2

An Overview of Kubernetes

Overview

In this chapter, we will have our first hands-on introduction to Kubernetes. This chapter will give you a brief overview of the different components of Kubernetes and how they work together. We will also try our hand at working with some fundamental Kubernetes components.

By the end of this chapter, you will have a single-node Minikube environment set up where you can run many of the exercises and activities in this book. You will be able to understand the high-level architecture of Kubernetes and identify the roles of the different components. You will also learn the basics required to migrate containerized applications to a Kubernetes environment.

Introduction

We ended the previous chapter by providing a brief and abstract introduction to Kubernetes, as well as some of its advantages. In this chapter, we will provide you with a much more concrete high-level understanding of how Kubernetes works. First, we will walk you through how to install Minikube, which is a handy tool that creates a single-node cluster and provides a convenient learning environment for Kubernetes. Then, we will take a 30,000-foot overview of all the components, including their responsibilities and how they interact with each other. After that, we will migrate the Docker application that we built in the previous chapter to Kubernetes and illustrate how it can enjoy the benefits afforded by Kubernetes, such as creating multiple replicas, and version updates. Finally, we will explain how the application responds to external and internal traffic.

Having an overview of Kubernetes is important before we dive deeper into the different aspects of it so that when we learn more specifics about the different aspects, you will have an idea of where they fit in the big picture. Also, when we go even further and explore how to use Kubernetes to deploy applications in a production environment, you will have an idea of how everything is taken care of in the background. This will also help you with optimization and troubleshooting.

Setting Up Kubernetes

Had you asked the question, "*How do you easily install Kubernetes?*" three years ago, it would have been hard to give a compelling answer. Embarrassing, but true. Kubernetes is a sophisticated system, and getting it installed and managing it well isn't an easy task.

However, as the Kubernetes community has expanded and matured, more and more user-friendly tools have emerged. As of today, based on your requirements, there are a lot of options to choose from:

- If you are using physical (bare metal) servers or virtual machines (VMs), Kubeadm is a good fit.
- If you're running on cloud environments, Kops and Kubespray can ease Kubernetes installation, as well as integration with the cloud providers. In fact, we will teach you how to deploy Kubernetes on AWS using Kops in *Chapter 11*, *Build Your Own HA Cluster*, and we will take another look at the various options we can use to set up Kubernetes.

- If you want to drop the burden of managing the Kubernetes control plane (which we will learn about later in this chapter), almost all cloud providers have their own Kubernetes managed service, such as Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), and IBM Kubernetes Service (IKS).
- If you just want a playground to study Kubernetes in, Minikube and Kind can help you spin up a Kubernetes cluster in minutes.

We will use Minikube extensively throughout this book as a convenient learning environment. But before we proceed to the installation process, let's take a closer look at Minikube itself.

An Overview of Minikube

Minikube is a tool that can be used to set up a single-node cluster, and it provides handy commands and parameters to configure the cluster. It aims at simplifying the complexity of Kubernetes' installation. Unlike other tools, it packs a VM containing all the core components of Kubernetes that get installed onto your host machine, all at once. This allows it to support any operating system, as long as there is a virtualization tool (also known as a Hypervisor) pre-installed. The following are the most common Hypervisors supported by Minikube:

- VirtualBox (works for all operating systems)
- KVM (Linux-specific)
- Hyperkit (macOS-specific)
- Hyper-V (Windows-specific)

Regarding the required hardware resources, the minimum requirement is 2 GB RAM and any dual-core CPU that supports virtualization (Intel VT or AMD-V), but you will, of course, need a more powerful machine if you are trying out heavier workloads.

Just like any other modern software, Kubernetes provides a handy command-line client called Kubectl that allows users to interact with the cluster conveniently. In the next exercise, we will set up Minikube and use some basic Kubectl commands. We will go into more detail about Kubectl in the next chapter.

Exercise 2.01: Getting Started with Minikube and Kubernetes Clusters

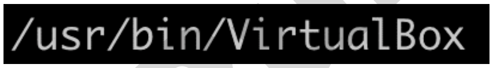
In this exercise, we will use Ubuntu 18.04 as the base operating system to install Minikube, using which we can start a single-node Kubernetes cluster easily. Once the Kubernetes cluster has been set up, you should be able to check its status and use **`kubectl`** to interact with it:

> **Note:**
>
> Since this exercise deals with software installations, you will need to be logged in as root/superuser. A simple way to switch to being a root user is to run the following command: **`sudo su -`**.
>
> In *step 9* of this exercise, we will create a regular user and then switch back to it.

1. First, ensure that VirtualBox is installed. You can confirm this by using the following command:

```
which VirtualBox
```

You should see the following output:



```
/usr/bin/VirtualBox
```

Figure 2.1: Path of the VirtualBox executable binary

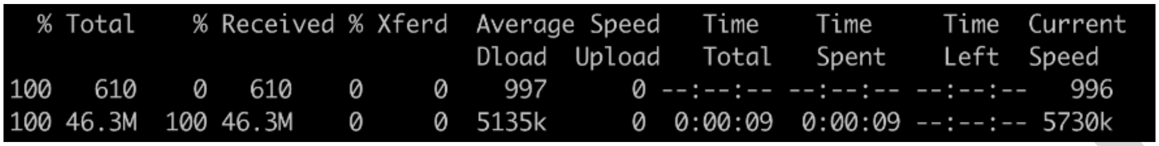
If VirtualBox has been successfully installed, the **`which`** command should show the path of the executable, as shown in the preceding screenshot. If not, then please ensure that you have installed VirtualBox as per the instructions provided in the *Preface*.

2. Download the Minikube standalone binary by using the following command:

```
curl -Lo minikube https://github.com/kubernetes/minikube/releases/
download/<version>/minikube-<ostype-arch> && chmod +x minikube
```

In this command, **`<version>`** should be replaced with a specific version, such as **`v1.5.2`** (which is the version we will use in this chapter) or **`latest`**. Depending on your host operating system, **`<ostype-arch>`** should be replaced with **`linux-amd64`** (for Ubuntu) or **`darwin-amd64`** (for macOS).

You should see the following output:



```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   610    0   610    0     0    997      0 --:--:-- --:--:-- --:--:--   996
100 46.3M  100 46.3M    0     0  5135k      0  0:00:09  0:00:09 --:--:-- 5730k
```

Figure 2.2: Downloading the Minikube binary

The preceding command contains two parts: the first command, **`curl`**, downloads the Minikube binary, while the second command, **`chmod`**, changes the permission to make it executable.

3. Move the binary into the system path (in the example, it's **`/usr/local/bin`**) so that we can directly run Minikube, regardless of which directory the command is run in:

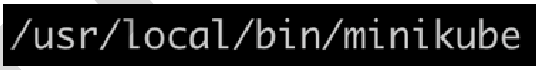
```
mv minikube /usr/local/bin
```

When executed successfully, the move (**`mv`**) command does not give a response in the Terminal.

4. After running the move command, we need to confirm that the Minikube executable is now in the correct location:

```
which minikube
```

You should see the following output:



```
/usr/local/bin/minikube
```

Figure 2.3: Path of the Minikube executable binary

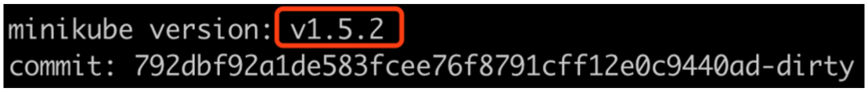
NOTE

If the **`which minikube`** command doesn't give you the expected result, you may need to explicitly add **`/usr/local/bin`** to your system path by running **`export PATH=$PATH:/usr/local/bin`**.

5. You can check the version of Minikube using the following command:

```
minikube version
```

You should see the following output:



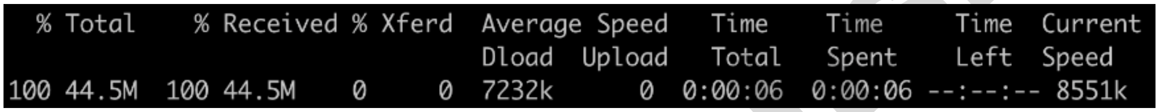
```
minikube version: v1.5.2
commit: 792dbf92a1de583fcee76f8791cff12e0c9440ad-dirty
```

Figure 2.4: Getting the version of Minikube

6. Now, let's download Kubectl version v1.16.2 (so that it's compatible with the version of Kubernetes that our setup of Minikube will create later) and make it executable by using the following command:

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.16.2/
bin/<ostype>/amd64/kubectl && chmod +x kubectl
```

You should see the following output:



```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 44.5M  100 44.5M    0     0  7232k      0  0:00:06  0:00:06 --:--:-- 8551k
```

Figure 2.5: Downloading the Kubectl binary

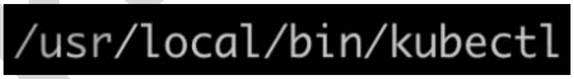
7. Then, move it to the system path, just like we did for the executable of Minikube earlier:

```
mv kubectl /usr/local/bin
```

8. Now, let's check whether the executable for Kubectl is at the correct path:

```
which kubectl
```

You should see the following response:



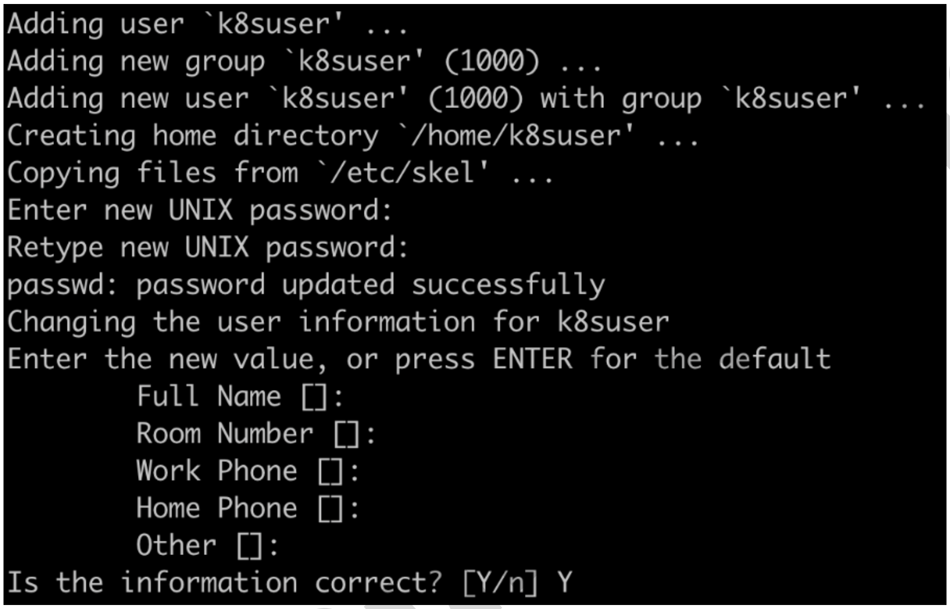
```
/usr/local/bin/kubectl
```

Figure 2.6: Path of the Kubectl binary

9. Since we are currently logged in as the **`root`** user, let's create a regular user called **`k8suser`** by running the following command:

```
useradd k8suser
```

Enter your desired password when you are prompted for it. You will also be prompted to enter other details, such as your full name. You may choose to skip those details by simply pressing *Enter*. You should see an output similar to the following:



```
Adding user `k8suser' ...
Adding new group `k8suser' (1000) ...
Adding new user `k8suser' (1000) with group `k8suser' ...
Creating home directory `/home/k8suser' ...
Copying files from `/etc/skel' ...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for k8suser
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] Y
```

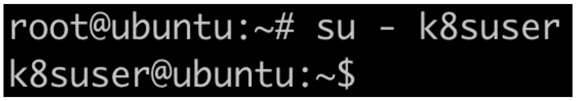
Figure 2.7: Creating a new Linux user

Enter **`Y`** and hit *Enter* to confirm the final prompt for creating a user, as shown at the end of the previous screenshot.

10. Now, switch user from **`root`** to **`k8suser`**:

```
su - k8suser
```

You should see the following output:



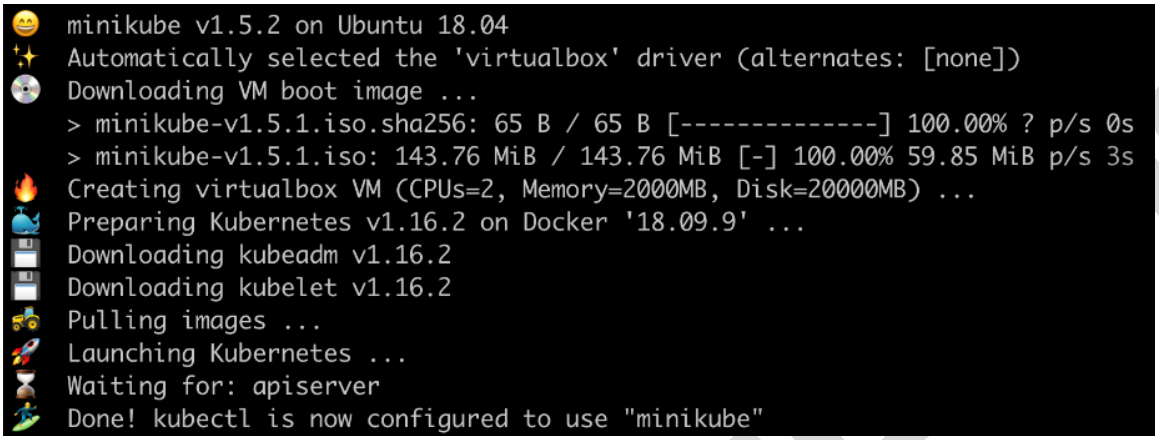
```
root@ubuntu:~# su - k8suser
k8suser@ubuntu:~$
```

Figure 2.8: Switching to a new Linux user

11. Now, we can create a Kubernetes cluster using **`minikube start`**:

```
minikube start --kubernetes-version=v1.16.2
```

It will take a few minutes to download the VM images and get everything set up. After Minikube has started up successfully, you should see a response that looks similar to the following:



```
😄  minikube v1.5.2 on Ubuntu 18.04
✨  Automatically selected the 'virtualbox' driver (alternates: [none])
💿  Downloading VM boot image ...
    > minikube-v1.5.1.iso.sha256: 65 B / 65 B [--------------] 100.00% ? p/s 0s
    > minikube-v1.5.1.iso: 143.76 MiB / 143.76 MiB [-] 100.00% 59.85 MiB p/s 3s
🔥  Creating virtualbox VM (CPUs=2, Memory=2000MB, Disk=20000MB) ...
🐳  Preparing Kubernetes v1.16.2 on Docker '18.09.9' ...
💾  Downloading kubeadm v1.16.2
💾  Downloading kubelet v1.16.2
🚜  Pulling images ...
🚀  Launching Kubernetes ...
⌛  Waiting for: apiserver
🏄  Done! kubectl is now configured to use "minikube"
```

Figure 2.9: Minikube first startup

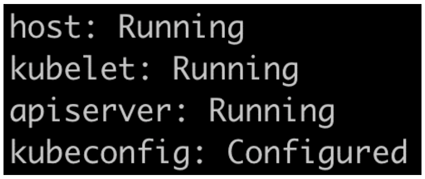
As we mentioned earlier, Minikube starts up a VM instance with all the components of Kubernetes inside it. By default, it uses VirtualBox, and you can use the **`--vm-driver`** flag to specify a particular hypervisor driver (such as **`hyperkit`** for macOS). Minikube also provides the **`--kubernetes-version`** flag so you can specify the Kubernetes version you want to use. If not specified, it will use the latest version that was available when the Minikube release was finalized. In this chapter, to ensure compatibility of the Kubernetes version with the Kubectl version, we have specified Kubernetes version **`v1.16.2`** explicitly.

The following commands should help establish that the Kubernetes cluster that was started by Minikube is running properly.

12. Use the following command to get the basic status of the various components of the cluster:

```
minikube status
```

You should see the following response:



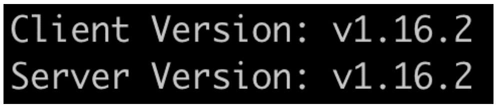
```
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Figure 2.10: Status of various components of Minikube

13. Now, let's look at the version of the Kubectl client and Kubernetes server:

```
kubectl version --short
```

You should see the following response:



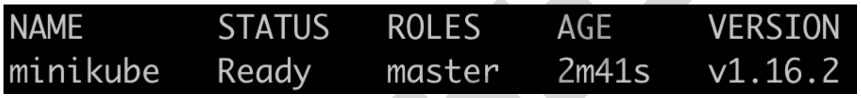
```
Client Version: v1.16.2
Server Version: v1.16.2
```

Figure 2.11: Getting the version of Minikube

14. Let's learn how many machines comprise the cluster and get some basic information about them:

```
kubectl get node
```

You should see a response similar to the following:



```
NAME       STATUS   ROLES    AGE     VERSION
minikube   Ready    master   2m41s   v1.16.2
```

Figure 2.12: Getting the list of nodes

After finishing this exercise, you should have Minikube set up with a single-node Kubernetes cluster. In the next section, we will enter the Minikube VM to take a look at how the cluster is composed and the various components of Kubernetes that make it work.

Kubernetes Components Overview

By completing the previous exercise, you have a single-node Kubernetes cluster up and running. Before playing your first concert, let's hold on a second and pull the curtains aside to take a look backstage to see how Kubernetes is architected behind the scenes, and then check how Minikube glues its various components together inside its VM.

Kubernetes has several core components that make the wheels of the machine turn. They are as follows:

- API server
- Etcd
- Controller manager
- Scheduler
- Kubelet

These components are critical for the functioning of a Kubernetes cluster.

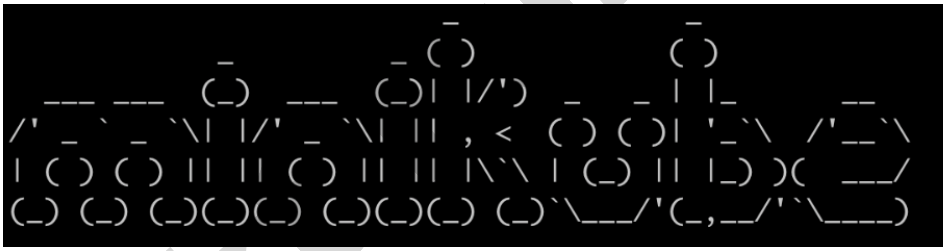
Besides these core components, you would deploy your applications in containers, which are bundled together as pods. We will learn more about pods in *Chapter* 5, *Pods*. These pods, and several other resources, are defined by something called API objects.

An **API object** describes how a certain resource should be honored in Kubernetes. We usually define API objects using a human-readable manifest file, and then use a tool (such as Kubectl) to parse it and hand it over to a Kubernetes API server. Kubernetes then tries to create the resource specified in the object and match its state to the desired state in the object definition, as mentioned in the manifest file. Next, we will walk you through how these components are organized and behave in a single-node cluster created by Minikube.

Minikube provides a command called **`minikube ssh`** that's used to gain SSH access from the host machine (in our machine, it's the physical machine running Ubuntu 18.04) to the **`minikube`** virtual machine, which serves as the sole node in our Kubernetes cluster. Let's see how that works:

```
minikube ssh
```

You will see the following output:



```
                         _             _
            _         _ ( )           ( )
  ___ ___  (_)  ___  (_)| |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\| || , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || || |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)
```

Figure 2.13: Accessing the Minikube VM via SSH

NOTE

All the commands that will be shown later in this section are presumed to have been run inside the Minikube VM, after running **`minikube ssh`**.

Container technology brings the convenience of encapsulating your application. Minikube is not exceptional – it leverages containers to glue the Kubernetes components together. In the Minikube VM, Docker is pre-installed so that it can manage the core Kubernetes components. You can take a look at this by running **`docker ps`**; however, the result may be overwhelming as it includes all the running containers – both the core Kubernetes components and add-ons, as well as all the columns – which will output a very large table.

To simplify the output and make it easier to read, we will pipe the output from **`docker ps`** into two other Bash commands:

1. **`grep -v pause`**: This will filter the results by not displaying the "sandbox" containers.

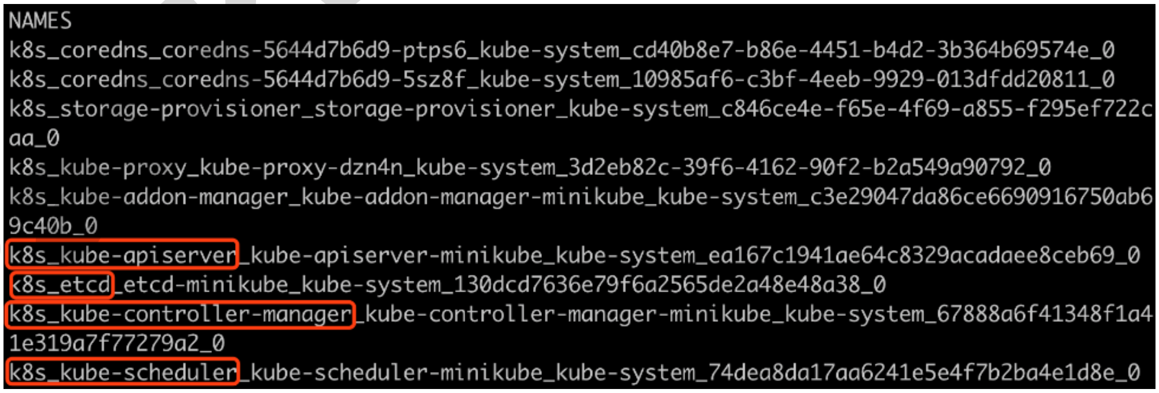
   Without **`grep -v pause`**, you would find that each container is "paired" with a "sandbox" container (in Kubernetes, it's implemented as a **`pause`** image). This is because, as mentioned in the previous chapter, Linux containers can be associated (or isolated) by joining the same (or different) Linux namespace. In Kubernetes, a "sandbox" container is used to bootstrap a Linux namespace, and then the containers that run the real application are able to join that namespace. Finer details about how all this works under the hood have been left out of scope for the sake of brevity.

2. **`awk '{print $NF}'`**: This will only print the last column with a container name.

Thus, the final command is as follows:

```
docker ps | grep -v pause | awk '{print $NF}'
```

You should see the following output:



```
NAMES
k8s_coredns_coredns-5644d7b6d9-ptps6_kube-system_cd40b8e7-b86e-4451-b4d2-3b364b69574e_0
k8s_coredns_coredns-5644d7b6d9-5sz8f_kube-system_10985af6-c3bf-4eeb-9929-013dfdd20811_0
k8s_storage-provisioner_storage-provisioner_kube-system_c846ce4e-f65e-4f69-a855-f295ef722c
aa_0
k8s_kube-proxy_kube-proxy-dzn4n_kube-system_3d2eb82c-39f6-4162-90f2-b2a549a90792_0
k8s_kube-addon-manager_kube-addon-manager-minikube_kube-system_c3e29047da86ce6690916750ab6
9c40b_0
k8s_kube-apiserver_kube-apiserver-minikube_kube-system_ea167c1941ae64c8329acadaee8ceb69_0
k8s_etcd_etcd-minikube_kube-system_130dcd7636e79f6a2565de2a48e48a38_0
k8s_kube-controller-manager_kube-controller-manager-minikube_kube-system_67888a6f41348f1a4
1e319a7f77279a2_0
k8s_kube-scheduler_kube-scheduler-minikube_kube-system_74dea8da17aa6241e5e4f7b2ba4e1d8e_0
```

Figure 2.14: Getting the list of containers by running the Minikube VM

The highlighted containers shown in the preceding screenshot are basically the core components of Kubernetes. We'll discuss each of these in detail in the following sections.

etcd

A distributed system may face various kinds of failures (network, storage, and so on) at any moment. To ensure it still works properly when failures arise, critical cluster metadata and state must be stored in a reliable way.

Kubernetes abstracts the cluster metadata and state as a series of API objects. For example, the node API object represents a Kubernetes worker node's specification, as well as its latest status.

Kubernetes uses **etcd** as the backend key-value database to persist the API objects during the life cycle of a Kubernetes cluster. It is important to note that nothing (internal cluster resources or external clients) is allowed to talk to etcd without going through the API server. Any updates to or requests from etcd are made only via calls to the API server.

In practice, etcd is usually deployed with multiple instances to ensure the data is persisted in a secure and fault-tolerant manner.

API Server

The API server allows standard APIs to access Kubernetes API objects. It is the only component that talks to backend storage (etcd).

Additionally, by leveraging the fact that it is the single point of contact for communicating to etcd, it provides a convenient interface for clients to "watch" any API objects that they may be interested in. Once the API object has been created, updated, or deleted, the client that is "watching" will get instant notifications so they can act upon those changes. The "watching" client is also known as the "controller," which has become a very popular entity that's used in both built-in Kubernetes objects and Kubernetes extensions.

