7

DEPLOYING CONTAINERS TO KUBERNETES

We're now ready to begin running containers on our working Kubernetes cluster. Because Kubernetes has a declarative API, we'll create various kinds of resources to run them, and we'll monitor the cluster to see what Kubernetes does for each type of resource.

Different containers have different use cases. Some might require multiple identical instances with autoscaling to perform well under load. Other containers might exist solely to run a one-time command. Still others may require a fixed ordering to enable selecting a single primary instance and providing controlled failover to a secondary instance. Kubernetes provides different *controller* resource types for each of those use cases. We'll look at each in turn, but we'll begin with the most fundamental of them, the *Pod*, which is utilized by all of those use cases.

Pods

A Pod is the most basic resource in Kubernetes and is how we run containers. Each Pod can have one or more containers within it. The Pod is used to

provide the process isolation we saw in Chapter 2. Linux kernel namespaces are used at the Pod and the container level:

`mnt` Mount points: each container has its own root filesystem; other mounts are available to all containers in the Pod.

`uts` Unix time sharing: isolated at the Pod level.

`ipc` Interprocess communication: isolated at the Pod level.

`pid` Process identifiers: isolated at the container level.

`net` Network: isolated at the Pod level.

The biggest advantage of this approach is that multiple containers can act like processes on the same virtual host, using the `localhost` address to communicate, while still being based on separate container images.

Deploying a Pod

To get started, let's create a Pod directly. Unlike the previous chapter, in which we used `kubectl run` to have the Pod specification created for us, we'll specify it directly using YAML so that we have complete control over the Pod and to prepare us for using controllers to create Pods, providing scalability and failover.

NOTE *The example repository for this book is at* https://github.com/book-of-kubernetes/examples. *See "Running Examples" on page xx for details on getting set up.*

The automation script for this chapter does a full cluster install with three nodes that run the control plane and regular applications, providing the smallest possible highly available cluster for testing. The automation also creates some YAML files for Kubernetes resources. Here's a basic YAML resource to create a Pod running NGINX:

nginx-pod.yaml

```
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
```

Pods are part of the *core* Kubernetes API, so we just specify a version number of `v1` for the `apiVersion`. Specifying `Pod` as the `kind` tells Kubernetes exactly what resource we're creating in the API group. We will see these fields in all of our Kubernetes resources.

The `metadata` field has many uses. For the Pod, we just need to provide the one required field of `name`. We don't specify the `namespace` in the metadata, so by default this Pod will end up in the `default` Namespace.

The remaining field, `spec`, tells Kubernetes everything it needs to know to run this Pod. For now we are providing the minimal information, which is a list of containers to run, but many other options are available. In this case, we have only one container, so we provide just the name and container image Kubernetes should use.

Let's add this Pod to the cluster. The automation added files to */opt*, so we can do it from `host01` as follows:

```
root@host01:~# kubectl apply -f /opt/nginx-pod.yaml
pod/nginx created
```

In Listing 7-1, we can check the Pod's status.

```
root@host01:~# kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE     IP               NODE    ...
nginx   1/1     Running   0          2m26s   172.31.25.202    host03 ...
```

Listing 7-1: Status of NGINX

It can take some time before the Pod shows `Running`, especially if you just set up your Kubernetes cluster and it's still busy deploying core components. Keep trying this `kubectl` command to check the status.

Instead of typing the `kubectl` command multiple times, you can also use `watch`. The `watch` command is a great way to observe changes in your cluster over time. Just add `watch` in front of your command, and it will be run for you every two seconds.

We added `-o wide` to the command to see the IP address and node assignment for this Pod. Kubernetes manages that for us. In this case, the Pod was scheduled on `host03`, so we need to go there to see the running container:

```
root@host03:~# crictl pods --name nginx
POD ID         CREATED         STATE  NAME   NAMESPACE  ...
9f1d6e0207d7e  19 minutes ago  Ready  nginx  default    ...
```

Run this command on whatever host your NGINX Pod is on.

If we collect the Pod ID, we can see the container as well:

```
root@host03:~# POD_ID=$(crictl pods -q --name nginx)
root@host03:~# crictl ps --pod $POD_ID
CONTAINER      IMAGE          CREATED         STATE    NAME   ...
9da09b3671418  4cdc5dd7eaadf  20 minutes ago  Running  nginx  ...
```

This output looks very similar to the output from `kubectl get` in Listing 7-1, which is not surprising given that our cluster gets that information from the `kubelet` service running on this node, which in turn uses the same Container Runtime Interface (CRI) API that `crictl` is also using to talk to the container engine.

