O'REILLY®

Compliments of NGINX

Kubernetes for Java Developers

Orchestrate Multi-Container Applications with Ease

Arun Gupta

Kubernetes for Java Developers

Orchestrate Multicontainer Applications with Ease

Arun Gupta

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Kubernetes for Java Developers
by Arun Gupta

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://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Nan Barber and Brian Foster
Production Editor: Melanie Yarbrough
Copyeditor: Rachel Monaghan
Proofreader: Amanda Kersey

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2017: First Edition

Revision History for the First Edition
2017-06-08: First Release

978-1-491-97326-4

[LSI]

Table of Contents

Foreword

I had the pleasure of meeting Arun at a meetup last year. He is a truly unique individual with a talent for explaining complex technical concepts so a wide range of people can understand them. The topic he was discussing that evening was Docker Swarm, and I left feeling like I could create my own Swarm app.

Arun was an early member of the Java software development team at Sun Microsystems and is now a prolific speaker and writer on Java-related topics. You can download his previous ebook, *Docker for Java Developers*, and the accompanying webinar, "Run Java in Docker Containers with NGINX", from the NGINX website.

Now, in this ebook, Arun introduces key Kubernetes concepts such as pods—collections of containers—and shows you how these entities interact with your Java application. The eruption of containers into the development landscape, led by Docker, highlights the need for ways to manage and orchestrate them. Kubernetes is the leading container orchestration tool.

NGINX is widely used with both technologies. NGINX works well with containers; it is one of the most frequently downloaded applications on Docker Hub, with over 10 million pulls to date. And NGINX is seeing increasing use with Kubernetes, including as an ingress controller. Arun's goal is to get you, as a Java developer, up to speed with Kubernetes in record time. This book in an important tool to help you carry out this mission. We hope you enjoy it.

— Faisal Memon, Product Marketer, NGINX, Inc.

Preface

A Java application typically consists of multiple components, such as an application server, a database, and a web server. Multiple instances of each component are started to ensure high availability. The components usually dynamically scale up and down to meet the always-changing demands of the application. These multiple components are run on a cluster of hosts or virtual machines to avoid a single point of failure.

In a containerized solution, each replica of each component is a container. So a typical application is a multicontainer, multihost application. A container orchestration system is required that can manage the cluster; schedule containers efficiently on different hosts; provide primitives, such as service discovery, that allow different containers to talk to each other; and supply network storage that can store database-persistent data. This system allows developers to focus on their core competency of writing the business logic as opposed to building the plumbing around it.

Kubernetes is an open source container orchestration framework that allows simplified deployment, scaling, and management of containerized applications. Originally created by Google, it was built upon the company's years of experience running production workloads in containers. It was donated to Cloud Native Computing Foundation (CNCF) in March 2016 to let it grow as a truly vendor-independent project. Different Kubernetes projects can be found on GitHub.

This book is targeted toward Java developers who are interested in learning the basic concepts of Kubernetes. The technology and the terminology are rapidly evolving, but the basics still remain relevant.

Chapter 1 explains them from the developer and operations perspective. Chapter 2 explains how to create a single-node local development cluster and how to get started with a multinode cluster. It also covers a simple Java application communicating with a database running on Kubernetes. Chapter 3 gets into more advanced concepts like stateful containers, scaling, performing health checks and rolling updates of an application, and sharing resources across the cluster. Chapter 4 details administrative aspects of Kubernetes. The examples in this book use the Java programming language, but the concepts are applicable for anybody interested in getting started with Kubernetes.

Acknowledgments

I would like to express gratitude to the people who made writing this book a fun experience. First and foremost, many thanks to O'Reilly for providing an opportunity to write it. The team provided excellent support throughout the editing, reviewing, proofreading, and publishing processes. At O'Reilly, Brian Foster believed in the idea and helped launch the project. Nan Barber was thorough and timely with her editing, which made the book fluent and consistent. Thanks also to the rest of the O'Reilly team, some of whom we may not have interacted with directly, but who helped in many other ways. Paul Bakker (@pbakker) and Roland Huss (@ro14nd) did an excellent technical review of the book, which ensured that the book stayed true to its purpose and explained the concepts in the simplest possible ways. A vast amount of information in this book is the result of delivering the "Kubernetes for Java Developers" presentation all around the world. A huge thanks goes to all the workshop attendees whose questions helped clarify my thoughts. Last but not least, I seek forgiveness from all those who have helped us over the past few months and whose names I have failed to mention.

CHAPTER 1

Kubernetes Concepts

Kubernetes is an open source orchestration system for managing containerized applications across multiple hosts, providing basic mechanisms for the deployment, maintenance, and scaling of applications.

Kubernetes, or "k8s" or "kube" for short, allows the user to declaratively specify the desired state of a cluster using high-level primitives. For example, the user may specify that she wants three instances of the WildFly server container running. Kubernetes' self-healing mechanisms, such as autorestarting, rescheduling, and replicating containers, then converge the actual state toward the desired state.

Kubernetes supports Docker and Rocket containers. An abstraction around the containerization layer allows for other container image formats and runtimes to be supported in the future. But while multiple container formats are supported, Docker is by far the most prevalent format used with Kubernetes.

All resource files in this chapter are available on GitHub.

Pods

A *pod* is the smallest deployable unit that can be created, scheduled, and managed. It's a logical collection of containers that belong to an application. Pods are created in a namespace. All containers in a pod share the namespace, volumes, and networking stack. This allows

containers in the pod to "find" each other and communicate using `localhost`.

Each resource in Kubernetes can be defined using a configuration file. For example, a WildFly pod can be defined with the configuration file shown in Example 1-1.

Example 1-1. Pod configuration

```
apiVersion: v1 ❶
kind: Pod ❷
metadata: ❸
  name: wildfly-pod ❹
  labels: ❺
    name: wildfly-pod
spec: ❻
  containers: ❼
  - name: wildfly ❽
    image: jboss/wildfly:10.1.0.Final ❽
    ports:
    - containerPort: 8080 ❾
```

This configuration file uses the following properties:

❶ `apiVersion` defines the version of the Kubernetes API. This is now fixed at `v1` and allows for the API to evolve in the future.

❷ `kind` defines the type of this resource—in this example, that value is `Pod`.

❸ `metadata` allows you to attach information about the resource.

❹ Each resource must have a `name` attribute. If this attribute is not set, then you must specify the `generateName` attribute, which is then used as a prefix to generate a unique name. Optionally, you can use a `namespace` property to specify a namespace for the pod. Namespaces provide a scope for names and are explained further in "Namespaces" on page 54.

In addition to these properties, there are two types of metadata: `metadata.labels` and `metadata.annotations`. They both are defined as key/value pairs.

❺ Labels are designed to specify identifying attributes of the object that are meaningful and relevant to the users, but which do not

directly imply semantics to the core system. Multiple labels can be attached to a resource. For example, `name: wildfly-pod` is a label assigned to this pod. Labels can be used to organize and to select subsets of objects.

Annotations are defined using `metadata.annotations[]`. They are designed to be nonidentifying arbitrary information attached to the object. Some information that can be recorded here is build/release information, such as release IDs, Git branch, and PR numbers.

❻ `spec` defines the specification of the resource, pod in our case.

❼ `containers` defines all the containers within the pod.

❽ Each container must have a uniquely identified `name` and `image` property. `name` defines the name of the container, and `image` defines the Docker image used for that container. Some other commonly used properties in this section are:

`args`
: A command array containing arguments to the entry point

`env`
: A list of environment variables in `key:value` format to set in the container

❾ `ports` define the list of ports to expose from the container. WildFly runs on port 8080, and thus that port is listed here. This allows other resources in Kubernetes to access this container on this port.

In addition, `restartPolicy` can be used to define the restart policy of all containers within the pod. `volumes[]` can be used to list volumes that can be mounted by containers belonging to the pod.

Pods are generally not created directly, as they do not survive node or scheduling failures. They are mostly created using a replication controller or deployment.

More details about the pod configuration file are available at the Kubernetes website.

Replication Controllers

A *replication controller* (RC) ensures that a specified number of pod "replicas" are running at any one time. Unlike manually created pods, the pods maintained by a replication controller are automatically replaced if they fail, get deleted, or are terminated. A replication controller ensures the recreation of a pod when the worker node fails or reboots. It also allows for both upscaling and downscaling the number of replicas.

A replication controller creating two instances of a WildFly pod can be defined as shown in Example 1-2.

Example 1-2. Replication controller configuration

```
apiVersion: v1
kind: ReplicationController ❶
metadata:
  name: wildfly-rc
spec:
  replicas: 2 ❷
  selector: ❸
    app: wildfly-rc-pod
  template: ❹
    metadata:
      labels:
        app: wildfly-rc-pod
    spec:
      containers:
      - name: wildfly
        image: jboss/wildfly:10.1.0.Final
        ports:
        - containerPort: 8080
```

The `apiVersion`, `kind`, `metadata`, and `spec` properties serve the same purpose in all configuration files.

This configuration file has the following additional properties:

❶ The value of `kind` is `ReplicationController`, which indicates that this resource is a replication controller.

❷ `replicas` defines the number of replicas of the pod that should concurrently run. By default, only one replica is created.

❸ `selector` is an optional property. The replication controller manages the pods that contain the labels defined by the `spec.selector` property. If specified, this value must match `spec.template.metadata.labels`.

All labels specified in the `selector` must match the labels on the selected pod.

❹ `template` is the only required field of `spec` in this case. The value of this field is exactly the same as a pod, except it is nested and does not have an `apiVersion` or `kind`. Note that `spec.template.metadata.labels` matches the value specified in `spec.selector`. This ensures that all pods started by this replication controller have the required metadata in order to be selected.

Each pod started by this replication controller has a name in the format *`<name-of-the-RC>-<hash-value-of-pod-template>`*. In our case, all names will be *`wildfly-rc-xxxxx`*, where *`xxxxx`* is the hash value of the pod template.

More details about replication controllers are available at the Kubernetes website.

Replica Sets

Replica sets are the next-generation replication controllers. Just like a replication controller, a replica set ensures that a specified number of pod replicas are running at any one time. The only difference between a replication controller and a replica set is the selector support.

For replication controllers, matching pods must satisfy all of the specified label constraints. The supported operators are `=`, `==`, and `!=`. The first two operators are synonyms and represent equality. The last operator represents inequality.

For replica sets, filtering is done according to a set of values. The supported operators are `in`, `notin`, and `exists` (only for the key). For example, a replication controller can select pods such as `environment = dev`. A replica set can select pods such as `environment in ["dev", "test"]`.

A replica set creating two instances of a WildFly pod can be defined as shown in Example 1-3.

Example 1-3. Replica set configuration

```
apiVersion: extensions/v1beta1 ❶
kind: ReplicaSet ❷
metadata:
  name: wildfly-rs
spec:
  replicas: 2
  selector:
    matchLabels: ❸
      app: wildfly-rs-pod ❹
    matchExpressions: ❺
      - {key: tier, operator: In, values: ["backend"]} ❻
      - {key: environment, operator: NotIn, values: ["prod"]} ❻
  template:
    metadata:
      labels:
        app: wildfly-rs-pod
        tier: backend
        environment: dev
    spec:
      containers:
      - name: wildfly
        image: jboss/wildfly:10.1.0.Final
        ports:
        - containerPort: 8080
```

The key differences between Examples 1-2 and 1-3 are as follows:

❶ The `apiVersion` property value is `extensions/v1beta1`. This means that this object is not part of the "core" API at this time, but is only a part of the `extensions` group. Read more about API versioning at the Kubernetes GitHub page.

❷ The value of `kind` is `Replicaset` and indicates the type of this resource.

❸ `matchLabels` defines the list of labels that must be on the selected pod. Each label is a key/value pair.

❹ `wildfly-rs-pod` is the exact label that must be on the selected pod.

❺ `matchExpressions` defines the list of pod selector requirements.

❻ Each expression can be defined as a combination of three key/value pairs. The keys are `key`, `operator`, and `values`. The values are one of the keys from the labels; one of the operators `In`, `NotIn`, `Exist`, or `DoesNotExist`; and a nonempty set of values, respectively.

   All the requirements, from both `matchLabels` and `matchExpressions`, must match for the pod to be selected.

Replica sets are generally never created on their own. Deployments own and manage replica sets to orchestrate pod creation, deletion, and updates. See the following section for more details about deployments.

More details about replica sets are available at the Kubernetes website.

Deployments

Deployments provide declarative updates for pods and replica sets. You can easily achieve the following functionality using deployment:

- Start a replication controller or replica set.
- Check the status of deployment.
- Update deployment to use a new image, without any outages.
- Roll back deployment to an earlier revision.