> **Note:**
>
> You will learn more about the API server in *Chapter 4*, *How to Communicate with Kubernetes (API Server)*, and about controllers in *Chapter 7*, *Kubernetes Controllers*.

Scheduler

The scheduler is responsible for distributing the incoming workloads to the most suitable node. The decision regarding distribution is made by the scheduler's understanding of the whole cluster, as well as a series of scheduling algorithms.

> **Note**
>
> You will learn more about the scheduler in *Chapter 18, Advanced Scheduling in Kubernetes*.

Controller Manager

As we mentioned earlier in the *API Server* subsection, the API server exposes ways to "watch" almost any API object and notify the watchers about the changes in the API objects being watched.

It works pretty much like a subscriber-publisher pattern. The controller manager acts as a typical subscriber and watches the API objects that it is interested in, and then attempts to make appropriate changes to move the current state toward the desired state described in the object.

For example, if it gets an update from the API server saying that an application claims two replicas, but right now there is only one living in the cluster, it will create the second one to make the application adhere to its desired replica number. The reconciliation process keeps running across the controller manager's life cycle to ensure that all applications stay in their expected state.

The controller manager aggregates various kinds of controllers to honor the semantics of API objects, such as Deployments and Services, which we will introduce later in this chapter.

Where Is the Kubelet?

etcd, the API server, the scheduler, and the controller manager comprise the control plane of Kubernetes. A machine that runs these components is called a master node. The kubelet, on the other hand, is deployed on each worker machine.

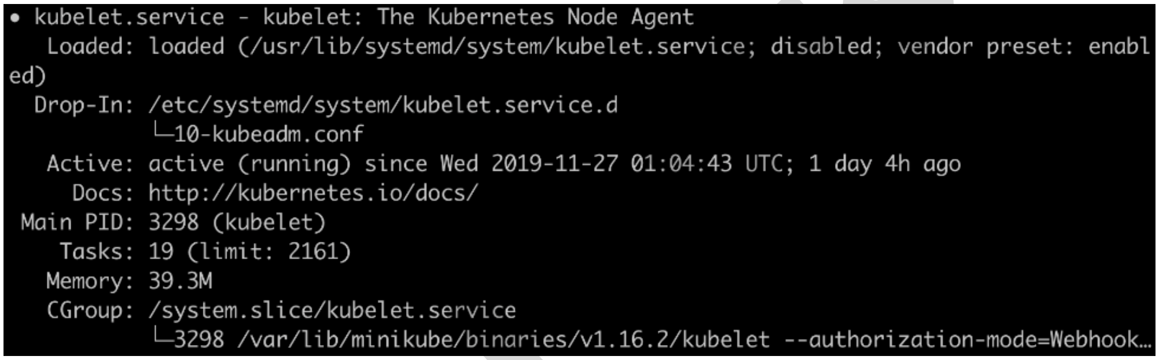
In our single-node Minikube cluster, the kubelet is deployed on the same node that carries the control plane components. However, in most production environments, it is not deployed on any of the master nodes. We will learn more about production environments when we deploy a multi-node cluster in *Chapter 11, Deployments and HA Kubernetes*.

The kubelet primarily aims at talking to the underlying container runtime (for example, Docker, containerd, or cri-o) to bring up the containers and ensure that the containers are running as expected. Also, it's responsible for sending the status update back to the API server.

However, as shown in the preceding screenshot, the **`docker ps`** command doesn't show anything named **`kubelet`**. To start, stop, or restart any software and make it auto-restart upon failure, usually, we need a tool to manage its life cycle. In Linux, systemd has that responsibility. In Minikube, the kubelet is managed by systemd and runs as a native binary instead of a Docker container. We can run the following command to check its status:

```
systemctl status kubelet
```

You should see an output similar to the following:



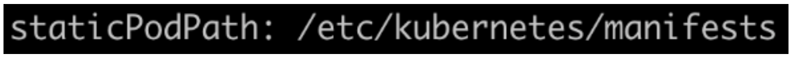
```
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; disabled; vendor preset: enabl
ed)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Wed 2019-11-27 01:04:43 UTC; 1 day 4h ago
     Docs: http://kubernetes.io/docs/
 Main PID: 3298 (kubelet)
    Tasks: 19 (limit: 2161)
   Memory: 39.3M
   CGroup: /system.slice/kubelet.service
           └─3298 /var/lib/minikube/binaries/v1.16.2/kubelet --authorization-mode=Webhook…
```

Figure 2.15: Status of kubelet

By default, the kubelet has the configuration for **`staticPodPath`** in its config file (which is stored at **`/var/lib/kubelet/config.yaml`**). Kubelet is instructed to continuously watch the changes in files under that path, and each file under that path represents a Kubernetes component. Let's understand what this means by first finding **`staticPodPath`** in the kubelet's config file:

```
grep "staticPodPath" /var/lib/kubelet/config.yaml
```

You should see the following output:



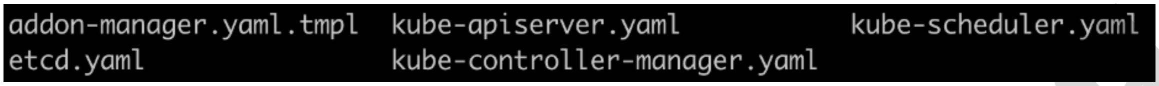
```
staticPodPath: /etc/kubernetes/manifests
```

Figure 2.16: Printing out the staticPodPath field from the manifest for kubelet

Now, let's see the contents of this path:

```
ls /etc/kubernetes/manifests
```

You should see the following output:



```
addon-manager.yaml.tmpl   kube-apiserver.yaml            kube-scheduler.yaml
etcd.yaml                 kube-controller-manager.yaml
```

Figure 2.17: Exploring the manifests of Kubernetes components

As shown in the list of files, the core components of Kubernetes are defined by objects that have a definition specified in YAML files. In the Minikube environment, in addition to managing the user-created pods, the kubelet also serves as a systemd equivalent in order to manage the life cycle of Kubernetes system-level components, such as the API server, the scheduler, the controller manager, and other add-ons. Once any of these YAML files is changed, the kubelet auto-detects that and updates the state of the cluster so that it matches the desired state defined in the updated YAML configuration.

We will stop here without diving deeper into the design of Minikube. In addition to "static components," the kubelet is also the manager of "regular applications" to ensure that they're running as expected on the node, and evicts pods according to the API specification or upon resource shortage.

kube-proxy

kube-proxy appears in the output of the **`docker ps`** command, but it was not present at **`/etc/kubernetes/manifests`** when we explored that directory in the previous subsection. This implies its role – it's positioned more as an add-on component instead of a core one.

kube-proxy is designed as a distributed network router that runs on every node. Its ultimate goal is to ensure that inbound traffic to a **Service** (this is an API object that we will introduce later) endpoint can be routed properly. Moreover, if there are multiple containers serving one application, it is able to balance the traffic in a round-robin manner by leveraging the underlying Linux iptables/IPVS technology.

There are also some other add-ons such as CoreDNS, though we will skip those so that we can focus on the core components and get a high-level picture of things.

> **Note**
>
> Sometimes, kube-proxy and CoreDNS are also considered core components of a Kubernetes installation. To some extent, that's technically true as they're mandatory in most cases; otherwise, the Service API object won't work. However, in this book, we're leaning more toward categorizing them as "add-ons" as they focus on the implementation of one particular Kubernetes API resource instead of a general workflow. Also, kube-proxy and CoreDNS are defined in **`addon-manager.yaml.tmpl`** instead of being portrayed on the same level as the other core Kubernetes components.

Kubernetes Architecture

In the previous section, we gained a first impression of the core Kubernetes components: etcd, the API server, the scheduler, the controller manager, and the kubelet. These components, plus other add-ons, comprise the Kubernetes architecture, which can be seen in the following diagram:

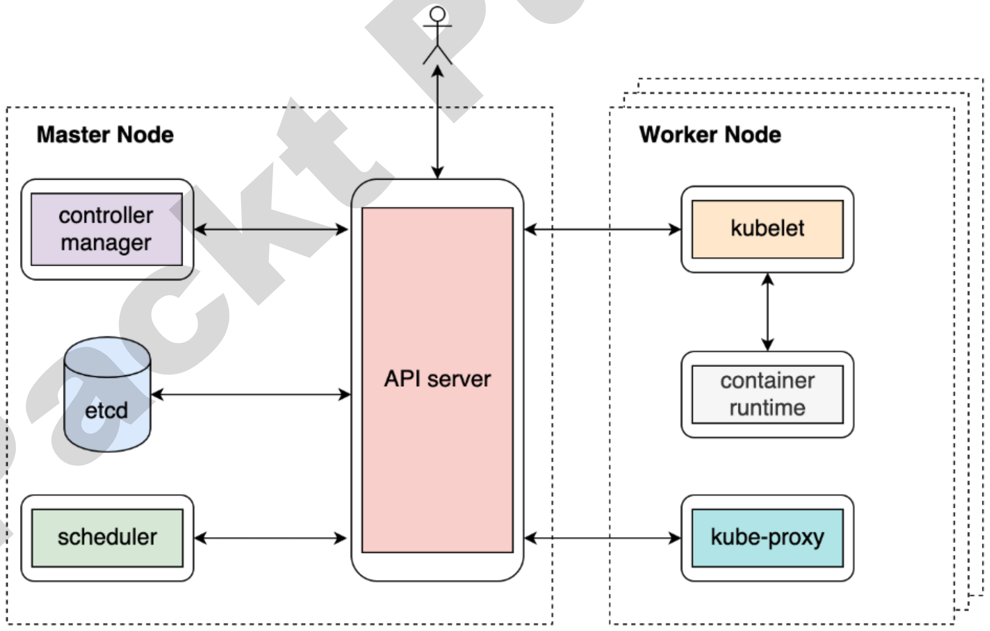


Figure 2.18: Kubernetes architecture

At this point, we won't look at each component in too much detail. However, at a high-level view, it's critical to understand how the components communicate with each other and why they're designed in that way.

The first thing to understand is which components the API server can interact with. From the preceding diagram, we can easily tell that the API server can talk to almost every other component (except the container runtime, which is handled by the kubelet) and that it also serves to interact with end users directly. This design makes the API server act as the "heart" of Kubernetes. Additionally, the API server also scrutinizes incoming requests and writes API objects into the backend storage (etcd). This, in other words, makes the API server the throttle of security control measures such as authentication, authorization, and auditing.

The second thing to understand is how the different Kubernetes components (except for the API server) interact with each other. It turns out that there is no explicit connection among them – the controller manager doesn't talk to the scheduler, nor does the kubelet talk to kube-proxy.

You read that right – they do need to work in coordination with each other to accomplish many functionalities, but they never directly talk to each other. Instead, they communicate implicitly via the API server. More precisely, they communicate by watching, creating, updating, or deleting corresponding API objects. This is also known as the controller/operator pattern.

Container Network Interface

There are several networking aspects to take into consideration, such as how a pod communicates with its host machine's network interface, how a node communicates with other nodes, and, eventually, how a pod communicates with any pod across different nodes. As the network infrastructure differs vastly in cloud or on-premise environments, Kubernetes chooses to solve those problems by defining a specification called the **Container Network Interface** (**CNI**). Different CNI providers can follow the same interface and implement their own logic that adheres to the Kubernetes standards to ensure that the whole Kubernetes network works. We will revisit the idea of the CNI in *Chapter 11*, *Cloud Deployments and HA Kubernetes*. For now, let's return to our discussion of how the different Kubernetes components work.

Later in this chapter, *Exercise 2.05*, *How Kubernetes Manages a Pod's Life Cycle*, will help you consolidate your understanding of this and clarify a few things, such as how the different Kubernetes components operate synchronously or asynchronously to ensure a typical Kubernetes workflow, and what would happen if one or more of these components malfunctions. The exercise will help you better understand the overall Kubernetes architecture. But before that, let's introduce our containerized application from the previous chapter to the Kubernetes world and explore a few benefits of Kubernetes.

Migrating Containerized Application to Kubernetes

In the previous chapter, we built a simple HTTP server called **`k8s-for-beginners`**, and it runs as a Docker container. It works perfectly for a sample application. However, what if you have to manage thousands of containers, and coordinate and schedule them properly? How can you upgrade a service without downtime? How do you keep a service healthy upon unexpected failure? These problems exceed the abilities of a system that simply uses containers alone. What we need is a platform that can orchestrate, as well as manage, our containers.

We have told you that Kubernetes is the solution that we need. Next, we will walk you through a series of exercises regarding how to orchestrate and run containers in Kubernetes using a Kubernetes native approach.

