CHAPTER 4

Working with Kubernetes Objects

> I can't understand why people are frightened of new ideas. I'm frightened of the old ones.
>
> —John Cage

In Chapter 2, you built and deployed an application to Kubernetes. In this chapter, you'll learn about the fundamental Kubernetes objects involved in that process: Pods, Deployments, and Services. You'll also find out how to use the essential Helm tool to manage applications in Kubernetes.

After working through the example in "Running the Demo App" on page 30, you should have a container image running in the Kubernetes cluster, but how does that actually work? Under the hood, the `kubectl run` command creates a Kubernetes resource called a Deployment. So what's that? And how does a Deployment actually run your container image?

Deployments

Think back to how you ran the demo app with Docker. The `docker container run` command started the container, and it ran until you killed it with `docker stop`.

But suppose that the container exits for some other reason: maybe the program crashed, or there was a system error, or your machine ran out of disk space, or a cosmic ray hit your CPU at the wrong moment (unlikely, but it does happen). Assuming this is a production application, that means you now have unhappy users, until someone can get to a terminal and type `docker container run` to start the container again.

That's an unsatisfactory arrangement. What you really want is a kind of supervisor program that continually checks that the container is running, and, if it ever stops,

starts it again immediately. On traditional servers, you can use a tool like `systemd`, `runit`, or `supervisord` to do this; Docker has something similar, and you won't be surprised to know that Kubernetes has a supervisor feature too: the Deployment.

Supervising and Scheduling

For each program that Kubernetes has to supervise, it creates a corresponding Deployment object, which records some information about the program: the name of the container image, the number of replicas you want to run, and whatever else it needs to know to start the container.

Working together with the Deployment resource is a kind of Kubernetes component called a *controller*. Controllers are basically pieces of code that run continuously in a loop, and watch the resources that they're responsible for, making sure they're present and working. If a given Deployment isn't running enough replicas, for whatever reason, the controller will create some new ones. (If there were too many replicas for some reason, the controller would shut down the excess ones. Either way, the controller makes sure that the real state matches the desired state.)

Actually, a Deployment doesn't manage replicas directly: instead, it automatically creates an associated object called a ReplicaSet, which handles that. We'll talk more about ReplicaSets in a moment in "ReplicaSets" on page 56, but since you generally interact only with Deployments, let's get more familiar with them first.

Restarting Containers

At first sight, the way Deployments behave might be a little surprising. If your container finishes its work and exits, the Deployment will restart it. If it crashes, or if you kill it with a signal, or terminate it with `kubectl`, the Deployment will restart it. (This is how you should think about it conceptually; the reality is a little more complicated, as we'll see.)

Most Kubernetes applications are designed to be long-running and reliable, so this behavior makes sense: containers can exit for all sorts of reasons, and in most cases all a human operator would do is restart them, so that's what Kubernetes does by default.

It's possible to change this policy for an individual container: for example, to never restart it, or to restart it only on failure, not if it exited normally (see "Restart Policies" on page 151). However, the default behavior (restart always) is usually what you want.

A Deployment's job is to watch its associated containers and make sure that the specified number of them is always running. If there are fewer, it will start more. If there are too many, it will terminate some. This is much more powerful and flexible than a traditional supervisor-type program.

Creating Deployments

Go ahead and create a Deployment using our demo container image in your local Kubernetes environment so we can dive into how they work:

```
kubectl create deployment demo --image=cloudnatived/demo:hello
deployment.apps/demo created
```

You can see all the Deployments active in your current *namespace* (see "Using Namespaces" on page 78) by running the following command:

```
kubectl get deployments
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
demo   1/1     1            1           37s
```

To get more detailed information on this specific Deployment, run the following command:

```
kubectl describe deployments/demo
Name:                   demo
Namespace:              default
...
Labels:                 app=demo
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=demo
...
```

As you can see, there's a lot of information here, most of which isn't important for now. Let's look more closely at the `Pod Template` section, though:

```
Pod Template:
  Labels:  app=demo
  Containers:
   demo:
    Image:        cloudnatived/demo:hello
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
...
```

You know that a Deployment contains the information Kubernetes needs to run the container, and here it is. But what's a Pod Template? Actually, before we answer that, what's a Pod?

