O'REILLY®

Compliments of heptio

FREE CHAPTERS

Kubernetes Up & Running

DIVE INTO THE FUTURE OF INFRASTRUCTURE

Kelsey Hightower,
Brendan Burns & Joe Beda

heptio

Unleash the Power of Kubernetes.

- Easily get your Kubernetes cluster up and running with Heptio training, services, and support
- Simplify operations with our open source tools and products

heptio.com

Kubernetes: Up and Running

Dive into the Future of Infrastructure

This excerpt contains Chapters 4–7 and 11–12 of *Kubernetes: Up and Running*. The final book is available for sale on oreilly.com and through other retailers.

Kelsey Hightower, Brendan Burns, and Joe Beda

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

Kubernetes: Up and Running
by Kelsey Hightower, Brendan Burns, and Joe Beda

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*.

Editor: Angela Rufino
Production Editor: Melanie Yarbrough
Copyeditor: Christina Edwards
Proofreader: Rachel Head
Indexer: Kevin Broccoli
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

September 2017: First Edition

Revision History for the First Edition
2017-09-05: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491935675* for release details.

978-1-491-93567-5

[LSI]

Table of Contents

Foreword

Three years ago, I was part of the small team that created Kubernetes—with the goal of radically simplifying the task of building, deploying, and maintaining distributed systems. We wanted to surface the lessons from 10+ years at Google to help companies across every industry power mission-critical applications. The uptake of Kubernetes since its inception has been truly incredible—the technology is actively being used by more than half of Fortune 100 companies and has incredibly strong community engagement witnessed in its 27,000 GitHub stars and 1,400 contributors.

I co-authored this book to help developers and operators get started with Kubernetes. The book is a practical guide to the core concepts of Kubernetes. It also delves into the reasoning behind choosing those concepts and how they can be used to improve the development, delivery, and maintenance of distributed applications. The excerpt begins with more introductory topics including how to take advantage of common commands for the kubectl command-line tool and how to deploy an application using pods, labels, and annotations, and follows with more advanced topics such as service discovery, ConfigMaps, secrets, and the deployment object.

I started Heptio with fellow Kubernetes co-creator Craig McLuckie to bring the power of cloud native systems and Kubernetes to every organization. While the internet giants were the first to reap the benefits of cloud native architectures, through container technology and API-driven practices, IT can transform itself into a driving force for any business in a hybrid, multi-cloud environment.

We hope you enjoy the excerpt, and that you consider Heptio to jump start your journey with Kubernetes.

— Joe Beda
Co-Founder and CTO, Heptio

CHAPTER 4

Common kubectl Commands

The kubectl command-line utility is a powerful tool, and in the following chapters you will use it to create objects and interact with the Kubernetes API. Before that, however, it makes sense to go over the basic kubectl commands that apply to all Kubernetes objects.

Namespaces

Kubernetes uses *namespaces* to organize objects in the cluster. You can think of each namespace as a folder that holds a set of objects. By default, the kubectl command-line tool interacts with the default namespace. If you want to use a different namespace, you can pass kubectl the --namespace flag. For example, kubectl --namespace=mystuff references objects in the mystuff namespace.

Contexts

If you want to change the default namespace more permanently, you can use a *context*. This gets recorded in a kubectl configuration file, usually located at $HOME/.kube/config. This configuration file also stores how to both find and authenticate to your cluster. For example, you can create a context with a different default namespace for your kubectl commands using:

```
$ kubectl config set-context my-context --namespace=mystuff
```

This creates a new context, but it doesn't actually start using it yet. To use this newly created context, you can run:

```
$ kubectl config use-context my-context
```

Contexts can also be used to manage different clusters or different users for authenticating to those clusters using the `--users` or `--clusters` flags with the `set-context` command.

Viewing Kubernetes API Objects

Everything contained in Kubernetes is represented by a RESTful resource. Throughout this book, we refer to these resources as *Kubernetes objects*. Each Kubernetes object exists at a unique HTTP path; for example, *https://your-k8s.com/api/v1/namespaces/default/pods/my-pod* leads to the representation of a pod in the default namespace named `my-pod`. The `kubectl` command makes HTTP requests to these URLs to access the Kubernetes objects that reside at these paths.

The most basic command for viewing Kubernetes objects via `kubectl` is `get`. If you run *`kubectl get <resource-name>`* you will get a listing of all resources in the current namespace. If you want to get a specific resource, you can use *`kubectl get <resource-name> <object-name>`*.

By default, `kubectl` uses a human-readable printer for viewing the responses from the API server, but this human-readable printer removes many of the details of the objects to fit each object on one terminal line. One way to get slightly more information is to add the `-o wide` flag, which gives more details, on a longer line. If you want to view the complete object, you can also view the objects as raw JSON or YAML using the `-o json` or `-o yaml` flags, respectively.

A common option for manipulating the output of `kubectl` is to remove the headers, which is often useful when combining `kubectl` with Unix pipes (e.g., `kubectl ... | awk ...`). If you specify the `--no-headers` flag, `kubectl` will skip the headers at the top of the human-readable table.

Another common task is extracting specific fields from the object. `kubectl` uses the JSONPath query language to select fields in the returned object. The complete details of JSONPath are beyond the scope of this chapter, but as an example, this command will extract and print the IP address of the pod:

```
$ kubectl get pods my-pod -o jsonpath --template={.status.podIP}
```

If you are interested in more detailed information about a particular object, use the `describe` command:

```
$ kubectl describe <resource-name> <obj-name>
```

This will provide a rich multiline human-readable description of the object as well as any other relevant, related objects and events in the Kubernetes cluster.

Creating, Updating, and Destroying Kubernetes Objects

Objects in the Kubernetes API are represented as JSON or YAML files. These files are either returned by the server in response to a query or posted to the server as part of an API request. You can use these YAML or JSON files to create, update, or delete objects on the Kubernetes server.

Let's assume that you have a simple object stored in *obj.yaml*. You can use `kubectl` to create this object in Kubernetes by running:

```
$ kubectl apply -f obj.yaml
```

Notice that you don't need to specify the resource type of the object; it's obtained from the object file itself.

Similarly, after you make changes to the object, you can use the `apply` command again to update the object:

```
$ kubectl apply -f obj.yaml
```

If you feel like making interactive edits, instead of editing a local file, you can instead use the `edit` command, which will download the latest object state, and then launch an editor that contains the definition:

```
$ kubectl edit <resource-name> <obj-name>
```

After you save the file, it will be automatically uploaded back to the Kubernetes cluster.

When you want to delete an object, you can simply run:

```
$ kubectl delete -f obj.yaml
```

But it is important to note that `kubectl` will not prompt you to confirm the delete. Once you issue the command, the object *will* be deleted.

Likewise, you can delete an object using the resource type and name:

```
$ kubectl delete <resource-name> <obj-name>
```

Labeling and Annotating Objects

Labels and annotations are tags for your objects. We'll discuss the differences in Chapter 6, but for now, you can update the labels and annotations on any Kubernetes object using the `annotate` and `label` commands. For example, to add the `color=red` label to a pod named `bar`, you can run:

```
$ kubectl label pods bar color=red
```

The syntax for annotations is identical.

By default, `label` and `annotate` will not let you overwrite an existing label. To do this, you need to add the `--overwrite` flag.

If you want to remove a label, you can use the `-<label-name>` syntax:

```
$ kubectl label pods bar -color
```

This will remove the `color` label from the pod named `bar`.

Debugging Commands

`kubectl` also makes a number of commands available for debugging your containers. You can use the following to see the logs for a running container:

```
$ kubectl logs <pod-name>
```

If you have multiple containers in your pod you can choose the container to view using the `-c` flag.

By default, `kubectl logs` lists the current logs and exits. If you instead want to continuously stream the logs back to the terminal without exiting, you can add the `-f` (follow) command-line flag.

You can also use the `exec` command to execute a command in a running container:

```
$ kubectl exec -it <pod-name> -- bash
```

This will provide you with an interactive shell inside the running container so that you can perform more debugging.

Finally, you can copy files to and from a container using the `cp` command:

```
$ kubectl cp <pod-name>:/path/to/remote/file /path/to/local/file
```

This will copy a file from a running container to your local machine. You can also specify directories, or reverse the syntax to copy a file from your local machine back out into the container.

Summary

`kubectl` is a powerful tool for managing your applications in your Kubernetes cluster. This chapter has illustrated many of the common uses for the tool, but `kubectl` has a great deal of built-in help available. You can start viewing this help with:

```
kubectl help
```

or:

```
kubectl help command-name
```

CHAPTER 5
Pods

In earlier chapters we discussed how you might go about containerizing your application, but in real-world deployments of containerized applications you will often want to colocate multiple applications into a single atomic unit, scheduled onto a single machine.

A canonical example of such a deployment is illustrated in Figure 5-1, which consists of a container serving web requests and a container synchronizing the filesystem with a remote Git repository.

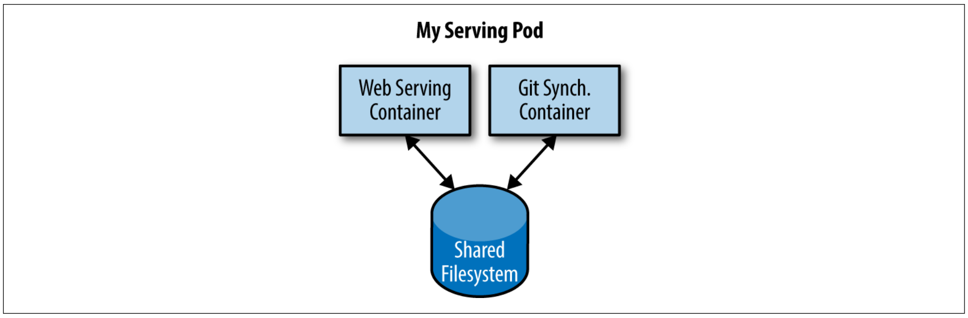


Figure 5-1. An example Pod with two containers and a shared filesystem

At first, it might seem tempting to wrap up both the web server and the Git synchronizer into a single container. After closer inspection, however, the reasons for the separation become clear. First, the two different containers have significantly different requirements in terms of resource usage. Take, for example, memory. Because the web server is serving user requests, we want to ensure that it is always available and responsive. On the other hand, the Git synchronizer isn't really user-facing and has a "best effort" quality of service.

Suppose that our Git synchronizer has a memory leak. We need to ensure that the Git synchronizer cannot use up memory that we want to use for our web server, since this can affect web server performance or even crash the server.

This sort of resource isolation is exactly the sort of thing that containers are designed to accomplish. By separating the two applications into two separate containers we can ensure reliable web server operation.

Of course, the two containers are quite symbiotic; it makes no sense to schedule the web server on one machine and the Git synchronizer on another. Consequently, Kubernetes groups multiple containers into a single, atomic unit called a *Pod*. (The name goes with the whale theme of Docker containers, since a Pod is also a group of whales.)

Pods in Kubernetes

A Pod represents a collection of application containers and volumes running in the same execution environment. Pods, not containers, are the smallest deployable artifact in a Kubernetes cluster. This means all of the containers in a Pod always land on the same machine.

Each container within a Pod runs in its own cgroup, but they share a number of Linux namespaces.

Applications running in the same Pod share the same IP address and port space (network namespace), have the same hostname (UTS namespace), and can communicate using native interprocess communication channels over System V IPC or POSIX message queues (IPC namespace). However, applications in different Pods are isolated from each other; they have different IP addresses, different hostnames, and more. Containers in different Pods running on the same node might as well be on different servers.

Thinking with Pods

One of the most common questions that occurs in the adoption of Kubernetes is "What should I put in a Pod?"

Sometimes people see Pods and think, "Aha! A WordPress container and a MySQL database container should be in the same Pod." However, this kind of Pod is actually an example of an antipattern for Pod construction. There are two reasons for this. First, Wordpress and its database are not truly symbiotic. If the WordPress container and the database container land on different machines, they still can work together quite effectively, since they communicate over a network connection. Secondly, you don't necessarily want to scale WordPress and the database as a unit. WordPress itself is mostly stateless, and thus you may want to scale your WordPress frontends in response to frontend load by creating more WordPress Pods. Scaling a MySQL database is much trickier, and you would be much more likely to increase the resources dedicated to a single MySQL Pod. If you group the WordPress and MySQL containers together in a single Pod, you are forced to use the same scaling strategy for both containers, which doesn't fit well.

In general, the right question to ask yourself when designing Pods is, "Will these containers work correctly if they land on different machines?" If the answer is "no," a Pod is the correct grouping for the containers. If the answer is "yes," multiple Pods is probably the correct solution. In the example at the beginning of this chapter, the two containers interact via a local filesystem. It would be impossible for them to operate correctly if the containers were scheduled on different machines.

In the remaining sections of this chapter, we will describe how to create, introspect, manage, and delete Pods in Kubernetes.

The Pod Manifest

Pods are described in a Pod manifest. The Pod manifest is just a text-file representation of the Kubernetes API object. Kubernetes strongly believes in *declarative configuration*. Declarative configuration means that you write down the desired state of the world in a configuration and then submit that configuration to a service that takes actions to ensure the desired state becomes the actual state.

Declarative configuration is different from *imperative configuration*, where you simply take a series of actions (e.g., `apt-get install foo`) to modify the world. Years of production experience have taught us that maintaining a written record of the system's desired state leads to a more manageable, reliable system. Declarative configuration enables numerous advantages, including code review for configurations as well as documenting the current state of the world for distributed teams. Additionally, it is the basis for all of the self-healing behaviors in Kubernetes that keep applications running without user action.

The Kubernetes API server accepts and processes Pod manifests before storing them in persistent storage (etcd). The scheduler also uses the Kubernetes API to find Pods that haven't been scheduled to a node. The scheduler then places the Pods onto nodes depending on the resources and other constraints expressed in the Pod manifests. Multiple Pods can be placed on the same machine as long as there are sufficient resources. However, scheduling multiple replicas of the same application onto the same machine is worse for reliability, since the machine is a single failure domain. Consequently, the Kubernetes scheduler tries to ensure that Pods from the same application are distributed onto different machines for reliability in the presence of such failures. Once scheduled to a node, Pods don't move and must be explicitly destroyed and rescheduled.

Multiple instances of a Pod can be deployed by repeating the workflow described here. However, ReplicaSets (Chapter 8) are better suited for running multiple instances of a Pod. (It turns out they're also better at running a single Pod, but we'll get into that later.)

Creating a Pod

The simplest way to create a Pod is via the imperative `kubectl run` command. For example, to run our same `kuard` server, use:

```
$ kubectl run kuard --image=gcr.io/kuar-demo/kuard-amd64:1
```

You can see the status of this Pod by running:

```
$ kubectl get pods
```

You may initially see the container as `Pending`, but eventually you will see it transition to `Running`, which means that the Pod and its containers have been successfully created.

Don't worry too much about the random strings attached to the end of the Pod name. This manner of creating a Pod actually creates it via `Deployment` and `ReplicaSet` objects, which we will cover in later chapters.