Pod Specification

A straightforward thought is that we wish to see what the equivalent API call or command to run a container in Kubernetes is. As explained in *Chapter 1*, *Understanding Kubernetes and Containers*, a container can join another container's namespace so that they can access each other's resources (for example, network, storage, and so on) without additional overhead. In the real world, some applications may need several containers working closely, either in parallel or in a particular order (the output of one will be processed by another). Also, some generic containers (for example, logging agent, network throttling agent, and so on) may need to work closely with their target containers.

Since an application may often need several containers, a container is not the minimum operational unit in Kubernetes; instead, it introduces a concept called **pods** to bundle one or multiple containers. Kubernetes provides a series of specifications to describe how this pod is supposed to be, including several specifics such as images, resource requests, startup commands, and more. To send this pod spec to Kubernetes, particularly to the Kubernetes API server, we're going to use Kubectl.

> **Note**
>
> We will learn more about pods in *Chapter 5, Pods*, but we will use them in this chapter for the purpose of simple demonstrations. You can refer to the complete list of available pod specifications at this link: https://godoc.org/k8s.io/api/core/v1#PodSpec.

Next, let's learn how to run a single container in Kubernetes by composing the pod spec file (also called the specification, manifest, config, or configuration file). In Kubernetes, you can use YAML or JSON to write this specification file, though YAML is commonly used since it is more human-readable and editable.

Consider the following YAML spec for a very simple pod:

```
kind: Pod
apiVersion: v1
metadata:
  name: k8s-for-beginners
spec:
  containers:
  - name: k8s-for-beginners
    image: packtworkshops/the-kubernetes-workshop:k8s-for-beginners
```

Let's go through the different fields briefly:

- **`kind`** tells Kubernetes which type of object you want to create. Here, we are creating a **`Pod`**. In later chapters, you will see many other kinds, such as **`Deployment`**, **`StatefulSet`**, **`ConfigMap`**, and so on.
- **`apiVersion`** specifies a particular version of an API object. Different versions may behave a bit differently.

- **`metadata`** includes some attributes that can be used to uniquely identify the pod, such as name and namespace. If we don't specify a namespace, it goes in the **`default`** namespace.
- **`spec`** contains a series of fields describing the pod. In this example, there is one container that has its image URL and name specified.

Pods are one of the simplest Kubernetes objects to deploy, so we will use them to learn how to deploy objects using YAML manifests in the following exercise.

Applying a YAML manifest

Once we have a YAML manifest ready, we can use **`kubectl apply -f <yaml file>`** or **`kubectl create -f <yaml file>`** to instruct the API server to persist the API resources defined in this manifest. When you create a pod from scratch for the first time, it doesn't make much difference which of the two commands you use. However, we may often need to modify the YAML (let's say, for example, if we want to upgrade the image version) and reapply it. If we use the **`kubectl create`** command, we have to delete and recreate it. However, with the **`kubectl apply`** command, we can rerun the same command and the delta change will be calculated and applied automatically by Kubernetes. This is very convenient from an operational point of view. For example, if we use some form of automation, it is much simpler to repeat the same command. So, we will use **`kubectl apply`** across the following exercise, regardless of whether it's the first time it's being applied or not.

Exercise 2.02: Running a Pod in Kubernetes

In the previous exercise, we started up Minikube and looked at the various Kubernetes components running as pods. Now, in this exercise, we shall deploy our own pod. Follow these steps to complete this exercise:

> **Note**
>
> If you have been trying out the commands from the *Kubernetes Components Overview* section, don't forget to leave the SSH session by using the **`exit`** command before beginning this exercise. Unless otherwise specified, all commands using **`kubectl`** should run on the host machine and not inside the Minikube VM.

1. In Kubernetes, we use a spec file to describe an API object such as a pod. As mentioned earlier, we will stick to YAML it is are more human-readable and editable. Create a file named **`k8s-for-beginners-pod.yaml`** (using any text editor of your choice) with the following content:

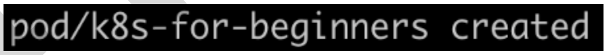
```
kind: Pod
apiVersion: v1
metadata:
  name: k8s-for-beginners
spec:
  containers:
  - name: k8s-for-beginners
    image: packtworkshops/the-kubernetes-workshop:k8s-for-beginners
```

> **Note**
>
> Please replace the image path in the last line of the preceding YAML file with the path to your own image that you created in the previous chapter.

2. On the host machine, run the following command to create this pod:

```
kubectl apply -f k8s-for-beginners-pod.yaml
```

You should see the following output:



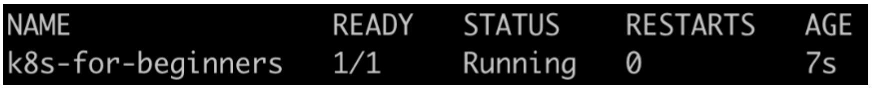
```
pod/k8s-for-beginners created
```

Figure 2.19: Creating k8s-for-beginners-pod

3. Now, we can use the following command to check the pod's status:

```
kubectl get pod
```

You should see the following response:



```
NAME                READY   STATUS    RESTARTS   AGE
k8s-for-beginners   1/1     Running   0          7s
```

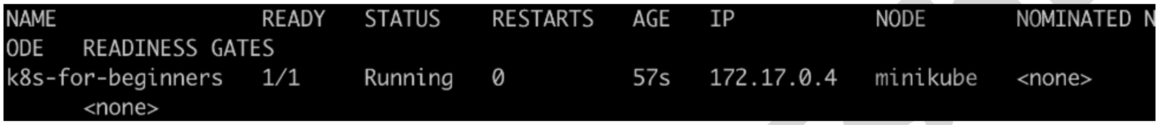
Figure 2.20: Getting the list of pods

Be default, **`kubectl get pod`** will list all the pods using a table format. In the preceding output, we can see the **`k8s-for-beginners`** pod is running properly and that it has one container that is ready (**`1/1`**). Moreover, Kubectl provides an additional flag called **`-o`** so we can adjust the output format. For example, **`-o yaml`** or **`-o json`** will return the full output of the pod API object in YAML or JSON format, respectively, as it's stored version in Kubernetes' backend storage (etcd).

4. You can use the following command to get more information about the pod:

```
kubectl get pod -o wide
```

You should see the following output:



```
NAME                READY   STATUS    RESTARTS   AGE   IP           NODE       NOMINATED N
ODE   READINESS GATES
k8s-for-beginners   1/1     Running   0          57s   172.17.0.4   minikube   <none>
      <none>
```

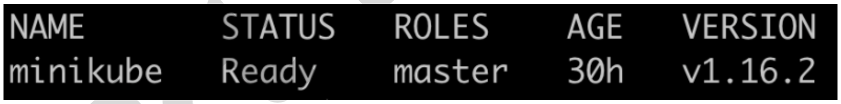
Figure 2.21: Getting more information about pods

As you can see, the output is still in the table format and we get additional information such as **`IP`** (the internal pod IP) and **`NODE`** (which node the pod is running on).

5. You can get the list of nodes in our cluster by running the following command:

```
kubectl get node
```

You should see the following response:



```
NAME       STATUS   ROLES    AGE   VERSION
minikube   Ready    master   30h   v1.16.2
```

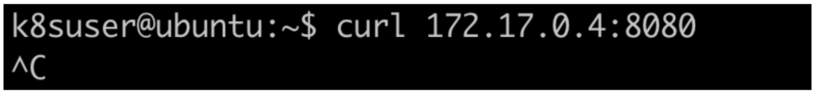
Figure 2.22: Getting the list of nodes

6. The IP listed in *Figure 2.21* refers to the internal IP Kubernetes assigned for this pod, and it's used for pod-to-pod communication, not for routing external traffic to pods. Hence, if you try to access this IP from outside the cluster, you will get nothing. You can try that using the following command from the host machine:

```
curl 172.17.0.4:8080
```

> **Note**
>
> Remember to change **`172.17.0.4`** to the value you get for your environment in *step 4*, as seen in *Figure 2.21*.

The **curl** command will just hang and return nothing, as shown here:



```
k8suser@ubuntu:~$ curl 172.17.0.4:8080
^C
```

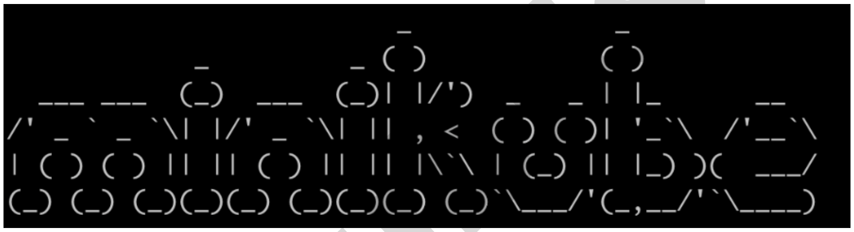
Figure 2.23: Trying to access our application via pod IP

You will need to press *Ctrl* + C to abort it.

7. In most cases, end users don't need to interact with the internal pod IP. However, just for observation purposes, let's walk into the cluster by opening an SSH session to the Minikube VM:

```
minikube ssh
```

You will see the following response in the Terminal:



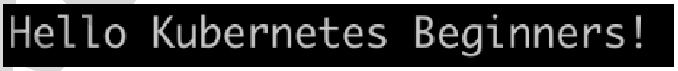
```
                         _             _
            _         _ ( )           ( )
  ___ ___  (_)  ___  (_)| |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\| || , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || || |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)
```

Figure 2.24: Accessing the Minikube VM via SSH

8. Now, try calling the IP from inside the Minikube VM to verify that it works:

```
curl 172.17.0.4:8080
```

You should get a successful response:



```
Hello Kubernetes Beginners!
```

Figure 2.25: Accessing our application from inside the Minikube VM

With this, we have successfully deployed our application in a pod on the Kubernetes cluster. We can confirm that it is working since we get a response when we call the application from inside the cluster. Now, you may end the SSH session using the **exit** command.

Service Specification

The last part of the previous section proves that network communication works great among different components inside the cluster. But in the real world, you would not expect users of your application to gain SSH access into your cluster to use your applications. So, you would want your application to be accessed externally.

To facilitate just that, Kubernetes provides a concept called a **Service** to abstract the network access to your application's pods. A Service acts as a network proxy to accept network traffic from external users, and then distributes it to internal pods. However, there should be a way to describe the association rule between the Service and the corresponding pods. Kubernetes uses labels, which are defined in the pod definitions, and label selectors, which are defined in the Service definition, to describe this relationship.

> **Note**
>
> You will learn more about labels and label selectors in *Chapter 6, Labels and Annotations*.

Let's consider the following sample spec for a Service:

```
kind: Service
apiVersion: v1
metadata:
  name: k8s-for-beginners
spec:
  selector:
    tier: frontend
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
```

Similar to a pod spec, here, we define **`kind`** and **`apiVersion`**, while **`name`** is defined under the **`metadata`** field. Under the **`spec`** field, there are several critical fields to take note of:

- **`selector`** defines the labels to be selected to match a relationship with the corresponding pods, which, as you will see in the following exercise, are supposed to be labeled properly.
- **`type`** defines the type of Service. If not specified, the default type is **`ClusterIP`**, which means it's only used within the cluster, that is, internally. Here, we specify it as **`NodePort`**. This means the Service will expose a port in each node of the cluster and associate the port with the corresponding pods. Another well-known type is called **`LoadBalancer`**, which is typically not implemented in a vanilla Kubernetes offering. Instead, Kubernetes delegates the implementation to each cloud provider, such as GKE, EKS, and so on.

- **ports** include a series of **port** fields, each with a **targetPort** field. The **targetPort** field is the actual port that's exposed by the destination pod.

  Thus, the Service can be accessed internally via **<service ip>:<port>**. Now, for example, if you have an NGINX pod running internally and listening at port 8080, then you should define **targetPort** as **8080**. You can specify any arbitrary number for the **port** field, such as **80** in this case. Kubernetes will set up and maintain the mapping between **<service IP>:<port>** and **<pod IP>:<targetPort>**. In the following exercise, we will learn how to access the Service from outside the cluster and bring external traffic inside the cluster via the Service.

In the following exercise, we will define Service manifests and create them using **kubectl apply** commands. You will learn that the common pattern for resolving problems in Kubernetes is to find out the proper API objects, then compose the detailed specs using YAML manifests, and finally create the objects to bring them into effect.