A WildFly replica set with three replicas can be defined using the configuration file shown in Example 1-4.

Example 1-4. Deployment configuration

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: wildfly-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: wildfly
```

```
spec:
  containers:
  - name: wildfly
    image: jboss/wildfly:10.1.0.Final
    ports:
    - containerPort: 8080
```

Two main differences from Example 1-2 are:

- The `apiVersion` property value is `extensions/v1beta1`. This means that this object is not part of the "core" API at this time and is only a part of the `extensions` group. Read more about API versioning at the Kubernetes GitHub page.
- The value of the `kind` property is `Deployment`, which indicates the type of resource.

More details about deployment are available in the Kubernetes user guide.

Services

A pod is ephemeral. Each pod is assigned a unique IP address. If a pod that belongs to a replication controller dies, then it is recreated and may be given a different IP address. Further, additional pods may be created using replication controllers. This makes it difficult for an application server such as WildFly to access a database such as Couchbase using its IP address.

A service is an abstraction that defines a logical set of pods and a policy by which to access them. The IP address assigned to a service does not change over time, and thus can be relied upon by other pods. Typically, the pods belonging to a service are defined by a label selector. This is similar to how pods belong to a replication controller.

This abstraction of selecting pods using labels enables a loose coupling. The number of pods in the replication controller may scale up or down, but the application server can continue to access the database using the service.

Multiple resources, such as a service and a replication controller, may be defined in the same configuration file. In this case, each resource definition in the configuration file needs to be separated by `---`.

For example, a WildFly service and a replication controller that creates matching pods can be defined as shown in Example 1-5.

Example 1-5. Service configuration

```
apiVersion: v1
kind: Service ❶
metadata:
  name: wildfly-service
spec:
  selector:
    app: wildfly-rc-pod ❷
  ports:
    - name: web ❸
      port: 8080
--- ❹
apiVersion: v1
kind: ReplicationController ❶
metadata:
  name: wildfly-rc
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: wildfly-rc-pod ❷
    spec:
      containers:
      - name: wildfly
        image: jboss/wildfly:10.1.0.Final
        ports:
        - containerPort: 8080
```

Multiple resources are created in the order they are specified in the file.

In this configuration file:

❶ There are two resources: a service and a replication controller.

❷ The service selects any pods that contain the label `app: wildfly-rc-pod`. The replication controller attaches those labels to the pod.

❸ `port` defines the port on which the service is accessible. A service can map an incoming port to any target port in the con-

tainer using `targetPort`. By default, `targetPort` is the same as `port`.

A service may expose multiple ports. In this case, each port must be given a unique name:

```
ports:
  - name: web
    port: 8080
```

❹ `---` is the separator between multiple resources.

By default, a service is available only inside the cluster. It can be exposed outside the cluster, as covered in "Exposing a Service" on page 58.

More details about services are available at the Kubernetes website.

Jobs

A *job* creates one or more pods and ensures that a specified number of them successfully complete. When the specified number of pods has successfully completed, the job itself is complete. The job will start a new pod if the pod fails or is deleted due to hardware failure.

This is different from a replication controller or a deployment, which ensure that a certain number of pods are always running. If a pod in a replication controller or deployment terminates, it is restarted. This makes replication controllers and deployments both long-running processes, which is well suited for an application server such as WildFly. But a job is completed only when the specified number of pods successfully completes, which is well suited for tasks that need to run only once. For example, a job may convert one image format to another. Restarting this pod in a replication controller would not only cause redundant work but may even be harmful in certain cases.

There are two main types of jobs:

Nonparallel jobs
: Job specification consists of a single pod. The job completes when the pod successfully terminates.

Parallel jobs
: A predefined number of pods successfully completes. Alternatively, a *work queue* pattern can be implemented where pods can

coordinate among themselves or with an external service to determine what each should work on.

A nonparallel job can be defined using the configuration file shown in Example 1-6.

Example 1-6. Job configuration

```
apiVersion: batch/v1 ❶
kind: Job ❷
metadata:
  name: wait
spec:
  template:
    metadata:
      name: wait
    spec: ❸
      containers:
      - name: wait
        image: ubuntu ❹
        command: ["sleep",  "20"] ❺
      restartPolicy: Never ❻
```

In this configuration file:

❶ Jobs are defined in their own API group using the path *batch/v1*.

❷ The `Job` value defines this resource to be of the type job.

❸ `spec` specifies the job resource as a pod template. This is similar to a replication controller.

❹ This job uses the base image of `ubuntu`. Usually, this will be a custom image that will perform the run-once task.

❺ By default, running the `ubuntu` image starts the shell. In this case, `command` overrides the default command and waits for 20 seconds. Note, this is only an example usage. The actual task would typically be done in the image itself.

❻ Each pod template must explicitly specify the `restartPolicy` equal to `Never` or `OnFailure`. A value of `Never` means that the pod is marked `Succeeded` or `Failed` depending upon the number of containers running and how they exited. A value of

`OnFailure` means the pod is restarted if the container in the pod exits with a failure. More details about these policies are available at the Kubernetes website.

Kubernetes 1.4 introduced a new alpha resource called `ScheduledJob`. This resource was renamed to `CronJob` starting in version 1.5.

`CronJob` allows you to manage time-based jobs. There are two primary use cases:

- Run jobs once at a specified point in time.
- Run jobs repeatedly at a specified point in time.

Note, this is an alpha resource, so it needs to be explicitly enabled.

Volumes

Pods are ephemeral and work well for a stateless container. They are restarted automatically when they die, but any data stored in their filesystem is lost with them. Stateful containers, such as Couchbase, require data to be persisted outside the lifetime of a container running inside a pod. This is where volumes help.

A *volume* is a directory that is accessible to the containers in a pod. The directory, the medium that backs it, and the contents within it are determined by the particular volume type used. A volume outlives any containers that run within the pod, and the data is preserved across container restarts.

Multiple types of volumes are supported. Some of the commonly used volume types are shown in Table 1-1.

Table 1-1. Common volume types in Kubernetes

Volume type	Mounts into your pod
`hostPath`	A file or directory from the host node's filesystem
`nfs`	Existing Network File System share
`awsElasticBlockStore`	An Amazon Web Service EBS volume
`gcePersistentDisk`	A Google Compute Engine persistent disk

Two properties need to be defined for a volume to be used inside a pod: `spec.volumes` to define the volume type, and `spec.containers.volumeMounts` to specify where to mount the volume. Multiple

volumes in a pod and multiple mount points in a container can be easily defined. A process in a container sees a filesystem view composed of the Docker image and volumes in the pod.

A volume defined in the pod configuration file is shown in Example 1-7.

Example 1-7. Volume configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: couchbase-pod
  labels:
    name: couchbase-pod
spec:
  containers:
  - name: couchbase
    image: arungupta/couchbase-oreilly:k8s ❶
    ports:
    - containerPort: 8091
    volumeMounts: ❷
    - mountPath: /var/couchbase/lib ❸
      name: couchbase-data ❹
  volumes: ❺
  - name: couchbase-data ❹
    hostPath: ❻
      path: /opt/data ❼
```

In this configuration file:

❶ The pod in the replication controller uses the image at *arungupta/oreilly-couchbase:k8s*. This image is created using Couchbase. It uses the Couchbase REST API to configure the Couchbase server and create a sample bucket in it.

❷ The `volumeMounts` property defines where the volume is mounted in the container.

❸ `mountPath` defines the path where the volume is mounted in the container.

❹ `name` refers to a named volume defined using `volumes`. This value must match the value of the `name` property of one of the volumes defined in `volumes`.

❺ `volumes` defines the volumes accessible to the pod.

❻ `hostPath` defines the type of volume mounted. This volume type is mounting a directory from host node's filesystem. A differe volume type may be specified here.

❼ */opt/data* is the path in the host node filesystem.

You can create an Amazon Elastic Block Storage (EBS) volume using the `aws ec2 create-volume` command. Alternatively, you can create a Google Cloud persistent disk using the `gcloud compute disks create` command. You can mount these volumes in the container using the `awsElasticBlockStore` and `gcePersistentDisk` volume types, respectively.

More details about volumes, including different types of volumes and how to configure them, are available at the Kubernetes website.

Architecture

The key components of the Kubernetes architecture are shown in Figure 1-1.

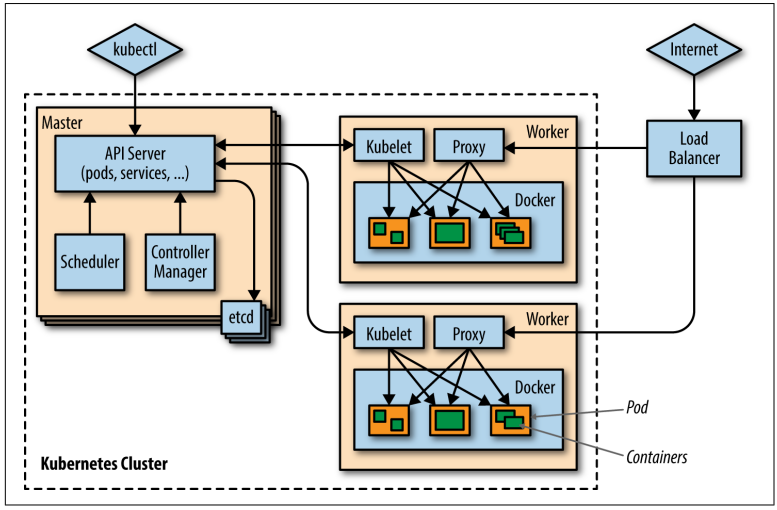

Figure 1-1. Kubernetes architecture

A Kubernetes cluster is a set of physical or virtual machines and other infrastructure resources that are used to run your applications.

Each machine is called a *node*. The machines that manage the cluster are called *master* nodes, and the machines that run the containers are called *worker* nodes. Each node handles the necessary services to run application containers.

The two typical interaction points with Kubernetes are `kubectl` and the client application running in the internet.

Master nodes

A master node is a central control plane that provides a unified view of the cluster. You can easily create a Kubernetes cluster with a single master node for development. Alternatively, you could create a Kubernetes cluster with high availability with multiple master nodes. Let's look at the key components in the master node:

`kubectl`
: This is a command-line tool that send commands to the master node to create, read, update, and delete resources. For example, it can request to create a pod by passing the pod configuration file, or it can query more details about the replicas running for a replica set. It reads container manifests as YAML or JSON files that describe each resource. A typical way to provide this manifest is using the configuration file as shown in the previous sections. This process is explained more in "Running Your First Java Application" on page 25.

API server
: Each command from `kubectl` is translated into a REST API and issued to the API server running inside the master node. The API server processes REST operations, validates them, and persists the state in a distributed watchable storage. This is implemented using `etcd` for Kubernetes.

Scheduler
: The scheduler works with the API server to schedule pods to the nodes. The scheduler has information about resources available on the worker nodes, as well as the ones requested by the pods. It uses this information to decide which node will be selected to deploy a specific pod.

Controller manager
: The controller manager is a daemon that watches the state of the cluster using the API server for different controllers and

reconciles the actual state with the desired one (e.g., the number of pods to run for a replica set). Some other controllers that come with Kubernetes are the namespace controller and the horizontal pod autoscaler.

`etcd`
: This is a simple, distributed, watchable, and consistent key/value store. It stores the persistent state of all REST API objects—for example, how many pods are deployed on each worker node, labels assigned to each pod (which can then be used to include the pods in a service), and namespaces for different resources. For reliability, `etcd` is typically run in a cluster.

Worker nodes

A worker node runs tasks as delegated by the master. Each worker node can run multiple pods:

Kubelet
: This is a service running on each node that manages containers and is managed by the master. It receives REST API calls from the master and manages the resources on that node. Kubelet ensures that the containers defined in the API call are created and started.

  Kubelet is a Kubernetes-internal concept and generally does not require direct manipulation.

Proxy
: This runs on each node, acting as a network proxy and load balancer for a service on a worker node. Client requests coming through an external load balancer will be redirected to the containers running in a pod through this proxy.

Docker
: Docker Engine is the container runtime running on each node. It understands the Docker image format and knows how to run Docker containers. Alternatively, Kubernetes may be configuired to use `rkt` as the container runtime. More details about that are available in the guide to running Kubernetes with `rkt`.

CHAPTER 2

Deploying a Java Application to Kubernetes

This chapter will explain how to start a Kubernetes cluster. It will also demonstrate CRUD (create, read, update, and delete) operations for different resources on this cluster. Service discovery and scaling the number of containers in a service will be discussed as well. Finally, you'll learn how to deploy a simple Java application to Kubernetes using Maven.