For now, you can delete this Pod by running:

```
$ kubectl delete deployments/kuard
```

We will now move on to writing a complete Pod manifest by hand.

Creating a Pod Manifest

Pod manifests can be written using YAML or JSON, but YAML is generally preferred because it is slightly more human-editable and has the ability to add comments. Pod manifests (and other Kubernetes API objects) should really be treated in the same way as source code, and things like comments help explain the Pod to new team members who are looking at them for the first time.

Pod manifests include a couple of key fields and attributes: mainly a `metadata` section for describing the Pod and its labels, a `spec` section for describing volumes, and a list of containers that will run in the Pod.

In Chapter 2 we deployed `kuard` using the following Docker command:

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  gcr.io/kuar-demo/kuard-amd64:1
```

A similar result can be achieved by instead writing Example 5-1 to a file named *kuard-pod.yaml* and then using `kubectl` commands to load that manifest to Kubernetes.

Example 5-1. kuard-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Running Pods

In the previous section we created a Pod manifest that can be used to start a Pod running `kuard`. Use the `kubectl apply` command to launch a single instance of `kuard`:

```
$ kubectl apply -f kuard-pod.yaml
```

The Pod manifest will be submitted to the Kubernetes API server. The Kubernetes system will then schedule that Pod to run on a healthy node in the cluster, where it will be monitored by the `kubelet` daemon process. Don't worry if you don't understand all the moving parts of Kubernetes right now; we'll get into more details throughout the book.

Listing Pods

Now that we have a Pod running, let's go find out some more about it. Using the `kubectl` command-line tool, we can list all Pods running in the cluster. For now, this should only be the single Pod that we created in the previous step:

```
$ kubectl get pods
NAME       READY     STATUS    RESTARTS   AGE
kuard      1/1       Running   0          44s
```

You can see the name of the Pod (`kuard`) that we gave it in the previous YAML file. In addition to the number of ready containers (`1/1`), the output also shows the status, the number of times the Pod was restarted, as well as the age of the Pod.

If you ran this command immediately after the Pod was created, you might see:

```
NAME       READY     STATUS    RESTARTS   AGE
kuard      0/1       Pending   0          1s
```

The `Pending` state indicates that the Pod has been submitted but hasn't been scheduled yet.

If a more significant error occurs (e.g., an attempt to create a Pod with a container image that doesn't exist), it will also be listed in the status field.

By default, the `kubectl` command-line tool tries to be concise in the information it reports, but you can get more information via command-line flags. Adding `-o wide` to any `kubectl` command will print out slightly more information (while still trying to keep the information to a single line). Adding `-o json` or `-o yaml` will print out the complete objects in JSON or YAML, respectively.

Pod Details

Sometimes, the single-line view is insufficient because it is too terse. Additionally, Kubernetes maintains numerous events about Pods that are present in the event stream, not attached to the Pod object.

To find out more information about a Pod (or any Kubernetes object) you can use the `kubectl describe` command. For example, to describe the Pod we previously created, you can run:

```
$ kubectl describe pods kuard
```

This outputs a bunch of information about the Pod in different sections. At the top is basic information about the Pod:

```
Name:           kuard
Namespace:      default
Node:           node1/10.0.15.185
Start Time:     Sun, 02 Jul 2017 15:00:38 -0700
Labels:         <none>
Annotations:    <none>
Status:         Running
IP:             192.168.199.238
Controllers:    <none>
```

Then there is information about the containers running in the Pod:

```
Containers:
  kuard:
    Container ID:  docker://055095…
    Image:         gcr.io/kuar-demo/kuard-amd64:1
    Image ID:      docker-pullable://gcr.io/kuar-demo/kuard-amd64@sha256:a580…
    Port:          8080/TCP
    State:         Running
      Started:     Sun, 02 Jul 2017 15:00:41 -0700
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-cg5f5 (ro)
```

Finally, there are events related to the Pod, such as when it was scheduled, when its image was pulled, and if/when it had to be restarted because of failing health checks:

```
Events:
  Seen From              SubObjectPath           Type      Reason    Message
  ---- ----              -------------           --------  ------    -------
  50s  default-scheduler                         Normal    Scheduled Success…
  49s  kubelet, node1    spec.containers{kuard}  Normal    Pulling   pulling…
  47s  kubelet, node1    spec.containers{kuard}  Normal    Pulled    Success…
  47s  kubelet, node1    spec.containers{kuard}  Normal    Created   Created…
  47s  kubelet, node1    spec.containers{kuard}  Normal    Started   Started…
```

Deleting a Pod

When it is time to delete a Pod, you can delete it either by name:

```
$ kubectl delete pods/kuard
```

or using the same file that you used to create it:

```
$ kubectl delete -f kuard-pod.yaml
```

When a Pod is deleted, it is *not* immediately killed. Instead, if you run `kubectl get pods` you will see that the Pod is in the `Terminating` state. All Pods have a termination *grace period*. By default, this is 30 seconds. When a Pod is transitioned to `Terminating` it no longer receives new requests. In a serving scenario, the grace period is important for reliability because it allows the Pod to finish any active requests that it may be in the middle of processing before it is terminated.

It's important to note that when you delete a Pod, any data stored in the containers associated with that Pod will be deleted as well. If you want to persist data across multiple instances of a Pod, you need to use `PersistentVolumes`, described at the end of this chapter.

Accessing Your Pod

Now that your Pod is running, you're going to want to access it for a variety of reasons. You may want to load the web service that is running in the Pod. You may want to view its logs to debug a problem that you are seeing, or even execute other commands in the context of the Pod to help debug. The following sections detail various ways that you can interact with the code and data running inside your Pod.

Using Port Forwarding

Later in the book, we'll show how to expose a service to the world or other containers using load balancers, but oftentimes you simply want to access a specific Pod, even if it's not serving traffic on the internet.

To achieve this, you can use the port-forwarding support built into the Kubernetes API and command-line tools.

When you run:

```
$ kubectl port-forward kuard 8080:8080
```

a secure tunnel is created from your local machine, through the Kubernetes master, to the instance of the Pod running on one of the worker nodes.

As long as the port-forward command is still running, you can access the Pod (in this case the `kuard` web interface) on *http://localhost:8080*.

Getting More Info with Logs

When your application needs debugging, it's helpful to be able to dig deeper than `describe` to understand what the application is doing. Kubernetes provides two commands for debugging running containers. The `kubectl logs` command downloads the current logs from the running instance:

```
$ kubectl logs kuard
```

Adding the `-f` flag will cause you to continuously stream logs.

The `kubectl logs` command always tries to get logs from the currently running container. Adding the `--previous` flag will get logs from a previous instance of the container. This is useful, for example, if your containers are continuously restarting due to a problem at container startup.

While using `kubectl logs` is useful for one-off debugging of containers in production environments, it's generally useful to use a log aggregation service. There are several open source log aggregation tools, like `fluentd` and `elasticsearch`, as well as numerous cloud logging providers. Log aggregation services provide greater capacity for storing a longer duration of logs, as well as rich log searching and filtering capabilities. Finally, they often provide the ability to aggregate logs from multiple Pods into a single view.

Running Commands in Your Container with exec

Sometimes logs are insufficient, and to truly determine what's going on you need to execute commands in the context of the container itself. To do this you can use:

```
$ kubectl exec kuard date
```

You can also get an interactive session by adding the `-it` flags:

```
$ kubectl exec -it kuard ash
```

Copying Files to and from Containers

At times you may need to copy files from a remote container to a local machine for more in-depth exploration. For example, you can use a tool like Wireshark to visualize `tcpdump` packet captures. Suppose you had a file called */captures/capture3.txt* inside a container in your Pod. You could securely copy that file to your local machine by running:

```
$ kubectl cp <pod-name>:/captures/capture3.txt ./capture3.txt
```

Other times you may need to copy files from your local machine into a container. Let's say you want to copy *$HOME/config.txt* to a remote container. In this case, you can run:

```
$ kubectl cp $HOME/config.txt <pod-name>:/config.txt
```

Generally speaking, copying files into a container is an antipattern. You really should treat the contents of a container as immutable. But occasionally it's the most immediate way to stop the bleeding and restore your service to health, since it is quicker than building, pushing, and rolling out a new image. Once the bleeding is stopped, however, it is critically important that you immediately go and do the image build and rollout, or you are guaranteed to forget the local change that you made to your container and overwrite it in the subsequent regularly scheduled rollout.

Health Checks

When you run your application as a container in Kubernetes, it is automatically kept alive for you using a *process health check*. This health check simply ensures that the main process of your application is always running. If it isn't, Kubernetes restarts it.

However, in most cases, a simple process check is insufficient. For example, if your process has deadlocked and is unable to serve requests, a process health check will still believe that your application is healthy since its process is still running.

To address this, Kubernetes introduced health checks for application *liveness*. Liveness health checks run application-specific logic (e.g., loading a web page) to verify that the application is not just still running, but is functioning properly. Since these liveness health checks are application-specific, you have to define them in your Pod manifest.

Liveness Probe

Once the `kuard` process is up and running, we need a way to confirm that it is actually healthy and shouldn't be restarted. Liveness probes are defined per container, which means each container inside a Pod is health-checked separately. In Example 5-2, we add a liveness probe to our `kuard` container, which runs an HTTP request against the `/healthy` path on our container.

Example 5-2. kuard-pod-health.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      livenessProbe:
        httpGet:
          path: /healthy
          port: 8080
        initialDelaySeconds: 5
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 3
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

The preceding Pod manifest uses an `httpGet` probe to perform an HTTP `GET` request against the `/healthy` endpoint on port 8080 of the `kuard` container. The probe sets an `initialDelaySeconds` of 5, and thus will not be called until five seconds after all the containers in the Pod are created. The probe must respond within the one-second timeout, and the HTTP status code must be equal to or greater than 200 and less than 400 to be considered successful. Kubernetes will call the probe every 10 seconds. If more than three probes fail, the container will fail and restart.

You can see this in action by looking at the `kuard` status page. Create a Pod using this manifest and then port-forward to that Pod:

```
$ kubectl apply -f kuard-pod-health.yaml
$ kubectl port-forward kuard 8080:8080
```

Point your browser to *http://localhost:8080*. Click the "Liveness Probe" tab. You should see a table that lists all of the probes that this instance of `kuard` has received. If you click the "fail" link on that page, `kuard` will start to fail health checks. Wait long enough and Kubernetes will restart the container. At that point the display will reset and start over again. Details of the restart can be found with `kubectl describe kuard`. The "Events" section will have text similar to the following:

```
Killing container with id docker://2ac946...:pod "kuard_default(9ee84...)"
container "kuard" is unhealthy, it will be killed and re-created.
```

Readiness Probe

Of course, liveness isn't the only kind of health check we want to perform. Kubernetes makes a distinction between *liveness* and *readiness*. Liveness determines if an application is running properly. Containers that fail liveness checks are restarted. Readiness describes when a container is ready to serve user requests. Containers that fail readiness checks are removed from service load balancers. Readiness probes are configured similarly to liveness probes. We explore Kubernetes services in detail in Chapter 7.

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

Types of Health Checks

In addition to HTTP checks, Kubernetes also supports `tcpSocket` health checks that open a TCP socket; if the connection is successful, the probe succeeds. This style of probe is useful for non-HTTP applications; for example, databases or other non–HTTP-based APIs.

Finally, Kubernetes allows `exec` probes. These execute a script or program in the context of the container. Following typical convention, if this script returns a zero exit code, the probe succeeds; otherwise, it fails. `exec` scripts are often useful for custom application validation logic that doesn't fit neatly into an HTTP call.

Resource Management

Most people move into containers and orchestrators like Kubernetes because of the radical improvements in image packaging and reliable deployment they provide. In addition to application-oriented primitives that simplify distributed system development, equally important is the ability to increase the overall utilization of the compute nodes that make up the cluster. The basic cost of operating a machine, either virtual or physical, is basically constant regardless of whether it is idle or fully loaded. Consequently, ensuring that these machines are maximally active increases the efficiency of every dollar spent on infrastructure.

Generally speaking, we measure this efficiency with the *utilization* metric. Utilization is defined as the amount of a resource actively being used divided by the amount of a resource that has been purchased. For example, if you purchase a one-core machine, and your application uses one-tenth of a core, then your utilization is 10%.

With scheduling systems like Kubernetes managing resource packing, you can drive your utilization to greater than 50%.

To achieve this, you have to tell Kubernetes about the resources your application requires, so that Kubernetes can find the optimal packing of containers onto purchased machines.

Kubernetes allows users to specify two different resource metrics. Resource *requests* specify the minimum amount of a resource required to run the application. Resource *limits* specify the maximum amount of a resource that an application can consume. Both of these resource definitions are described in greater detail in the following sections.

Resource Requests: Minimum Required Resources

With Kubernetes, a Pod requests the resources required to run its containers. Kubernetes guarantees that these resources are available to the Pod. The most commonly requested resources are CPU and memory, but Kubernetes has support for other resource types as well, such as GPUs and more.

For example, to request that the `kuard` container lands on a machine with half a CPU free and gets 128 MB of memory allocated to it, we define the Pod as shown in Example 5-3.

Example 5-3. kuard-pod-resreq.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Resources are requested per container, not per Pod. The total resources requested by the Pod is the sum of all resources requested by all containers in the Pod. The reason for this is that in many cases the different containers have very different CPU requirements. For example, in the web server and data synchronizer Pod, the web server is user-facing and likely needs a great deal of CPU, while the data synchronizer can make do with very little.

Request limit details

Requests are used when scheduling Pods to nodes. The Kubernetes scheduler will ensure that the sum of all requests of all Pods on a node does not exceed the capacity of the node. Therefore, a Pod is guaranteed to have at least the requested resources when running on the node. Importantly, "request" specifies a minimum. It does not specify a maximum cap on the resources a Pod may use. To explore what this means, let's look at an example.

Imagine that we have container whose code attempts to use all available CPU cores. Suppose that we create a Pod with this container that requests 0.5 CPU. Kubernetes schedules this Pod onto a machine with a total of 2 CPU cores.

As long as it is the only Pod on the machine, it will consume all 2.0 of the available cores, despite only requesting 0.5 CPU.

If a second Pod with the same container and the same request of 0.5 CPU lands on the machine, then each Pod will receive 1.0 cores.

If a third identical Pod is scheduled, each Pod will receive 0.66 cores. Finally, if a fourth identical Pod is scheduled, each Pod will receive the 0.5 core it requested, and the node will be at capacity.

CPU requests are implemented using the `cpu-shares` functionality in the Linux kernel.

Memory requests are handled similarly to CPU, but there is an important difference. If a container is over its memory request, the OS can't just remove memory from the process, because it's been allocated. Consequently, when the system runs out of memory, the `kubelet` terminates containers whose memory usage is greater than their requested memory. These containers are automatically restarted, but with less available memory on the machine for the container to consume.

Since resource requests guarantee resource availability to a Pod, they are critical to ensuring that containers have sufficient resources in high-load situations.

Capping Resource Usage with Limits

In addition to setting the resources required by a Pod, which establishes the minimum resources available to the Pod, you can also set a maximum on a Pod's resource usage via resource *limits*.

In our previous example we created a `kuard` Pod that requested a minimum of 0.5 of a core and 128 MB of memory. In the Pod manifest in Example 5-4, we extend this configuration to add a limit of 1.0 CPU and 256 MB of memory.

Example 5-4. kuard-pod-reslim.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
        limits:
          cpu: "1000m"
          memory: "256Mi"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

When you establish limits on a container, the kernel is configured to ensure that consumption cannot exceed these limits. A container with a CPU limit of 0.5 cores will only ever get 0.5 cores, even if the CPU is otherwise idle. A container with a memory limit of 256 MB will not be allowed additional memory (e.g., `malloc` will fail) if its memory usage exceeds 256 MB.

Persisting Data with Volumes

When a Pod is deleted or a container restarts, any and all data in the container's filesystem is also deleted. This is often a good thing, since you don't want to leave around cruft that happened to be written by your stateless web application. In other cases, having access to persistent disk is an important part of a healthy application. Kubernetes models such persistent storage.

Using Volumes with Pods

To add a volume to a Pod manifest, there are two new stanzas to add to our configuration. The first is a new `spec.volumes` section. This array defines all of the volumes that may be accessed by containers in the Pod manifest. It's important to note that not all containers are required to mount all volumes defined in the Pod. The second addition is the `volumeMounts` array in the container definition. This array defines the volumes that are mounted into a particular container, and the path where each volume should be mounted. Note that two different containers in a Pod can mount the same volume at different mount paths.

The manifest in Example 5-5 defines a single new volume named `kuard-data`, which the `kuard` container mounts to the `/data` path.

Example 5-5. kuard-pod-vol.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  volumes:
    - name: "kuard-data"
      hostPath:
        path: "/var/lib/kuard"
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      volumeMounts:
        - mountPath: "/data"
          name: "kuard-data"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Different Ways of Using Volumes with Pods

There are a variety of ways you can use data in your application. The following are a few, and the recommended patterns for Kubernetes.

Communication/synchronization

In the first example of a Pod, we saw how two containers used a shared volume to serve a site while keeping it synchronized to a remote Git location. To achieve this, the Pod uses an `emptyDir` volume. Such a volume is scoped to the Pod's lifespan, but it can be shared between two containers, forming the basis for communication between our Git sync and web serving containers.

Cache

An application may use a volume that is valuable for performance, but not required for correct operation of the application. For example, perhaps the application keeps prerendered thumbnails of larger images. Of course, they can be reconstructed from the original images, but that makes serving the thumbnails more expensive. You want such a cache to survive a container restart due to a health check failure, and thus `emptyDir` works well for the cache use case as well.

Persistent data

Sometimes you will use a volume for truly persistent data—data that is independent of the lifespan of a particular Pod, and should move between nodes in the cluster if a node fails or a Pod moves to a different machine for some reason. To achieve this, Kubernetes supports a wide variety of remote network storage volumes, including widely supported protocols like NFS or iSCSI as well as cloud provider network storage like Amazon's Elastic Block Store, Azure's Files and Disk Storage, as well as Google's Persistent Disk.

Mounting the host filesystem

Other applications don't actually need a persistent volume, but they do need some access to the underlying host filesystem. For example, they may need access to the */dev* filesystem in order to perform raw block-level access to a device on the system. For these cases, Kubernetes supports the `hostDir` volume, which can mount arbitrary locations on the worker node into the container.

The previous example uses the `hostDir` volume type. The volume created is */var/lib/kuard* on the host.

Persisting Data Using Remote Disks

Oftentimes, you want the data a Pod is using to stay with the Pod, even if it is restarted on a different host machine.

To achieve this, you can mount a remote network storage volume into your Pod. When using network-based storage, Kubernetes automatically mounts and unmounts the appropriate storage whenever a Pod using that volume is scheduled onto a particular machine.

There are numerous methods for mounting volumes over the network. Kubernetes includes support for standard protocols such as NFS and iSCSI as well as cloud provider–based storage APIs for the major cloud providers (both public and private). In many cases, the cloud providers will also create the disk for you if it doesn't already exist.

Here is an example of using an NFS server:

```
...
# Rest of pod definition above here
volumes:
    - name: "kuard-data"
      nfs:
        server: my.nfs.server.local
        path: "/exports"
```

Putting It All Together

Many applications are stateful, and as such we must preserve any data and ensure access to the underlying storage volume regardless of what machine the application runs on. As we saw earlier, this can be achieved using a persistent volume backed by network-attached storage. We also want to ensure a healthy instance of the application is running at all times, which means we want to make sure the container running `kuard` is ready before we expose it to clients.

Through a combination of persistent volumes, readiness and liveness probes, and resource restrictions Kubernetes provides everything needed to run stateful applications reliably. Example 5-6 pulls this all together into one manifest.

Example 5-6. kuard-pod-full.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  volumes:
    - name: "kuard-data"
      nfs:
        server: my.nfs.server.local
        path: "/exports"
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
        limits:
          cpu: "1000m"
          memory: "256Mi"
      volumeMounts:
        - mountPath: "/data"
```

```
          name: "kuard-data"
      livenessProbe:
        httpGet:
          path: /healthy
          port: 8080
        initialDelaySeconds: 5
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 3
      readinessProbe:
        httpGet:
          path: /ready
          port: 8080
        initialDelaySeconds: 30
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 3
```

Persistent volumes are a deep topic that has many different details: in particular, the manner in which persistent volumes, persistent volume claims, and dynamic volume provisioning work together. There is a more in-depth examination of the subject in Chapter 13.

Summary

Pods represent the atomic unit of work in a Kubernetes cluster. Pods are comprised of one or more containers working together symbiotically. To create a Pod, you write a Pod manifest and submit it to the Kubernetes API server by using the command-line tool or (less frequently) by making HTTP and JSON calls to the server directly.

Once you've submitted the manifest to the API server, the Kubernetes scheduler finds a machine where the Pod can fit and schedules the Pod to that machine. Once scheduled, the `kubelet` daemon on that machine is responsible for creating the containers that correspond to the Pod, as well as performing any health checks defined in the Pod manifested.

Once a Pod is scheduled to a node, no rescheduling occurs if that node fails. Additionally, to create multiple replicas of the same Pod you have to create and name them manually. In a later chapter we introduce the `ReplicaSet` object and show how you can automate the creation of multiple identical Pods and ensure that they are recreated in the event of a node machine failure.

CHAPTER 6

Labels and Annotations

Kubernetes was made to grow with you as your application scales both in size and complexity. With this in mind, labels and annotations were added as foundational concepts. Labels and annotations let you work in sets of things that map to how *you* think about your application. You can organize, mark, and cross-index all of your resources to represent the groups that make the most sense for your application.

Labels are key/value pairs that can be attached to Kubernetes objects such as Pods and ReplicaSets. They can be arbitrary, and are useful for attaching identifying information to Kubernetes objects. Labels provide the foundation for grouping objects.

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

Labels

Labels provide identifying metadata for objects. These are fundamental qualities of the object that will be used for grouping, viewing, and operating.

The motivations for labels grew out of Google's experience in running large and complex applications. There were a couple of lessons that emerged from this experience. See the great *Site Reliability Engineering* by Betsy Beyer et al. (O'Reilly) for some deeper background on how Google approaches production systems.

The first lesson is that production abhors a singleton. When deploying software, users will often start with a single instance. However, as the application matures, these singletons often multiply and become sets of objects. With this in mind, Kubernetes uses labels to deal with *sets* of objects instead of single instances.

The second lesson is that any hierarchy imposed by the system will fall short for many users. In addition, user grouping and hierarchy are subject to change over time. For instance, a user may start out with the idea that all apps are made up of many services. However, over time, a service may be shared across multiple apps. Kubernetes labels are flexible enough to adapt to these situations and more.

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

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

Table 6-1 shows valid label keys and values.

Table 6-1. Label examples

Key	Value
`acme.com/app-version`	`1.0.0`
`appVersion`	`1.0.0`
`app.version`	`1.0.0`
`kubernetes.io/cluster-service`	`true`

Applying Labels

Here we create a few deployments (a way to create an array of Pods) with some interesting labels. We'll take two apps (called `alpaca` and `bandicoot`) and have two environments for each. We will also have two different versions.

1. First, create the alpaca-prod deployment and set the ver, app, and env labels:

   ```
   $ kubectl run alpaca-prod \
     --image=gcr.io/kuar-demo/kuard-amd64:1 \
     --replicas=2 \
     --labels="ver=1,app=alpaca,env=prod"
   ```

2. Next, create the alpaca-test deployment and set the ver, app, and env labels with the appropriate values:

   ```
   $ kubectl run alpaca-test \
     --image=gcr.io/kuar-demo/kuard-amd64:2 \
     --replicas=1 \
     --labels="ver=2,app=alpaca,env=test"
   ```

3. Finally, create two deployments for bandicoot. Here we name the environments prod and staging:

   ```
   $ kubectl run bandicoot-prod \
     --image=gcr.io/kuar-demo/kuard-amd64:2 \
     --replicas=2 \
     --labels="ver=2,app=bandicoot,env=prod"
   $ kubectl run bandicoot-staging \
     --image=gcr.io/kuar-demo/kuard-amd64:2 \
     --replicas=1 \
     --labels="ver=2,app=bandicoot,env=staging"
   ```

At this point you should have four deployments—alpaca-prod, alpaca-staging, bandicoot-prod, and bandicoot-staging:

```
$ kubectl get deployments --show-labels

NAME                ... LABELS
alpaca-prod         ... app=alpaca,env=prod,ver=1
alpaca-test         ... app=alpaca,env=test,ver=2
bandicoot-prod      ... app=bandicoot,env=prod,ver=2
bandicoot-staging   ... app=bandicoot,env=staging,ver=2
```

We can visualize this as a Venn diagram based on the labels (Figure 6-1).

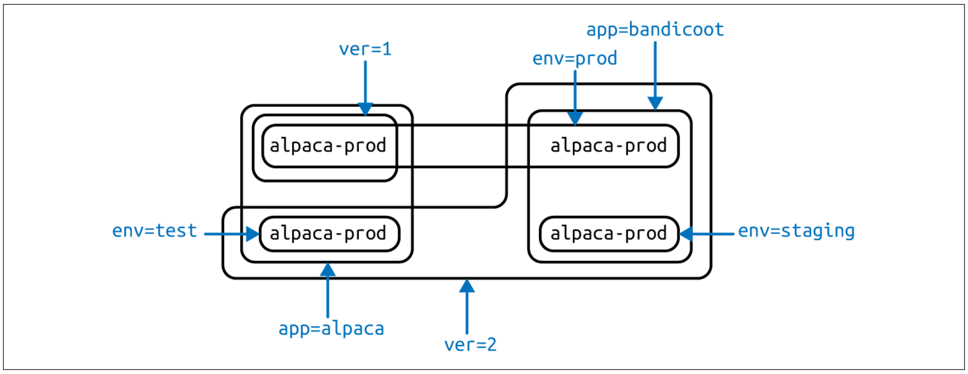


Figure 6-1. Visualization of labels applied to our deployments

Modifying Labels

Labels can also be applied (or updated) on objects after they are created.

```
$ kubectl label deployments alpaca-test "canary=true"
```

There is a caveat to be aware of here. In this example, the `kubectl label` command will only change the label on the deployment itself; it won't affect the objects (ReplicaSets and Pods) the deployment creates. To change those, you'll need to change the template embedded in the deployment (see Chapter 6).

You can also use the `-L` option to `kubectl get` to show a label value as a column:

```
$ kubectl get deployments -L canary

NAME                DESIRED   CURRENT   ... CANARY
alpaca-prod         2         2         ... <none>
alpaca-test         1         1         ... true
bandicoot-prod      2         2         ... <none>
bandicoot-staging   1         1         ... <none>
```

You can remove a label by applying a dash suffix:

```
$ kubectl label deployments alpaca-test "canary-"
```

Label Selectors

Label selectors are used to filter Kubernetes objects based on a set of labels. Selectors use a simple Boolean language. They are used both by end users (via tools like `kubectl`) and by different types of objects (such as how ReplicaSet relates to its Pods).

Each deployment (via a ReplicaSet) creates a set of Pods using the labels specified in the template embedded in the deployment. This is configured by the `kubectl run` command.

Running the `kubectl get pods` command should return all the Pods currently running in the cluster. We should have a total of six `kuard` Pods across our three environments:

```
$ kubectl get pods --show-labels

NAME                              ... LABELS
alpaca-prod-3408831585-4nzfb      ... app=alpaca,env=prod,ver=1,...
alpaca-prod-3408831585-kga0a      ... app=alpaca,env=prod,ver=1,...
alpaca-test-1004512375-3r1m5      ... app=alpaca,env=test,ver=2,...
bandicoot-prod-373860099-0t1gp    ... app=bandicoot,env=prod,ver=2,...
bandicoot-prod-373860099-k2wcf    ... app=bandicoot,env=prod,ver=2,...
bandicoot-staging-1839769971-3ndv ... app=bandicoot,env=staging,ver=2,...
```

You may see a new label that we haven't seen yet: `pod-template-hash`. This label is applied by the deployment so it can keep track of which pods were generated from which template versions. This allows the deployment to manage updates in a clean way, as will be covered in depth in Chapter 12.

If we only wanted to list pods that had the `ver` label set to `2` we could use the `--selector` flag:

```
$ kubectl get pods --selector="ver=2"

NAME                               READY     STATUS    RESTARTS   AGE
alpaca-test-1004512375-3r1m5       1/1       Running   0          3m
bandicoot-prod-373860099-0t1gp     1/1       Running   0          3m
bandicoot-prod-373860099-k2wcf     1/1       Running   0          3m
bandicoot-staging-1839769971-3ndv5 1/1       Running   0          3m
```

If we specify two selectors separated by a comma, only the objects that satisfy both will be returned. This is a logical AND operation:

```
$ kubectl get pods --selector="app=bandicoot,ver=2"

NAME                               READY     STATUS    RESTARTS   AGE
bandicoot-prod-373860099-0t1gp     1/1       Running   0          4m
bandicoot-prod-373860099-k2wcf     1/1       Running   0          4m
bandicoot-staging-1839769971-3ndv5 1/1       Running   0          4m
```

We can also ask if a label is one of a set of values. Here we ask for all pods where the `app` label is set to `alpaca` or `bandicoot` (which will be all six pods):

```
$ kubectl get pods --selector="app in (alpaca,bandicoot)"
```

```
NAME                                  READY     STATUS    RESTARTS   AGE
alpaca-prod-3408831585-4nzfb          1/1       Running   0          6m
alpaca-prod-3408831585-kga0a          1/1       Running   0          6m
alpaca-test-1004512375-3r1m5          1/1       Running   0          6m
bandicoot-prod-373860099-0t1gp        1/1       Running   0          6m
bandicoot-prod-373860099-k2wcf        1/1       Running   0          6m
bandicoot-staging-1839769971-3ndv5    1/1       Running   0          6m
```

Finally, we can ask if a label is set at all. Here we are asking for all of the deployments with the `canary` label set to anything:

```
$ kubectl get deployments --selector="canary"

NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
alpaca-test   1         1         1            1           7m
```

There are also "negative" versions of each of these, as shown in Table 6-2.