Pods

A *Pod* is the Kubernetes object that represents a group of one or more containers (*pod* is also the name for a group of whales, which fits in with the vaguely seafaring flavor of Kubernetes metaphors).

Why doesn't a Deployment just manage an individual container directly? The answer is that sometimes a set of containers needs to be scheduled together, running on the same node, and communicating locally, perhaps sharing storage. This is where Kubernetes starts to grow beyond simply running containers directly on a host using something like Docker. It manages entire combinations of containers, their configuration, and storage, etc. across a cluster of nodes.

For example, a blog application might have one container that syncs content with a Git repository, and an NGINX web server container that serves the blog content to users. Since they share data, the two containers need to be scheduled together in a Pod. In practice, though, many Pods only have one container, as in this case. (See "What Belongs in a Pod?" on page 138 for more about this.)

So a Pod specification (*spec* for short) has a list of `containers`, and in our example there is only one container, `demo`:

```
demo:
 Image:         cloudnatived/demo:hello
```

The `Image` spec is our demo Docker container image from Docker Hub, which is all the information a Kubernetes Deployment needs to start the Pod and keep it running.

And that's an important point. The `kubectl create deployment` command didn't actually create the Pod directly. Instead it created a Deployment, and *then* the Deployment created a ReplicaSet, which created the Pod. The Deployment is a declaration of your desired state: "A Pod should be running with the `demo` container inside it."

ReplicaSets

Deployments don't manage Pods directly. That's the job of the ReplicaSet object.

A ReplicaSet is responsible for a group of identical Pods, or *replicas*. If there are too few (or too many) Pods, compared to the specification, the ReplicaSet controller will start (or stop) some Pods to rectify the situation.

Deployments, in turn, manage ReplicaSets, and control how the replicas behave when you update them—by rolling out a new version of your application, for example (see "Deployment Strategies" on page 244). When you update the Deployment, a new ReplicaSet is created to manage the new Pods, and when the update is completed, the old ReplicaSet and its Pods are terminated.

In Figure 4-1, each ReplicaSet (V1, V2, V3) represents a different version of the application, with its corresponding Pods.

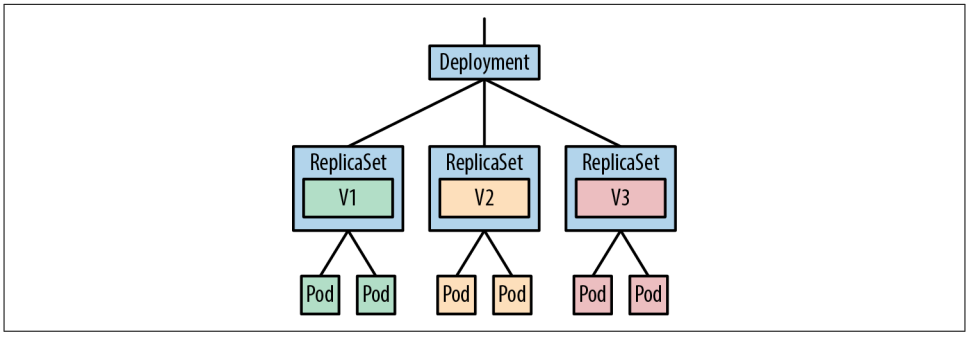

Figure 4-1. Deployments, ReplicaSets, and Pods

Usually, you won't interact with ReplicaSets directly, since Deployments do the work for you—but it's useful to know what they are.