All resource files in this chapter are available in the GitHub repo for this book.

Managing Kubernetes Cluster

You can create a Kubernetes cluster on your laptop, virtual machine (VM), cloud provider, or a rack of bare-metal servers. For development purposes, it's easiest to start with a one-node cluster using Minikube. For production purposes, hosted solutions or cloud-based solutions provide the scalability and higher availability you'll need. A complete list of solutions to create Kubernetes clusters is available at the Kubernetes website.

This section will explain how to create and shut down a development Kubernetes cluster using Minikube. The deployment of such an application typically happens on a cloud platform such as Amazon Web Services (AWS). A common way to start a single-master

cluster and a highly available cluster using Kops on AWS is also explained.

Development Cluster Using Minikube

Minikube runs a single-node Kubernetes cluster inside a VM on your laptop. This allows you to try out Kubernetes on your local machine easily. Minikube packages and configures a Linux VM, the container runtime, and all Kubernetes components, optimized for local development.

As of release 0.18.0, Minikube is distributed in a binary form for macOS, Windows, and Linux. Complete instructions, including the latest release, are available on GitHub.

This book will use the 0.18.0 release on macOS for the Kubernetes cluster, unless specified otherwise.

Start cluster

Download the Minikube binary as follows:

```
curl -Lo minikube \
https://storage.googleapis.com/minikube/releases/v0.18.0/
    minikube-darwin-amd64 \
&& chmod +x minikube
```

The `kubectl` binary, which manages the cluster, will need to be downloaded separately. Complete download instructions are available on the Kubernetes website. The latest release of `kubectl` can be downloaded as follows:

```
curl -LO https://storage.googleapis.com/kubernetes-release/
release/$(curl -s https://storage.googleapis.com/kubernetes-
release/release/stable.txt)/bin/darwin/amd64/kubectl &&
chmod +x kubectl
```

Note that both the binaries are downloaded in the same directory. `kubectl` should in included in the `PATH` of the terminal where your Kubernetes cluster will be started.

Start the cluster as shown in Example 2-1.

Example 2-1. Start the Kubernetes cluster using Minikube

```
minikube start
Starting local Kubernetes cluster...
Starting VM...
```

```
Downloading Minikube ISO
 89.51 MB / 89.51 MB [============================================]
 100.00% 0s
SSH-ing files into VM...
Setting up certs...
Starting cluster components...
Connecting to cluster...
Setting up kubeconfig...
Kubectl is now configured to use the cluster.
```

The example starts by downloading the ISO base image and creating the VirtualBox VM. Then it generates the SSL certificates that allow a secure connection to the VM. Next, all the needed components—such as the API server, Docker engine, and `etcd`—are started. All the configuration information is stored in the *kubeconfig* file. The default location of this file is *~/.kube/config*. Finally, the `kubectl` CLI is configured to connect to this cluster using the configuration information.

As of this writing, Minikube starts a Kubernetes 1.6.0 cluster. For a complete list of Kubernetes versions, use the command `minikube get-k8s-versions`. You'll see the following output:

```
The following Kubernetes versions are available:
    - v1.6.0
    - v1.6.0-rc.1
    - v1.6.0-beta.4
    - v1.6.0-beta.3
    - v1.6.0-beta.2
    - v1.6.0-alpha.1
    - v1.6.0-alpha.0
    - v1.5.3
    - v1.5.2
    - v1.5.1
    - v1.4.5
    - v1.4.3
    - v1.4.2
    - v1.4.1
    - v1.4.0
    - v1.3.7
    - v1.3.6
    - v1.3.5
    - v1.3.4
    - v1.3.3
    - v1.3.0
```

You can start a different version of Kubernetes cluster as follows:

```
minikube start --kubernetes-version=<version>
```

The value of `<version>` needs to be one of the versions listed in the output of the previous command.

For more details about client and server version, use the command `kubectl version`. The output looks like the following:

```
Client Version: version.Info{Major:"1", Minor:"6", GitVersion:
    "v1.6.2", GitCommit:"477efc3cbe6a7effca06bd1452fa356e2201e
    1ee", GitTreeState:"clean", BuildDate:"2017-04-19T20:33:
    11Z", GoVersion:"go1.7.5", Compiler:"gc", Platform:"darwin
    ----/amd64"}
Server Version: version.Info{Major:"1", Minor:"6", GitVersion:
"v1.6.0", GitCommit:"fff5156092b56e6bd60fff75aad4dc9de6b6ef37",
GitTreeState:"dirty", BuildDate:"2017-04-07T20:46:46Z",
GoVersion:"go1.7.3", Compiler:"gc", Platform:"linux/amd64"}
```

For more details about the cluster, use the command `kubectl cluster-info`, which gives the following output:

```
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443/api/v1/proxy/
    namespaces/kube-system/services/kube-dns
kubernetes-dashboard is running at https://192.168.99.100:8443/
    api/v1/proxy/namespaces/kube-system/services/kubernetes-
    dashboard

To further debug and diagnose cluster problems, use 'kubectl
    cluster-info dump'.
```

The IP address may be different in your case. You can find exact value of the IP address using the `minikube ip` command. Other commonly used commands are shown in Table 2-1.

Table 2-1. Minikube common commands

Command	Purpose
`stop`	Stops a running local Kubernetes cluster
`delete`	Deletes a local Kubernetes cluster
`delete`	Deletes a local Kubernetes cluster
`get-k8s-versions`	Gets the list of available Kubernetes versions available for Minikube

To view the complete list of commands for Minikube, use the command `minikube --help`.

Kubernetes dashboard

The Kubernetes dashboard is a general-purpose, web-based UI for Kubernetes clusters. It provides an overview of applications running

on the cluster, as well as allowing you to create or modify individual Kubernetes resources and workloads, such as replica sets, jobs, services, and pods. The dashboard can be used to manage the cluster as well.

You can view this dashboard using the command `minikube dashboard`. A default view of the dashboard is shown in Figure 2-1.

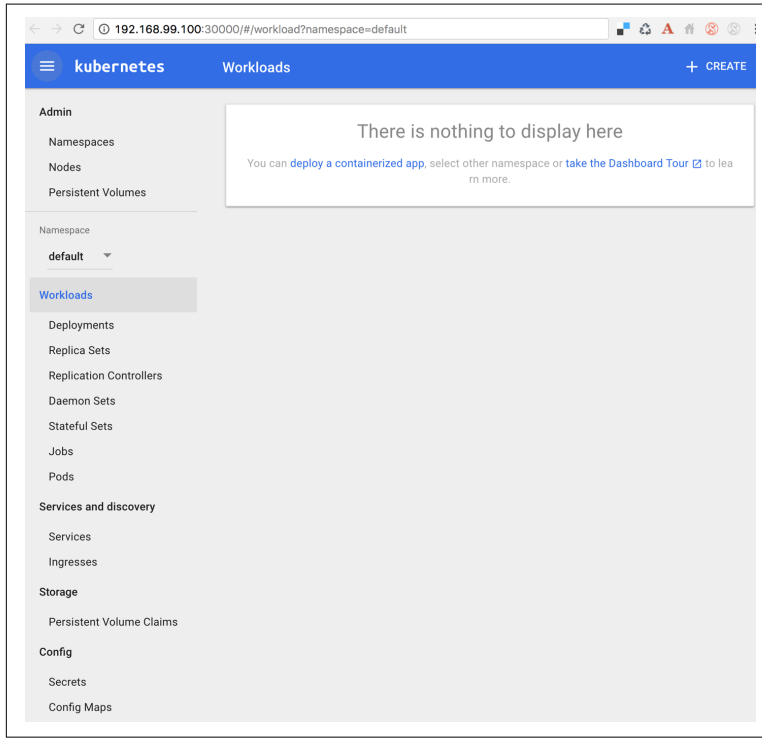

Figure 2-1. Kubernetes dashboard

Shut down the cluster

You can stop the cluster using `minikube stop`.

Multimaster Production Cluster on AWS

A complete list of options for running a Kubernetes cluster is explained in the Kubernetes documentation. The cluster can be started as a hosted solution created and maintained by the provider. Alternatively, you can start it on a wide variety of cloud providers by using a few commands. You can also start it using on-premise VMs

or on bare metal. These approaches are typically more involved, though.

This section will explain how to set up a Kubernetes cluster on AWS using *Kops*. Kops, short for *Kubernetes operations*, is one of the ways to create a highly available Kubernetes cluster on cloud platforms such as AWS. The `kubectl` script is the CLI for running commands against clusters. Kops provide the same functionality for clusters as `kubectl` does for running commands against clusters.

Running a Kubernetes cluster using Kops on AWS has some requirements:

- An AWS account with full access, which can be created at the AWS website
- The AWS Command Line Interface
- The latest Kops release
- The `kubectl` CLI

A complete list of options to start a highly available cluster is provided at the Kubernetes website.

Start the cluster

The "Getting Started" guide on GitHub provides complete details about starting a cluster using Kops on AWS.

You'll need to meet the following requirements before you can start a cluster:

1. Create a top-level or subdomain where the Kubernetes cluster will be hosted.
2. Create a Amazon Route 53 hosted zone mapping to this domain.
3. Create an S3 bucket to store your cluster configuration.
4. Download the `kubectl` CLI.

Amazon EC2 is hosted in multiple locations worldwide. These locations are composed of *regions* and *availability zones* (AZ). Each region is a separate geographic area, and has multiple, isolated availability zones. A highly available Kubernetes cluster can be created across multiple AZs of a region, but not across regions.

You can start a multimaster Kubernetes cluster on AWS as shown in Example 2-2.

Example 2-2. Start multimaster Kubernetes cluster on AWS

```
kops-darwin-amd64 \
    create cluster \ ❶
    --name=kubernetes.arungupta.me \ ❷
    --cloud=aws \ ❸
    --zones=us-west-2a,us-west-2b,us-west-2c \ ❹
    --master-zones=us-west-2a,us-west-2b,us-west-2c \ ❺
    --master-size=m4.large \ ❻
    --node-count=3 \ ❼
    --node-size=m4.2xlarge \ ❽
    --state=s3://my-kops \ ❾
    --yes ❿
```

❶ This is the command to create the cluster.

❷ Here we define the domain where the Kubernetes cluster will be created.

❸ This is optional if the cloud provider can be inferred from zones.

❹ Here we specify multiple zones (must be an odd number) to create multiple masters across the AZ.

❺ This line defines the zones in which to run the master.

❻ Here we provide the instance size of masters.

❼ This specifies the number of nodes in the cluster.

❽ This specifies the instance size of each node.

❾ This is the S3 bucket where the cluster configuration is stored.

❿ Here we specify the immediate creation of the cluster. Otherwise, only the state is stored in the bucket, and the cluster needs to be created separately.