Table 6-2. Selector operators

Operator	Description
`key=value`	`key` is set to `value`
`key!=value`	`key` is not set to `value`
`key in (value1, value2)`	`key` is one of `value1` or `value2`
`key notin (value1, value2)`	`key` is not one of `value1` or `value2`
`key`	`key` is set
`!key`	`key` is not set

Label Selectors in API Objects

When a Kubernetes object refers to a set of other Kubernetes objects, a label selector is used. Instead of a simple string as described in the previous section, a parsed structure is used.

For historical reasons (Kubernetes doesn't break API compatibility!), there are two forms. Most objects support a newer, more powerful set of selector operators.

A selector of `app=alpaca,ver in (1, 2)` would be converted to this:

```
selector:
  matchLabels:
    app: alpaca
  matchExpressions:
    - {key: ver, operator: In, values: [1, 2]} ❶
```

❶ Compact YAML syntax. This is an item in a list (`matchExpressions`) that is a map with three entries. The last entry (`values`) has a value that is a list with two items.

All of the terms are evaluated as a logical AND. The only way to represent the `!=` operator is to convert it to a `NotIn` expression with a single value.

The older form of specifying selectors (used in `ReplicationControllers` and services) only supports the `=` operator. This is a simple set of key/value pairs that must all match a target object to be selected.

The selector `app=alpaca,ver=1` would be represented like this:

```
selector:
  app: alpaca
  ver: 1
```

Annotations

Annotations provide a place to store additional metadata for Kubernetes objects with the sole purpose of assisting tools and libraries. They are a way for other programs driving Kubernetes via an API to store some opaque data with an object. Annotations can be used for the tool itself or to pass configuration information between external systems.

While labels are used to identify and group objects, annotations are used to provide extra information about where an object came from, how to use it, or policy around that object. There is overlap, and it is a matter of taste as to when to use an annotation or a label. When in doubt, add information to an object as an annotation and promote it to a label if you find yourself wanting to use it in a selector.

Annotations are used to:

- Keep track of a "reason" for the latest update to an object.
- Communicate a specialized scheduling policy to a specialized scheduler.
- Extend data about the last tool to update the resource and how it was updated (used for detecting changes by other tools and doing a smart merge).
- Build, release, or image information that isn't appropriate for labels (may include a Git hash, timestamp, PR number, etc.).
- Enable the `Deployment` object (Chapter 12) to keep track of ReplicaSets that it is managing for rollouts.
- Provide extra data to enhance the visual quality or usability of a UI. For example, objects could include a link to an icon (or a base64-encoded version of an icon).
- Prototype alpha functionality in Kubernetes (instead of creating a first-class API field, the parameters for that functionality are instead encoded in an annotation).

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

Users should avoid using the Kubernetes API server as a general-purpose database. Annotations are good for small bits of data that are highly associated with a specific resource. If you want to store data in Kubernetes but you don't have an obvious object to associate it with, consider storing that data in some other, more appropriate database.

Defining Annotations

Annotation keys use the same format as label keys. However, because they are often used to communicate information between tools, the "namespace" part of the key is more important. Example keys include `deployment.kubernetes.io/revision` or `kubernetes.io/change-cause`.

The value component of an annotation is a free-form string field. While this allows maximum flexibility as users can store arbitrary data, because this is arbitrary text, there is no validation of any format. For example, it is not uncommon for a JSON document to be encoded as a string and stored in an annotation. It is important to note that the Kubernetes server has no knowledge of the required format of annotation values. If annotations are used to pass or store data, there is no guarantee the data is valid. This can make tracking down errors more difficult.

Annotations are defined in the common `metadata` section in every Kubernetes object:

```
...
metadata:
  annotations:
    example.com/icon-url: "https://example.com/icon.png"
...
```

Annotations are very convenient and provide powerful loose coupling. However, they should be used judiciously to avoid an untyped mess of data.

Cleanup

It is easy to clean up all of the deployments that we started in this chapter:

```
$ kubectl delete deployments --all
```

If you want to be more selective you can use the `--selector` flag to choose which deployments to delete.

Summary

Labels are used to identify and optionally group objects in a Kubernetes cluster. Labels are also used in selector queries to provide flexible runtime grouping of objects such as pods.

Annotations provide object-scoped key/value storage of metadata that can be used by automation tooling and client libraries. Annotations can also be used to hold configuration data for external tools such as third-party schedulers and monitoring tools.

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

CHAPTER 7
Service Discovery

Kubernetes is a very dynamic system. The system is involved in placing Pods on nodes, making sure they are up and running, and rescheduling them as needed. There are ways to automatically change the number of pods based on load (such as horizontal pod autoscaling, discussed in Chapter 8). The API-driven nature of the system encourages others to create higher and higher levels of automation.

While the dynamic nature of Kubernetes makes it easy to run a lot of things, it creates problems when it comes to *finding* those things. Most of the traditional network infrastructure wasn't built for the level of dynamism that Kubernetes presents.

What Is Service Discovery?

The general name for this class of problems and solutions is *service discovery*. Service discovery tools help solve the problem of finding which processes are listening at which addresses for which services. A good service discovery system will enable users to resolve this information quickly and reliably. A good system is also low-latency; clients are updated soon after the information associated with a service changes. Finally, a good service discovery system can store a richer definition of what that service is. For example, perhaps there are multiple ports associated with the service.

The Domain Name System (DNS) is the traditional system of service discovery on the internet. DNS is designed for relatively stable name resolution with wide and efficient caching. It is a great system for the internet but falls short in the dynamic world of Kubernetes.

Unfortunately, many systems (for example, Java, by default) look up a name in DNS directly and never re-resolve. This can lead to clients caching stale mappings and talking to the wrong IP. Even with short TTLs and well-behaved clients, there is a natural delay between when a name resolution changes and the client notices. There are natural limits to the amount and type of information that can be returned in a typical DNS query, too. Things start to break past 20–30 A records for a single name. SRV records solve some problems but are often very hard to use. Finally, the way that clients handle multiple IPs in a DNS record is usually to take the first IP address and rely on the DNS server to randomize or round-robin the order of records. This is no substitute for more purpose-built load balancing.

The Service Object

Real service discovery in Kubernetes starts with a `Service` object.

A `Service object` is a way to create a named label selector. As we will see, the `Service` object does some other nice things for us too.

Just as the `kubectl run` command is an easy way to create a Kubernetes deployment, we can use `kubectl expose` to create a service. Let's create some deployments and services so we can see how they work:

```
$ kubectl run alpaca-prod \
  --image=gcr.io/kuar-demo/kuard-amd64:1 \
  --replicas=3 \
  --port=8080 \
  --labels="ver=1,app=alpaca,env=prod"
$ kubectl expose deployment alpaca-prod
$ kubectl run bandicoot-prod \
  --image=gcr.io/kuar-demo/kuard-amd64:2 \
  --replicas=2 \
  --port=8080 \
  --labels="ver=2,app=bandicoot,env=prod"
$ kubectl expose deployment bandicoot-prod
$ kubectl get services -o wide

NAME             CLUSTER-IP    ... PORT(S)  ... SELECTOR
alpaca-prod      10.115.245.13 ... 8080/TCP ... app=alpaca,env=prod,ver=1
bandicoot-prod   10.115.242.3  ... 8080/TCP ... app=bandicoot,env=prod,ver=2
kubernetes       10.115.240.1  ... 443/TCP  ... <none>
```

After running these commands, we have three services. The ones we just created are `alpaca-prod` and `bandicoot-prod`. The `kubernetes` service is automatically created for you so that you can find and talk to the Kubernetes API from within the app.

If we look at the `SELECTOR` column, we see that the `alpaca-prod` service simply gives a name to a selector and specifies which ports to talk to for that service. The `kubectl expose` command will conveniently pull both the label selector and the relevant ports (8080, in this case) from the deployment definition.

Furthermore, that service is assigned a new type of virtual IP called a *cluster IP*. This is a special IP address the system will load-balance across all of the pods that are identified by the selector.

To interact with services, we are going to port-forward to one of the `alpaca` pods. Start and leave this command running in a terminal window. You can see the port forward working by accessing the `alpaca` pod at *http://localhost:48858*:

```
$ ALPACA_POD=$(kubectl get pods -l app=alpaca \
    -o jsonpath='{.items[0].metadata.name}')
$ kubectl port-forward $ALPACA_POD 48858:8080
```

Service DNS

Because the cluster IP is virtual it is stable and it is appropriate to give it a DNS address. All of the issues around clients caching DNS results no longer apply. Within a namespace, it is as easy as just using the service name to connect to one of the pods identified by a service.

Kubernetes provides a DNS service exposed to Pods running in the cluster. This Kubernetes DNS service was installed as a system component when the cluster was first created. The DNS service is, itself, managed by Kubernetes and is a great example of Kubernetes building on Kubernetes. The Kubernetes DNS service provides DNS names for cluster IPs.

You can try this out by expanding the "DNS Resolver" section on the `kuard` server status page. Query the A record for `alpaca-prod`. The output should look something like this:

```
;; opcode: QUERY, status: NOERROR, id: 12071
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;alpaca-prod.default.svc.cluster.local. IN       A

;; ANSWER SECTION:
alpaca-prod.default.svc.cluster.local.  30      IN      A       10.115.245.13
```

The full DNS name here is `alpaca-prod.default.svc.cluster.local.`. Let's break this down:

`alpaca-prod`
: The name of the service in question.

`default`
: The namespace that this service is in.

`svc`*:: Recognizing that this is a service. This allows Kubernetes to expose other types of things as DNS in the future.* `cluster.local.`

> The base domain name for the cluster. This is the default and what you will see for most clusters. Administrators may change this to allow unique DNS names across multiple clusters.

When referring to a service in your own namespace you can just use the service name (`alpaca-prod`). You can also refer to a service in another namespace with `alpaca-prod.default`. And, of course, you can use the fully qualified service name (`alpaca-prod.default.svc.cluster.local.`). Try each of these out in the "DNS Resolver" section of `kuard`.

Readiness Checks

Oftentimes when an application first starts up it isn't ready to handle requests. There is usually some amount of initialization that can take anywhere from under a second to several minutes. One nice thing the `Service` object does is track which of your pods are ready via a readiness check. Let's modify our deployment to add a readiness check:

```
$ kubectl edit deployment/alpaca-prod
```

This command will fetch the current version of the `alpaca-prod` deployment and bring it up in an editor. After you save and quit your editor, it'll then write the object back to Kubernetes. This is a quick way to edit an object without saving it to a YAML file.

Add the following section:

```
spec:
  ...
  template:
    ...
    spec:
      containers:
        ...
        name: alpaca-prod
        readinessProbe:
          httpGet:
            path: /ready
            port: 8080
          periodSeconds: 2
          initialDelaySeconds: 0
          failureThreshold: 3
          successThreshold: 1
```

This sets up the pods this deployment will create so that they will be checked for readiness via an HTTP `GET` to `/ready` on port 8080. This check is done every 2 seconds starting as soon as the pod comes up. If three successive checks fail, then the pod will be considered not ready. However, if only one check succeeds, then the pod will again be considered ready.

Only ready pods are sent traffic.

Updating the deployment definition like this will delete and recreate the `alpaca` pods. As such, we need to restart our `port-forward` command from earlier:

```
$ ALPACA_POD=$(kubectl get pods -l app=alpaca \
    -o jsonpath='{.items[0].metadata.name}')
$ kubectl port-forward $ALPACA_POD 48858:8080
```

Open your browser to *http://localhost:48858* and you should see the debug page for that instance of `kuard`. Expand the "Readiness Check" section. You should see this page update every time there is a new readiness check from the system, which should happen every 2 seconds.

In another terminal window, start a `watch` command on the endpoints for the `alpaca-prod` service. Endpoints are a lower-level way of finding what a service is sending traffic to and are covered later in this chapter. The `--watch` option here causes the `kubectl` command to hang around and output any updates. This is an easy way to see how a Kubernetes object changes over time:

```
$ kubectl get endpoints alpaca-prod --watch
```

Now go back to your browser and hit the "Fail" link for the readiness check. You should see that the server is not returning 500s. After three of these this server is removed from the list of endpoints for the service. Hit the "Succeed" link and notice that after a single readiness check the endpoint is added back.

This readiness check is a way for an overloaded or sick server to signal to the system that it doesn't want to receive traffic anymore. This is a great way to implement graceful shutdown. The server can signal that it no longer wants traffic, wait until existing connections are closed, and then cleanly exit.

Press Control-C to exit out of both the `port-forward` and `watch` commands in your terminals.

Looking Beyond the Cluster

So far, everything we've covered in this chapter has been about exposing services inside of a cluster. Oftentimes the IPs for pods are only reachable from within the cluster. At some point, we have to allow new traffic in!

The most portable way to do this is to use a feature called `NodePorts`, which enhance a service even further. In addition to a cluster IP, the system picks a port (or the user can specify one), and every node in the cluster then forwards traffic to that port to the service.

With this feature, if you can reach any node in the cluster you can contact a service. You use the NodePort without knowing where any of the Pods for that service are running. This can be integrated with hardware or software load balancers to expose the service further.

Try this out by modifying the alpaca-prod service:

```
$ kubectl edit service alpaca-prod
```

Change the spec.type field to NodePort. You can also do this when creating the service via kubectl expose by specifying --type=NodePort. The system will assign a new NodePort:

```
$ kubectl describe service alpaca-prod

Name:                   alpaca-prod
Namespace:              default
Labels:                 app=alpaca
                        env=prod
                        ver=1
Annotations:            <none>
Selector:               app=alpaca,env=prod,ver=1
Type:                   NodePort
IP:                     10.115.245.13
Port:                   <unset> 8080/TCP
NodePort:               <unset> 32711/TCP
Endpoints:              10.112.1.66:8080,10.112.2.104:8080,10.112.2.105:8080
Session Affinity:       None
No events.
```

Here we see that the system assigned port 32711 to this service. Now we can hit any of our cluster nodes on that port to access the service. If you are sitting on the same network, you can access it directly. If your cluster is in the cloud someplace, you can use SSH tunneling with something like this:

```
$ ssh <node> -L 8080:localhost:32711
```

Now if you open your browser to *http://localhost:8080* you will be connected to that service. Each request that you send to the service will be randomly directed to one of the Pods that implement the service. Reload the page a few times and you will see that you are randomly assigned to different pods.

When you are done, exit out of the SSH session.

Cloud Integration

Finally, if you have support from the cloud that you are running on (and your cluster is configured to take advantage of it) you can use the LoadBalancer type. This builds on NodePorts by additionally configuring the cloud to create a new load balancer and direct it at nodes in your cluster.