Maintaining Desired State

Kubernetes controllers continually check the desired state specified by each resource against the actual state of the cluster, and make any necessary adjustments to keep them in sync. This process is called the *reconciliation loop*, because it loops forever, trying to reconcile the actual state with the desired state.

For example, when you first create the `demo` Deployment, there is no `demo` Pod running. So Kubernetes will start the required Pod immediately. If it ever stops, Kubernetes will start it again, so long as the Deployment still exists.

Let's verify that right now by removing the Pod manually. First, check that the Pod is indeed running:

```
kubectl get pods --selector app=demo
NAME                    READY   STATUS    RESTARTS   AGE
demo-794889fc8d-5ddsg   1/1     Running   0          33s
```

Note that the name of the Pod will be unique for you. You can also see the ReplicaSet that created this Pod by running:

```
kubectl get replicaset --selector app=demo
NAME              DESIRED   CURRENT   READY   AGE
demo-794889fc8d   1         1         1       64s
```

See how the ReplicaSet has a randomly generated ID that matches the beginning part of the demo Pod name above? In this example, the `demo-794889fc8d` ReplicaSet created one Pod named `demo-794889fc8d-5ddsg`.

Now, run the following command to remove the Pod:

```
kubectl delete pods --selector app=demo
pod "demo-794889fc8d-bdbcp" deleted
```

List the Pods again:

```
kubectl get pods --selector app=demo
NAME                    READY     STATUS        RESTARTS   AGE
demo-794889fc8d-qbcxm   1/1       Running       0          5s
demo-794889fc8d-bdbcp   0/1       Terminating   0          1h
```

You may catch the original Pod shutting down (its status is `Terminating`), but it's already been replaced by a new Pod, which is only five seconds old. You can also see that the new Pod has the same ReplicaSet, `demo-794889fc8d`, but a new unique Pod name `demo-794889fc8d-qbcxm`. That's the reconciliation loop at work.

You told Kubernetes, by means of the Deployment you created, that the `demo` Pod should *always* be running one replica. It takes you at your word, and even if you delete the Pod yourself, Kubernetes assumes you must have made a mistake, and helpfully starts a new Pod to replace it for you.

Once you've finished experimenting with the Deployment, shut it down and clean up using the following command:

```
kubectl delete all --selector app=demo
pod "demo-794889fc8d-qbcxm" deleted
deployment.apps "demo" deleted
replicaset.apps "demo-794889fc8d" deleted
```

The Kubernetes Scheduler

We've said things like *the Deployment will create Pods* and *Kubernetes will start the required Pod*, without really explaining how that happens.

The Kubernetes *scheduler* is the component responsible for this part of the process. When a Deployment (via its associated ReplicaSet) decides that a new replica is needed, it creates a Pod resource in the Kubernetes database. Simultaneously, this Pod is added to a queue, which is like the scheduler's inbox.

The scheduler's job is to watch its queue of unscheduled Pods, grab the next Pod from it, and find a node to run it on. It will use a few different criteria, including the Pod's resource requests, to choose a suitable node, assuming there is one available (we'll talk more about this process in Chapter 5).

Once the Pod has been scheduled on a node, the kubelet running on that node picks it up and takes care of actually starting its containers (see "Node Components" on page 35).

When you deleted a Pod in "Maintaining Desired State" on page 57, it was the ReplicaSet that spotted this and started a replacement. It *knows* that a `demo` Pod should be running on its node, and if it doesn't find one, it will start one. (What

would happen if you shut the node down altogether? Its Pods would become unscheduled and go back into the scheduler's queue, to be reassigned to other nodes.)

Stripe engineer Julia Evans has written a delightfully clear explanation of how scheduling works in Kubernetes.

Resource Manifests in YAML Format

Now that you know how to run an application in Kubernetes, is that it? Are you done? Not quite. Using the `kubectl create` command to create a Deployment is useful, but limited. Suppose that you want to change something about the Deployment spec: the image name or version, say. You could delete the existing Deployment (using `kubectl delete`) and create a new one with the right fields. But let's see if we can do better.