For more details about the cluster, use the command `kubectl cluster-info`, which outputs the following:

```
Kubernetes master is running at https://api.kubernetes.arun
    gupta.me
KubeDNS is running at https://api.kubernetes.arungupta.me/api/
    v1/proxy/namespaces/kube-system/services/kube-dns

To further debug and diagnose cluster problems, use 'kubectl
    cluster-info dump'.
```

You can find a list of all the nodes in the cluster using `kubectl get nodes` command; you'll see the following output:

```
NAME                                        STATUS    AGE       VERSION
ip-172-20-109-249.compute.internal   Ready,master   7m   v1.5.2
ip-172-20-123-1.compute.internal     Ready,node     6m   v1.5.2
ip-172-20-32-140.compute.internal    Ready,master   7m   v1.5.2
ip-172-20-61-111.compute.internal    Ready,node     6m   v1.5.2
ip-172-20-82-253.compute.internal    Ready,master   7m   v1.5.2
ip-172-20-82-57.compute.internal     Ready,node     6m   v1.5.2
```

By default, a cluster created using Kops does not have the UI dashboard. But you can include it as an add-on:

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/
    kops/master/addons/kubernetes-dashboard/v1.5.0.yaml
deployment "kubernetes-dashboard" created
service "kubernetes-dashboard" created
```

The URL for the dashboard will be derived based upon the domain name used for starting the Kubernetes cluster. The Kubernetes dashboard is now available at *https://api.kubernetes.arungupta.me/ui*. To obtain the login credentials for the dashboard, use the command `kubectl config view`. The dashboard by itself looks like Figure 2-2.

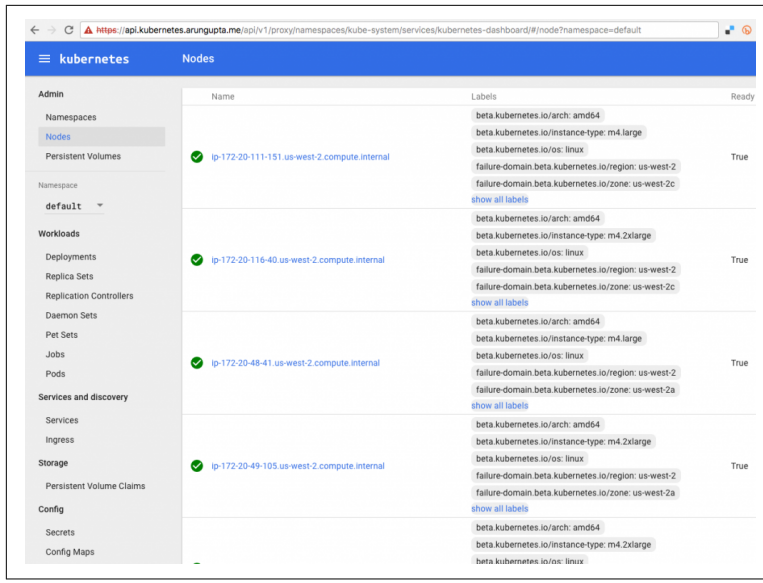

Figure 2-2. Kubernetes dashboard with Kops

Shut down the cluster

You can shut down and delete the cluster using the command shown in Example 2-3.

Example 2-3. Shut down multimaster Kubernetes cluster on AWS

```
kops-darwin-amd64 \
    delete cluster \
    --name=kubernetes.arungupta.me \
    --state=s3://kops-couchbase \
    --yes
```

Running Your First Java Application

Running a Java application in Kubernetes requires you to create different Kubernetes resources. A resource will use the Docker image, which contains all the application dependencies. The book *Docker for Java Developers* (O'Reilly) provides a reference on how to package your Java applications as Docker images.

`arungupta/oreilly-hello-java` is a Docker image that uses `openjdk` as the base image. This image is built using the `Dockerfile` at this book's GitHub page. Running this container prints the JDK

version. Running a Java application would require packaging all dependencies in the Docker image.

The easiest way to run a container on the Kubernetes cluster is to use the `kubectl run` command.

Create a pod using the `kubectl run` command:

```
kubectl run hello-java --image=arungupta/oreilly-hello-java
```

This creates a deployment. Check the status of deployment as follows:

```
kubectl get deployments
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-java   1         1         1            0           15s
```

This output shows that one replica of the pod is desired but is not available at this time.

Check the pods in the deployment like so:

```
kubectl get pods
NAME                          READY     STATUS
hello-java-888248798-8ingx    0/1       ContainerCreating

RESTARTS   AGE
0          25s
```

The first run of the pod requires the Docker image to be downloaded on the node where the pod is created. This is indicated by the status `ContainerCreating`. Adding the `-w` switch watches for any change in the object's state, and the output is updated accordingly.

Check logs from the pod using the pod's name:

```
kubectl logs hello-java-888248798-8ingx
openjdk version "1.8.0_102"
OpenJDK Runtime Environment (build 1.8.0_102-8u102-b14.1-1~bpo
    8+1-b14)
OpenJDK 64-Bit Server VM (build 25.102-b14, mixed mode)
```

The output shows the JDK version, as expected.

If you wait a few seconds to watch the output of the `kubectl get -w pods` command, you should see the following:

```
NAME             READY     STATUS             RESTARTS   AGE
hello-java       0/1       Completed          0          9m
hello-java-xxx   0/1       Completed          0          5s
hello-java-xxx   0/1       Completed          1          6s
hello-java-xxx   0/1       CrashLoopBackOff   1          7s
```

```
hello-java-xxx   0/1       Completed          2          29s
hello-java-xxx   0/1       CrashLoopBackOff   2          41s
hello-java-xxx   0/1       Completed          3          59s
```

By default, a pod's restart policy is set to `Always`. This means that if the pod terminates, then Kubernetes will restart the pod with an exponential backoff. The pod in our case just needs to print the Java version and terminate. Kubernetes attempts to restart the terminated pod. One way to tell Kubernetes to not restart the pod is to start the pod with a restart policy of `Never`. You can do so by modifying the `kubectl run` command and specifying the restart policy.

In order to do that, we need to delete the deployment first:

```
kubectl delete deployments hello-java
```

And then create the pod as shown:

```
kubectl run hello-java -- \
image=arungupta/oreilly-hello-java
```

This command will not create a deployment; instead, it creates a single pod. `kubectl get pods` returns the list of running pods. This pod terminates after printing the Java version. So, to see the exact name of the pod in this case, use the `kubectl get pods --show-all` command:

```
kubectl get pods --show-all
NAME         READY     STATUS      RESTARTS   AGE
hello-java   0/1       Completed   0          16m
```

You can then delete this pod using the command `kubectl delete pod hello-java`. Now, if you check the list of pods using the `kubectl get pods` command, an empty list will be returned.

Starting the WildFly Application Server

Now that you've created your first Java pod, start the WildFly application server container:

```
kubectl run hello-wildfly --image=jboss/wildfly:10.1.0.Final
    --port=8080
deployment "hello-wildfly" created
```

`--port` defines the port number that this container exposes. This only exposes the port within the Kubernetes cluster though, not outside.

Get the list of deployments like so:

```
kubectl get deployments
NAME            DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-wildfly   1         1         1            1           10s
```

This shows that the expected and current number of pods is 1. The pod is up to date with the image and is available.

To get more details about the deployment, use the `kubectl describe` command:

```
kubectl describe deployments hello-wildfly
Name:           hello-wildfly
Namespace:      default
CreationTimestamp:  Wed, 03 May 2017 14:22:09 -0700
Labels:         run=hello-wildfly
Annotations:        deployment.kubernetes.io/revision=1
Selector:       run=hello-wildfly
Replicas:       1 desired | 1 updated | 1 total | 1 available
    | 0 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Pod Template:
  Labels:   run=hello-wildfly
  Containers:
   hello-wildfly:
    Image:      jboss/wildfly:10.1.0.Final
    Port:       8080/TCP
    Environment:    <none>
    Mounts:     <none>
  Volumes:      <none>
Conditions:
  Type      Status  Reason
  ----      ------  ------
  Available     True    MinimumReplicasAvailable
OldReplicaSets: <none>
NewReplicaSet:  hello-wildfly-135927814 (1/1 replicas created)
Events:
  FirstSeen LastSeen    Count   From
  --------- --------    -----   ----
  39s       39s     1   deployment-controller

  SubObjectPath
  -------------
        Normal    ScalingReplicaSet   Scaled up replica
                  set hello-wildfly-135927814 to 1
```

A single replica of WildFly pod is created as part of this deployment.

The container started by this deployment is accessible only inside the cluster. Expose the deployment as a service using the `kubectl expose` command:

```
kubectl expose \
deployment hello-wildfly \
--name=hello-wildfly-service \
--port=8080 \
--target-port=8080
```

`--port` defines the port on which the service should listen. `--target-port` is the port on the container that the service should direct traffic to. In our case, service is listening on port `8080` and will redirect to port `8080` on the container.

All the exposed resources are available on the Kubernetes API server. You can access them by starting a proxy to the API server using the `kubectl proxy` command.

By default, this allows the resources to be accessible at this URL:

> *http://localhost:8001/api/v1/proxy/namespaces/default/<resource-type>/<resource-name>*

In our case, the deployed application is available here:

> *http://localhost:8001/api/v1/proxy/namespaces/default/services/hello-wildfly-service/index.html*

The link shows the main page of WildFly as shown in Figure 2-3.

Figure 2-3. WildFly home page using Kubernetes' Proxy

Some common options to configure for this proxy are:

- `--port` can be used to specify the port on which to run the proxy.
- `--address` is the IP address on which to serve.
- `--www` allows you to serve static files from a given directory under the specified prefix.

Delete the deployment as follows:

```
kubectl delete deployment/hello-wildfly
```

Managing Kubernetes Resources Using a Configuration File

A more common way to create a Kubernetes resource is by using the `kubectl create` command. This command can take one or more configuration files using the `-f` option.

You can easily create a WildFly pod using the configuration file in Example 1-1. This resource configuration file is at this book's GitHub page:

```
kubectl create -f wildfly-pod.yml
```

The output of this command is:

```
pod "wildfly-pod" created
```

The pod's name, its metadata, the list of containers within it, the Docker image name for each container, ports for each container, and other information about the pod and containers within it are all specified in this configuration file. You can easily confirm the list of created pods using the `kubectl get pods` command:

```
NAME          READY     STATUS    RESTARTS   AGE
wildfly-pod   1/1       Running   0          7s
```

You can also easily delete resources created using the configuration file:

```
kubectl delete -f wildfly-pod.yml
```

You can create a replication controller with two replicas of the pod using the configuration file in Example 1-2. This resource configuration file is in this book's GitHub repo:

```
kubectl create -f wildfly-rc.yml
```

The replication controller's name, the number of replicas, the metadata and specification for each pod, and other information is all specified in this configuration file. To see the replication controller and pods created, use the command `kubectl get rc,pods`, which gives the following output:

```
NAME            DESIRED   CURRENT   READY     AGE
rc/wildfly-rc   2         2         2         20s

NAME                   READY     STATUS    RESTARTS   AGE
po/wildfly-rc-7pf08    1/1       Running   0          20s
po/wildfly-rc-bbh52    1/1       Running   0          20s
```

Both the replication controller and the pods created from it are shown in the output. Different type of resources are separated with a blank line.

You can create multiple resources by using multiple resource configuration files and specify them using the `-f` option to the `kubectl create` command. Alternatively, all resources can be listed together in a single resource configuration file as shown in Example 1-5.

You can create a service with two WildFly pods using the configuration file at this book's GitHub page:

```
kubectl create -f wildfly-service.yml
```

The output of this command is:

```
service "wildfly-service" created
replicationcontroller "wildfly-rc" created
```

To see the multiple resources created using this configuration file, use the command `kubectl get service,rc`, which gives the following output:

```
NAME            DESIRED   CURRENT   READY     AGE
rc/wildfly-rc   2         2         2         13s

NAME                        CLUSTER-IP  EXTERNAL-IP PORT(S)   AGE
svc/hello-wildfly-service   10.0.0.157  <none>      8080/TCP  2h
svc/kubernetes              10.0.0.1    <none>      443/TCP   4h
svc/wildfly-service         10.0.0.220  <none>      8080/TCP  13s
```

You can scale the number of pod replicas in the replication controller using the command `kubectl scale`. For example, the number of replicas can be set to 4 like so:

```
kubectl scale --replicas=4 rc/wildfly-rc
```

Checking the list of pods using the command `kubectl get rc/wildfly-rc` now shows:

```
NAME         DESIRED   CURRENT   READY     AGE
wildfly-rc   4         4         4         3m
```

New pods created by this command have the labels `app: wildfly-rc-pod`. This is the `selector` label on the service as well. So the number of pods in that service has now increased from two to four.

Resources created by this configuration file can be deleted as follows:

```
kubectl delete -f wildfly-service.yml
```

Alternatively, you can delete multiple resources by specifying their names:

```
kubectl delete service/wildfly-service rc/wildfly-rc
```

Service Discovery for Java Application and a Database

A typical application consists of an application server, a web server, a messaging server, a database, and possibly some other components. Multiple replicas of each component are generally started to avoid a single point of failure (SPOF) in a distributed architecture. These components need to dynamically scale and be able to discover each other with a well-defined name.

This model maps well to Kubernetes. Each component can be started as multiple pods through a replication controller and scaled easily with the `kubectl scale` command. Pods in the replication controller can also be exposed as a service. This allows different components of the application to interact with each other. The `labels` on the pod created by the replication controller match the `selector` on the service.

This section will explain how to run a Couchbase database service, run a Spring Boot application that uses the Couchbase service, and store a JSON document in it (see Figure 2-4). This concept can be applied for two or more pods to communicate.

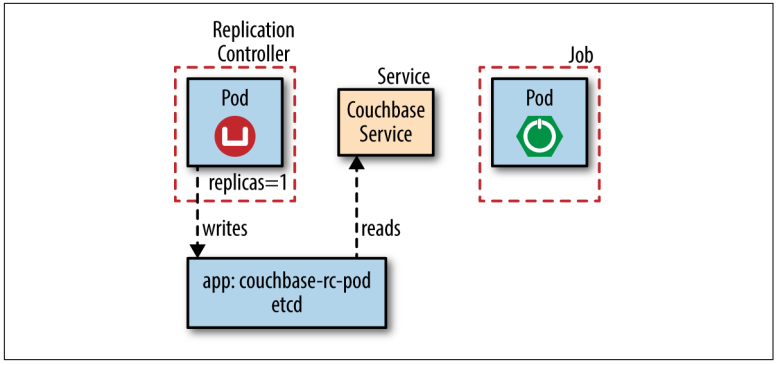

Figure 2-4. Bootiful Couchbase

Start the Couchbase service using the configuration file in Example 2-4. This file is available at this book's GitHub repo.

Example 2-4. Couchbase service