Pod Details and Logging

The ability to use `crictl` with the underlying container engine to explore a container running in the cluster is valuable, but it does require us to connect to the specific host running the container. Much of the time, we can avoid that by using `kubectl` commands to inspect Pods from anywhere by connecting to our cluster's API server. Let's move back to `host01` and explore the NGINX Pod further.

In Chapter 6, we saw how we could use `kubectl describe` to see the status and event log for a cluster node. We can use the same command to see the status and configuration details of other Kubernetes resources. Here's the event log for our NGINX Pod:

```
root@host01:~# kubectl describe pod nginx
Name:         nginx
Namespace: ❶ default
...
Containers:
  nginx:
    Container ID:   containerd://9da09b3671418...
...
❷ Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  22m   default-scheduler  Successfully assigned ...
  Normal  Pulling    22m   kubelet            Pulling image "nginx"
  Normal  Pulled     21m   kubelet            Successfully pulled image ...
  Normal  Created    21m   kubelet            Created container nginx
  Normal  Started    21m   kubelet            Started container nginx
```

We can use `kubectl describe` with many different Kubernetes resources, so we first tell `kubectl` that we are interested in a Pod and provide the name. Because we didn't specify a Namespace, Kubernetes will look for this Pod in the `default` Namespace ❶.

NOTE *We use the `default` Namespace for most of the examples in this book to save typing, but it's a good practice to use multiple Namespaces to keep applications separate, both to avoid naming conflicts and to manage access control. We look at Namespaces in more detail in Chapter 11.*

The `kubectl describe` command output provides an event log ❷, which is the first place to look for issues when we have problems starting a container.

Kubernetes takes a few steps when deploying a container. First, it needs to schedule it onto a node, which requires that node to be available with sufficient resources. Then, control passes to `kubelet` on that node, which has to interact with the container engine to pull the image, create a container, and start it.

After the container is started, `kubelet` collects the standard out and standard error. We can view this output by using the `kubectl logs` command:

```
root@host01:~# kubectl logs nginx
...
2021/07/13 22:37:03 [notice] 1#1: start worker processes
2021/07/13 22:37:03 [notice] 1#1: start worker process 33
2021/07/13 22:37:03 [notice] 1#1: start worker process 34
```

The `kubectl logs` command always refers to a Pod because Pods are the basic resource used to run containers, and our Pod has only one container, so we can just specify the name of the Pod as a single parameter to `kubectl logs`. As before, Kubernetes will look in the `default` Namespace because we didn't specify the Namespace.

The container output is available even if the container has exited, so the `kubectl logs` command is the place to look if a container is pulled and started successfully but then crashes. Of course, we have to hope that the container printed a log message explaining why it crashed. In Chapter 10, we look at what to do if we can't get a container going and don't have any log messages.

We're done with the NGINX Pod, so let's clean it up:

```
root@host01:~# kubectl delete -f /opt/nginx-pod.yaml
pod "nginx" deleted
```

We can use the same YAML configuration file to delete the Pod, which is convenient when we have multiple Kubernetes resources defined in a single file, as a single command will delete all of them. The `kubectl` command uses the name of each resource defined in the file to perform the delete.

Deployments

To run a container, we need a Pod, but that doesn't mean we generally want to create the Pod directly. When we create a Pod directly, we don't get all of the scalability and failover that Kubernetes offers, because Kubernetes will run only one instance of the Pod. This Pod will be allocated to a node only on creation, with no re-allocation even if the node fails.

To get scalability and failover, we instead need to create a controller to manage the Pod for us. We'll look at multiple controllers that can run Pods, but let's start with the most common: the *Deployment*.

Creating a Deployment

A Deployment manages one or more *identical* Kubernetes Pods. When we create a Deployment, we provide a Pod template. The Deployment then creates Pods matching that template with the help of a *ReplicaSet*.

DEPLOYMENTS AND REPLICASETS

Kubernetes has evolved its controller resources over time. The first type of controller, the *ReplicationController*, provided only basic functionality. It was replaced by the ReplicaSet, which has improvements in how it identifies which Pods to manage.