Exercise 2.03: Accessing a Pod via a Service

In the previous exercise, we observed that an internal pod IP doesn't work for anyone outside the cluster. In this exercise, we will create Services that will act as connectors to map the external requests to the destination pods so that we can access the pods externally without entering the cluster. Follow these steps to complete this exercise:

1. Firstly, let's tweak the pod spec from *Exercise 2.02*, *Running a Pod in Kubernetes*, to apply some labels. Modify the contents of the **k8s-for-beginners-pod1.yaml** file, as follows:

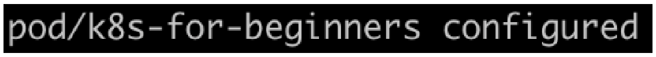
```
kind: Pod
apiVersion: v1
metadata:
  name: k8s-for-beginners
  labels:
    tier: frontend
spec:
  containers:
  - name: k8s-for-beginners
    image: packtworkshops/the-kubernetes-workshop:k8s-for-beginners
```

   Here, we added a label pair, **tier: frontend**, under the **labels** field.

2. Because the pod name remains the same, let's rerun the **apply** command so that Kubernetes knows that we're trying to update the pod's spec, instead of creating a new pod:

```
kubectl apply -f k8s-for-beginners-pod1.yaml
```

You should see the following response:



```
pod/k8s-for-beginners configured
```

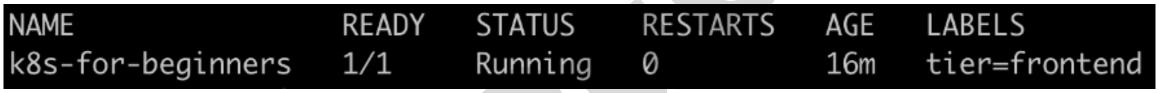
Figure 2.26: Updating k8s-for-beginners-pod

Behind the scenes, for the **kubectl apply** command, Kubectl generates the difference of the specified YAML and the stored version in the Kubernetes server-side storage (that is, etcd). If the request is valid (that is, we have not made any errors in the specification format or the command), Kubectl will send an HTTP patch to the Kubernetes API server. Hence, only the delta changes will be applied. If you look at the message that's returned, you'll see it says **pod/k8s-for-beginners configured** instead of **created**, so we can be sure it's applying the delta changes and not creating a new pod.

3. You can use the following command to explicitly display the labels that have been applied to existing pods:

```
kubectl get pod --show-labels
```

You should see the following response:



```
NAME                READY   STATUS    RESTARTS   AGE   LABELS
k8s-for-beginners   1/1     Running   0          16m   tier=frontend
```

Figure 2.27: Getting the list of pods with labels

Now that the pod has the **tier: frontend** attribute, we're ready to create a Service and link it to the pods.

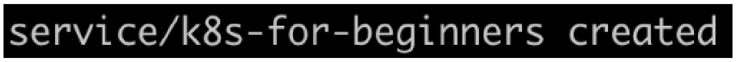
4. Create a file named **k8s-for-beginners-svc.yaml** with the following content:

```
kind: Service
apiVersion: v1
metadata:
  name: k8s-for-beginners
spec:
  selector:
    tier: frontend
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
```

5. Now, let's create the Service using the following command:

```
kubectl apply -f k8s-for-beginners-svc.yaml
```

   You should see the following response:



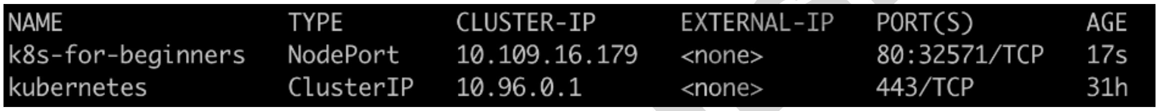
```
service/k8s-for-beginners created
```

Figure 2.28: Creating the k8s-for-beginners-svc Service

6. Use the **`get`** command to return the list of created Services and confirm whether our Service is online:

```
kubectl get service
```

   You should see the following response:



```
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
k8s-for-beginners   NodePort    10.109.16.179   <none>        80:32571/TCP   17s
kubernetes          ClusterIP   10.96.0.1       <none>        443/TCP        31h
```

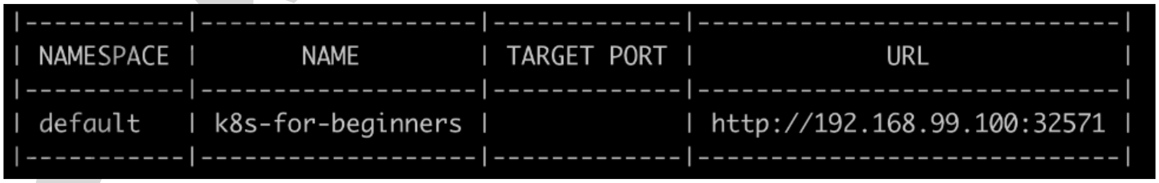
Figure 2.29: Getting the list of Services

   So, you may have noticed that the **`PORT(S)`** column outputs **`80:32571/TCP`**. Port **`32571`** is an auto-generated port that's exposed on every node, which is done intentionally so that external users can access it. Now, before moving on to the next step, exit the SSH session.

7. Now, we have the "external port" as **`32571`**, but we still need to find the external IP. Minikube provides a utility we can use to easily access the **`NodePort`** Service:

```
minikube service k8s-for-beginners
```

   You should see a response that looks similar to the following:



```
|-----------|-------------------|-------------|-----------------------------|
| NAMESPACE |       NAME        | TARGET PORT |             URL             |
|-----------|-------------------|-------------|-----------------------------|
| default   | k8s-for-beginners |             | http://192.168.99.100:32571 |
|-----------|-------------------|-------------|-----------------------------|
```

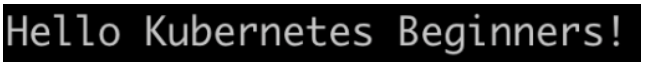
Figure 2.30: Getting the URL and port to access the NodePort Service

   Depending on your environment, this may also automatically open a browser web page so you can access the Service. From the URL, you will be able to see that the Service port is **`32606`**. The external IP is actually the IP of the Minikube VM.

8. You can also access our application from outside the cluster via the command line:

```
    curl http://192.168.99.100:32571
```

   You should see the following response:



```
Hello Kubernetes Beginners!
```

Figure 2.31: Accessing the application

As a summary, in this exercise, we created a **`NodePort`** Service to enable external users to access the internal pods without entering the cluster. Under the hood, there are several layers of traffic transitions that make this happen:

- The first layer is from the external user to the machine IP at the auto-generated random port (3XXXX).
- The second layer is from the random port (3XXXX) to the Service IP (10.X.X.X) at port 80.
- The third layer is from the Service IP (10.X.X.X) to the ultimate pod IP at port 8080.

The following is a diagram illustrating these interactions:

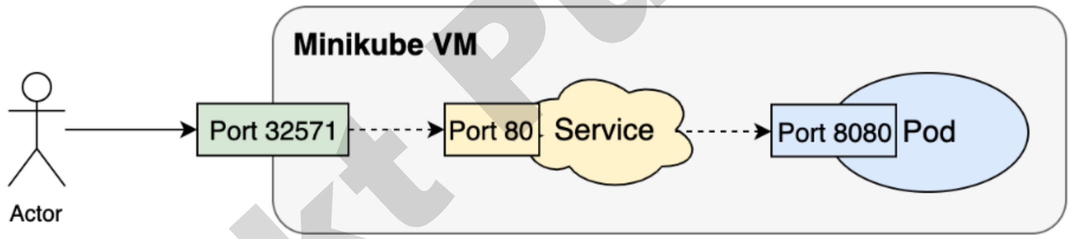


Figure 2.32: Routing traffic from a user outside the cluster to the pod running our application

Services and Pods

In *step* 3 of the previous exercise, you may have noticed that the Service tries to match pods by labels (the **`selector`** field under the **`spec`** section) instead of using a fixed pod name or something similar. From a pod's perspective, it doesn't need to know which Service is bringing traffic to it. (In some rare cases, it can even be mapped to multiple Services; that is, multiple Services may be sending traffic to a pod.)

This label-based matching mechanism is widely used in Kubernetes. It enables the API objects to be loosely coupled at runtime. For example, you can specify **`tier: frontend`** as the label selector, which will, in turn, be associated with the pods that are labeled as **`tier: frontend`**.

Due to this, by the time the Service is created, it doesn't matter if the backing pods exist or not. It's totally acceptable for backing pods to be created later, and after they are created, the Service object will become associated with the correct pods. Internally, the whole mapping logic is implemented by the service controller, which is part of the controller manager component. It's also possible that a Service may have two matching pods at a time, and later a third pod is created with matching labels, or one of the existing pods gets deleted. In either case, the service controller can detect such changes and ensure that users can always access their application via the Service endpoint.

It's a very commonly used pattern in Kubernetes to orchestrate your application using different kinds of API objects, and then glue them together by using labels or other loosely coupled conventions. It's also the key part of container orchestration.

Delivering Kubernetes-Native Applications

In the previous sections, we migrated a Docker-based application to Kubernetes and successfully accessed it from inside the Minikube VM, as well as externally. Now, let's see what other benefits Kubernetes can provide if we design our application from the ground up so that it can be deployed using Kubernetes.

Along with the increasing usage of your application, it may be common to run several replicas of certain pods to serve a business functionality. In this case, grouping different containers in a pod alone is not sufficient. We need to go ahead and create groups of pods that are working together. Kubernetes provides several abstractions for groups of pods, such as Deployments, DaemonSets, Jobs, CronJobs, and so on. Just like the Service object, these objects can also be created by using a spec that's been defined in a YAML file.

To start understanding the benefits of Kubernetes, let's use a Deployment to demonstrate how to replicate (scale up/down) an application in multiple pods.

Abstracting groups of pods using Kubernetes gives us the following advantages:

- **Creating replicas of pods for redundancy**: This is the main advantage of abstractions of groups of pods such as Deployments. A Deployment can create several pods with the given spec. A Deployment will automatically ensure that the pods that it creates are online, and it will automatically replace any pods that fail.

- **Easy upgrades and rollbacks**: Kubernetes provides different strategies that you can use to upgrade your applications, as well as rolling versions back. This is important because in modern software development, software is often developed iteratively and updates are released frequently. An upgrade can change anything in the Deployment specification. It can be an update of labels or any other field(s), an image version upgrade, an update on its embedded containers, and so on.

Let's take a look at some notable aspects of the spec of a sample Deployment:

```
k8s-for-beginners-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: k8s-for-beginners
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: k8s-for-beginners
    image: packtworkshops/the-kubernetes-workshop:k8s-for-beginners
```

In addition to wrapping the pod spec as a "template," a Deployment must also specify its own kind (**`Deployment`**), as well as the API version (**`apps/v1`**).

> **Note**
>
> For some historical reason, the spec name **`apiVersion`** is still being used. But technically speaking, it literally means apiGroupVersion. In the preceding Deployment example, it belongs to the **`apps`** group and is version **`v1`**.

In the Deployment spec, the **replicas** field instructs Kubernetes to start three pods using the pod spec defined in the **template** field. The **selector** field plays the same role as we saw in the case of the Service – it aims to associate the Deployment object with specific pods in a loosely coupled manner. This is particularly useful if you want to bring any preexisting pods under the management of your new Deployment.

The replica number defined in a Deployment or other similar API object represents the desired state of how many pods are supposed to be running continuously. If some of these pods fail for some unexpected reason, Kubernetes will automatically detect that and create a corresponding number of pods to take their place. We will explore that in the following exercise.

We'll see a Deployment in action in the following exercise.