Because Kubernetes is inherently a *declarative* system, continuously reconciling actual state with desired state, all you need to do is change the desired state—the Deployment spec—and Kubernetes will do the rest. How do you do that?

Resources Are Data

All Kubernetes resources, such as Deployments or Pods, are represented by records in its internal database. The reconciliation loop watches the database for any changes to those records, and takes the appropriate action. In fact, all the `kubectl create` command does is add a new record in the database corresponding to a Deployment, and Kubernetes does the rest.

But you don't need to use `kubectl create` in order to interact with Kubernetes. You can also create and edit the resource *manifest* (the specification for the desired state of the resource) directly. You can (and should) keep the manifest file in a version control system, and instead of running imperative commands to make on-the-fly changes, you can change your manifest files and then tell Kubernetes to read the updated data.

Deployment Manifests

The usual format for Kubernetes manifest files is YAML, although it can also understand the JSON format. So what does the YAML manifest for a Deployment look like?

Have a look at our example for the demo application (*hello-k8s/k8s/deployment.yaml*):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo
  labels:
    app: demo
spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - name: demo
          image: cloudnatived/demo:hello
          ports:
          - containerPort: 8888
```

At first glance, this looks complicated, but it's mostly boilerplate. The only interesting parts are the same information that you've already seen in various forms: the container image name and port. When you gave this information to `kubectl create` earlier, it created the equivalent of this YAML manifest behind the scenes and submitted it to Kubernetes.

Using kubectl apply

To use the full power of Kubernetes as a declarative infrastructure as code system, submit YAML manifests to the cluster yourself, using the `kubectl apply` command.

Try it with our example Deployment manifest, *hello-k8s/k8s/deployment.yaml* in the demo repository.[1]

Run the following commands in your cloned copy of the demo repo:

```
cd hello-k8s
kubectl apply -f k8s/deployment.yaml
deployment.apps "demo" created
```

After a few seconds, a `demo` Pod should be running:

```
kubectl get pods --selector app=demo
NAME                    READY   STATUS    RESTARTS   AGE
demo-c77cc8d6f-nc6fm    1/1     Running   0          13s
```

We're not quite done, though, because in order to connect to the `demo` Pod with a web browser, we're going to create a Service, which is a Kubernetes resource that lets you connect to your deployed Pods (more on this in a moment).

First, let's explore what a Service is, and why we need one.

1 *k8s*, pronounced *kates*, is a common abbreviation for *Kubernetes*, following the geeky pattern of abbreviating words as a *numeronym*: their first and last letters, plus the number of letters in between (*k-8-s*). See also *i18n* (internationalization), *a11y* (accessibility), and *o11y* (observability).

Service Resources

Suppose you want to make a network connection to a Pod (such as our example application). How do you do that? You could find out the Pod's IP address and connect directly to that address and the app's port number. But the IP address may change when the Pod is restarted, so you'll have to keep looking it up to make sure it's up-to-date.

Worse, there may be multiple replicas of the Pod, each with different addresses. Every other application that needs to contact the Pod would have to maintain a list of those addresses, which doesn't sound like a great idea.

Fortunately, there's a better way: a Service resource gives you a single, unchanging IP address or DNS name that will be automatically routed to any matching Pod. Later on in "Ingress" on page 173, we will talk about the Ingress resource, which allows for more advanced routing and using TLS certificates.

But for now, let's take a closer look at how a Kubernetes Service works.

You can think of a Service as being like a web proxy or a load balancer, forwarding requests to a set of *backend* Pods (Figure 4-2). However, it isn't restricted to web ports: a Service can forward traffic from any port to any other port, as detailed in the `ports` part of the spec.

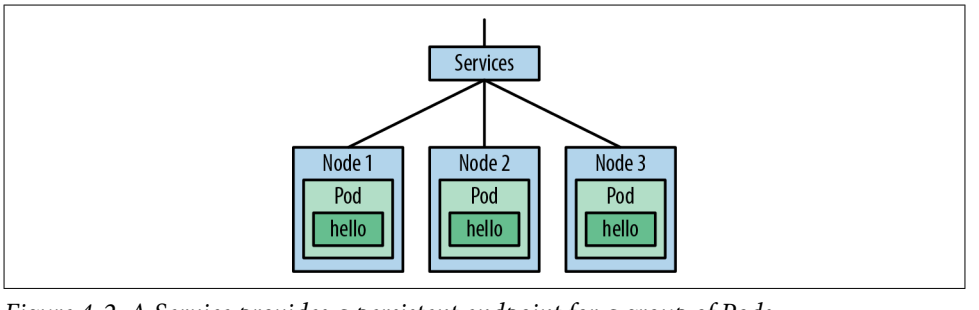

Figure 4-2. A Service provides a persistent endpoint for a group of Pods

Here's the YAML manifest of the Service for our demo app:

```
apiVersion: v1
kind: Service
metadata:
  name: demo
  labels:
    app: demo
spec:
  ports:
  - port: 8888
    protocol: TCP
    targetPort: 8888
```

```
  selector:
    app: demo
  type: ClusterIP
```

You can see that it looks somewhat similar to the Deployment resource we showed earlier. However, the `kind` is `Service`, instead of `Deployment`, and the `spec` just includes a list of `ports`, plus a `selector` and a `type`.

If you zoom in a little, you can see that the Service is forwarding its port 8888 to the Pod's port 8888:

```
...
ports:
- port: 8888
  protocol: TCP
  targetPort: 8888
```

The `selector` is the part that tells the Service how to route requests to particular Pods. Requests will be forwarded to any Pods matching the specified set of labels; in this case, just `app: demo` (see "Labels" on page 155). In our example, there's only one Pod that matches, but if there were multiple Pods, the Service would send each request to a randomly selected one.[2]

In this respect, a Kubernetes Service is a little like a traditional load balancer, and, in fact, both Services and Ingresses can automatically create cloud load balancers (see "Ingress" on page 173).

For now, the main thing to remember is that a Deployment manages a set of Pods for your application, and a Service gives you a single entry point for requests to those Pods.

Go ahead and apply the manifest now, to create the Service:

```
kubectl apply -f k8s/service.yaml
service "demo" created

kubectl port-forward service/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

As before, `kubectl port-forward` will connect the `demo` pod to a port on your local machine so that you can connect to *http://localhost:9999/* with your web browser.

Once you're satisfied that everything is working correctly, run the following command to clean up before moving on to the next section:

2 This is the default load-balancing algorithm; Kubernetes versions 1.10+ support other algorithms too, such as *least connection*. See the Kubernetes documentation for more.

```
kubectl delete -f k8s/
deployment.apps "demo" deleted
service "demo" deleted
```

You can use `kubectl delete` with a label selector, as we did earlier on, to delete all resources that match the selector (see "Labels" on page 155). Alternatively, you can use `kubectl delete -f`, as here, with a directory of manifests. All the resources described by the manifest files will be deleted.

Exercise

Modify the *k8s/deployment.yaml* file to change the number of replicas to 3. Reapply the manifest using `kubectl apply` and check that you get three `demo` Pods instead of one, using `kubectl get pods`.

Querying the Cluster with kubectl

The `kubectl` tool is the Swiss Army knife of Kubernetes: it applies configuration, creates, modifies, and destroys resources, and can also query the cluster for information about the resources that exist, as well as their status.

We've already seen how to use `kubectl get` to query Pods and Deployments. You can also use it to see what nodes exist in your cluster.