Part of the reason to replace ReplicationControllers with ReplicaSets is that ReplicationControllers were becoming more and more complicated, making the code difficult to maintain. The new approach splits up controller responsibility between ReplicaSets and Deployments. ReplicaSets are responsible for basic Pod management, including monitoring Pod status and performing failover. Deployments are responsible for tracking changes to the Pod template caused by configuration changes or container image updates. Deployments and ReplicaSets work together, but the Deployment creates its own ReplicaSet, so we usually need to interact only with Deployments. For this reason, I use the term *Deployment* generically to refer to features provided by the ReplicaSet, such as monitoring Pods to provide the requested number of replicas.

Here's the YAML file we'll use to create an NGINX Deployment:

nginx-deploy.yaml

```
---
kind: Deployment
apiVersion: apps/v1
metadata:
❶ name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
   ❷ labels:
        app: nginx
 ❸ spec:
     containers:
     - name: nginx
       image: nginx
    ❹ resources:
        requests:
          cpu: "100m"
```

Deployments are in the `apps` API group, so we specify `apps/v1` for `apiVersion`. Like every Kubernetes resource, we need to provide a unique name ❶ to keep this Deployment separate from any others we might create.

The Deployment specification has a few important fields, so let's look at them in detail. The `replicas` field tells Kubernetes how many identical instances of the Pod we want. Kubernetes will work to keep this many Pods

running. The next field, `selector`, is used to enable the Deployment to find its Pods. The content of `matchLabels` must exactly match the content in the `template.metadata.labels` field ❷, or Kubernetes will reject the Deployment.

Finally, the content of `template.spec` ❸ will be used as the `spec` for any Pods created by this Deployment. The fields here can include any configuration we can provide for a Pod. This configuration matches *nginx-pod.yaml* that we looked at earlier except that we add a CPU resource request ❹ so that we can configure autoscaling later on.

Let's create our Deployment from this YAML resource file:

```
root@host01:~# kubectl apply -f /opt/nginx-deploy.yaml
deployment.apps/nginx created
```

We can track the status of the Deployment with `kubectl get`:

```
root@host01:~# kubectl get deployment nginx
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   3/3     3            3           4s
```

When the Deployment is fully up, it will report that it has three replicas ready and available, which means that we now have three separate NGINX Pods managed by this Deployment:

```
root@host01:~# kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
nginx-6799fc88d8-6vn44   1/1     Running   0          18s
nginx-6799fc88d8-dcwx5   1/1     Running   0          18s
nginx-6799fc88d8-sh8qs   1/1     Running   0          18s
```

The name of each Pod begins with the name of the Deployment. Kubernetes adds some random characters to build the name of the ReplicaSet, followed by more random characters so that each Pod has a unique name. We don't need to create or manage the ReplicaSet directly, but we can use `kubectl get` to see it:

```
root@host01:~# kubectl get replicasets
NAME               DESIRED   CURRENT   READY   AGE
nginx-6799fc88d8   3         3         3       30s
```

Although we generally interact only with Deployments, it is important to know about the ReplicaSet, as some specific errors encountered when creating Pods are only reported in the ReplicaSet event log.

The `nginx` prefix on the ReplicaSet and Pod names are purely for convenience. The Deployment does not use names to match itself to Pods. Instead, it uses its selector to match the labels on the Pod. We can see these labels if we run `kubectl describe` on one of the three Pods:

```
root@host01:~# kubectl describe pod nginx-6799fc88d8-6vn44
Name:         nginx-6799fc88d8-6vn44
Namespace:    default
```

```
...
Labels:       app=nginx
...
```

This matches the Deployment's selector:

```
root@host01:~# kubectl describe deployment nginx
Name:                   nginx
Namespace:              default
...
Selector:               app=nginx
...
```

The Deployment queries the API server to identify Pods matching its selector. Whereas the Deployment uses the programmatic API, the `kubectl get` command in the following example generates a similar API server query, giving us an opportunity to see how that works:

```
root@host01:~# kubectl get all -l app=nginx
NAME                     READY   STATUS    RESTARTS   AGE
nginx-6799fc88d8-6vn44   1/1     Running   0          69s
nginx-6799fc88d8-dcwx5   1/1     Running   0          69s
nginx-6799fc88d8-sh8qs   1/1     Running   0          69s

NAME                                 DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-6799fc88d8   3         3         3       69s
```

Using `kubectl get all` in this case allows us to list multiple different kinds of resources as long as they match the selector. As a result, we see not only the three Pods but also the ReplicaSet that was created by the Deployment to manage those Pods.