Edit the alpaca-prod service again (kubectl edit service alpaca-prod) and change spec.type to LoadBalancer.

If you do a kubectl get services right away you'll see that the EXTERNAL-IP column for alpaca-prod now says <pending>. Wait a bit and you should see a public address assigned by your cloud. You can look in the console for your cloud account and see the configuration work that Kubernetes did for you:

```
$ kubectl describe service alpaca-prod

Name:                   alpaca-prod
Namespace:              default
Labels:                 app=alpaca
                        env=prod
                        ver=1
Selector:               app=alpaca,env=prod,ver=1
Type:                   LoadBalancer
IP:                     10.115.245.13
LoadBalancer Ingress:   104.196.248.204
Port:                   <unset> 8080/TCP
NodePort:               <unset> 32711/TCP
Endpoints:              10.112.1.66:8080,10.112.2.104:8080,10.112.2.105:8080
Session Affinity:       None
Events:
  FirstSeen ... Reason                Message
  --------- ... ------                -------
  3m        ... Type                  NodePort -> LoadBalancer
  3m        ... CreatingLoadBalancer  Creating load balancer
  2m        ... CreatedLoadBalancer   Created load balancer
```

Here we see that we have an address of 104.196.248.204 now assigned to the alpaca-prod service. Open up your browser and try!

This example is from a cluster launched and managed on the Google Cloud Platform via GKE. However, the way a LoadBalancer is configured is specific to a cloud. In addition, some clouds have DNS-based load balancers (e.g., AWS ELB). In this case you'll see a hostname here instead of an IP. Also, depending on the cloud provider, it may still take a little while for the load balancer to be fully operational.

Advanced Details

Kubernetes is built to be an extensible system. As such, there are layers that allow for more advanced integrations. Understanding the details of how a sophisticated concept like services is implemented may help you troubleshoot or create more advanced integrations. This section goes a bit below the surface.

Endpoints

Some applications (and the system itself) want to be able to use services without using a cluster IP. This is done with another type of object called `Endpoints`. For every `Service` object, Kubernetes creates a buddy `Endpoints` object that contains the IP addresses for that service:

```
$ kubectl describe endpoints alpaca-prod

Name:           alpaca-prod
Namespace:      default
Labels:         app=alpaca
                env=prod
                ver=1
Subsets:
  Addresses:            10.112.1.54,10.112.2.84,10.112.2.85
  NotReadyAddresses:    <none>
  Ports:
    Name        Port    Protocol
    ----        ----    --------
    <unset>     8080    TCP

No events.
```

To use a service, an advanced application can talk to the Kubernetes API directly to look up endpoints and call them. The Kubernetes API even has the capability to "watch" objects and be notified as soon as they change. In this way a client can react immediately as soon as the IPs associated with a service change.

Let's demonstrate this. In a terminal window, start the following command and leave it running:

```
$ kubectl get endpoints alpaca-prod --watch
```

It will output the current state of the endpoint and then "hang":

```
NAME          ENDPOINTS                                            AGE
alpaca-prod   10.112.1.54:8080,10.112.2.84:8080,10.112.2.85:8080   1m
```

Now open up *another* terminal window and delete and recreate the deployment backing alpaca-prod:

```
$ kubectl delete deployment alpaca-prod
$ kubectl run alpaca-prod \
  --image=gcr.io/kuar-demo/kuard-amd64:1 \
  --replicas=3 \
  --port=8080 \
  --labels="ver=1,app=alpaca,env=prod"
```

If you look back at the output from the watched endpoint, you will see that as you deleted and re-created these pods, the output of the command reflected the most up-to-date set of IP addresses associated with the service. Your output will look something like this:

```
NAME          ENDPOINTS                                            AGE
alpaca-prod   10.112.1.54:8080,10.112.2.84:8080,10.112.2.85:8080   1m
alpaca-prod   10.112.1.54:8080,10.112.2.84:8080   1m
alpaca-prod   <none>    1m
alpaca-prod   10.112.2.90:8080   1m
alpaca-prod   10.112.1.57:8080,10.112.2.90:8080   1m
alpaca-prod   10.112.0.28:8080,10.112.1.57:8080,10.112.2.90:8080   1m
```

The `Endpoints` object is great if you are writing new code that is built to run on Kubernetes from the start. But most projects aren't in this position! Most existing systems are built to work with regular old IP addresses that don't change that often.

Manual Service Discovery

Kubernetes services are built on top of label selectors over pods. That means that you can use the Kubernetes API to do rudimentary service discovery without using a `Service` object at all! Let's demonstrate.

With `kubectl` (and via the API) we can easily see what IPs are assigned to each pod in our example deployments:

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

NAME                            ... IP          ... LABELS
alpaca-prod-12334-87f8h    ... 10.112.1.54 ... app=alpaca,env=prod,ver=1
alpaca-prod-12334-jssmh    ... 10.112.2.84 ... app=alpaca,env=prod,ver=1
alpaca-prod-12334-tjp56    ... 10.112.2.85 ... app=alpaca,env=prod,ver=1
bandicoot-prod-5678-sbxzl  ... 10.112.1.55 ... app=bandicoot,env=prod,ver=2
bandicoot-prod-5678-x0dh8  ... 10.112.2.86 ... app=bandicoot,env=prod,ver=2
```

This is great, but what if you have a ton of pods? You'll probably want to filter this based on the labels applied as part of the deployment. Let's do that for just the `alpaca` app:

```
$ kubectl get pods -o wide --selector=app=alpaca,env=prod

NAME                         ... IP          ...
alpaca-prod-3408831585-bpzdz ... 10.112.1.54 ...
alpaca-prod-3408831585-kncwt ... 10.112.2.84 ...
alpaca-prod-3408831585-l9fsq ... 10.112.2.85 ...
```

At this point we have the basics of service discovery! We can always use labels to identify the set of pods we are interested in, get all of the pods for those labels, and dig out the IP address. But keeping the correct set of labels to use in sync can be tricky. This is why the `Service` object was created.

kube-proxy and Cluster IPs

Cluster IPs are stable virtual IPs that load-balance traffic across all of the endpoints in a service. This magic is performed by a component running on every node in the cluster called the `kube-proxy` (Figure 7-1).

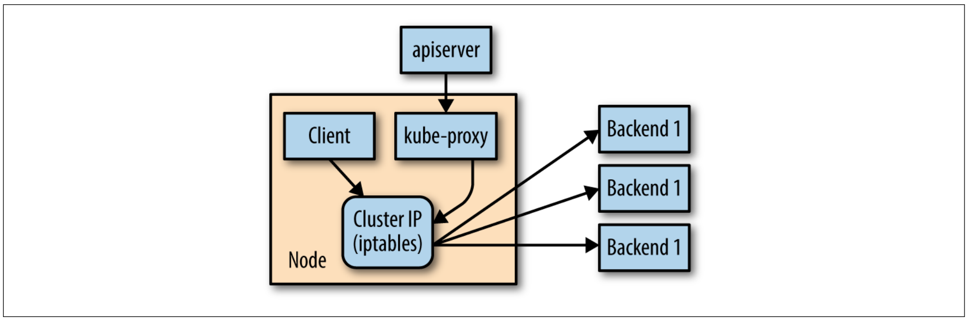


Figure 7-1. Configuring and using a cluster IP

In Figure 7-1, the `kube-proxy` watches for new services in the cluster via the API server. It then programs a set of `iptables` rules in the kernel of that host to rewrite the destination of packets so they are directed at one of the endpoints for that service. If the set of endpoints for a service changes (due to pods coming and going or due to a failed readiness check) the set of `iptables` rules is rewritten.

The cluster IP itself is usually assigned by the API server as the service is created. However, when creating the service, the user can specify a specific cluster IP. Once set, the cluster IP cannot be modified without deleting and recreating the `Service` object.

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

In addition, any explicit cluster IP requested must come from that range and not already be in use.

Cluster IP Environment Variables

While most users should be using the DNS services to find cluster IPs, there are some older mechanisms that may still be in use. One of these is injecting a set of environment variables into pods as they start up.

To see this in action, let's look at the console for the bandicoot instance of kuard. Enter the following commands in your terminal:

```
$ BANDICOOT_POD=$(kubectl get pods -l app=bandicoot \
    -o jsonpath='{.items[0].metadata.name}')
$ kubectl port-forward $BANDICOOT_POD 48858:8080
```

Now open your browser to *http://localhost:48858* to see the status page for this server. Expand the "Environment" section and note the set of environment variables for the `alpaca` service. The status page should show a table similar to Table 7-1.

Table 7-1. Service environment variables

Name	Value
ALPACA_PROD_PORT	tcp://10.115.245.13:8080
ALPACA_PROD_PORT_8080_TCP	tcp://10.115.245.13:8080
ALPACA_PROD_PORT_8080_TCP_ADDR	10.115.245.13
ALPACA_PROD_PORT_8080_TCP_PORT	8080
ALPACA_PROD_PORT_8080_TCP_PROTO	tcp
ALPACA_PROD_SERVICE_HOST	10.115.245.13
ALPACA_PROD_SERVICE_PORT	8080

The two main environment variables to use are `ALPACA_PROD_SERVICE_HOST` and `ALPACA_PROD_SERVICE_PORT`. The other environment variables are created to be compatible with (now deprecated) Docker link variables.

A problem with the environment variable approach is that it requires resources to be created in a specific order. The services must be created before the pods that reference them. This can introduce quite a bit of complexity when deploying a set of services that make up a larger application. In addition, using *just* environment variables seems strange to many users. For this reason, DNS is probably a better option.

Cleanup

Run the following commands to clean up all of the objects created in this chapter:

```
$ kubectl delete services,deployments -l app
```

Summary

Kubernetes is a dynamic system that challenges traditional methods of naming and connecting services over the network. The `Service` object provides a flexible and powerful way to expose services both within the cluster and beyond. With the techniques covered here you can connect services to each other and expose them outside the cluster.

While using the dynamic service discovery mechanisms in Kubernetes introduces some new concepts and may, at first, seem complex, understanding and adapting these techniques is key to unlocking the power of Kubernetes. Once your application can dynamically find services and react to the dynamic placement of those applications, you are free to stop worrying about where things are running and when they move. It is a critical piece of the puzzle to start to think about services in a logical way and let Kubernetes take care of the details of container placement.

CHAPTER 11

ConfigMaps and Secrets

It is a good practice to make container images as reusable as possible. The same image should be able to be used for development, staging, and production. It is even better if the same image is general purpose enough to be used across applications and services. Testing and versioning get riskier and more complicated if images need to be recreated for each new environment. But then how do we specialize the use of that image at runtime?

This is where ConfigMaps and secrets come into play. ConfigMaps are used to provide configuration information for workloads. This can either be fine-grained information (a short string) or a composite value in the form of a file. Secrets are similar to ConfigMaps but focused on making sensitive information available to the workload. They can be used for things like credentials or TLS certificates.

ConfigMaps

One way to think of a ConfigMap is as a Kubernetes object that defines a small filesystem. Another way is as a set of variables that can be used when defining the environment or command line for your containers. The key thing is that the ConfigMap is combined with the Pod right before it is run. This means that the container image and the pod definition itself can be reused across many apps by just changing the ConfigMap that is used.

Creating ConfigMaps

Let's jump right in and create a ConfigMap. Like many objects in Kubernetes, you can create these in an immediate, imperative way or you can create them from a manifest on disk. We'll start with the imperative method.

First, suppose we have a file on disk (called *my-config.txt*) that we want to make available to the Pod in question, as shown in Example 11-1.

Example 11-1. my-config.txt

```
# This is a sample config file that I might use to configure an application
parameter1 = value1
parameter2 = value2
```

Next, let's create a ConfigMap with that file. We'll also add a couple of simple key/value pairs here. These are referred to as literal values on the command line:

```
$ kubectl create configmap my-config \
  --from-file=my-config.txt \
  --from-literal=extra-param=extra-value \
  --from-literal=another-param=another-value
```

The equivalent YAML for the ConfigMap object we just created is:

```
$ kubectl get configmaps my-config -o yaml

apiVersion: v1
data:
  another-param: another-value
  extra-param: extra-value
  my-config.txt: |
    # This is a sample config file that I might use to configure an application
    parameter1 = value1
    parameter2 = value2
kind: ConfigMap
metadata:
  creationTimestamp: ...
  name: my-config
  namespace: default
  resourceVersion: "13556"
  selfLink: /api/v1/namespaces/default/configmaps/my-config
  uid: 3641c553-f7de-11e6-98c9-06135271a273
```

As you can see, the ConfigMap is really just some key/value pairs stored in an object. The interesting stuff happens when you try to *use* a ConfigMap.

Using a ConfigMap

There are three main ways to use a ConfigMap:

Filesystem
: You can mount a ConfigMap into a Pod. A file is created for each entry based on the key name. The contents of that file are set to the value.

Environment variable
: A ConfigMap can be used to dynamically set the value of an environment variable.

Command-line argument
: Kubernetes supports dynamically creating the command line for a container based on ConfigMap values.

Let's create a manifest for `kuard` that pulls all of these together, as shown in Example 11-2.

Example 11-2. kuard-config.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard-config
spec:
  containers:
    - name: test-container
      image: gcr.io/kuar-demo/kuard-amd64:1
      imagePullPolicy: Always
      command:
        - "/kuard"
        - "$(EXTRA_PARAM)"
      env:
        - name: ANOTHER_PARAM
          valueFrom:
            configMapKeyRef:
              name: my-config
              key: another-param
        - name: EXTRA_PARAM
          valueFrom:
            configMapKeyRef:
              name: my-config
              key: extra-param
      volumeMounts:
        - name: config-volume
          mountPath: /config
  volumes:
    - name: config-volume
      configMap:
        name: my-config
  restartPolicy: Never
```

For the filesystem method, we create a new volume inside the pod and give it the name `config-volume`. We then define this volume to be a ConfigMap volume and point at the ConfigMap to mount. We have to specify where this gets mounted into the `kuard` container with a `volumeMount`. In this case we are mounting it at `/config`.

Environment variables are specified with a special `valueFrom` member. This references the ConfigMap and the data key to use within that ConfigMap.