If you are running minikube, it should look something like this:

```
kubectl get nodes
NAME       STATUS   ROLES                  AGE   VERSION
minikube   Ready    control-plane,master   17d   v1.21.2
```

If you want to see resources of all types, use `kubectl get all`. (In fact, this doesn't show literally *all* resources, just the most common types, but we won't quibble about that for now.)

To see comprehensive information about an individual Pod (or any other resource), use `kubectl describe`:

```
kubectl describe pod/demo-dev-6c96484c48-69vss
Name:         demo-794889fc8d-7frgb
Namespace:    default
Priority:     0
Node:         minikube/192.168.49.2
Start Time:   Mon, 02 Aug 2021 13:21:25 -0700
...
Containers:
  demo:
```

```
    Container ID:   docker://646aaf7c4baf6d...
    Image:          cloudnatived/demo:hello
...
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
...
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  1d    default-scheduler  Successfully assigned demo-dev...
  Normal  Pulling    1d    kubelet            pulling image "cloudnatived/demo...
...
```

In the example output, you can see that `kubectl` gives you some basic information about the container itself, including its image identifier and status, along with an ordered list of events that have happened to the container. (We'll learn a lot more about the power of `kubectl` in Chapter 7.)

Taking Resources to the Next Level

You now know everything you need to know to deploy applications to Kubernetes clusters using declarative YAML manifests. But there's a lot of repetition in these files: for example, you've repeated the name `demo`, the label selector `app: demo`, and the port `8888` several times.

Shouldn't you be able to just specify those values once, and then reference them wherever they occur through the Kubernetes manifests?

For example, it would be great to be able to define variables called something like `container.name` and `container.port`, and then use them wherever they're needed in the YAML files. Then, if you needed to change the name of the app or the port number it listens on, you'd only have to change them in one place, and all the manifests would be updated automatically.

Fortunately, there's a tool for that, and in the final section of this chapter we'll show you a little of what it can do.

Helm: A Kubernetes Package Manager

One popular package manager for Kubernetes is called Helm, and it works just the way we've described in the previous section. You can use the `helm` command-line tool to install and configure applications (your own or anyone else's), and you can create packages called Helm *charts*, which completely specify the resources needed to run the application, its dependencies, and its configurable settings.

Helm is part of the Cloud Native Computing Foundation family of projects (see "Cloud Native" on page 15), which reflects its stability and widespread adoption.

It's important to realize that a Helm chart, unlike the binary software packages used by tools like APT or Yum, doesn't actually include the container image itself. Instead, it simply contains metadata about where the image can be found, just as a Kubernetes Deployment does.

When you install the chart, Kubernetes itself will locate and download the binary container image from the place you specified. In fact, a Helm chart is really just a convenient wrapper around Kubernetes YAML manifests.

Installing Helm

Follow the Helm installation instructions for your operating system.

To verify that Helm is installed and working, run:

```
helm version
version.BuildInfo{Version:"v3...GoVersion:"go1.16.5"}
```

Once this command succeeds, you're ready to start using Helm.

Installing a Helm Chart

What would the Helm chart for our demo application look like? In the *hello-helm3* directory, you'll see a *k8s* subdirectory, which in the previous example (`hello-k8s`) contained just the Kubernetes manifest files to deploy the application. Now it contains a Helm chart, in the *demo* directory:

```
ls k8s/demo
Chart.yaml  production-values.yaml  staging-values.yaml  templates  values.yaml
```

We'll see what all these files are for in "What's Inside a Helm Chart?" on page 222, but for now, let's use Helm to install the demo application. First, clean up the resources from any previous deployments:

```
kubectl delete all --selector app=demo
```