It may seem strange that the Deployment uses a selector rather than just tracking the Pods it created. However, this design makes it easier for Kubernetes to be self-healing. At any time, a Kubernetes node might go offline, or we might have a network split, during which some control nodes lose their connection to the cluster. If a node comes back online, or the cluster needs to recombine after a network split, Kubernetes must be able to look at the current state of all of the running Pods and figure out what changes are required to achieve the desired state. This might mean that a Deployment that started an additional Pod as the result of a node disconnection would need to shut down a Pod when that node reconnects so that the cluster can maintain the appropriate number of replicas. Using a selector avoids the need for the Deployment to remember all the Pods it has ever created, even Pods on failed nodes.

Monitoring and Scaling

Because the Deployment is monitoring its Pods to make sure we have the correct number of replicas, we can delete a Pod, and it will be automatically re-created:

```
root@host01:~# kubectl delete pod nginx-6799fc88d8-6vn44
pod "nginx-6799fc88d8-6vn44" deleted
root@host01:~# kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
nginx-6799fc88d8-dcwx5   1/1     Running   0          3m52s
nginx-6799fc88d8-dtddk   1/1     Running   0          ❶ 14s
nginx-6799fc88d8-sh8qs   1/1     Running   0          3m52s
```

As soon as the old Pod is deleted, the Deployment created a new Pod ❶. Similarly, if we change the number of replicas for the Deployment, Pods are automatically updated. Let's add another replica:

```
root@host01:~# kubectl scale --replicas=4 deployment nginx
deployment.apps/nginx scaled
root@host01:~# kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
nginx-6799fc88d8-dcwx5   1/1     Running   0          8m22s
nginx-6799fc88d8-dtddk   1/1     Running   0          4m44s
nginx-6799fc88d8-kk7r6   1/1     Running   0          ❶ 5s
nginx-6799fc88d8-sh8qs   1/1     Running   0          8m22s
```

The first command sets the number of replicas to four. As a result, Kubernetes needs to start a new identical Pod to meet the number we requested ❶. We can scale the Deployment by updating the YAML file and re-running `kubectl apply`, or we can use the `kubectl scale` command to edit the Deployment directly. Either way, this is a declarative approach; we are updating the Deployment's resource declaration; Kubernetes then updates the actual state of the cluster to match.

Similarly, scaling the Deployment down causes Pods to be automatically deleted:

```
root@host01:~# kubectl scale --replicas=2 deployment nginx
deployment.apps/nginx scaled
root@host01:~# kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
nginx-6799fc88d8-dcwx5   1/1     Running   0          10m
nginx-6799fc88d8-sh8qs   1/1     Running   0          10m
```

When we scale down, Kubernetes selects two Pods to terminate. These Pods take a moment to finish shutting down, at which point we have only two NGINX Pods running.

Autoscaling

For an application that is receiving real requests from users, we would choose the number of replicas necessary to provide a quality application, while scaling down when possible to reduce the amount of resources used by our application. Of course, the load on our application is constantly changing, and it would be tedious to monitor each component of our application continually to scale it independently. Instead, we can have the cluster perform the monitoring and scaling for us using a *HorizontalPodAutoscaler*. The term *horizontal* in this case just refers to the fact that the autoscaler can update the number of replicas of the same Pod managed by a controller.

To configure autoscaling, we create a new resource with a reference to our Deployment. The cluster then monitors resources used by the Pods and reconfigures the Deployment as needed. We could add a HorizontalPodAutoscaler to our Deployment using the `kubectl autoscale` command, but using a YAML resource file so that we can keep the autoscale configuration under version control is better. Here's the YAML file:

nginx-scaler.yaml

```
---
❶ apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
  labels:
    app: nginx
spec:
❷ scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
❸ minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: ❹ 50
```

In the `metadata` field, we add the label `app: nginx`. This does not change the behavior of the resource; its only purpose is to ensure that this resource shows up if we use an `app=nginx` label selector in a `kubectl get` command. This style of tagging the components of an application with consistent metadata is a good practice to help others understand what resources go together and to make debugging easier.

This YAML configuration uses version 2 of the autoscaler configuration ❶. Providing new versions of API resource groups is how Kubernetes accommodates future capability without losing any of its backward compatibility. Generally, alpha and beta versions are released for new resource groups before the final configuration is released, and there is at least one version of overlap between the beta version and the final release to enable seamless upgrades.