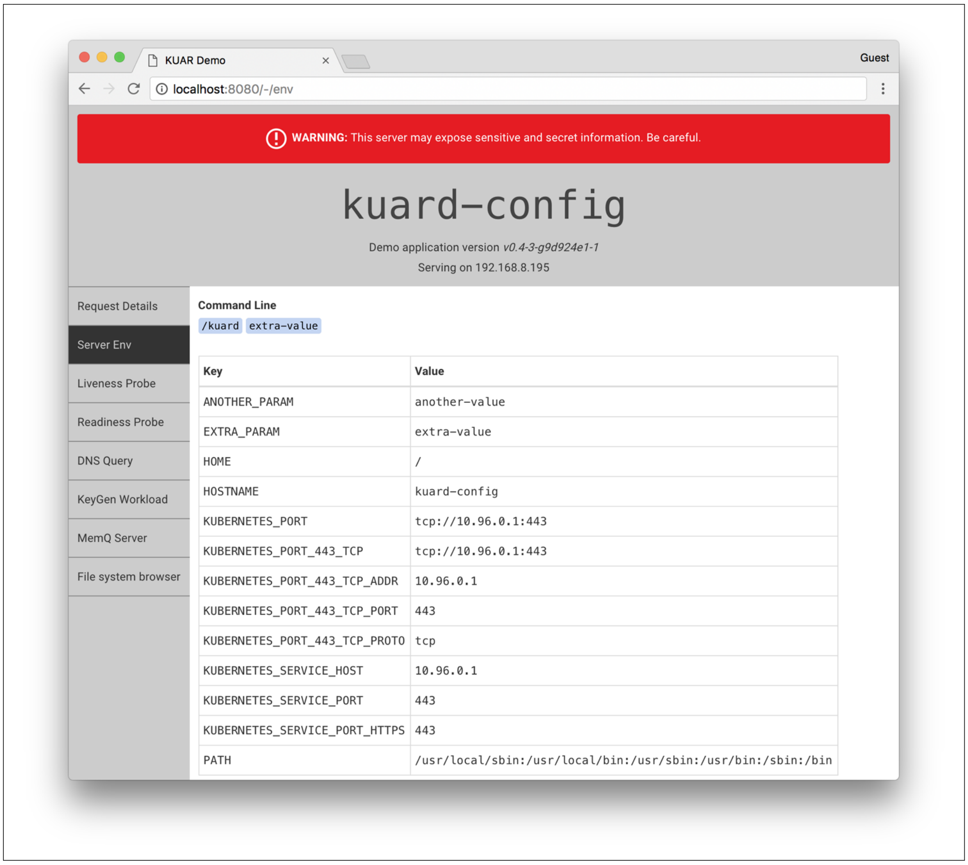
Command-line arguments build on environment variables. Kubernetes will perform the correct substitution with a special `$(<env-var-name>)` syntax.

Run this Pod and let's port-forward to examine how the app sees the world:

```
$ kubectl apply -f kuard-config.yaml
$ kubectl port-forward kuard-config 8080
```

Now point your browser at *http://localhost:8080*. We can look at how we've injected configuration values into the program in all three ways.

Click on the "Server Env" tab on the left. This will show the command line that the app was launched with along with its environment, as shown in Figure 11-1.



Command Line

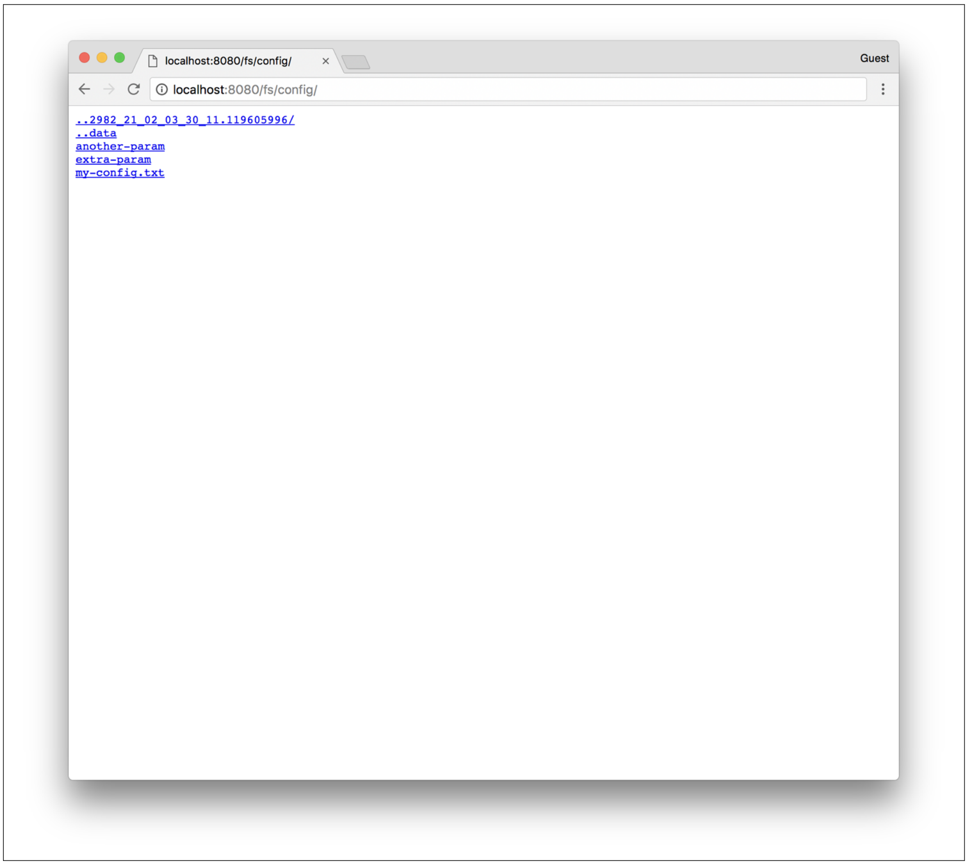
```
/kuard extra-value
```

Key	Value
ANOTHER_PARAM	another-value
EXTRA_PARAM	extra-value
HOME	/
HOSTNAME	kuard-config
KUBERNETES_PORT	tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP	tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_ADDR	10.96.0.1
KUBERNETES_PORT_443_TCP_PORT	443
KUBERNETES_PORT_443_TCP_PROTO	tcp
KUBERNETES_SERVICE_HOST	10.96.0.1
KUBERNETES_SERVICE_PORT	443
KUBERNETES_SERVICE_PORT_HTTPS	443
PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

Figure 11-1. kuard showing its environment

Here we can see that we've added two environment variables (`ANOTHER_PARAM` and `EXTRA_PARAM`) whose values are set via the ConfigMap. Furthermore, we've added an argument to the command line of `kuard` based on the `EXTRA_PARAM` value.

Next, click on the "File system browser" tab (Figure 11-2). This lets you explore the filesystem as the application sees it. You should see an entry called `/config`. This is a volume created based on our ConfigMap. If you navigate into that, you'll see that a file has been created for each entry of the ConfigMap. You'll also see some hidden files (prepended with ..) that are used to do a clean swap of new values when the ConfigMap is updated.



```
..2982_21_02_03_30_11.119605996/
..data
another-param
extra-param
my-config.txt
```

Figure 11-2. The /config directory as seen through kuard

Secrets

While ConfigMaps are great for most configuration data, there is certain data that is extra-sensitive. This can include passwords, security tokens, or other types of private keys. Collectively, we call this type of data "secrets." Kubernetes has native support for storing and handling this data with care.

Secrets enable container images to be created without bundling sensitive data. This allows containers to remain portable across environments. Secrets are exposed to pods via explicit declaration in pod manifests and the Kubernetes API. In this way the Kubernetes secrets API provides an application-centric mechanism for exposing sensitive configuration information to applications in a way that's easy to audit and leverages native OS isolation primitives.

Depending on your requirements, Kubernetes secrets may not be secure enough for you. As of Kubernetes version 1.6, anyone with root access on any node has access to all secrets in the cluster. While Kubernetes utilizes native OS containerization primitives to only expose Pods to secrets they are supposed to see, isolation between nodes is still a work in progress.

Kubernetes version 1.7 improves this situation quite a bit. When properly configured, it both encrypts stored secrets and restricts the secrets that each individual node has access to.

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

Creating Secrets

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

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

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

The first step in creating a secret is to obtain the raw data we want to store. The TLS key and certificate for the `kuard` application can be downloaded by running the following commands (please don't use these certificates outside of this example):

```
$ curl -O https://storage.googleapis.com/kuar-demo/kuard.crt
$ curl -O https://storage.googleapis.com/kuar-demo/kuard.key
```

With the *kuard.crt* and *kuard.key* files stored locally, we are ready to create a secret. Create a secret named `kuard-tls` using the `create secret` command:

```
$ kubectl create secret generic kuard-tls \
  --from-file=kuard.crt \
  --from-file=kuard.key
```

The `kuard-tls` secret has been created with two data elements. Run the following command to get details:

```
$ kubectl describe secrets kuard-tls

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

Type:          Opaque

Data
====
kuard.crt:     1050 bytes
kuard.key:     1679 bytes
```

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

Consuming Secrets

Secrets can be consumed using the Kubernetes REST API by applications that know how to call that API directly. However, our goal is to keep applications portable. Not only should they run well in Kubernetes, but they should run, unmodified, on other platforms.

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

Secrets volumes

Secret data can be exposed to pods using the secrets volume type. Secrets volumes are managed by the `kubelet` and are created at pod creation time. Secrets are stored on tmpfs volumes (aka RAM disks) and, as such, are not written to disk on nodes.

Each data element of a secret is stored in a separate file under the target mount point specified in the volume mount. The `kuard-tls` secret contains two data elements: *kuard.crt* and *kuard.key*. Mounting the `kuard-tls` secrets volume to `/tls` results in the following files:

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

The following pod manifest (Example 11-3) demonstrates how to declare a secrets volume, which exposes the `kuard-tls` secret to the `kuard` container under `/tls`.

Example 11-3. kuard-secret.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard-tls
spec:
  containers:
    - name: kuard-tls
      image: gcr.io/kuar-demo/kuard-amd64:1
      imagePullPolicy: Always
      volumeMounts:
      - name: tls-certs
        mountPath: "/tls"
        readOnly: true
  volumes:
    - name: tls-certs
      secret:
        secretName: kuard-tls
```

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

```
$ kubectl apply -f kuard-secret.yaml
```

Connect to the pod by running:

```
$ kubectl port-forward kuard-tls 8443:8443
```

Now navigate your browser to *https://localhost:8443*. You should see some invalid certificate warnings as this is a self-signed certificate for *kuard.example.com*. If you navigate past this warning, you should see the `kuard` server hosted via HTTPS. Use the "File system browser" tab to find the certificates on disk.

Private Docker Registries

A special use case for secrets is to store access credentials for private Docker registries. Kubernetes supports using images stored on private registries, but access to those images requires credentials. Private images can be stored across one or more private registries. This presents a challenge for managing credentials for each private registry on every possible node in the cluster.

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

Use the `create secret docker-registry` to create this special kind of secret:

```
$ kubectl create secret docker-registry my-image-pull-secret \
  --docker-username=<username> \
  --docker-password=<password> \
  --docker-email=<email-address>
```

Enable access to the private repository by referencing the image pull secret in the pod manifest file, as shown in Example 11-4.

Example 11-4. kuard-secret-ips.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard-tls
spec:
  containers:
    - name: kuard-tls
      image: gcr.io/kuar-demo/kuard-amd64:1
      imagePullPolicy: Always
      volumeMounts:
      - name: tls-certs
        mountPath: "/tls"
        readOnly: true
  imagePullSecrets:
  - name:  my-image-pull-secret
  volumes:
    - name: tls-certs
      secret:
        secretName: kuard-tls
```

Naming Constraints

The key names for data items inside of a secret or ConfigMap are defined to map to valid environment variable names. They may begin with a dot followed by a letter or number. Following characters include dots, dashes, and underscores. Dots cannot be repeated and dots and underscores or dashes cannot be adjacent to each other. More formally, this means that they must conform to the regular expression `[.]?[a-zA-Z0-9]([.]?[-_a-zA-Z0-9]*[a-zA-Z0-9])*`. Some examples of valid and invalid names for ConfigMaps or secrets are given in Table 11-1.

Table 11-1. ConfigMap and secret key examples

Valid key name	Invalid key name
`.auth_token`	`Token..properties`
`Key.pem`	`auth file.json`
`config_file`	`_password.txt`

When selecting a key name consider that these keys can be exposed to pods via a volume mount. Pick a name that is going to make sense when specified on a command line or in a config file. Storing a TLS key as `key.pem` is more clear than `tls-key` when configuring applications to access secrets.

ConfigMap data values are simple UTF-8 text specified directly in the manifest. As of Kubernetes 1.6, ConfigMaps are unable to store binary data.

Secret data values hold arbitrary data encoded using base64. The use of base64 encoding makes it possible to store binary data. This does, however, make it more difficult to manage secrets that are stored in YAML files as the base64-encoded value must be put in the YAML.

Managing ConfigMaps and Secrets

Secrets and ConfigMaps are managed through the Kubernetes API. The usual `create`, `delete`, `get`, and `describe` commands work for manipulating these objects.

Listing

You can use the `kubectl get secrets` command to list all secrets in the current namespace:

```
$ kubectl get secrets

NAME                  TYPE                                  DATA      AGE
default-token-f5jq2   kubernetes.io/service-account-token   3         1h
kuard-tls             Opaque                                2         20m
```

Similarly, you can list all of the ConfigMaps in a namespace:

```
$ kubectl get configmaps

NAME        DATA      AGE
my-config   3         1m
```

`kubectl describe` can be used to get more details on a single object:

```
$ kubectl describe configmap my-config

Name:           my-config
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data
====
another-param:  13 bytes
```

```
    extra-param:    11 bytes
    my-config.txt:  116 bytes
```

Finally, you can see the raw data (including values in secrets!) with something like `kubectl get configmap my-config -o yaml` or `kubectl get secret kuard-tls -o yaml`.

Creating

The easiest way to create a secret or a ConfigMap is via `kubectl create secret generic` or `kubectl create configmap`. There are a variety of ways to specify the data items that go into the secret or ConfigMap. These can be combined in a single command:

`--from-file=<filename>`
: Load from the file with the secret data key the same as the filename.

`--from-file=<key>=<filename>`
: Load from the file with the secret data key explicitly specified.

`--from-file=<directory>`
: Load all the files in the specified directory where the filename is an acceptable key name.

`--from-literal=<key>=<value>`
: Use the specified key/value pair directly.

Updating

You can update a ConfigMap or secret and have it reflected in running programs. There is no need to restart if the application is configured to reread configuration values. This is a rare feature but might be something you put in your own applications.

The following are three ways to update ConfigMaps or secrets.

Update from file

If you have a manifest for your ConfigMap or secret, you can just edit it directly and push a new version with `kubectl replace -f <filename>`. You can also use `kubectl apply -f <filename>` if you previously created the resource with `kubectl apply`.

Due to the way that datafiles are encoded into these objects, updating a configuration can be a bit cumbersome as there is no provision in `kubectl` to load data from an external file. The data must be stored directly in the YAML manifest.

The most common use case is when the ConfigMap is defined as part of a directory or list of resources and everything is created and updated together. Oftentimes these manifests will be checked into source control.

It is generally a bad idea to check secret YAML files into source control. It is too easy to push these files someplace public and leak your secrets.

Recreate and update

If you store the inputs into your ConfigMaps or secrets as separate files on disk (as opposed to embedded into YAML directly), you can use `kubectl` to recreate the manifest and then use it to update the object.