Then run the following command:

```
helm upgrade --install demo ./k8s/demo
NAME: demo
LAST DEPLOYED: Mon Aug  2 13:37:21 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

If you use your `kubectl get deployment` and `kubectl get service` commands that you learned earlier, you will see that Helm has created a Deployment resource (which starts a Pod) and a Service, just as in the previous examples. The `helm upgrade --install` command also creates a Kubernetes Secret with a Type of `helm.sh/release.v1` to track the release.

Charts, Repositories, and Releases

These are the three most important Helm terms you need to know:

- A *chart* is a Helm package, containing all the resource definitions necessary to run an application in Kubernetes.
- A *repository* is a place where charts can be collected and shared.
- A *release* is a particular instance of a chart running in a Kubernetes cluster.

There are some parallels with Helm resources to Docker containers:

- A Helm *repository* is a server where charts are stored and downloaded from clients, similar to how a container registry stores and serves container images, like Docker Hub.
- A Helm *release* is when a chart is installed into a cluster, much like when a published Docker image is launched as a running container.

Helm charts can be downloaded and installed from repository servers, or installed directly by pointing to a local path of a directory containing the Helm YAML files on the filesystem.

One chart can be installed many times into the same cluster. For example, you might be running multiple copies of the Redis chart for various applications, each serving as a backend for different websites. Each separate instance of the Helm chart is a distinct release.

You may also want to centrally install something in your cluster used by all of your apps, like Prometheus for centralized monitoring, or the NGINX Ingress Controller for handling incoming web requests.

Listing Helm Releases

To check what releases you have running at any time, run `helm list`:

```
helm list
NAME  NAMESPACE REVISION  UPDATED STATUS    CHART
demo  default   1         ...     deployed  demo-1.0.1
```

To see the exact status of a particular release, run `helm status` followed by the name of the release. You'll see the same information that you did when you first deployed the release.

Later in the book, we'll show you how to build your own Helm charts for your applications (see "What's Inside a Helm Chart?" on page 222). For now, just know that Helm is a handy way to install applications from public charts.

Many popular applications are hosted in various Helm repositories and maintained by the package providers. You can add Helm repositories and install their charts, and you can also host and publish your own Helm charts for your own applications.

You can see many examples of popular Helm charts hosted on Artifact Hub, another CNCF project.

Summary

This is primarily a book about using Kubernetes, not diving deep into the details of how Kubernetes works. Our aim is to show you what Kubernetes can *do*, and bring you quickly to the point where you can run real workloads in production. However, it's useful to know at least some of the main pieces of machinery you'll be working with, such as Pods and Deployments. In this chapter, we've briefly introduced some of the most important ones. We also recommend *Managing Kubernetes*, *Production Kubernetes*, and the *Kubernetes the Hard Way* repo for those looking to get more familiar with what is going on under the hood.

As fascinating as the technology is to geeks like us, we're also interested in getting stuff done. Therefore, we haven't exhaustively covered every kind of resource Kubernetes provides, because there are a *lot*, and many of them you almost certainly won't need (at least, not yet).

The key points we think you need to know right now are:

- The Pod is the fundamental unit of work in Kubernetes, specifying a single container or group of communicating containers that are scheduled together.
- A Deployment is a high-level Kubernetes resource that declaratively manages Pods, deploying, scheduling, updating, and restarting them when necessary.
- A Service is the Kubernetes equivalent of a load balancer or proxy, routing traffic to its matching Pods via a single, well-known, durable IP address or DNS name.
- The Kubernetes scheduler watches for a Pod that isn't yet running on any node, finds a suitable node for it, and instructs the kubelet on that node to run the Pod.
- Resources like Deployments are represented by records in Kubernetes's internal database. Externally, these resources can be represented by text files (known as *manifests*) in YAML format. The manifest is a declaration of the desired state of the resource.
- `kubectl` is the main tool for interacting with Kubernetes, allowing you to apply manifests, query resources, make changes, delete resources, and do many other tasks.
- Helm is a Kubernetes package manager. It simplifies configuring and deploying Kubernetes applications, allowing you to use a single set of bundled manifests and templates used to generate parameterized Kubernetes YAML files, instead of having to maintain the raw YAML files yourself.