Version 2 of the autoscaler supports multiple resources. Each resource is used to calculate a vote on the desired number of Pods, and the largest number wins. Adding support for multiple resources requires a change in the YAML layout, which is a common reason for the Kubernetes maintainers to create a new resource version.

We specify our NGINX Deployment ❷ as the target for the autoscaler using its API resource group, kind, and name, which is enough to uniquely identify any resource in a Kubernetes cluster. We then tell the autoscaler to monitor the CPU utilization of the Pods that belong to the Deployment ❹. The autoscaler will work to keep average CPU utilization by the Pods close to 50 percent over the long run, scaling up or down as necessary. However, the number of replicas will never go beyond the range we specify ❸.

Let's create our autoscaler using this configuration:

```
root@host01:~# kubectl apply -f /opt/nginx-scaler.yaml
horizontalpodautoscaler.autoscaling/nginx created
```

We can query the cluster to see that it was created:

```
root@host01:~# kubectl get hpa
NAME    REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
nginx   Deployment/nginx   0%/50%    1         10        3          96s
```

The output shows the autoscaler's target reference, the current and desired resource utilization, and the maximum, minimum, and current number of replicas.

We use `hpa` as an abbreviation for `horizontalpodautoscaler`. Kubernetes allows us to use either singular or plural names and provides abbreviations for most of its resources to save typing. For example, we can type `deploy` for `deployment` and even `po` for `pods`. Every extra keystroke counts!

The autoscaler uses CPU utilization data that the `kubelet` is already collecting from the container engine. This data is centralized by the metrics server we installed as a cluster add-on. Without that cluster add-on, there would be no utilization data, and the autoscaler would not make any changes to the Deployment. In this case, because we're not really using our NGINX server instances, they aren't consuming any CPU, and the Deployment is scaled down to a single Pod, the minimum we specified:

```
root@host01:~# kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
nginx-6799fc88d8-dcwx5   1/1     Running   0          15m
```

The autoscaler has calculated that only one Pod is needed and has scaled the Deployment to match. The Deployment then selected a Pod to terminate to reach the desired scale.

For accuracy, the autoscaler will not use CPU data from the Pod if it recently started running, and it has logic to prevent it from scaling up or down too often, so if you ran through these examples very quickly you might need to wait a few minutes before you see it scale.

We explore Kubernetes resource utilization metrics in more detail when we look at limiting resource usage in Chapter 14.

Other Controllers

Deployments are the most generic and commonly used controller, but Kubernetes has some other useful options. In this section, we explore *Job*s and *CronJob*s, *StatefulSets*, and *DaemonSets*.

Jobs and CronJobs

Deployments are great for application components because we usually want one or more instances to stay running indefinitely. However, for cases for which we need to run a command, either once or on a schedule, we can use a Job. The primary difference is a Deployment ensures that any container that stops running is restarted, whereas a Job can check the exit code of the main process and restart only if the exit code is non-zero, indicating failure.

A Job definition looks very similar to a Deployment:

sleep-job.yaml

```
---
apiVersion: batch/v1
kind: Job
metadata:
  name: sleep
spec:
  template:
    spec:
      containers:
      - name: sleep
        image: busybox
        command:
          - "/bin/sleep"
          - "30"
      restartPolicy: OnFailure
```

The `restartPolicy` can be set to `OnFailure`, in which case the container will be restarted for a non-zero exit code, or to `Never`, in which case the Job will be completed when the container exits regardless of the exit code.

We can create and view the Job and the Pod it has created:

```
root@host01:~# kubectl apply -f /opt/sleep-job.yaml
job.batch/sleep created
root@host01:~# kubectl get job
NAME    COMPLETIONS   DURATION   AGE
sleep   0/1           3s         3s
root@host01:~# kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
...
sleep-fgcnz              1/1     Running   0          10s
```

The Job has created a Pod per the specification provided in the YAML file. The Job reflects `0/1` completions because it is waiting for its Pod to exit successfully.

When the Pod has been running for 30 seconds, it exits with a code of zero, indicating success, and the Job and Pod status are updated accordingly:

```
root@host01:~# kubectl get jobs
NAME    COMPLETIONS   DURATION   AGE
sleep   1/1           31s        40s
root@host01:~# kubectl get pods
NAME                     READY   STATUS      RESTARTS   AGE
nginx-65db7cf9c9-2wcng   1/1     Running     0          31m
sleep-fgcnz              0/1     Completed   0          43s
```

The Pod is still available, which means that we could review its logs if desired, but it shows a status of `Completed`, so Kubernetes will not try to restart the exited container.