```
apiVersion: v1
kind: Service ❶
metadata:
  name: couchbase-service
spec:
  selector:
    app: couchbase-rc-pod ❷
  ports:
    - name: admin
      port: 8091
    - name: views
      port: 8092
    - name: query
      port: 8093
    - name: memcached
      port: 11210
---
apiVersion: v1
kind: ReplicationController ❶
metadata:
  name: couchbase-rc
spec:
  replicas: 1 ❸
  template:
    metadata:
      labels:
        app: couchbase-rc-pod ❷
    spec:
      containers:
      - name: couchbase
        image: arungupta/oreilly-couchbase ❹
        ports:
        - containerPort: 8091
        - containerPort: 8092
        - containerPort: 8093
        - containerPort: 11210
```

In this example:

❶ There are two resources: one service and a replication controller with one replica of the pod.

❷ The label specified in the selector `app: couchbase-rc-pod` matches the label on the pod created by the replication controller. This enables all pods created by the replication controller to be part of the service.

❸ One replica of the pod is created.

❹ The *arungupta/oreilly-couchbase* image is built using the `Docker file` defined in this GitHub repo. This image configures the Couchbase database using Couchbase REST API and creates a sample bucket.

Create this resource as follows:

```
kubectl create -f app-couchbase-service.yml
```

This creates the replication controller, the pod for the replication controller, and the service that includes this pod. Find the pod's name using the command `kubectl get -w pods`:

```
NAME                READY  STATUS              RESTARTS   AGE
couchbase-rc-8r6j1  0/1    ContainerCreating   0          10s
couchbase-rc-8r6j1  1/1    Running   0         23s
```

`-w` watches for changes in the requested object. The status of the pod is `ContainerCreating` and then `Running`. The first run of this command involves downloading the Docker image from the Docker hub, so it may take some time for the pod to start, depending upon your internet connectivity.

Check logs for the pod using the command `kubectl logs couchbase-rc-8r6j1`:

```
Starting Couchbase Server -- Web UI available at http://<ip>:8091
*   Trying 127.0.0.1...
  % Total    % Received % Xferd  Average Speed   Time    Time
                                 Dload  Upload   Total   Spent
  0     0    0     0    0     0      0      0 --:--:-- --:--:--
> POST /pools/default HTTP/1.1
> User-Agent: curl/7.40.0-DEV
> Host: 127.0.0.1:8091

. . .

<
{ [286 bytes data]
100   325  100   286  100    39    140     19  0:00:02  0:00:02
* Connection #0 to host 127.0.0.1 left intact
{"newBaseUri":"http://127.0.0.1:8091/"}{
    "requestID": "9f109317-5764-4d75-bce1-2ec7b1f502cc",
    "signature": null,
    "results": [
    ],
    "status": "success",
```

```
        "metrics": {
            "elapsedTime": "2.029432083s",
            "executionTime": "2.028563628s",
            "resultCount": 0,
            "resultSize": 0
        }
    }
    /entrypoint.sh couchbase-server
```

This log shows the output from the REST API for configuring Couchbase. It also shows that the Couchbase pod has started correctly.

A Spring Boot application that connects to this Couchbase database and stores a JSON document in the bucket is defined at this GitHub page. The Fabric8 Maven plugin is used to create the Docker image `arungupta/bootiful-couchbase`.

Run the Spring Boot application using the configuration file in Example 2-5. This file is in this book's GitHub repo.

Example 2-5. Spring Boot application

```
apiVersion: batch/v1
kind: Job ❶
metadata:
  name: bootiful-couchbase
  labels:
    name: bootiful-couchbase-pod
spec:
  template:
    metadata:
      name: bootiful-couchbase-pod
    spec:
      containers:
      - name: bootiful-couchbase
        image: arungupta/bootiful-couchbase ❷
      restartPolicy: Never ❸
```

In this example:

❶ This resource is a run-once job. More details about this type of resource are given in "Jobs" on page 10.

❷ The Spring Boot application that accesses the Couchbase database, and stores a JSON document in it, is packaged as a Docker image.

❸ The Java application is executed once, so the restart policy for the container needs to be set to `Never`. This type of resource ensures one successful completion of the pod.

The last missing piece that connects the Java application and Couchbase is `application.properties`. This file is used by Spring Boot to bootstrap the application configuration:

```
spring.couchbase.bootstrap-hosts=couchbase-service
spring.couchbase.bucket.name=books
```

To connect to the Couchbase service, the Spring Boot application uses the `spring.couchbase.bootstrap-hosts` property value. The value of this propery is the service name used in Example 2-4.

Create this resource like so:

```
kubectl create -f app-bootiful-couchbase.yml
```

This creates a run-once job that leads to at least one successful completion of a pod.

Get the status of the pods using the command `kubectl get pods`:

```
NAME                      READY  STATUS            RESTARTS AGE
bootiful-couchbase-rzzgq  0/1    ContainerCreating 0        3s
couchbase-rc-7a544        1/1    Running           0        2m
```

`bootiful-couchbase-zyq8h` is the pod that is created by this job. Once the Docker image is downloaded, a pod for the job is created and leads to successful completion. Getting a list of the pods again will show only the Couchbase pod:

```
NAME                READY     STATUS    RESTARTS   AGE
couchbase-rc-7a544  1/1       Running   0          3m
```

Note how the pod `bootiful-couchbase-rzzgq` is not shown in this list. This is because the pod has likely completed successfully.

To see the complete list of pods, use the command `kubectl get pods --show-all`, which gives the result:

```
NAME                      READY     STATUS      RESTARTS   AGE
bootiful-couchbase-rzzgq  0/1       Completed   0          1m
couchbase-rc-8r6j1        1/1       Running     0          13m
```

The pod `bootiful-couchbase-rzzgq` status shows as `Completed`. You can see logs for the completed pod by using the `kubectl logs bootiful-couchbase-rzzgq` command. The output looks like:

```
exec java -javaagent:/opt/agent-bond/agent-bond.jar=jolokia{
    {host=0.0.0.0}},jmx_exporter{{9779:/opt/agent-bond/jmx_
    exporter_config.yml}} -cp . -jar /deployments/bootiful-
    couchbase.jar
I> No access restrictor found, access to any MBean is allowed
Jolokia: Agent started with URL http://172.17.0.5:8778/jolokia/
2017-05-04 00:23:05.101:INFO:ifasjipjsoejs.Server:jetty-8.y.z-
    SNAPSHOT
2017-05-04 00:23:05.136:INFO:ifasjipjsoejs.AbstractConnector:
Started SelectChannelConnector@0.0.0.0:9779

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.4.0.RELEASE)


. . .


2017-05-04 00:23:08.330  INFO 1 --- [      cb-io-1-2]
    com.couchbase.client.core.node.Node      : Connected to
    Node couchbase-service
2017-05-04 00:23:08.479  INFO 1 --- [-computations-3]
    c.c.c.core.config.ConfigurationProvider  : Opened bucket
    books
2017-05-04 00:23:09.094  INFO 1 --- [           main]
    o.s.j.e.a.AnnotationMBeanExporter        : Registering
    beans for JMX exposure on startup
Book{isbn=978-1-4919-1889-0, name=Minecraft Modding with
    Forge, cost=29.99}
2017-05-04 00:23:09.413  INFO 1 --- [           main]
    org.example.webapp.Application           : Started
    Application in 3.833 seconds (JVM running for 4.693)
2017-05-04 00:23:09.444  INFO 1 --- [      Thread-16]
    s.c.a.AnnotationConfigApplicationContext : Closing
    org.springframework.context.annotation.AnnotationConfig
    ApplicationContext@4eb7f003: startup date [Thu May 04
    00:23:06 GMT 2017]; root of context hierarchy
2017-05-04 00:23:09.447  INFO 1 --- [      Thread-16]
    o.s.j.e.a.AnnotationMBeanExporter        : Unregistering
    JMX-exposed beans on shutdown
2017-05-04 00:23:09.463  INFO 1 --- [      cb-io-1-1]
    com.couchbase.client.core.node.Node      : Disconnected
    from Node couchbase-service
2017-05-04 00:23:09.467  INFO 1 --- [      Thread-16]
    c.c.c.core.config.ConfigurationProvider  : Closed
    bucket books
```

The output shows:

1. The Spring Boot application is started.
2. The application is connected to the Couchbase service.
3. The Couchbase bucket is opened.
4. The JSON document is stored in the Couchbase instance.
5. The Couchbase bucket is closed.
6. The application is disconnected from the Couchbase service.

The key output in the log is the following statement:

```
Book{isbn=978-1-4919-1889-0, name=Minecraft Modding with Forge,
    cost=29.99}
```

This is the JSON document that is stored in the Couchbase database.

Kubernetes Maven Plugin

Fabric8 is an opinionated and open source integrated developer platform for continuous delivery of microservices using Kubernetes. One of the key components of Fabric8 is `fabric8-maven-plugin`. This is a Maven plugin that provides a seamless integration of the Kubernetes lifecycle with Maven. More details about the plugin are available at fabric8-maven-plugin website.

The plugin serves two purposes:

- Manage Docker images—for example, build images, run containers, and push the images to any Docker registry.
- Manage Kubernetes resources—for example, create Kubernetes resource configuration files, create a Kubernetes development cluster, and apply the resources to a cluster.

Java developers are already familiar with how to use Maven to create archives (e.g., JAR or WAR). This plugin extends the functionality of Maven to use the known targets for Kubernetes.

The plugin coordinates are shown in Example 2-6.

Example 2-6. Kubernetes Maven plugin coordinates

```
<plugin>
  <groupId>io.fabric8</groupId>
```

```
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.3.5</version>
</plugin>
```

Some of the commonly used goals of the plugin are listed in Table 2-2.

Table 2-2. Kubernetes Maven plugin goals

Goal	Description
`fabric8:build`	Build Docker images using the information specified in *pom.xml* or referring to an external `Dockerfile`
`fabric8:resource`	Generate Kubernetes resource configuration files
`fabric8:deploy`	Build your Docker image, generate Kubernetes resources, and deploy them to a Kubernetes cluster
`fabric8:undeploy`	Delete the Kubernetes resources deployed via `fabric8:run` or `fabric8:deploy`
`fabric8:log`	Show the logs of the running application
`fabric8:cluster-start`	Download Minikube and `kubectl` to create, configure, and start a Kubernetes development cluster
`fabric8:cluster-stop`	Stop a development cluster

The complete set of goals for this plugin is listed at the plugin's website.

A zero-config setup allows you to use the plugin with opinionated defaults. This is possible because of several generators that provide reasonable defaults for the commonly used build types, like a plain Java build or a Spring Boot application. If the build type is identified as a certain type it makes reasonable defaults, like the base image, which ports to expose, and the startup command. They can be configured, but offer only a few options.

For example, adding the plugin definition shown in Example 2-7 to your *pom.xml* file will generate Kubernetes resource configuration files in the *target/classes/META-INF/fabric8/kubernetes* directory before the `build` target is invoked. This ensures that the Kubernetes resources are packaged in the Docker image built using the `build` target.

Example 2-7. Kubernetes Maven plugin resource generation

```
<plugin>
    <groupId>io.fabric8</groupId>
```

```
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>3.3.5</version>
    <executions>
        <execution>
            <goals>
                <goal>resource</goal>
                <goal>build</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

A complete working sample of how to configure a simple Java Maven application with this plugin is available in this book's GitHub repo.

The plugin can also be configured with the `<configuration>` section to override the defaults.

OpenShift is Red Hat's open source container application platform. This plugin can also be used to generate OpenShift descriptors.

More details about the Kubernetes Maven plugin are available at the plugin's website.

CHAPTER 3

Advanced Concepts

This chapter addresses how to run stateful containers using persistent volumes and stateful set. It covers different aspects of pods, including automatic scaling of pods using horizontal pod autoscaling, configuring a singleton instance of a pod using daemon sets, and performing health checks of a pod using probes. Using namespaces to share resources in a Kubernetes cluster across different groups is also explained. In addition, the chapter describes how to perform a rolling update of an application using replication controllers and deployments. And finally, it covers exposing a service outside the cluster.

