0.1:8000/stats
{
  "kind": "stats",
  "queues": [
    {
      "name": "d674925d-2e64-702c-54d0-0164908af570",
      "depth": 100
    }
  ]
}
```

Now we are ready to kick off a job to consume the work queue until it's empty.

Creating the Consumer Job

Create the parallel-consumers job:

```
$ kubectl create -f jobs/parallel-consumers.yaml
job "parallel-consumers" created
```

Once the job has been created you can view the pods backing the job:

```
$ kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
memq                        1/1       Running   0          12m
parallel-consumers-5hfmk    1/1       Running   0          4s
parallel-consumers-6smdg    1/1       Running   0          4s
parallel-consumers-vugv8    1/1       Running   0          4s
parallel-consumers-xsbn6    1/1       Running   0          4s
parallel-consumers-zee3f    1/1       Running   0          4s
```

Notice there are 5 pods running in parallel. These pods will continue to run until the job queue is empty.

Summary

Kubernetes can handle running both long running workloads such as web applications and short lived workloads such as batch jobs on the same cluster. The Job abstraction allows you to model batch job patterns ranging from simple one-time tasks to parallel jobs that process many items until work has been exhausted.

Jobs are a low level primitive and can be used directly for simple workloads, but for more complex jobs with dependencies then some form of orchestration will need to be built on top.

CHAPTER 10
Secrets

Applications rely on configuration data to control run-time behavior such as determining which TCP port to listen on for client requests. Most configuration data can safely be stored in configuration files or passed to applications via command line arguments without need for concern. However, sensitive configuration data such as database credentials must be handled with care and exposed only to authorized applications.

A common approach towards managing sensitive configuration data is to employ configuration management tools[1], which do an excellent job of allowing users to declare and transport configuration data across one or more machines. Did you catch that? Configuration management tools target machines, not applications, when distributing configuration data. This means some configuration settings run the risk of being exposed to every application running on a specific machine.

The Secrets API overcomes the limitations of configuration management tools and provides an application-centric mechanism for exposing sensitive configuration to applications in a way that's easy to audit and leverages native OS isolation primitives.

Secrets are used to store sensitive configuration data such as passwords, auth tokens, and TLS keys. Secrets enable container images to be created without bundling sensitive data, which means containers can remain portable across environments. Secrets are exposed to pods via explicit declaration in pod manifests and the Kubernetes API.

The remainder of this chapter will explore how to create and manage Kubernetes secrets, and also lay out best practices for exposing secrets to pods that require them.

1 Puppet, Chef and Ansible come to mind here.

Creating Secrets

Secrets are created using the Kubernetes API or the kubectl command line tool. Secrets hold one or more data elements as a collection of key/value pairs.

Naming Requirements

Secret key names follow the naming conventions defined under the DNS_SUBDOMAIN[2] rfc, and can optionally start with a leading dot. One thing to note is that instead of being case insensitive, secret key names with uppercase characters are invalid. The following table provides examples of valid and invalid secret names.

Table 10-1. A Table

Valid Key Name	Invalid Key Name
.auth_token	Token.properties
key.pem	auth file.json
id_rsa	_password.txt

When selecting a secret's key name consider that secrets can be exposed to pods via a volume mount. Storing a TLS key as "key.pem" is more clear than "tls-key" when configuring applications to access secrets.

Value Requirements

Secret data values hold arbitrary data encoded using base64. The use of base64 encoding makes it easy to store binary data.

Creating Secrets with kubectl

In this section we will create a secret to store a TLS key and certificate for the helloworld application that meets the storage requirements listed above.

The helloworld container image does not bundle a TLS certificate or key. This allows the helloworld container to remain portable across environments and distributable through public Docker repositories.

2 *http://www.ietf.org/rfc/rfc1035.txt*

The first step in creating a secret is to obtain the raw data we want to store. The TLS key and certificate for the helloworld application can be downloaded by running the following commands:

```
$ curl -O https://storage.googleapis.com/kuar/cert.pem
$ curl -O https://storage.googleapis.com/kuar/key.pem
```

With the cert.pem and key.pem files stored locally we are ready to create a secret. Create a secret named helloworld-tls using the `create secret` command:

```
$ kubectl create secret generic helloworld-tls \
  --from-file=key.pem \
  --from-file=cert.pem
```

The helloworld-tls secret has been created with two data elements:

```
$ kubectl describe secrets helloworld-tls

Name:        helloworld-tls
Namespace:   default
Labels:      <none>
Annotations: <none>

Type:        Opaque

Data
====
cert.pem:   1472 bytes
key.pem:    1679 bytes
```

With the helloworld-tls secret in place we can consume it from a pod by using a secrets volume.

Consuming Secrets

Secrets can be consumed using the Kubernetes REST API, but it requires that applications be modified to communicate with the Kubernetes API server. Our goal is to keep applications portable. Not only should they run well in Kubernetes, but should run, unmodified, on other platforms.

Instead of accessing secrets through the API server we can use a secrets volume.

Secrets Volumes

Secret data can be exposed to pods using the secret volume type. Secret volumes are managed by the kubelet and are created at pod creation time. Secrets are stored on tmpfs volumes and not written to disks on nodes.

Each data element of a secret is stored in a separate file under the target mount point specified in the volume mount. The helloworld-tls secret contains two data elements: cert.pem and key.pem. Mounting the helloworld-tls secret volume to /etc/tls will request in the following files:

```
/etc/tls/cert.pem
/etc/tls/key.pem
```

The following pod manifest demonstrates how to declare a secret volume which exposes the helloworld-tls secret to the helloworld container under /etc/tls.

```
apiVersion: v1
kind: Pod
metadata:
  name: helloworld-tls
spec:
  containers:
    - name: helloworld-tls
      image: kelseyhightower/helloworld-tls:v1
      volumeMounts:
        name: helloworld-tls-certs
        mountPath: "/etc/tls"
        readOnly: true
  volumes:
    - name: helloworld-tls-certs
      secret:
        secretName: helloworld-tls-certs
```

Create the helloworld-tls pod using kubectl and observe the log output from the running pod:

```
$ kubectl create -f helloworld-tls-pod.yaml

$ kubectl logs helloworld-tls
```

Managing Secrets

Secrets are managed through the Kubernetes API.

Listing secrets

Use the `get secrets` command to list all secrets in the current namespace. command:

```
$ kubectl get secrets

NAME                  TYPE                                  DATA      AGE
default-token-cxf05   kubernetes.io/service-account-token   2         1h
helloworld-tls        Opaque                                2         5s
```

To get more details for a specific secret using the `describe secrets` command:

```
$ kubectl describe secrets helloworld-tls

Name:        helloworld-tls
Namespace:   default
Labels:      <none>
Annotations: <none>

Type:        Opaque

Data
====
cert.pem:   1472 bytes
key.pem:    1679 bytes
```

Private Docker Registries

A common use case for secrets is to store access credentials for private Docker registries. Kubernetes supports using images stored on private registries, but access to those images require credentials. Private images can be stored across one or more private Registries which presents the challenge of managing credentials for each private repository on every possible node in the cluster.

Image Pull Secrets leverage the secrets API to automate the distribution of private registry credentials. Image Pull Secrets are stored just like normal secrets but are consumed through the `spec.imagePullSecrets` Pod specification field.

Run the `docker login` command to generate a dockercfg file containing the docker credentials required to login to the private repository.

```
$ docker login
Username: janedoe
Password: ●●●●●●●●●●
Email: jdoe@example.com
WARNING: login credentials saved in /Users/jdoe/.dockercfg.
Login Succeeded
```

Use the `create secret docker-registry` command to convert the .dockercfg file to a Kubernetes secret:

```
$ kubectl create secret docker-registry registrykey \
  --from-file="~/.dockercfg"
```

Enable access to the private repository by referencing the registrykey pull secret in the pod manifest file:

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: helloworld-tls
spec:
  containers:
    - name: helloworld-tls
      image: kelseyhightower/helloworld-tls:v1
      volumeMounts:
        name: helloworld-tls-certs
        mountPath: "/etc/tls"
        readOnly: true
      imagePullSecrets:
        - name: registrykey
  volumes:
    - name: helloworld-tls-certs
      secret:
        secretName: helloworld-tls-certs
```

```
$kubectl create -f helloworld-tls-pod.yaml
```

Summary

Secrets provide secure storage for sensitive data. The secrets API and volume type provide a secure transportation mechanism for sensitive data and automates the process of exposing that data to the pods that need it.