A CronJob is a controller that creates Jobs on a schedule. For example, we could set up our sleep Job to run once per day:

sleep-cronjob.yaml

```
---
apiVersion: batch/v1
kind: CronJob
metadata:
  name: sleep
spec:
❶ schedule: "0 3 * * *"
❷ jobTemplate:
  spec:
    template:
      spec:
        containers:
          - name: sleep
            image: busybox
            command:
              - "/bin/sleep"
              - "30"
        restartPolicy: OnFailure
```

The entire contents of the Job specification are embedded inside the `jobTemplate` field ❷. To this, we add a `schedule` ❶ that follows the standard format for the Unix `cron` command. In this case, `0 3 * * *` indicates that a Job should be created at 3:00 AM every day.

One of Kubernetes' design principles is that anything could go down at any time. For a CronJob, if the cluster has an issue during the time the Job would be scheduled, the Job might not be scheduled, or it might be scheduled twice, this means that you should take care to write Jobs in an idempotent way so that they can handle missing or duplicated scheduling.

If we create this CronJob

```
root@host01:~# kubectl apply -f /opt/sleep-cronjob.yaml
cronjob.batch/sleep created
```

it now exists in the cluster, but it does not immediately create a Job or a Pod:

```
root@host01:~# kubectl get jobs
NAME    COMPLETIONS   DURATION   AGE
sleep   1/1           31s        2m32s
root@host01:~# kubectl get pods
NAME                       READY   STATUS      RESTARTS   AGE
nginx-65db7cf9c9-2wcng     1/1     Running     0          33m
sleep-fgcnz                0/1     Completed   0          2m23s
```

Instead, the CronJob will create a new Job each time its schedule is triggered.

StatefulSets

So far, we've been looking at controllers that create identical Pods. With both Deployments and Jobs, we don't really care which Pod is which, or where it is deployed, as long as we run enough instances at the right time. However, that doesn't always match the behavior we want. For example, even though a Deployment can create Pods with persistent storage, the storage must either be brand new for each new Pod, or the same storage must be shared across all Pods. That doesn't align well with a "primary and secondary" architecture such as a database. For those cases, we want specific storage to be attached to specific Pods.

At the same time, because Pods can come and go due to hardware failures or upgrades, we need a way to manage the replacement of a Pod so that each Pod is attached to the right storage. This is the purpose of a *StatefulSet*. A StatefulSet identifies each Pod with a number, starting at zero, and each Pod receives matching persistent storage. When a Pod must be replaced, the new Pod is assigned the same numeric identifier and is attached to the same storage. Pods can look at their hostname to determine their identifier, so a StatefulSet is useful both for cases with a fixed primary instance as well as cases for which a primary instance is dynamically chosen.

We'll explore a lot more details related to Kubernetes StatefulSets in the next several chapters, including persistent storage and Services. For this

chapter, we'll look at a basic example of a StatefulSet and then build on it as we introduce other important concepts.

For this simple example, let's create two Pods and show how they each get unique storage that stays in place even if the Pod is replaced. We'll use this YAML resource:

sleep-set.yaml

```
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
   name: sleep
spec:
❶ serviceName: sleep
  replicas: 2
  selector:
    matchLabels:
      app: sleep
  template:
    metadata:
      labels:
        app: sleep
    spec:
      containers:
        - name: sleep
          image: busybox
          command:
            - "/bin/sleep"
            - "3600"
       ❷ volumeMounts:
            - name: sleep-volume
              mountPath: /storagedir
❸ volumeClaimTemplates:
    - metadata:
        name: sleep-volume
      spec:
        storageClassName: longhorn
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 10Mi
```

There are a few important differences here compared to a Deployment or a Job. First, we must declare a `serviceName` to tie this StatefulSet to a Kubernetes Service ❶. This connection is used to create a Domain Name Service (DNS) entry for each Pod. We must also provide a template for the StatefulSet to use to request persistent storage ❸ and then tell Kubernetes where to mount that storage in our container ❷.

The actual *sleep-set.yaml* file that the automation scripts install includes the `sleep` Service definition. We cover Services in detail in Chapter 9.