All resource files in this chapter are available at this book's GitHub repo.

Persistent Volume

Stateful containers, such as Couchbase, need to store data outside the container. This allows the data to be preserved independent of container and pod restarts. To accomplish this, Kubernetes provides *volumes*, which are essentially directories accessible to the pod. The lifecycle of a volume is tied to the pod that created it. Pods can store data on this volume and preserve data across container restarts. But the volume ceases to exist along with the pod. Moreover, pods are ephemeral and so may be rescheduled on a different host. This means the data cannot be stored on a host as well.

Kubernetes *persistent* volumes solve this problem by providing persistent, cluster-scoped storage for applications that require long-lived data.

Creating and using a persistent volume is a three-step process:

Provision
: The administrator provisions a networked storage solution in the cluster, such as AWS ElasticBlockStore (EBS) volumes. This is called a *persistent volume* (PV).

Request storage
: The user requests storage for pods by using *claims*. Claims specify levels of resources (CPU and memory), sizes and access modes (e.g., can be mounted once read/write or many times write-only). This is called `PersistentVolumeClaim` (PVC).

Use claim
: Claims are mounted as volumes and used in pods for storage.

For an AWS Elastic Block Storage to be used as a persistent volume:

- The nodes on which pods are running must be AWS EC2 instances.
- Those instances need to be in the same region and availability zone as the EBS volume.
- EBS supports only a single EC2 instance mounting a volume.

You can create an AWS EBS volume using AWS CLI:

```
aws ec2 create-volume \
--region us-west-2 \
--availability-zone us-west-2a \
--size 5 \
--volume-type gp2
```

Here's the output of this command:

```
{
    "AvailabilityZone": "us-west-2a",
    "Encrypted": false,
    "VolumeType": "gp2",
    "VolumeId": "vol-0e04a9f45ad0cc01d",
    "State": "creating",
    "Iops": 100,
    "SnapshotId": "",
    "CreateTime": "2017-05-04T22:37:35.506Z",
```

```
        "Size": 5
    }
```

The attribute `VolumeId` is the unique volume identifier. It is used to create a persistent volume using the configuration file shown in Example 3-1.

Example 3-1. Kubernetes persistent volume

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: couchbase-pv
  labels:
    type: amazonEBS
spec:
  capacity:
    storage: 5Gi ❶
  accessModes:
    - ReadWriteOnce ❷
  awsElasticBlockStore:
    volumeID: vol-0e04a9f45ad0cc01d ❸
    fsType: ext4
```

This configuration file specifies the following:

❶ The capacity of this storage is 5 GB.

❷ Storage can be mounted by only one node for read/write by a single node.

❸ A unique volume identifier is created using the AWS CLI.

You can create the persistent volume using the command `kubectl create -f volume-pv.yml`.

You can view the list of persistent volumes using the `kubectl get pv` command:

```
NAME           CAPACITY   ACCESSMODES   RECLAIMPOLICY
couchbase-pv   5Gi        RWO           Retain

STATUS      CLAIM     STORAGECLASS   REASON    AGE
Available                                      8s
```

The status of the volume as `Available`.

You can create a PVC using the configuration file shown in Example 3-2.

Example 3-2. Kubernetes persistent volume claim

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: couchbase-pvc
  labels:
    type: amazonEBS
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

The PVC is created using `kubectl create -f volume-pvc.yml`. Check the created PV and PVC using the command `kubectl get pv,pvc`:

```
kubectl get pv,pvc
NAME                CAPACITY    ACCESSMODES   RECLAIMPOLICY
pv/couchbase-pv     5Gi         RWO           Retain

STATUS      CLAIM                       STORAGECLASS   REASON
Available

AGE
3m

NAME                                          CAPACITY
pv/pvc-9a765e86-311b-11e7-be2d-08002724bd66   3Gi

ACCESSMODES   RECLAIMPOLICY   STATUS      CLAIM
RWO           Delete          Bound       default/couchbase-pvc

STORAGECLASS   REASON   AGE
standard               1m

NAME                STATUS
pvc/couchbase-pvc   Bound

VOLUME                                    CAPACITY
pvc-9a765e86-311b-11e7-be2d-08002724bd66 3Gi

ACCESSMODES   STORAGECLASS   AGE
RWO           standard       1m
```

To claim the PVC, create a Couchbase replication controller using the configuration file shown in Example 3-3.

Example 3-3. Couchbase with persistent volume claim

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: couchbase
spec:
  replicas: 1
  template:
    metadata:
      name: couchbase-rc-pod
      labels:
        name: couchbase-rc-pod
        context: couchbase-pv
    spec:
      containers:
      - name: couchbase-rc-pod
        image: arungupta/oreilly-couchbase ❶
        volumeMounts: ❷
        - mountPath: "/opt/couchbase/var"
          name: mypd
        ports:
        - containerPort: 8091
        - containerPort: 8092
        - containerPort: 8093
        - containerPort: 11210
      volumes: ❸
      - name: mypd
        persistentVolumeClaim:
          claimName: couchbase-pvc
```

In this example:

❶ The resource defines a replication controller using the `arungupta/couchbase` Docker image.

❷ We define which volumes are going to be mounted. */opt/couchbase/var* is the directory where the Couchbase server stores all the data. The mount is given the name `mypd`.

❸ We specify different volumes that can be used in this replication controller definition. The `name` attribute is referenced in `volumeMounts`. `claimName` is defined in Example 3-2.

You can create the replication controller using the command `kubectl create -f volume-couchbase.yml`.

Stateful Sets

Typically, pods are stateless. So if one of them is unhealthy or superseded by a newer version, then Kubernetes disposes of it. If it is part of a replication controller, then another pod will be started. This notion of treating pods as "cattle" work for stateless applications. Stateful pods, like Couchbase, have a stronger notion of identity. Such pods are called "pets," as they need to be managed more carefully and are typically long-lived entities with persistent storage.

Kubernetes 1.3 introduced the pet set as an *alpha* resource to define stateful pods. Kubernetes 1.5 renamed the pet set to stateful set and also made it a *beta* resource. Different API versions, such as alpha and beta, imply different levels of stability and support.

A stateful set ensures that a specified number of "pets" with unique identities are running at any given time. Each pet is a stateful pod. The identity of a pod is composed of:

- A stable hostname, available in DNS
- An ordinal index
- Stable storage, linked to the ordinal and hostname

Stateful applications typically have the following requirements:

- Discovery of peers for quorum
- Stable persistent storage
- Startup/teardown ordering

For example, creating a *n*-node Couchbase cluster involves these steps:

1. Create *n* nodes.
2. Pick any node, and add other *n*–1 nodes to it to create a homogeneous cluster.
3. When a new node starts, it can join the cluster by talking to any node in the cluster.

A stateful set requires that there be 0..*N*–1 pods. Each pod has a deterministic name in the format `<statefulset-name>-<ordinal>`, and a unique identity. The identity of a pod sticks to it, regardless of which node it is (re)scheduled on.

A stateful set for Couchbase cluster can be created as shown in Example 3-4.

Example 3-4. Couchbase stateful set configuration

```
apiVersion: apps/v1beta1 ❶
kind: StatefulSet ❷
metadata:
  name: couchbase
spec:
  serviceName: "couchbase" ❸
  replicas: 2
  template:
    metadata:
      labels:
        app: couchbase
      annotations:
        pod.alpha.kubernetes.io/initialized: "true"
    spec: ❹
      terminationGracePeriodSeconds: 0
      containers:
      - name: couchbase
        image: arungupta/couchbase:k8s-statefulset ❺
        ports:
        - containerPort: 8091
        volumeMounts: ❻
        - name: couchbase-data ❼
          mountPath: /opt/couchbase/var ❽
        env:
          - name: COUCHBASE_MASTER ❾
            value: "couchbase-0.couchbase.default.svc.cluster
                    .local" ❿
          - name: AUTO_REBALANCE ⓫
            value: "false"
  volumeClaimTemplates: ⓬
  - metadata:
      name: couchbase-data ⓭
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi ⓮
```

In this configuration file:

❶ `StatefulSet` is a beta resource and defined in its own API group using the path *apps/v1beta1*.

❷ The `StatefulSet` value defines this resource to be of the type stateful set.

❸ We define the pod's deterministic name by using the value of this attribute and attaching an ordinal number to it. In this case, the first stateful pod will be named `couchbase-0`, the second pod will be named `couchbase-1`, and so on.

❹ Each stateful set will be defined as a pod. This section defines the specification of the pod.

❺ Each pod is created using this image. It uses the Couchbase Docker image and configures it using Couchbase REST API. This image is built using `Dockerfile`.

❻ We define the volume that is mounted per pod.

❼ We define the name of the volume that is specified later in the configuration file.

❽ The Couchbase Docker container persists the data at */opt/couchbase/var*. In this case, the data will be persisted to the volume defined in ❼.

❾ The Docker image uses the environment variable `COUCHBASE_MASTER` to define the Couchbase bootstrap node. Other nodes connect to this node to create a cluster.

❿ This value is the deterministic name of the Couchbase stateful pod. A fully qualified name for this stateful set is defined using the format `$(`*`service name`*`).$(`*`namespace`*`).svc.cluster.local`. The `$(`*`service name`*`)` is defined `<`*`statefulset-name`*`>-<`*`ordinal`*`>`. The first ordinal for `StatefulSet` is 0. `default` is the namespace. `cluster.local` is the default domain name for the cluster.

⓫ The `AUTO_REBALANCE` property defines whether the Couchbase cluster needs to be rebalanced.

⓬ It is recommended that you manually rebalance the cluster after adding a few nodes. The environment value is set to `false` to indicate no rebalance.

❸ The `PersistentVolumeClaim` specification is used to define the volume claim. The required volume is created by the Kubernetes cluster on the underlying cloud provider.

❹ 1GB of storage space is requested for each volume.

In addition to this configuration file, a headless service is also required to control the domain within which pods are created.

More details about using this stateful set and creating a Couchbase cluster, including headless service, are available on GitHub.

More details about stateful sets are available at this book's GitHub page.

Horizontal Pod Autoscaling

Horizontal pod autoscaling automatically scales the numbers of pods in a replication controller, deployment, or replica based on observed CPU utilization. There is also an alpha support for application-provided metrics.

By default, the autoscaler checks every 30 seconds and adjusts the number of replicas to match the observed average CPU utilization to the target specified by the user.

Heapster provides container cluster monitoring and performance analysis. The autoscaler uses Heapster to collect CPU utilization information, so it must be installed in the cluster for autoscaling to work. Heapster may be turned on by deault depending upon how the Kubernetes cluster was installed. Alternatively, it can be installed using an add-on.

You can define a web application that is deployed using `webapp-deployment`, and scales between 2 and 10 pods, based upon 80% CPU utilization, using the configuration file shown in Example 3-5.

Example 3-5. Horizontal pod scaling configuration

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: webapp-autoscaler
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
```

```
    kind: Deployment
    name: webapp-deployment
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 80
```

More details about horizontal pod autoscaling are available at the Kubernetes website.

Daemon Sets

A *daemon set* ensures that a copy of the pod runs on a selected set of nodes. By default, all nodes in the cluster are selected, but you can specify criteria to select a limited number of nodes instead.

Pods defined in the daemon set are created on a node when it's added to the cluster, and removed when it's removed from the cluster.

Common use cases for a daemon set include running a log daemon such as `logstash`, an exporter for machine metrics such as `prom/node-exporter`, a monitoring agent such as New Relic agent, or a cluster storage daemon such as `glusterd`.

You can define a configuration to run a Prometheus monitoring agent pod on all nodes of the cluster using Example 3-6.

Example 3-6. Daemon set configuration

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: prom-exporter
spec:
  template:
    metadata:
      labels:
        tier: monitoring
        name: prom-exporter
    spec:
      containers:
      - name: prometheus
        image: prom/node-exporter
```

More details about daemon sets are available at the Kubernetes website.

Checking the Health of a Pod

Kubernetes provides diagnostic probes to perform a health check on pods. There are two types of probes: *liveness* and *readiness*.

A liveness probe indicates whether the container is live (i.e., running). A liveness probe for Couchbase can be specified as shown in Example 3-7.