This will look something like this:

```
$ kubectl create secret generic kuard-tls \
  --from-file=kuard.crt --from-file=kuard.key \
  --dry-run -o yaml | kubectl replace -f -
```

This command line first creates a new secret with the same name as our existing secret. If we just stopped there, the Kubernetes API server would return an error complaining that we are trying to create a secret that already exists. Instead, we tell `kubectl` not to actually send the data to the server but instead to dump the YAML that it *would have* sent to the API server to `stdout`. We then pipe that to `kubectl replace` and use `-f -` to tell it to read from `stdin`. In this way we can update a secret from files on disk without having to manually base64-encode data.

Edit current version

The final way to update a ConfigMap is to use `kubectl edit` to bring up a version of the ConfigMap in your editor so you can tweak it (you could also do this with a secret, but you'd be stuck managing the base64 encoding of values on your own):

```
$ kubectl edit configmap my-config
```

You should see the ConfigMap definition in your editor. Make your desired changes and then save and close your editor. The new version of the object will be pushed to the Kubernetes API server.

Live updates

Once a ConfigMap or secret is updated using the API, it'll be automatically pushed to all volumes that use that ConfigMap or secret. It may take a few seconds, but the file listing and contents of the files, as seen by `kuard`, will be updated with these new values. Using this live update feature you can update the configuration of applications without restarting them.

Currently there is no built-in way to signal an application when a new version of a ConfigMap is deployed. It is up to the application (or some helper script) to look for the config files to change and reload them.

Using the file browser in `kuard` (accessed through `kubectl port-forward`) is a great way to interactively play with dynamically updating secrets and ConfigMaps.

Summary

ConfigMaps and secrets are a great way to provide dynamic configuration in your application. They allow you to create a container image (and pod definition) once and reuse it in different contexts. This can include using the exact same image as you move from dev to staging to production. It can also include using a single image across multiple teams and services. Separating configuration from application code will make your applications more reliable and reusable.

CHAPTER 12

Deployments

So far, you have seen how to package your application as a container, create a replicated set of these containers, and use services to load-balance traffic to your service. All of these objects are used to build a single instance of your application. They do little to help you manage the daily or weekly cadence of releasing new versions of your application. Indeed, both Pods and ReplicaSets are expected to be tied to specific container images that don't change.

The `Deployment` object exists to manage the release of new versions. Deployments represent deployed applications in a way that transcends any particular software version of the application. Additionally, Deployments enable you to easily move from one version of your code to the next version of your code. This "rollout" process is configurable and careful. It waits for a user-configurable amount of time between upgrading individual Pods. It also uses health checks to ensure that the new version of the application is operating correctly, and stops the deployment if too many failures occur.

Using Deployments you can simply and reliably roll out new software versions without downtime or errors. The actual mechanics of the software rollout performed by a Deployment is controlled by a Deployment controller that runs in the Kubernetes cluster itself. This means you can let a Deployment proceed unattended and it will still operate correctly and safely. This makes it easy to integrate Deployments with numerous continuous delivery tools and services. Further, running server-side makes it safe to perform a rollout from places with poor or intermittent internet connectivity. Imagine rolling out a new version of your software from your phone while riding on the subway. Deployments make this possible and safe!

When Kubernetes was first released, one of the most popular demonstrations of its power was the "rolling update," which showed how you could use a single command to seamlessly update a running application without taking any downtime or losing requests. This original demo was based on the `kubectl rolling-update` command, which is still available in the command-line tool, but its functionality has largely been subsumed by the `Deployment` object.

Your First Deployment

At the beginning of this book, you created a Pod by running `kubectl run`. It was something similar to:

```
$ kubectl run nginx --image=nginx:1.7.12
```

Under the hood, this was actually creating a `Deployment` object.

You can view this `Deployment` object by running:

```
$ kubectl get deployments nginx
NAME    DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx   1         1         1            1           13s
```

Deployment Internals

Let's explore how Deployments actually work. Just as we learned that ReplicaSets manage Pods, Deployments manage ReplicaSets. As with all relationships in Kubernetes, this relationship is defined by labels and a label selector. You can see the label selector by looking at the `Deployment` object:

```
$ kubectl get deployments nginx \
  -o jsonpath --template {.spec.selector.matchLabels}

map[run:nginx]
```

From this you can see that the Deployment is managing a ReplicaSet with the labels `run=nginx`. We can use this in a label selector query across ReplicaSets to find that specific ReplicaSet:

```
$ kubectl get replicasets --selector=run=nginx

NAME               DESIRED   CURRENT   READY     AGE
nginx-1128242161   1         1         1         13m
```

Now let's see the relationship between a Deployment and a ReplicaSet in action. We can resize the Deployment using the imperative `scale` command:

```
$ kubectl scale deployments nginx --replicas=2

deployment "nginx" scaled
```

Now if we list that ReplicaSet again, we should see:

```
$ kubectl get replicasets --selector=run=nginx

NAME               DESIRED   CURRENT   READY     AGE
nginx-1128242161   2         2         2         13m
```

Scaling the Deployment has also scaled the ReplicaSet it controls.

Now let's try the opposite, scaling the ReplicaSet:

```
$ kubectl scale replicasets nginx-1128242161 --replicas=1

replicaset "nginx-1128242161" scaled
```

Now `get` that ReplicaSet again:

```
$ kubectl get replicasets --selector=run=nginx

NAME               DESIRED   CURRENT   READY     AGE
nginx-1128242161   2         2         2         13m
```

That's odd. Despite our scaling the ReplicaSet to one replica, it still has two replicas as its desired state. What's going on? Remember, Kubernetes is an online, self-healing system. The top-level `Deployment` object is managing this ReplicaSet. When you adjust the number of replicas to one, it no longer matches the desired state of the Deployment, which has `replicas` set to `2`. The Deployment controller notices this and takes action to ensure the observed state matches the desired state, in this case readjusting the number of replicas back to two.

If you ever want to manage that ReplicaSet directly, you need to delete the Deployment (remember to set `--cascade` to `false`, or else it will delete the ReplicaSet and Pods as well!).

Creating Deployments

Of course, as has been stated elsewhere, you should have a preference for declarative management of your Kubernetes configurations. This means maintaining the state of your deployments in YAML or JSON files on disk.

As a starting point, download this Deployment into a YAML file:

```
$ kubectl get deployments nginx --export -o yaml > nginx-deployment.yaml
$ kubectl replace -f nginx-deployment.yaml --save-config
```

If you look in the file, you will see something like this:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  labels:
    run: nginx
  name: nginx
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      run: nginx
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        run: nginx
    spec:
      containers:
      - image: nginx:1.7.12
        imagePullPolicy: Always
      dnsPolicy: ClusterFirst
      restartPolicy: Always
```

A lot of read-only and default fields were removed in the preceding listing for brevity. We also need to run `kubectl replace --save-config`. This adds an annotation so that, when applying changes in the future, `kubectl` will know what the last applied configuration was for smarter merging of configs. If you always use `kubectl apply`, this step is only required after the first time you create a Deployment using `kubectl create -f`.

The Deployment spec has a very similar structure to the ReplicaSet spec. There is a Pod template, which contains a number of containers that are created for each replica managed by the Deployment. In addition to the Pod specification, there is also a `strategy` object:

```
...
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
...
```

The `strategy` object dictates the different ways in which a rollout of new software can proceed. There are two different strategies supported by Deployments: `Recreate` and `RollingUpdate`.

These are discussed in detail later in this chapter.

Managing Deployments

As with all Kubernetes objects, you can get detailed information about your Deployment via the `kubectl describe` command:

```
$ kubectl describe deployments nginx

Name:                   nginx
Namespace:              default
CreationTimestamp:      Sat, 31 Dec 2016 09:53:32 -0800
Labels:                 run=nginx
Selector:               run=nginx
Replicas:               2 updated | 2 total | 2 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:         <none>
NewReplicaSet:          nginx-1128242161 (2/2 replicas created)
Events:
  FirstSeen   ...   Message
  ---------   ...   -------
  5m          ...   Scaled up replica set nginx-1128242161 to 1
  4m          ...   Scaled up replica set nginx-1128242161 to 2
```

In the output of `describe` there is a great deal of important information.

Two of the most important pieces of information in the output are `OldReplicaSets` and `NewReplicaSet`. These fields point to the ReplicaSet objects this Deployment is currently managing. If a Deployment is in the middle of a rollout, both fields will be set to a value. If a rollout is complete, `OldReplicaSets` will be set to `<none>`.

In addition to the `describe` command, there is also the `kubectl rollout` command for deployments. We will go into this command in more detail later on, but for now, you can use `kubectl rollout history` to obtain the history of rollouts associated with a particular Deployment. If you have a current Deployment in progress, then you can use `kubectl rollout status` to obtain the current status of a rollout.

Updating Deployments

Deployments are declarative objects that describe a deployed application. The two most common operations on a Deployment are scaling and application updates.

Scaling a Deployment

Although we previously showed how you could imperatively scale a Deployment using the `kubectl scale` command, the best practice is to manage your Deployments declaratively via the YAML files, and then use those files to update your Deployment. To scale up a Deployment, you would edit your YAML file to increase the number of replicas:

```
...
spec:
  replicas: 3
...
```

Once you have saved and committed this change, you can update the Deployment using the `kubectl apply` command:

```
$ kubectl apply -f nginx-deployment.yaml
```

This will update the desired state of the Deployment, causing it to increase the size of the ReplicaSet it manages, and eventually create a new Pod managed by the Deployment:

```
$ kubectl get deployments nginx

NAME     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx    3         3         3            3           4m
```

Updating a Container Image

The other common use case for updating a Deployment is to roll out a new version of the software running in one or more containers. To do this, you should likewise edit the deployment YAML file, though in this case you are updating the container image, rather than the number of replicas:

```
...
      containers:
      - image: nginx:1.9.10
        imagePullPolicy: Always
...
```

We are also going to put an annotation in the template for the Deployment to record some information about the update:

```
...
spec:
  ...
  template:
    annotations:
      kubernetes.io/change-cause: "Update nginx to 1.9.10"
...
```

Make sure you add this annotation to the template and not the `Deployment` itself. Also, do not update the `change-cause` annotation when doing simple scaling operations. A modification of `change-cause` is a significant change to the template and will trigger a new rollout.

Again, you can use `kubectl apply` to update the Deployment:

```
$ kubectl apply -f nginx-deployment.yaml
```

After you update the Deployment it will trigger a rollout, which you can then monitor via the `kubectl rollout` command:

```
$ kubectl rollout status deployments nginx
deployment nginx successfully rolled out
```

You can see the old and new ReplicaSets managed by the deployment along with the images being used. Both the old and new ReplicaSets are kept around in case you want to roll back:

```
$ kubectl get replicasets -o wide

NAME               DESIRED   CURRENT   READY   ...   IMAGE(S)       ...
nginx-1128242161   0         0         0       ...   nginx:1.7.12   ...
nginx-1128635377   3         3         3       ...   nginx:1.9.10   ...
```

If you are in the middle of a rollout and you want to temporarily pause it for some reason (e.g., if you start seeing weird behavior in your system and you want to investigate), you can use the `pause` command:

```
$ kubectl rollout pause deployments nginx
deployment "nginx" paused
```

If, after investigation, you believe the rollout can safely proceed, you can use the `resume` command to start up where you left off:

```
$ kubectl rollout resume deployments nginx
deployment "nginx" resumed
```

Rollout History

Kubernetes Deployments maintain a history of rollouts, which can be useful both for understanding the previous state of the Deployment and to roll back to a specific version.

You can see the deployment history by running:

```
$ kubectl rollout history deployment nginx

deployments "nginx"
REVISION        CHANGE-CAUSE
```

```
1               <none>
2               Update nginx to 1.9.10
```

The revision history is given in oldest to newest order. A unique revision number is incremented for each new rollout. So far we have two: the initial deployment, the update of the image to `nginx:1.9.10`.

If you are interested in more details about a particular revision, you can add the `--revision` flag to view details about that specific revision:

```
$ kubectl rollout history deployment nginx --revision=2

deployments "nginx" with revision #2
  Labels:       pod-template-hash=2738859366
        run=nginx
  Annotations:  kubernetes.io/change-cause=Update nginx to 1.9.10
  Containers:
   nginx:
    Image:      nginx:1.9.10
    Port:
    Volume Mounts:      <none>
    Environment Variables:      <none>
  No volumes.
```

Let's do one more update for this example. Update the nginx version to 1.10.2 by modifying the container version number and updating the `change-cause` annotation. Apply it with `kubectl apply`. Our history should now have three entries:

```
$ kubectl rollout history deployment nginx

deployments "nginx"
REVISION        CHANGE-CAUSE
1               <none>
2               Update nginx to 1.9.10
3               Update nginx to 1.10.2
```

Let's say there is an issue with the latest release and you want to roll back while you investigate. You can simply undo the last rollout:

```
$ kubectl rollout undo deployments nginx
deployment "nginx" rolled back
```

The `undo` command works regardless of the stage of the rollout. You can undo both partially completed and fully completed rollouts. An undo of a rollout is actually simply a rollout in reverse (e.g., from *v2* to *v1*, instead of from *v1* to *v2*), and all of the same policies that control the rollout strategy apply to the undo strategy as well. You can see the `Deployment` object simply adjusts the desired replica counts in the managed ReplicaSets:

```
$ kubectl get replicasets -o wide

NAME               DESIRED   CURRENT   READY   ...   IMAGE(S)       ...
nginx-1128242161   0         0         0       ...   nginx:1.7.12   ...
nginx-1570155864   0         0         0       ...   nginx:1.10.2   ...
nginx-2738859366   3         3         3       ...   nginx:1.9.10   ...
```

When using declarative files to control your production systems, you want to, as much as possible, ensure that the checked-in manifests match what is actually running in your cluster. When you do a `kubectl rollout undo` you are updating the production state in a way that isn't reflected in your source control.

An alternate (and perhaps preferred) way to undo a rollout is to revert your YAML file and `kubectl apply` the previous version. In this way your "change tracked configuration" more closely tracks what is really running in your cluster.

Let's look at our deployment history again:

```
$ kubectl rollout history deployment nginx

REVISION          CHANGE-CAUSE
1                 <none>
3                 Update nginx to 1.10.2
4                 Update nginx to 1.9.10
```

Revision 2 is missing! It turns out that when you roll back to a previous revision, the Deployment simply reuses the template and renumbers it so that it is the latest revision. What was revision 2 before is now reordered into revision 4.

We previously saw that you can use the `kubectl rollout undo` command to roll back to a previous version of a deployment. Additionally, you can roll back to a specific revision in the history using the `--to-revision` flag:

```
$ kubectl rollout undo deployments nginx --to-revision=3
deployment "nginx" rolled back
$ kubectl rollout history deployment nginx
deployments "nginx"
REVISION          CHANGE-CAUSE
1                 <none>
4                 Update nginx to 1.9.10
5                 Update nginx to 1.10.2
```

Again, the `undo` took revision 3, applied it, and renumbered it as revision 5.

Specifying a revision of `0` is a shorthand way of specifying the previous revision. In this way, `kubectl rollout undo` is equivalent to `kubectl rollout undo --to-revision=0`.

By default, the complete revision history of a Deployment is kept attached to the Deployment object itself. Over time (e.g., years) this history can grow fairly large, so it is recommended that if you have Deployments that you expect to keep around for a long time you set a maximum history size for the Deployment revision history, to limit the total size of the `Deployment` object. For example, if you do a daily update you may limit your revision history to 14, to keep a maximum of 2 weeks' worth of revisions (if you don't expect to need to roll back beyond 2 weeks).

To accomplish this, use the `revisionHistoryLimit` property in the Deployment specification:

```
...
spec:
  # We do daily rollouts, limit the revision history to two weeks of
  # releases as we don't expect to roll back beyond that.
  revisionHistoryLimit: 14
...
```

Deployment Strategies

When it comes time to change the version of software implementing your service, a Kubernetes Deployment supports two different rollout strategies:

- `Recreate`
- `RollingUpdate`

Recreate Strategy

The recreate strategy is the simpler of the two rollout strategies. It simply updates the ReplicaSet it manages to use the new image and terminates all of the Pods associated with the Deployment. The ReplicaSet notices that it no longer has any replicas, and re-creates all Pods using the new image. Once the Pods are re-created, they are running the new version.

While this strategy is fast and simple, it has one major drawback—it is potentially catastrophic, and will almost certainly result in some site downtime. Because of this, the recreate strategy should only be used for test deployments where a service is not user-facing and a small amount of downtime is acceptable.

RollingUpdate Strategy

The `RollingUpdate` strategy is the generally preferable strategy for any user-facing service. While it is slower than `Recreate`, it is also significantly more sophisticated and robust. Using `RollingUpdate`, you can roll out a new version of your service while it is still receiving user traffic, without any downtime.

As you might infer from the name, the rolling update strategy works by updating a few Pods at a time, moving incrementally until all of the Pods are running the new version of your software.

Managing multiple versions of your service

Importantly, this means that for a period of time, both the new and the old version of your service will be receiving requests and serving traffic. This has important implications for how you build your software. Namely, it is critically important that each version of your software, and all of its clients, is capable of talking interchangeably with both a slightly older and a slightly newer version of your software.

As an example of why this is important, consider the following scenario:

> You are in the middle of rolling out your frontend software; half of your servers are running version 1 and half are running version 2. A user makes an initial request to your service and downloads a client-side JavaScript library that implements your UI. This request is serviced by a version 1 server and thus the user receives the version 1 client library. This client library runs in the user's browser and makes subsequent API requests to your service. These API requests happen to be routed to a version 2 server; thus, version 1 of your JavaScript client library is talking to version 2 of your API server. If you haven't ensured compatibility between these versions, your application won't function correctly.

At first, this might seem like an extra burden. But in truth, you always had this problem; you may just not have noticed. Concretely, a user can make a request at time `t` just before you initiate an update. This request is serviced by a version 1 server. At `t_1` you update your service to version 2. At `t_2` the version 1 client code running on the user's browser runs and hits an API endpoint being operated by a version 2 server. No matter how you update your software, you have to maintain backward and forward compatibility for reliable updates. The nature of the rolling update strategy simply makes it more clear and explicit that this is something to think about.

Note that this doesn't just apply to JavaScript clients—the same thing is true of client libraries that are compiled into other services that make calls to your service. Just because you updated doesn't mean they have updated their client libraries. This sort of backward compatibility is critical to decoupling your service from systems that depend on your service. If you don't formalize your APIs and decouple yourself, you are forced to carefully manage your rollouts with all of the other systems that call into your service. This kind of tight coupling makes it extremely hard to produce the necessary agility to be able to push out new software every week, let alone every hour or every day. In the de-coupled architecture shown in Figure 12-1, the frontend is isolated from the backend via an API contract and a load balancer, whereas in the coupled architecture, a thick client compiled into the frontend is used to connect directly to the backends.

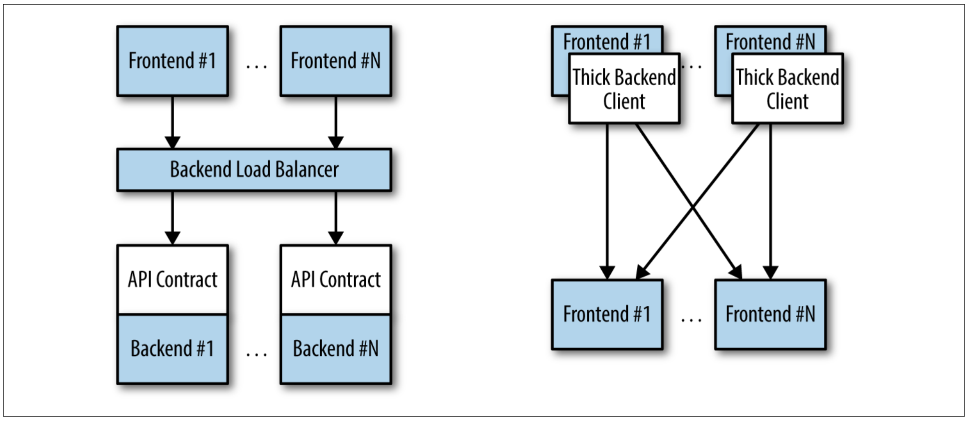


Figure 12-1. Diagrams of both de-coupled (left) and couple (right) application architectures

Configuring a rolling update

`RollingUpdate` is a fairly generic strategy; it can be used to update a variety of applications in a variety of settings. Consequently, the rolling update itself is quite configurable; you can tune its behavior to suit your particular needs. There are two parameters you can use to tune the rolling update behavior: `maxUnavailable` and `maxSurge`.

The `maxUnavailable` parameter sets the maximum number of Pods that can be unavailable during a rolling update. It can either be set to an absolute number (e.g., `3` meaning a maximum of three Pods can be unavailable) or to a percentage (e.g., `20%` meaning a maximum of 20% of the desired number of replicas can be unavailable).

Generally speaking, using a percentage is a good approach for most services, since the value is correctly applicable regardless of the desired number of replicas in the Deployment. However, there are times when you may want to use an absolute number (e.g., limiting the maximum unavailable pods to one).

At its core, the `maxUnavailable` parameter helps tune how quickly a rolling update proceeds. For example, if you set `maxUnavailable` to `50%`, then the rolling update will immediately scale the old ReplicaSet down to 50% of its original size. If you have four replicas, it will scale it down to two replicas. The rolling update will then replace the removed pods by scaling the new ReplicaSet up to two replicas, for a total of four replicas (two old, two new). It will then scale the old ReplicaSet down to zero replicas, for a total size of two new replicas. Finally, it will scale the new ReplicaSet up to four replicas, completing the rollout. Thus, with `maxUnavailable` set to `50%`, our rollout completes in four steps, but with only 50% of our service capacity at times.

Consider instead what happens if we set `maxUnavailable` to `25%`. In this situation, each step is only performed with a single replica at a time and thus it takes twice as many steps for the rollout to complete, but availability only drops to a minimum of 75% during the rollout. This illustrates how `maxUnavailable` allows us to trade rollout speed for availability.

> The observant among you will note that the recreate strategy is actually identical to the rolling update strategy with `maxUnavailable` set to `100%`.

Using reduced capacity to achieve a successful rollout is useful either when your service has cyclical traffic patterns (e.g., much less traffic at night) or when you have limited resources, so scaling to larger than the current maximum number of replicas isn't possible.

However, there are situations where you don't want to fall below 100% capacity, but you are willing to temporarily use additional resources in order to perform a rollout. In these situations, you can set the `maxUnavailable` parameter to `0%`, and instead control the rollout using the `maxSurge` parameter. Like `maxUnavailable`, `maxSurge` can be specified either as a specific number or a percentage.

The `maxSurge` parameter controls how many extra resources can be created to achieve a rollout. To illustrate how this works, imagine we have a service with 10 replicas. We set `maxUnavailable` to `0` and `maxSurge` to `20%`. The first thing the rollout will do is scale the new ReplicaSet up to 2 replicas, for a total of 12 (120%) in the service. It will then scale the old ReplicaSet down to 8 replicas, for a total of 10 (8 old, 2 new) in the service. This process proceeds until the rollout is complete. At any time, the capacity of the service is guaranteed to be at least 100% and the maximum extra resources used for the rollout are limited to an additional 20% of all resources.

> Setting `maxSurge` to `100%` is equivalent to a blue/green deployment. The Deployment controller first scales the new version up to 100% of the old version. Once the new version is healthy, it immediately scales the old version down to 0%.

Slowing Rollouts to Ensure Service Health

The purpose of a staged rollout is to ensure that the rollout results in a healthy, stable service running the new software version. To do this, the Deployment controller always waits until a Pod reports that it is ready before moving on to updating the next Pod.

The Deployment controller examines the Pod's status as determined by its readiness checks. Readiness checks are part of the Pod's health probes, and they are described in detail in Chapter 5. If you want to use Deployments to reliably roll out your software, you *have* to specify readiness health checks for the containers in your Pod. Without these checks the Deployment controller is running blind.

Sometimes, however, simply noticing that a Pod has become ready doesn't give you sufficient confidence that the Pod actually is behaving correctly. Some error conditions only occur after a period of time. For example, you could have a serious memory leak that still takes a few minutes to show up, or you could have a bug that is only triggered by 1% of all requests. In most real-world scenarios, you want to wait a period of time to have high confidence that the new version is operating correctly before you move on to updating the next Pod.

For deployments, this time to wait is defined by the `minReadySeconds` parameter:

```
...
spec:
  minReadySeconds: 60
...
```

Setting `minReadySeconds` to `60` indicates that the Deployment must wait for 60 seconds *after* seeing a Pod become healthy before moving on to updating the next Pod.

In addition to waiting a period of time for a Pod to become healthy, you also want to set a timeout that limits how long the system will wait. Suppose, for example, the new version of your service has a bug and immediately deadlocks. It will never become ready, and in the absence of a timeout, the Deployment controller will stall your rollout forever.

The correct behavior in such a situation is to time out the rollout. This in turn marks the rollout as failed. This failure status can be used to trigger alerting that can indicate to an operator that there is a problem with the rollout.

At first blush, timing out a rollout might seem like a unnecessary complication. However, increasingly, things like rollouts are being triggered by fully automated systems with little to no human involvement. In such a situation, timing out becomes a critical exception, which can either trigger an automated rollback of the release or create a ticket/event that triggers human intervention.

To set the timeout period, the Deployment parameter `progressDeadlineSeconds` is used:

```
...
spec:
  progressDeadlineSeconds: 600
...
```

This example sets the progress deadline to 10 minutes. If any particular stage in the rollout fails to progress in 10 minutes, then the Deployment is marked as failed, and all attempts to move the Deployment forward are halted.

It is important to note that this timeout is given in terms of Deployment *progress*, not the overall length of a Deployment. In this context progress is defined as any time the deployment creates or deletes a Pod. When that happens, the timeout clock is reset to zero. Figure 12-2 is an illustration of the deployment lifecycle.

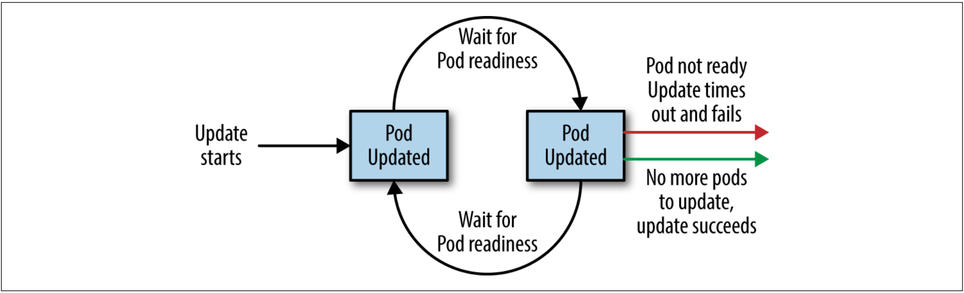

Update starts
Pod Updated
Wait for Pod readiness
Pod Updated
Wait for Pod readiness
Pod not ready Update times out and fails
No more pods to update, update succeeds

Figure 12-2. The Kubernetes Deployment lifecycle

Deleting a Deployment

If you ever want to delete a deployment, you can do it either with the imperative command:

```
$ kubectl delete deployments nginx
```

or using the declarative YAML file we created earlier:

```
$ kubectl delete -f nginx-deployment.yaml
```

In either case, by default, deleting a Deployment deletes the entire service. It will delete not just the Deployment, but also any ReplicaSets being managed by the

Deployment, as well as any Pods being managed by the ReplicaSets. As with ReplicaSets, if this is not the desired behavior, you can use the `--cascade=false` flag to exclusively delete the `Deployment` object.

Summary

At the end of the day, the primary goal of Kubernetes is to make it easy for you to build and deploy reliable distributed systems. This means not just instantiating the application once, but managing the regularly scheduled rollout of new versions of that software service. Deployments are a critical piece of reliable rollouts and rollout management for your services.

About the Authors

Kelsey Hightower has worn every hat possible throughout his career in tech, and enjoys leadership roles focused on making things happen and shipping software. Kelsey is a strong open source advocate focused on building simple tools that make people smile. When he is not slinging Go code, you can catch him giving technical workshops covering everything from programming to system administration.

Joe Beda started his career at Microsoft working on Internet Explorer (he was young and naive). Throughout 7 years at Microsoft and 10 at Google, Joe has worked on GUI frameworks, real-time voice and chat, telephony, machine learning for ads, and cloud computing. Most notably, while at Google, Joe started the Google Compute Engine and, along with Brendan and Craig McLuckie, created Kubernetes. Joe is now CTO of Heptio, a startup he founded along with Craig. Joe proudly calls Seattle home.

Brendan Burns began his career with a brief stint in the software industry followed by a PhD in Robotics focused on motion planning for human-like robot arms. This was followed by a brief stint as a professor of computer science. Eventually, he returned to Seattle and joined Google, where he worked on web search infrastructure with a special focus on low-latency indexing. While at Google, he created the Kubernetes project with Joe and Craig McLuckie. Brendan is currently a Director of Engineering at Microsoft Azure.

Colophon

The animal on the cover of *Kubernetes: Up and Running* is a bottlenose dolphin (*Tursiops truncatus*).

Bottlenose dolphins live in groups typically of 10–30 members, called pods, but group size varies from single individuals to more than 1,000. Dolphins often work as a team to harvest fish schools, but they also hunt individually. Dolphins search for prey primarily using echolocation, which is similar to sonar.

The Bottlenose dolphin is found in most tropical to temperate oceans; its color is grey, with the shade of grey varying among populations; it can be bluish-grey, brownish-grey, or even nearly black, and is often darker on the back from the rostrum to behind the dorsal fin. Bottlenose dolphins have the largest brain to body mass ratio of any mammal on Earth, sharing close ratios with those of humans and other great apes, which more than likely attributes to their incredibly high intelligence and emotional intelligence.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *British Quadrapeds*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.