Let's create the `sleep` StatefulSet:

```
root@host01:~# kubectl apply -f /opt/sleep-set.yaml
```

The StatefulSet creates two Pods:

```
root@host01:~# kubectl get statefulsets
NAME    READY   AGE
sleep   2/2     1m14s
root@host01:~# kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
...
sleep-0   1/1     Running   0          57s
sleep-1   1/1     Running   0          32s
```

The persistent storage for each Pod is brand new, so it starts empty. Let's create some content. The easiest way to do that is from within the container itself, using `kubectl exec`, which allows us to run commands inside a container, similar to `crictl`. The `kubectl exec` command works no matter what host the container is on, even if we're connecting to our Kubernetes API server from outside the cluster.

Let's write each container's hostname to a file and print it out so that we can verify it worked:

```
root@host01:~# kubectl exec sleep-0 -- /bin/sh -c \
  'hostname > /storagedir/myhost'
root@host01:~# kubectl exec sleep-0 -- /bin/cat /storagedir/myhost
sleep-0
root@host01:~# kubectl exec sleep-1 -- /bin/sh -c \
  'hostname > /storagedir/myhost'
root@host01:~# kubectl exec sleep-1 -- /bin/cat /storagedir/myhost
sleep-1
```

Each of our Pods now has unique content in its persistent storage. Let's delete one of the Pods and verify that its replacement inherits its predecessor's storage:

```
root@host01:~# kubectl delete pod sleep-0
pod "sleep-0" deleted
root@host01:~# kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
...
sleep-0   1/1     Running   0          28s
sleep-1   1/1     Running   0          8m18s
root@host01:~# kubectl exec sleep-0 -- /bin/cat /storagedir/myhost
sleep-0
```

After deleting `sleep-0`, we see a new Pod created with the same name, which is different from the Deployment for which a random name was generated for every new Pod. Additionally, for this new Pod, the file we created previously is still present because the StatefulSet attached the same persistent storage to the new Pod it created when the old one was deleted.

Daemon Sets

The *DaemonSet* controller is like a StatefulSet in that the DaemonSet also runs a specific number of Pods, each with a unique identity. However, the DaemonSet runs exactly one Pod per node, which is useful primarily for control plane and add-on components for a cluster, such as a network or storage plug-in.

Our cluster already has multiple DaemonSets installed, so let's look at the `calico-node` DaemonSet that's already running, which runs on each node to provide network configuration for all containers on that node.

The `calico-node` DaemonSet is in the `calico-system` Namespace, so we'll specify that Namespace to request information about the DaemonSet:

```
root@host01:~# kubectl -n calico-system get daemonsets
NAME          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   ...
calico-node   3         3         3       3            3           ...
```

Our cluster has three nodes, so the `calico-node` DaemonSet has created three instances. Here's the configuration of this DaemonSet in YAML format:

```
root@host01:~# kubectl -n calico-system get daemonset calico-node -o yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
...
  name: calico-node
  namespace: calico-system
...
spec:
...
  selector:
    matchLabels:
      k8s-app: calico-node
...
```

The `-o yaml` parameter to `kubectl get` prints out the configuration and status of one or more resources in YAML format, allowing us to inspect Kubernetes resources in detail.

The selector for this DaemonSet expects a label called `k8s-app` to be set to `calico-node`. We can use this to show just the Pods that this DaemonSet creates:

```
root@host01:~# kubectl -n calico-system get pods \
  -l k8s-app=calico-node -o wide
NAME                READY   STATUS    ... NODE   ...
calico-node-h9kjh   1/1     Running   ... host01 ...
calico-node-rcfk7   1/1     Running   ... host03 ...
calico-node-wj876   1/1     Running   ... host02 ...
```

The DaemonSet has created three Pods, each of which is assigned to one of the nodes in our cluster. If we add additional nodes to our cluster, the DaemonSet will schedule a Pod on the new nodes as well.

Final Thoughts

This chapter explored Kubernetes from the perspective of a regular cluster user, creating controllers that in turn create Pods with containers. Having this core knowledge of controller resource types is essential for building our applications. At the same time, it's important to remember that Kubernetes is using the container technology we explored in Part I.

One key aspect of container technology is the ability to isolate containers in separate network namespaces. Running containers in a Kubernetes cluster adds additional requirements for networking because we now need to connect containers running on different cluster nodes. In the next chapter, we consider multiple approaches to make this work as we look at overlay networks.