Example 3-7. Liveness probe

```
apiVersion: extensions/v1beta1
kind: Deployment ❶
metadata:
  name: couchbase
spec:
  replicas: 1 ❶
  template:
    metadata:
      labels:
        app: couchbase
    spec:
      containers:
      - name: couchbase
        image: arungupta/oreilly-couchbase ❶
        ports:
        - containerPort: 8091
        livenessProbe: ❷
          httpGet: ❸
            path: /pools ❹
            port: 8091 ❺
          initialDelaySeconds: 30 ❻
          timeoutSeconds: 1 ❼
```

In this configuration file:

❶ We create a deployment resource with one replica using the image at *arungupta/oreilly-couchbase*.

❷ A liveness probe is identified by `livenessProbe`. Alternatively, a readiness probe is identified by `readinessProbe` and indicates whether the container is ready to service requests. If the liveness probe fails, Kubernetes will kill the container and the container will be subjected to its `RestartPolicy`. If the readiness probe fails, Kubernetes will remove the pod's IP address from the endpoints of all services that match the pod.

❸ The probe performs the diagnostic using one of the handlers described in Table 3-1.

Table 3-1. Handlers used by the probe to check the health of a pod

Handler	Diagnostic	Success criteria
`exec`	Executes a command inside the container	Command exits with status code 0
`tcpSocket`	Runs a TCP check against the container's IP address on a specified port	Port is open and connection can be established
`httpGet`	Performs an HTTP GET against the container's IP address on a specified port and path	Status code is between 200 and 399

❹ `/pools` is the HTTP URI path that is invoked by the probe.

❺ `8091` is the container port on which the diagnostic is run.

❻ The first diagnostic is run after the time specified here, 30 seconds in our case.

❼ The probe is marked failed if the diagnostic times out after the time specified here, 1 second in our case.

Often a live container may be performing some background task. The container may not be ready to receive a request in this case, or may not be able to respond within a specified time. For example, a Couchbase cluster may be getting rebalanced. In this case, the container is still relevant and does not need to be killed. But it may not be ready to process requests. So the liveness probe may succeed but the readiness probe may fail.

More details about health-checking a pod are at the Kubernetes website.

Namespaces

All resources in a Kubernetes cluster are created in a default namespace. A pod will run with unbounded CPU and memory requests/limits. A Kubernetes namespace allows you to partition created resources into a logically named group. Each namespace provides:

- A unique *scope* for resources to avoid name collisions

- *Policies* to ensure appropriate authority is granted to trusted users
- Ability to specify *constraints* for resource consumption

This allows a Kubernetes cluster to share resources by multiple groups and provide different levels of quality of service (QoS) for each group. Resources created in one namespace are hidden from other namespaces. Multiple namespaces can be created, each potentially with different constraints. This list of namespaces can be obtained using the following command:

```
kubectl get namespace
```

This command shows the following output:

```
NAME          STATUS    AGE
default       Active    27d
kube-public   Active    27d
kube-system   Active    27d
```

Any pod, service, or replication controller will be created in this `default` namespace. The `kube-public` namespace is for bootstrap discovery. The `kube-system` namespace is reserved for resources created by the Kubernetes cluster.

More details about namespaces are available at the Kubernetes website.

Each namespace can be assigned a resource quota. As mentioned, by default, a pod will run with unbounded CPU and memory requests/limits. Specifying a quota allows you to restrict how much of the cluster resources can be consumed across all pods in a namespace.

You can create a resource quota using Example 3-8.

Example 3-8. Resource quota configuration

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota
spec:
  hard:
    cpu: "20"
    memory: 10Gi
    pods: "10"
    replicationcontrollers: "20"
    services: "5"
```

This configuration file requests 20 cores of CPU, 10 GB of memory across all pods, and a total of 10 pods, 20 replication controllers, and 5 services in the default namespace.

You can view the allocated quota using the `kubectl describe quota` command:

```
Name:         quota
Namespace:    default
Resource               Used  Hard
--------               ----  ----
cpu                    0     20
memory                 0     1Gi
pods                   0     10
replicationcontrollers 0     20
services               1     5
```

Some of the common resources that are supported by the quota system are shown in Table 3-2.

Table 3-2. Resource quotas

Resource	Description
`cpu`	Total requested CPU usage (number of cores)
`memory`	Total requested memory usage
`pods`	Total number of active pods where the phase is pending or active
`services`	Total number of services
`replicationcontrollers`	Total number of replication controllers
`resourcequotas`	Total number of resource quotas
`secrets`	Total number of secrets

More details about resource quotas are available at the Kubernetes website.

Rolling Updates

A rolling update is a method of deploying a new version of the application with zero downtime. You achieve this by updating pods one by one instead of updating all the pods at once.

Kubernetes allows rolling updates of replication controllers and deployments.

The `kubectl rolling-update` command is an imperative command and can be used to update the pods managed by a replication controller. The command updates one pod at a time. The steps in this process are as follows:

1. Create a new replication controller with the updated pod configuration.
2. Increase the number of pod replicas for the new replication controller.
3. Decreases the number of pod replicas for the original replication controller.
4. Delete the original replication controller.
5. Rename the new replication controller to the original replication controller.

Example 3-9 shows a rolling update of a replication controller with a new Docker image.

Example 3-9. Rolling update replication controller

```
kubectl.sh rolling-update webapp-rc --image=arungupta/wildfly-app:2
```

This command changes the Docker image of the existing pods of the replication controller `webapp-rc` to the new image at *arungupta/wildfly-app:2*, one at a time.

Example 3-10 shows a rolling update of a deployment with a new Docker image.

Example 3-10. Rolling update deployment

```
kubectl set image \
deployment/webapp-deployment \
webapp=arungupta/wildfly-app:2
```

This command changes the Docker image of the existing pods of the deployment `webapp-deployment` to the new image at *arungupta/wildfly-app:2*, one at a time.

You can check the deployment history using the command `kubectl rollout history deployment webapp-deployment`.

More details about rolling deployment are available at the Kubernetes website.

Exposing a Service

A service may need to be exposed outside of your cluster or on the public internet. You can define this behavior using the `type` property. This property can take three values:

`ClusterIP`
: This is the default and means that the service will be reachable only from inside the cluster.

`NodePort`
: This value exposes the service on a port on each node of the cluster (the same port on each node):

```
spec:
  type: NodePort
  ports:
    - name: web
      port: 8080
      nodePort: 30001
```

  The `nodePort` value must be within the fixed range 30000–32767.

  This allows you to set up your own custom load balancer.

`LoadBalancer`
: This option is valid only on cloud providers that support external load balancers. It builds upon the `NodePort` type. The cloud provider exposes an external load balancer that forwards the request from the load balancer to each node and the exposed port:

```
spec:
  . . .
  type: LoadBalancer
```

CHAPTER 4
Administration

This chapter explains how to get more details about the Kubernetes cluster. It introduces the Kubernetes dashboard and basic CLI commands, discusses application logs and other means of debugging the application, and covers monitoring an application's performance using add-on components of Heapster, InfluxDB, and Grafana.

The Kubernetes administration guide provides a comprehensive set of docs on Kubernetes administration.

Cluster Details

Once the cluster is up and running, often you'll want to get more details about it. You can obtain these details using `kubectl`. In addition, you can use the Kubernetes dashboard, a general-purpose, web-based UI, to view this information as well. It can be used to manage the cluster and applications running in the cluster.

The dashboard is accessible at the URI *http://<kubernetes-master>/ui*. This URI is redirected to this URI:

> *http://<kubernetes-master>:8443/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard*

It provides an overview of applications running in the cluster, different Kubernetes resources, and details about the cluster itself.

The dashboard is not enabled by default and so must be explicitly enabled. Installing Kubernetes Addons explains how to enable the

dashboard for a cluster installed using Kops. "Running Kubernetes Locally via Minikube" explains how to enable the dashboard for a cluster started using Minikube.

Figure 4-1 shows a view of dashboard with a Couchbase cluster and a WildFly replication controller.

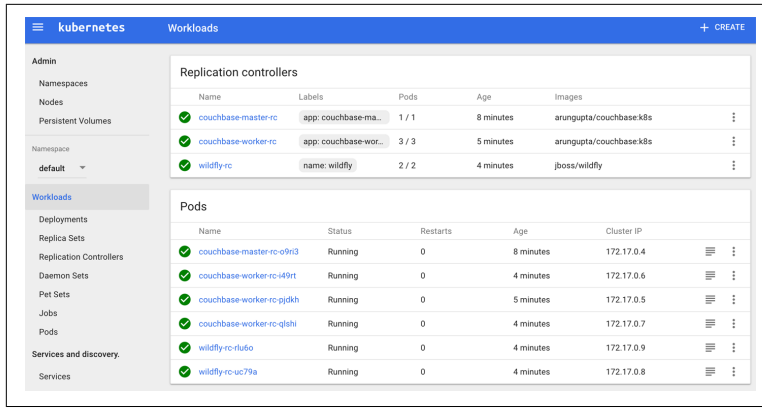

Figure 4-1. Kubernetes dashboard

It shows all the nodes and namespaces in the cluster. Once you choose a namespace, you'll see different resources within that namespace, such as deployments, replica sets, replication controllers, and daemon sets. You can create and manage each resource by uploading the resource configuration file.

The information in the dashboard is also accessible via the `kubectl` commands.

The `kubectl cluster-info` command displays the addresses of the master and services with the label `kubernetes.io/cluster-service=true`. The output from the Minikube install is shown in Example 4-1.

Example 4-1. Kubectl cluster-info output

```
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443/api/. . ./
    kube-dns
    kubernetes-dashboard is running at https://192.168.99.100:8443/
    api/. . ./kubernetes-dashboard

To further debug and diagnose cluster problems, use 'kubectl
    cluster-info dump'.
```

The output shows the URI of the master, the DNS service, and the dashboard.

Example 4-2 shows similar output from a Kubernetes cluster running on AWS.

Example 4-2. Kubectl cluster-info output from AWS

```
Kubernetes master is running at https://api.kubernetes.arungupta.me
KubeDNS is running at https://api.kubernetes.arungupta.me/api/.../
    kube-dns

To further debug and diagnose cluster problems, use 'kubectl
    cluster-info dump'.
```

This output does not have the dashboard URL because that component has not been installed yet.

As the output states, complete details about the cluster can be obtained with the `kubectl cluster-info dump` command.

More details about the client (i.e., `kubectl` CLI) and the server (i.e., Kubernetes API server) can be obtained with the `kubectl version` command. The output looks like the following:

```
Client Version: version.Info{Major:"1", Minor:"6", GitVersion:
    "v1.6.2", GitCommit:"477efc3cbe6a7effca06bd1452fa356e2201e
    1ee", GitTreeState:"clean", BuildDate:"2017-04-19T20:33:11
    Z", GoVersion:"go1.7.5", Compiler:"gc", Platform:"darwin/
    amd64"}
Server Version: version.Info{Major:"1", Minor:"6", GitVersion:
    "v1.6.0", GitCommit:"fff5156092b56e6bd60fff75aad4dc9de6b6ef
    37", GitTreeState:"dirty", BuildDate:"2017-04-07T20:46:46Z",
    GoVersion:"go1.7.3", Compiler:"gc", Platform:"linux/amd64"}
```

The output prints two lines, one each for the client and server. The value of the `Major` and `Minor` attributes defines the Kubernetes API server used by each one. Both use version 1.4 in our case. Other detailed information about the binary requesting and serving the Kubernetes API is displayed as well.

The `kubectl get nodes` command provides basic information—name, status, and age—about each node in the cluster. `kubectl describe nodes` provides detailed information about each node in the cluster. This include pods running on a node; CPU and memory requests and limits for each pod; labels, events, and conditions for each node in the cluster.

The `kubectl top` command displays resource consumption for nodes or pods. For nodes, it shows how many cores and memory are allocated to each node. It shows percent utilization for both of them as well.

Application Logs

Accessing application logs is typically the first step to debugging any errors. These logs provide valuable information about what might have gone wrong. Kubernetes provides integrated support for logging during development and production.

The `kubectl logs` command prints standard output and standard error output from the container(s) in a pod. If there is only one container in the pod, the container name is optional. If the pod consists of multiple containers, the container name must be specified—for example, `kubectl logs <pod-name> <container-name>`.

Some other relevant options for the command are:

- `-f` streams the log.
- `--tail=<n>` displays the last `<n>` files from the log.
- `-p` prints the logs from the previous instance of the container in a pod, if it exists.

You can view the complete set of options using `kubectl logs --help`.

This command is typically useful during early development stages, when there is only a handful of pods. An application typically consists of multiple replication controllers, each of which may create multiple pods. Existing pods may be terminated and new pods may be created by Kubernetes, based upon your application. Viewing application logs across multiple pods and containers using this command may not be the most efficient approach.

Kubernetes supports cluster-level logging that allows you to collect, manage, and query the logs of an application composed of multiple pods. Cluster-level logging allows you to collect logs that persist beyond the lifetime of the pod's container images or the lifetime of the pod or even cluster.

A typical usage is to manage these logs using ElasticSearch and Kibana. The logs can also be ingested in Google Cloud Logging. Additional logfiles from the container, specific to the application, can be sent to the cluster's ElasticSearch or Google Cloud Logging service.

Debugging Applications

As explained in the previous section, debugging an application typically requires looking at application logs. If that approach does not work, you need to start getting more details about the resources.

The `kubectl get` command is used to display basic information about one or more resources. Some of the common resource names are `pods` (aka `pod`), `replicasets` (aka `rs`), `services` (aka `svc`), and `deployments` (aka `deploy`). For example, `kubectl get pods` will display the list of pods as follows:

```
NAME                          READY     STATUS    RESTARTS   AGE
couchbase-master-rc-o9ri3     1/1       Running   0          1h
couchbase-worker-rc-i49rt     1/1       Running   0          1h
couchbase-worker-rc-pjdkh     1/1       Running   0          1h
couchbase-worker-rc-qlshi     1/1       Running   0          1h
wildfly-rc-rlu6o              1/1       Running   0          1h
wildfly-rc-uc79a              1/1       Running   1          1h
```

`kubectl --help` shows the complete list of resource names that can be used with this command.

By default, basic details about each resource are shown. To display only the name for each resource, use the `-o name` option. To see a complete JSON or YAML representation of the resource, use the `-o json` and `-o yaml` options, respectively.

You can use `-w` to watch for state changes of a resource. This is particularly useful when you're creating pods using replication controllers. It allows you to see the pod going through different stages.

With multiple applications deployed in the cluster, it's likely that the pods created by each application will have specific labels. You can use the `-l` option to query resources, using the selector (label query) to filter on.

The `kubectl describe` command can be used to get more details about a specific resource or a group of resources. For example, the `kubectl get svc` command shows the list of all services:

```
Name:                   couchbase-master-service
Namespace:              default
Labels:                 app=couchbase-master-service
Selector:               app=couchbase-master-pod
Type:                   LoadBalancer
IP:                     10.0.0.235
Port:                   <unset> 8091/TCP
NodePort:               <unset> 31854/TCP
Endpoints:              172.17.0.4:8091
Session Affinity:       None
No events.

Name:                   kubernetes
Namespace:              default
Labels:                 component=apiserver
                        provider=kubernetes
Selector:               <none>
Type:                   ClusterIP
IP:                     10.0.0.1
Port:                   https   443/TCP
Endpoints:              10.0.2.15:8443
Session Affinity:       ClientIP
```

You can use the `kubectl get events` command to see all events in the cluster. These events provide a high-level view of what is happening in the cluster. To obtain events from a specific namespace, you can use the `--namespace=<namespace>` option.

`kubectl attach` can be used to attach to a process that is already running inside an existing container. This is possible only if the container's specification has the `stdin` and `tty` attributes set to `true`, as shown in Example 4-3.

Example 4-3. WildFly replica set with TTY

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: wildfly-rs
spec:
  replicas: 2
  selector:
    matchLabels:
      app: wildfly-rs-pod
  template:
    metadata:
      labels:
        app: wildfly-rs-pod
    spec:
```

```
containers:
- name: wildfly
  image: jboss/wildfly
  stdin: true
  tty: true
  ports:
  - containerPort: 8080
```

This configuration is very similar to Example 1-3. The main difference is the aforementioned `stdin` and `tty` attributes set to `true` in the container specification.

The `kubectl proxy` command runs a proxy to the Kubernetes API server. By default, the command starts a reverse proxy server at 127.0.0.1:8001. Then the list of pods can be accessed at the URI *http://127.0.0.1:8001/api/v1/pods*. Similarly, the list of services can be accessed at *http://127.0.0.1:8001/api/v1/services* and the list of replication controllers at *http://127.0.0.1:8001/api/v1/replicationcontrollers*. Other Kubernetes resources can be accessed at a similar URI as well.

You can start the proxy on a different port using `kubectl proxy --port=8080`.

The `kubectl exec` command allows you to execute a command in the container. For example, you can connect to a bash shell in the container using `kubectl exec <pod-id> -it bash`.

Containers within the pods are not directly accessible outside the cluster unless they are exposed via services. The `kubectl port-forward` command allows you to forward one or more local ports to a pod. For example, `kubectl port-forward couchbase-master-rc-o9ri3 8091` will forward port 8091 on the localhost to the port exposed in the Couchase pod started by the `couchbase-master-rc` replication controller. Now, the Couchbase web console can be accessed at *http://localhost:8091*. This could be very useful for debugging without exposing the pod outside the cluster.

The `kubectl --help` command provides a complete list of commands for the `kubectl` CLI.

Check out the Kubernetes website for more details on debugging replication controller and pods, debugging services, and see GitHub for a more comprehensive list of debugging tips.

Application Performance Monitoring

You can monitor the performance of an application in a Kubernetes cluster at multiple levels: whole cluster, services, pods, and containers. Figure 4-2 shows how this information is collected.

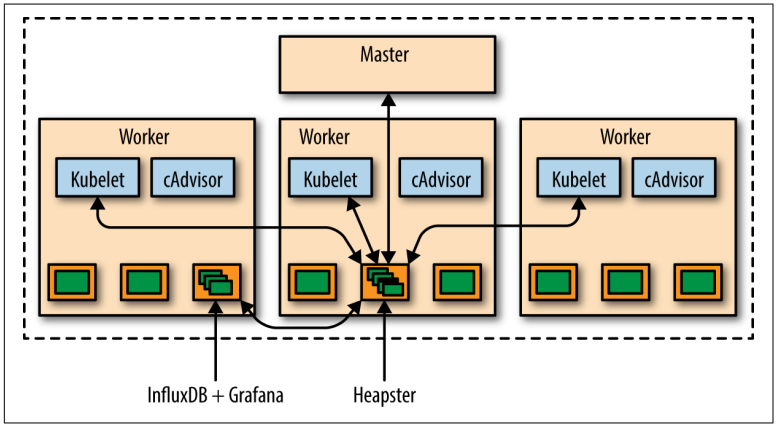

Figure 4-2. Kubernetes resource monitoring

The key components in this image are:

Heapster
: Heapster is a cluster-wide aggregator of monitoring and event data. It supports Kubernetes natively and works on all Kubernetes setups. Heapster runs as a pod in the cluster, similar to how any other Kubernetes application would run. The Heapster pod discovers all nodes in the cluster and queries usage information from each node's Kubelet.

cAdvisor
: The Kubelet itself fetches usage information data from cAdvisor. cAdvisor (container advisor) provides information on resource usage and performance characteristics of a running container. Specifically, for each container it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage, and network statistics. This data is exported by container and machine-wide.

InfluxDB
: InfluxDB is an open source database written in Go specifically to handle time series data with high-availability and high-

performance requirements. It exposes an easy-to-use API to write and fetch time series data. Heapster in Kubernetes is set up to use InfluxDB as the storage backend by default on most Kubernetes clusters. Other storage backends, such as Google Cloud Monitoring, are supported as well.

Grafana
: Grafana is an open source metric analytics and visualization suite. It is most commonly used for visualizing time series data for infrastructure and application analytics. It is available out of the box in a Kubernetes cluster. The Grafana container serves Grafana's UI, which provides an easy-to-configure dashboard interface for visualizing application performance in a Kubernetes cluster.

The default dashboard for Kubernetes contains an example dashboard that monitors resource usage of the cluster and the pods within it. This dashboard can easily be customized and expanded.

If you are using a Kubernetes cluster on AWS, then Heapster, InfluxDB, and Grafana are already available. If you're using Minikube or Kops, then these add-ons need to be enabled.

You can obtain complete details about different endpoints in the cluster using the `kubectl cluster-info` command. The output looks as shown in Example 4-2.

Access the Grafana endpoint URI in a browser. Use `kubectl config view` to get the login name and password.

The cluster dashboard is shown in Figure 4-3.

Figure 4-3. Cluster monitoring Grafana dashboard

The dashboard shows overall CPU, memory, network, and filesystem usage. This information is displayed per node as well. Additional data points can be configured on a custom dashboard.

The pods dashboard is shown in Figure 4-4.

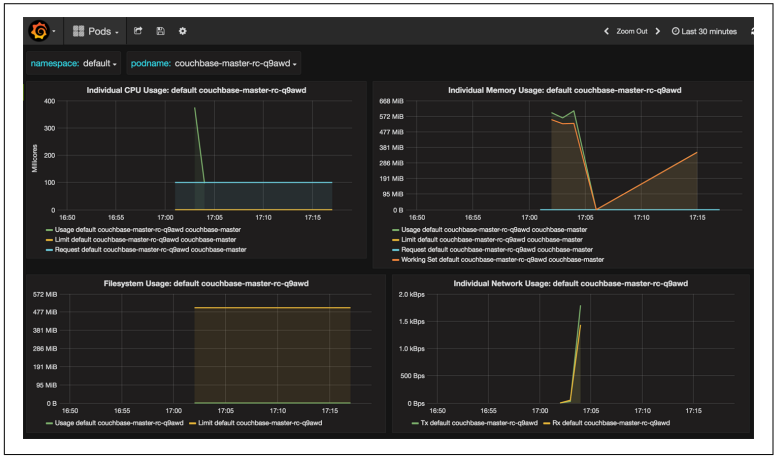

Figure 4-4. Pods monitoring Grafana dashboard

CPU, memory, network, and filesystem usage for a pod is displayed. You can choose different pods from the drop-down list.

In addition to the default monitoring tools in Kubernetes, some of the popular open source and commercial offerings are by Sysdig, Weaveworks, and Datadog.

CHAPTER 5
Conclusion

It would be nice if developers could write an application and test it locally on their machine, and operations teams could deploy the exact same application on any infrastructure of their choice. Although the technology is continuously improving, this still requires quite a bit of hand-crafting for the majority of cases.

Container-based solutions have gone a long way toward bridging this impedance mismatch by packaging applications as Docker containers in an easy and portable way. These applications can run in your on-premises data center or on public clouds. You can hack your own scripts for orchestration, but that requires a significant effort on your part to maintain them. This is exactly where an orchestration platform such as Kubernetes is helpful.

With 1,000+ contributors and 240 releases over the past 2.5 years as of this writing, Kubernetes is one of the fastest-moving open source projects on GitHub. It is supported on all major cloud providers, such as Amazon Web Services, Google Cloud, and Microsoft Azure, as well as on managed services like Google Container Engine, Azure Container Service, and Red Hat OpenShift. This integration with existing toolchains makes adoption seamless and delivers significant benefits.

By introducing another layer of abstraction between applications and cloud providers, Kubernetes enables a cloud-agnostic development and deployment platform. Developers can work on applications, create Docker images, author Kubernetes configuration files, and test them using Minikube or a development cluster. Operations

teams can then create a production-ready cluster on a cloud provider and use the same configuration files to deploy the application.

Docker's container donation to the Cloud Native Computing Foundation only enables a closer collaboration between the two projects. Kubernetes already supports different container formats. The Open Container Initiative is working on creating open industry standards around container formats and runtime. Projects like CRI-O will enable standardized support for containers in Kubernetes.

Even though not a strict requirement, containers simplify microservices deployment. There are some common tenets behind both: the single responsibility principle, isolation, explicitly published interface, service discovery, and ability to scale. All of these aspects are embraced by Kubernetes as well.

There is a lot of excitement and an equal amount of work happening in this space. This is not just hype—containers provide the real and credible benefits of simplicity and portability. Kubernetes orchestration simplifies the plumbing needed to get those containers up and running at all times.

If you have yet to explore the world of container orchestration, let me just say, "Toto, we're not in Kansas anymore!"

About the Author

Arun Gupta is a Principal Open Source Technologist at Amazon Web Services. He has built and led developer communities for 10+ years at Sun, Oracle, Red Hat, and Couchbase. He has deep expertise in leading cross-functional teams to develop and execute strategy, content, marketing campaigns, and programs. Prior to that he led engineering teams at Sun and is a founding member of the Java EE team. Gupta has authored more than 2,000 blog posts on technology. He has extensive speaking experience in more than 40 countries on myriad topics and has been a JavaOne Rock Star for four years in a row. Gupta also founded the Devoxx4Kids chapter in the US and continues to promote technology education among children. An author of several books on technology, an avid runner, a globe trotter, a Java Champion, a JUG leader, a NetBeans Dream Team member, and a Docker Captain, he is easily accessible at @arungupta.