Exercise 2.04: Scaling a Kubernetes Application

In Kubernetes, it's easy to increase the number of replicas running the application by updating the **replicas** field of a Deployment spec. In this exercise, we'll experiment with how to scale a Kubernetes application up and down. Follow these steps to complete this exercise:

1. Create a file named **k8s-for-beginners-deploy.yaml** using the content shown here:

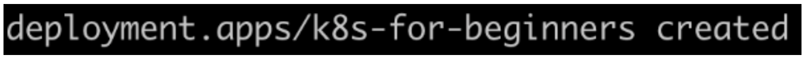
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: k8s-for-beginners
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: k8s-for-beginners
    image: packtworkshops/the-kubernetes-workshop:k8s-for-beginners
```

If you take a closer look, you'll see that this Deployment spec is largely based on the pod spec from earlier exercises (**k8s-for-beginners-pod1.yaml**), which you can see under the **template** field.

2. Next, we can use Kubectl to create the Deployment:

```
kubectl apply -f k8s-for-beginners-deploy.yaml
```

You should see the following output:



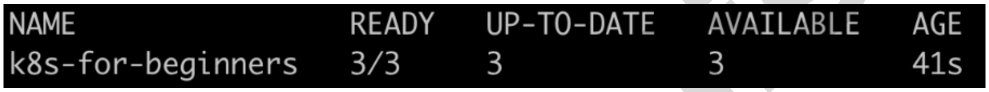
```
deployment.apps/k8s-for-beginners created
```

Figure 2.33: Creating the k8s-for-beginners Deployment

3. Given that the Deployment has been created successfully, we can use the following command to show all the Deployment's statuses, such as their names, running pods, and so on:

```
kubectl get deploy
```

You should get the following response:



```
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
k8s-for-beginners   3/3     3            3           41s
```

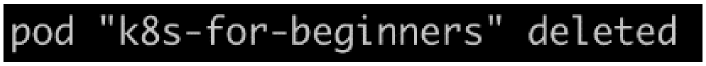
Figure 2.34: Getting the list of Deployments

> **Note**
>
> As shown in the previous command, we are using **`deploy`** instead of **`deployment`**. Both of these will work and **`deploy`** is an allowed short name for **`deployment`**. You can find a quick list of some commonly used short names at this link: https://kubernetes.io/docs/reference/kubectl/overview/#resource-types.
>
> You can also view the short names by running **`kubectl get`**, without specifying the resource type.

4. A pod called **`k8s-for-beginners`** exists that we created in the previous exercise. To ensure that we see only the pods being managed by the Deployment, let's delete the older pod:

```
kubectl delete pod k8s-for-beginners
```

You should see the following response:



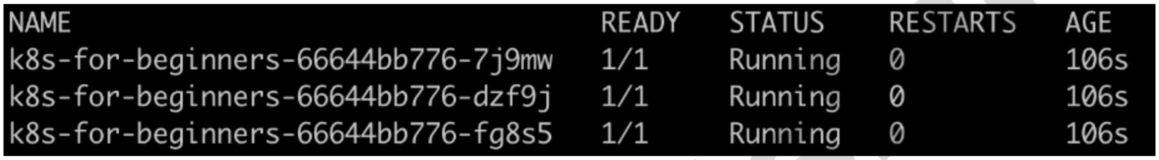
```
pod "k8s-for-beginners" deleted
```

Figure 2.35: Deleting the k8s-for-beginners pod

5. Now, get a list of all the pods:

```
kubectl get pod
```

You should see the following response:



```
NAME                                 READY   STATUS    RESTARTS   AGE
k8s-for-beginners-66644bb776-7j9mw   1/1     Running   0          106s
k8s-for-beginners-66644bb776-dzf9j   1/1     Running   0          106s
k8s-for-beginners-66644bb776-fg8s5   1/1     Running   0          106s
```

Figure 2.36: Getting the list of pods

The Deployment has created three pods, and their labels (specified in the **labels** field in *step 1*) happen to match the Service we created in the previous section. So, what will happen if we try to access the Service? Will the network traffic going to the Service be smartly routed to the new three pods? Let's test this out.

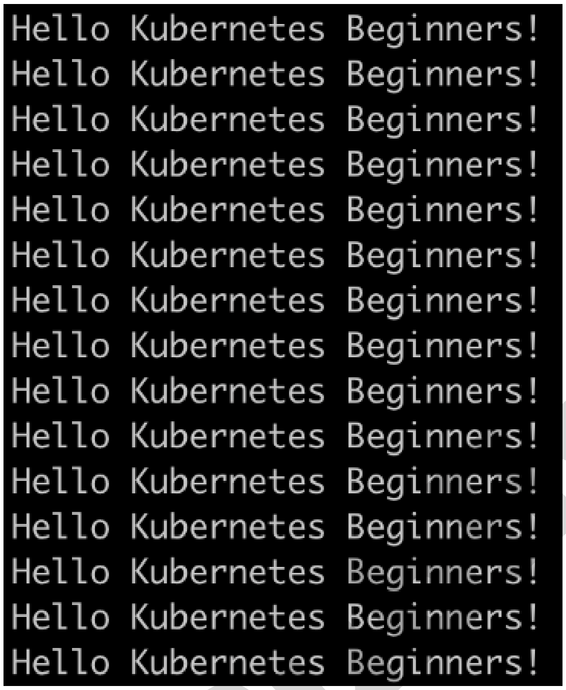
6. To see how the traffic is distributed to the three pods, we can simulate a number of consecutive requests to the Service endpoint by running the **curl** command inside a Bash **for** loop, as follows:

```
for i in $(seq 1 30); do curl <minikube vm ip>:<service node port>; done
```

Note

In this command, use the same IP and port that you used in the previous exercise if you are running the same instance of Minikube. If you have restarted Minikube or have made any other changes, please get the proper IP of your Minikube cluster by following *step 9* of the previous exercise.

Once you've run the command with the proper IP and port, you should see the following output:



```
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
Hello Kubernetes Beginners!
```

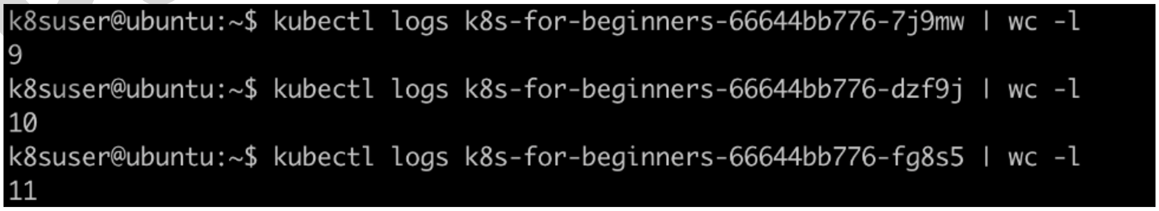
Figure 2.37: Repeatedly accessing our application

From the output, we can tell that all 30 requests get the expected response.

7. You can run **`kubectl logs <pod name>`** to check the log of each pod. Let's go one step further and figure out the exact number of requests each pod has responded to, which might help us find out whether the traffic was evenly distributed. To do that, we can pipe the logs of each pod into the **`wc`** command to get the number of lines:

```
kubectl logs <pod name> | wc -l
```

Run the preceding command three times, copying the pod name you obtained, as shown in *Figure 2.36*:



```
k8suser@ubuntu:~$ kubectl logs k8s-for-beginners-66644bb776-7j9mw | wc -l
9
k8suser@ubuntu:~$ kubectl logs k8s-for-beginners-66644bb776-dzf9j | wc -l
10
k8suser@ubuntu:~$ kubectl logs k8s-for-beginners-66644bb776-fg8s5 | wc -l
11
```

Figure 2.38: Getting the logs of each of the three pod replicas running our application

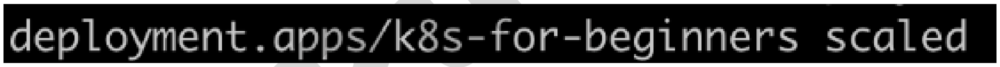
The result shows that the three pods handled 9, 10, and 11 requests, respectively. Due to the small sample size, the distribution is not absolutely even (that is, 10 for each), but it is sufficient to indicate the default round-robin distribution strategy used by a Service.

> **Note**
>
> You can read more about how kube-proxy leverages iptables to perform the internal load balancing by looking at the official documentation: https://kubernetes.io/docs/concepts/services-networking/service/#proxy-mode-iptables.

8. Next, let's learn how to scale up a Deployment. There are two ways of accomplishing this: one way is to modify the Deployment's YAML config, where we can set the value of **`replicas`** to another number (such as 5), while the other way is to use the **`kubectl scale`** command, as follows:

```
kubectl scale deploy k8s-for-beginners --replicas=5
```

You should see the following response:



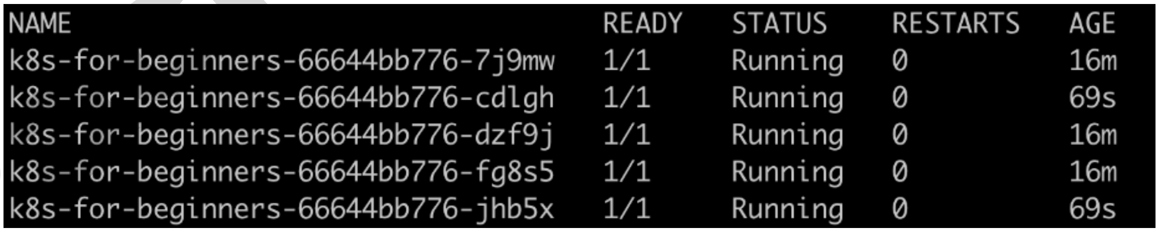
```
deployment.apps/k8s-for-beginners scaled
```

Figure 2.39: Scaling the Deployment

9. Let's verify whether there are five pods running:

```
kubectl get pod
```

You should see a response similar to the following:



```
NAME                                READY   STATUS    RESTARTS   AGE
k8s-for-beginners-66644bb776-7j9mw  1/1     Running   0          16m
k8s-for-beginners-66644bb776-cdlgh  1/1     Running   0          69s
k8s-for-beginners-66644bb776-dzf9j  1/1     Running   0          16m
k8s-for-beginners-66644bb776-fg8s5  1/1     Running   0          16m
k8s-for-beginners-66644bb776-jhb5x  1/1     Running   0          69s
```

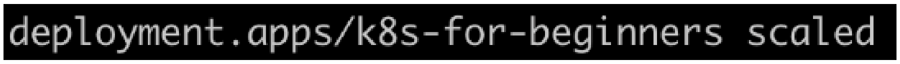
Figure 2.40: Getting the list of pods

The output shows that the existing three pods are kept and that two new pods are created.

10. Similarly, you can specify replicas that are smaller than the current number. In our example, let's say that we want to shrink the replica's number to 2. The command for this would look as follows:

```
kubectl scale deploy k8s-for-beginners --replicas=2
```

You should see the following response:



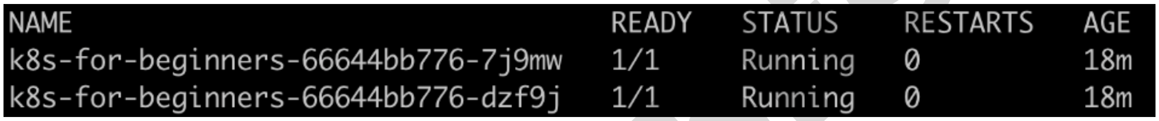
```
deployment.apps/k8s-for-beginners scaled
```

Figure 2.41: Scaling the Deployment

11. Now, let's verify the number of pods:

```
kubectl get pod
```

You should see a response similar to the following:



```
NAME                                 READY   STATUS    RESTARTS   AGE
k8s-for-beginners-66644bb776-7j9mw   1/1     Running   0          18m
k8s-for-beginners-66644bb776-dzf9j   1/1     Running   0          18m
```

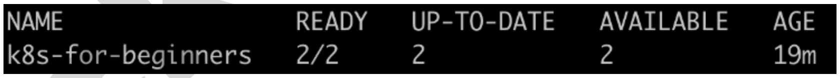
Figure 2.42: Getting the list of pods

As shown in the preceding screenshot, there are two pods, and they are both running as expected. Thus, in Kubernetes' terms, we can say, "the Deployment is in its desired state."

12. We can run the following command to verify this:

```
kubectl get deploy
```

You should see the following response:



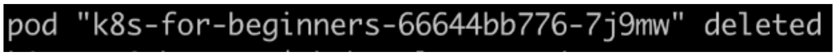
```
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
k8s-for-beginners   2/2     2            2           19m
```

Figure 2.43: Getting the list of Deployments

13. Now, let's see what happens if we delete one of the two pods:

```
kubectl delete pod <pod name>
```

You should get the following response:



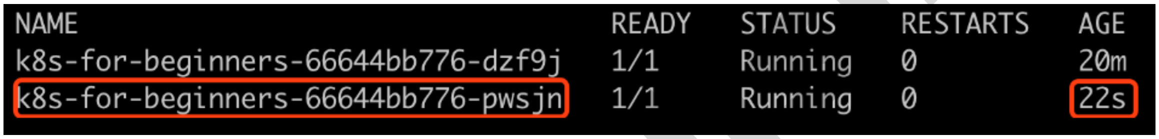
```
pod "k8s-for-beginners-66644bb776-7j9mw" deleted
```

Figure 2.44: Deleting one of the pod replicas

14. Check the status of the pods to see what has happened:

```
kubectl get pod
```

You should see the following response:



```
NAME                                 READY   STATUS    RESTARTS   AGE
k8s-for-beginners-66644bb776-dzf9j   1/1     Running   0          20m
k8s-for-beginners-66644bb776-pwsjn   1/1     Running   0          22s
```

Figure 2.45: Getting the list of pods

We can see that there are still two pods. From the output, it's worth noting that the first pod name is the same as the second pod in *Figure 2.42* (this is the one that was not deleted), but that the highlighted pod name is different from any of the pods in *Figure 2.42*. This indicates that the highlighted one is the pod that was newly created to replace the deleted one. The Deployment created a new pod so that the number of running pods satisfies the desired state of the Deployment.

In this exercise, we have learned how to scale a deployment up and down. You can scale other similar Kubernetes objects, such as DaemonSets and StatefulSets, in the same way. Also, for such objects, Kubernetes will try to auto-recover the failed pods.

Pod Life Cycle and Kubernetes Components

The previous sections in this chapter briefly described the Kubernetes components and how they work internally with each other. On the other hand, we also demonstrated how to use some Kubernetes API objects (pods, Services, and Deployments) to compose your applications.

But how is a Kubernetes API object managed by different Kubernetes components? Let's consider a pod as an example. Its life cycle can be illustrated as follows:

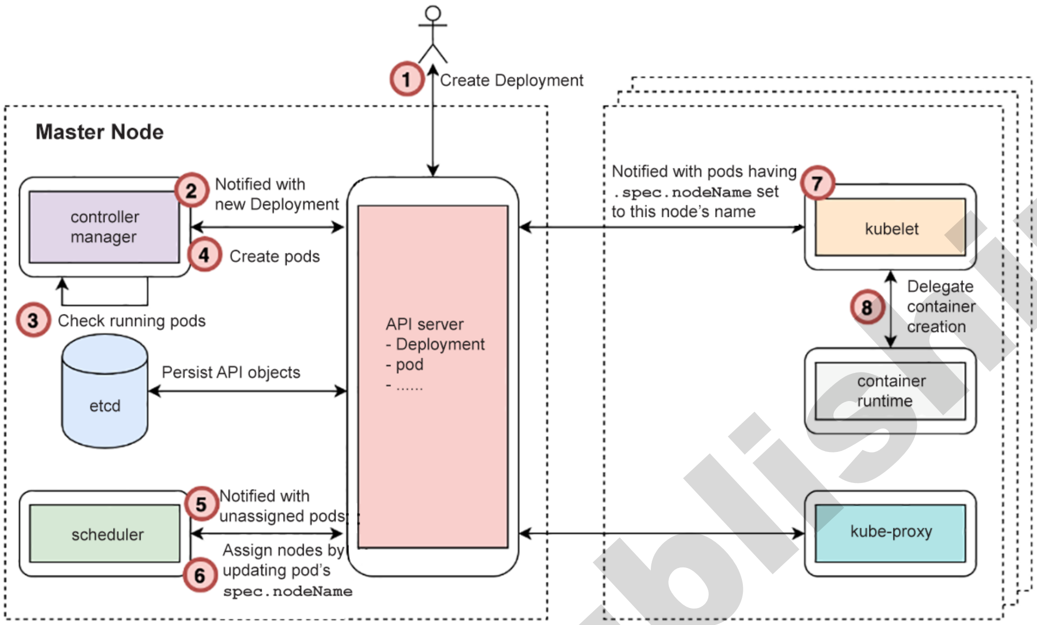

1 Create Deployment
Master Node
controller manager
2 Notified with new Deployment
4 Create pods
3 Check running pods
etcd
Persist API objects
API server
- Deployment
- pod
- ......
scheduler
5 Notified with unassigned pods
6 Assign nodes by updating pod's spec.nodeName
7 Notified with pods having .spec.nodeName set to this node's name
kubelet
8 Delegate container creation
container runtime
kube-proxy

Figure 2.46: The process behind the creation of a pod

This entire process can be broken down as follows:

1. A user starts to deploy an application by sending a Deployment YAML manifest to the Kubernetes API server. The API server verifies the request and checks whether it's valid. If it is, it persists the Deployment API object to its backend datastore (etcd).

> **Note**
>
> For any step that evolves by modifying API objects, interactions have to happen between etcd and the API server, so we don't list the interactions as extra steps explicitly.

2. By now, the pod hasn't been created yet. The controller manager gets a notification from the API server that a Deployment has been created.
3. Then, the controller manager checks whether the desired number of replica pods are running already.

4. If there are not enough pods running, it creates the appropriate number of pods. The creation of pods is accomplished by sending a request with the pod spec to the API server. It's quite similar to how a user would apply the Deployment YAML, but with the major difference being that this happens inside the controller manager in a programmatic manner.
5. Although pods have been created, they're nothing but some API objects stored in etcd. Now, the scheduler gets a notification from the API server saying that new pods have been created and no node has been assigned for them to run.
6. The scheduler checks the resource usage, as well as existing pods allocation, and then calculates the node that fits best for each new pod. At the end of this step, the scheduler sends an update request to the API server by setting the pod's **`nodeName`** spec to the chosen node.
7. By now, the pods have been assigned a proper node to run on. However, there are no physical containers running. In other words, the application doesn't work yet. Each kubelet (running on different worker nodes) gets notifications, indicating that some pods are expected to be run. Each kubelet will then check whether the pods to be run have been assigned the node that a kubelet is running on.
8. Once the kubelet determines that a pod is supposed to be on its node, it calls the underlying container runtime (Docker, containerd, or cri-o, for instance) to spin up the containers on the host. Once the containers are up, the kubelet is responsible for reporting its status back to the API server.

With this basic flow in mind, you should now have a vague understanding of the answers to the following questions:

- Who is in charge of pod creation? What's the state of the pod upon creation?
- Who is responsible for placing a pod? What's the state of the pod after placement?
- Who brings up the concrete containers?
- Who is in charge of the overall message delivery process to ensure that all components work together?

In the following exercise, we will use a series of concrete experiments to help you solidify this understanding. This will allow you to see how things work in practice.

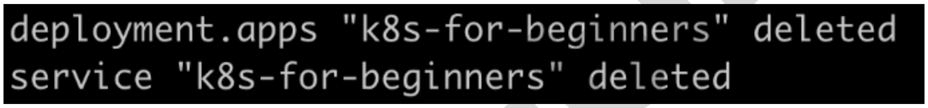
Exercise 2.05: How Kubernetes Manages a Pod's Life Cycle

As a Kubernetes cluster comprises multiple components, and each component works simultaneously, it's usually difficult to know what's exactly happening in each phase of a pod's life cycle. To solve this problem, we will use a film editing technique to "play the whole life cycle in slow motion," so as to observe each phase. We will turn off the master plane components and then attempt to create a pod. Then, we will respond to the errors that we see, and slowly bring each component online, one by one. This will allow us to slow down and examine each stage of the process of pod creation "frame by frame." Follow these steps to complete this exercise:

1. First, let's delete the Deployment and Service we created earlier by using the following command:

```
kubectl delete deploy k8s-for-beginners && kubectl delete service k8s-for-
beginners
```

You should see the following response:



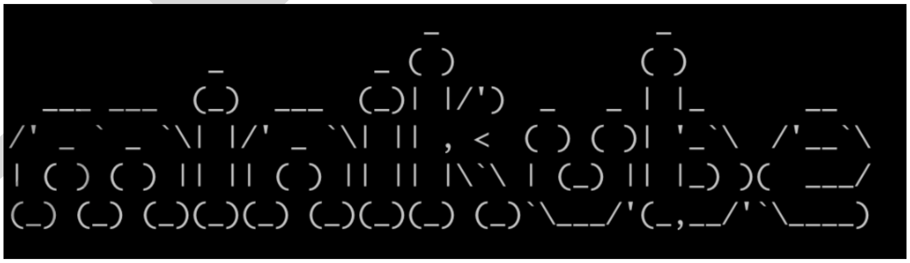
```
deployment.apps "k8s-for-beginners" deleted
service "k8s-for-beginners" deleted
```

Figure 2.47: Deleting pods from previous exercises

2. Prepare two Terminal sessions: one (host Terminal) to run commands on your host machine and another (Minikube Terminal) to pass commands inside the Minikube VM via SSH. Thus, your Minikube session will be initiated like this:

```
minikube ssh
```

You will see the following output:



```
                         _             _
            _         _ ( )           ( )
  ___ ___  (_)  ___  (_)| |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\| || , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || || |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)
```

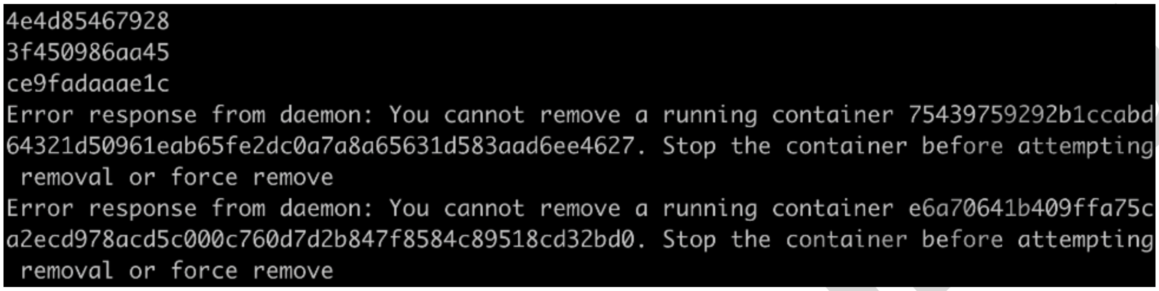
Figure 2.48: Accessing the Minikube VM via SSH

> **Note**
>
> All **`kubectl`** commands are expected to be run in the host Terminal session, while all **`docker`** commands are to be run in the Minikube Terminal session.

3. In the Minikube session, clean up all stopped Docker containers:

```
docker rm $(docker ps -a -q)
```

You should see the following output:



```
4e4d85467928
3f450986aa45
ce9fadaaae1c
Error response from daemon: You cannot remove a running container 75439759292b1ccabd
64321d50961eab65fe2dc0a7a8a65631d583aad6ee4627. Stop the container before attempting
 removal or force remove
Error response from daemon: You cannot remove a running container e6a70641b409ffa75c
a2ecd978acd5c000c760d7d2b847f8584c89518cd32bd0. Stop the container before attempting
 removal or force remove
```

Figure 2.49: Cleaning up all stopped Docker containers

You may see some error messages such as "You cannot remove a running container ...". This is because the preceding **`docker rm`** command runs against all containers (**`docker ps -a -q`**), but it won't stop any running containers.

4. In the Minikube session, stop the kubelet by running the following command:

```
sudo systemctl stop kubelet
```

This command does not show any response upon successful execution.

Note

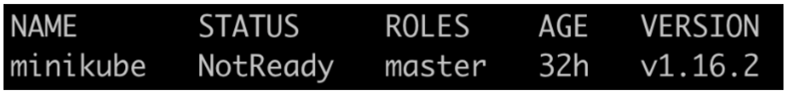
Later in this exercise, we will manually stop and start other Kubernetes components, such as the API server, that are managed by the kubelet in a Minikube environment. Hence, it's required that you stop the kubelet first in this exercise; otherwise, the kubelet will automatically restart its managed components.

Note that in typical production environments, unlike Minikube, it's not necessary to run the kubelet on the master node to manage the master plane components; the kubelet is only a mandatory component on worker nodes.

5. After 30 seconds, check the cluster's status by running the following command in your host Terminal session:

```
kubectl get node
```

You should see the following response:



```
NAME       STATUS     ROLES    AGE   VERSION
minikube   NotReady   master   32h   v1.16.2
```

Figure 2.50: Getting the list of nodes

It's expected that the status of the **`minkube`** node is changed to **`NotReady`** because the kubelet has been stopped.

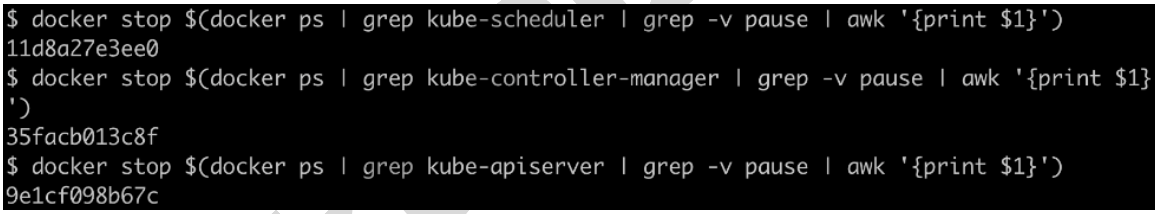
6. In your Minikube session, stop **`kube-scheduler`**, **`kube-controller-manager`**, and **`kube-apiserver`**. As we saw earlier, all of these are running as Docker containers. Hence, you can use the following commands, one after the other:

```
docker stop $(docker ps | grep kube-scheduler | grep -v pause | awk '{print
$1}')

docker stop $(docker ps | grep kube-controller-manager | grep -v pause | awk
'{print $1}')

docker stop $(docker ps | grep kube-apiserver | grep -v pause | awk '{print
$1}')
```

You should see the following responses:



```
$ docker stop $(docker ps | grep kube-scheduler | grep -v pause | awk '{print $1}')
11d8a27e3ee0
$ docker stop $(docker ps | grep kube-controller-manager | grep -v pause | awk '{print $1}
')
35facb013c8f
$ docker stop $(docker ps | grep kube-apiserver | grep -v pause | awk '{print $1}')
9e1cf098b67c
```

Figure 2.51: Stopping the containers running Kubernetes components

As we explained in the *Kubernetes Components Overview* section, the **`grep -v pause | awk`** command can fetch the exact container ID of the required Docker containers. Then, the **`docker pause`** command can pause that running Docker container.

Now, the three major Kubernetes components have been stopped.

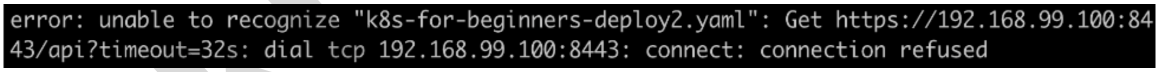
7. Now, you need to create a Deployment spec on your host machine. Create a file named **`k8s-for-beginners-deploy2.yaml`** with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: k8s-for-beginners
spec:
  replicas: 1
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: k8s-for-beginners
    image: packtworkshops/the-kubernetes-workshop:k8s-for-beginners
```

8. Try to create the Deployment by running the following command on your host session:

```
kubectl apply -f k8s-for-beginners-deploy2.yaml
```

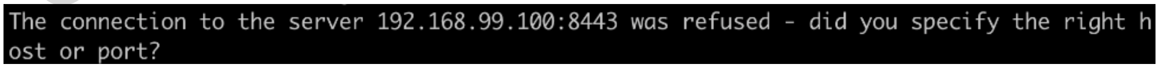
You should see a response similar to this:



```
error: unable to recognize "k8s-for-beginners-deploy2.yaml": Get https://192.168.99.100:84
43/api?timeout=32s: dial tcp 192.168.99.100:8443: connect: connection refused
```

Figure 2.52: Trying to create a new Deployment

Unsurprisingly, we got a network timeout error since we intentionally stopped the Kubernetes API server. If the API server is down, you cannot run any **`kubectl`** commands or use any equivalent tools (such as Kubernetes Dashboard) that rely on API requests:



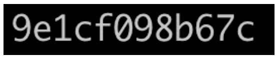
```
The connection to the server 192.168.99.100:8443 was refused - did you specify the right h
ost or port?
```

Figure 2.53: Trying to get a list of nodes

9. Let's see what happens if we restart the API server and try to create the Deployment once more. Restart the API server container by running the following command in your Minikube session:

```
docker start $(docker ps -a | grep kube-apiserver | grep -v pause | awk
'{print $1}')
```

This command tries to find the container ID of the stopped container carrying the API server, and then it starts it. You should get a response like this:



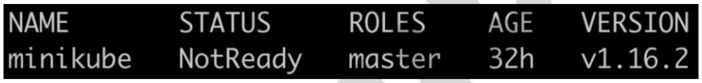
```
9e1cf098b67c
```

Figure 2.54: Starting up the container for the Kubernetes API server

10. Wait for 10 seconds. Then, check whether the API server is online. You can run any simple kubectl command for this. Let's try getting the list of nodes by running the following command in the host session:

```
kubectl get node
```

You should see the following response:



```
NAME       STATUS     ROLES    AGE   VERSION
minikube   NotReady   master   32h   v1.16.2
```

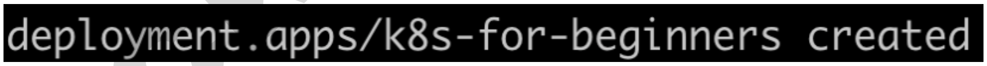
Figure 2.55: Getting the list of nodes

As you can see, we are able to get a response without errors.

11. Let's try to create the Deployment again:

```
kubectl apply -f k8s-for-beginners-deploy2.yaml
```

You should see the following response:



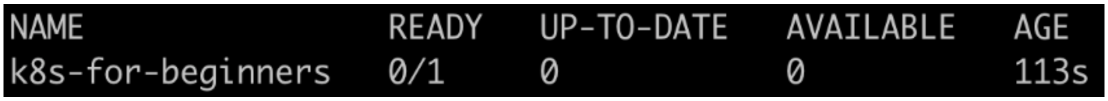
```
deployment.apps/k8s-for-beginners created
```

Figure 2.56: Trying to create the new Deployment

12. Let's check whether the Deployment has been created successfully by running the following command:

```
kubectl get deploy
```

You should see the following response:



```
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
k8s-for-beginners   0/1     0            0           113s
```

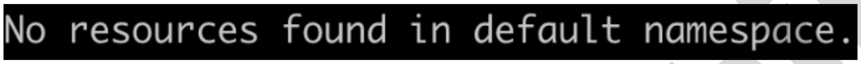
Figure 2.57: Getting the list of Deployments

From the preceding screenshot, there seems to be something wrong as in the **READY** column, we can see **0/1**, which indicates that there are 0 pods associated with this Deployment, while the desired number is 1 (which we specified in the **replicas** field in the Deployment spec).

13. Let's check that all the pods that are online:

```
kubectl get pod
```

You should get a response similar to the following:



```
No resources found in default namespace.
```

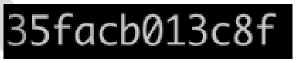
Figure 2.58: Getting the list of pods

We can see that our pod has not been created. This is because the Kubernetes API server only creates the API objects; the implementation of any API object is carried out by other components. For example, in the case of Deployment, it's **kube-controller-manager** that creates the corresponding pod(s).

14. Now, let's restart **kube-controller-manager**. Run the following command in your Minikube session:

```
docker start $(docker ps -a | grep kube-controller-manager | grep -v pause |
awk '{print $1}')
```

You should see a response similar to the following:



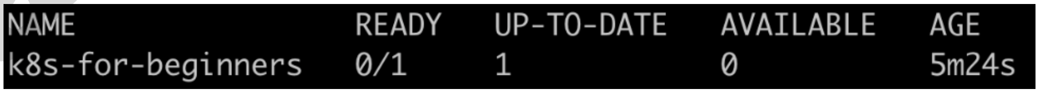
```
35facb013c8f
```

Figure 2.59: Starting up the container for the controller manager

15. After waiting for a few seconds, check the status of the Deployment by running the following command in the host session:

```
kubectl get deploy
```

You should see the following response:



```
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
k8s-for-beginners   0/1     1            0           5m24s
```

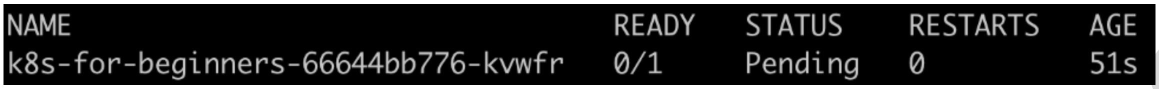
Figure 2.60: Getting the list of Deployments

As we can see, the pod that we are looking for is still not online.

16. Now, check the status of the pod:

```
kubectl get pod
```

You should see the following response:



```
NAME                                READY   STATUS    RESTARTS   AGE
k8s-for-beginners-66644bb776-kvwfr  0/1     Pending   0          51s
```

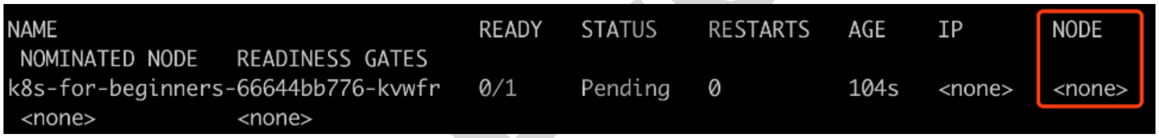
Figure 2.61: Getting the list of pods

This is different from *Figure* 2.58, as in this case, one pod was created by **kube-controller-manager**. However, we can see **Pending** under the **STATUS** column. This is because assigning a pod to a suitable node is not the responsibility of **kube-controller-manager**; it's the responsibility of kube-scheduler.

17. Before starting kube-scheduler, let's take a look at some additional information about the pod:

```
kubectl get pod -o wide
```

You should see the following response:



```
NAME                                READY   STATUS    RESTARTS   AGE    IP       NODE
 NOMINATED NODE   READINESS GATES
k8s-for-beginners-66644bb776-kvwfr  0/1     Pending   0          104s   <none>   <none>
 <none>           <none>
```

Figure 2.62: Getting more information about the pod

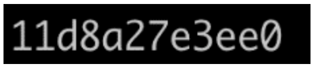
The highlighted **NODE** column indicates that no node has been assigned to this pod. This proves that the scheduler is not working properly, which we know because we took it offline. If the scheduler were to be online, this response would indicate that there is no place to land this pod.

> **Note**
>
> You will learn a lot more about pod scheduling in *Chapter 17, Advanced Scheduling in Kubernetes*.

18. Let's restart **kube-scheduler** by running the following command in the Minikube session:

```
docker start $(docker ps -a | grep kube-scheduler | grep -v pause | awk
'{print $1}')
```

You should see a response similar to the following:



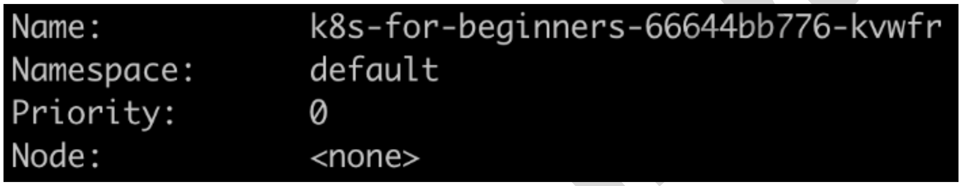
```
11d8a27e3ee0
```

Figure 2.63: Starting up the container for the Kubernetes scheduler

19. We can verify that **`kube-scheduler`** is working by running the following command in the host session:

```
kubectl describe pod k8s-for-beginners-66644bb776-kvwfr
```

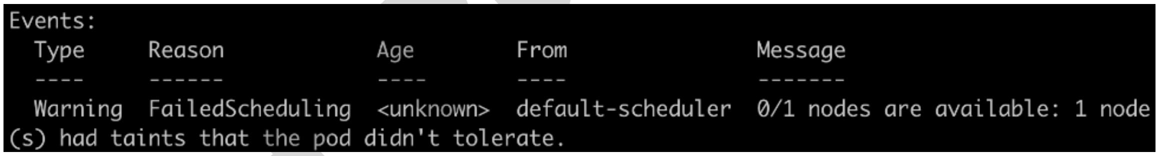
Please get the pod name from the response you get at *step* 17, as seen in *Figure* 2.62. You should see the following output:



```
Name:           k8s-for-beginners-66644bb776-kvwfr
Namespace:      default
Priority:       0
Node:           <none>
```

Figure 2.64: Describing the pod

We are truncating the output screenshots for better presentation. Please take a look at the following excerpt, highlighting the **`Events`** section:



```
Events:
  Type     Reason            Age        From               Message
  ----     ------            ----       ----               -------
  Warning  FailedScheduling  <unknown>  default-scheduler  0/1 nodes are available: 1 node
(s) had taints that the pod didn't tolerate.
```

Figure 2.65: Examining the events reported by the pod

In the **`Events`** section, we can see that **`kube-scheduler`** has tried scheduling, but it reports that there is no node available. Why is that?

This is because, earlier, we stopped the kubelet, and the Minikube environment is a single-node cluster, so there is no available node(s) with a functioning kubelet for the pod to be placed.

20. Let's restart the kubelet by running the following command in the Minikube session:

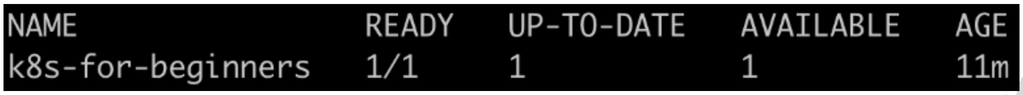
```
sudo systemctl start kubelet
```

This should not give any response in the Terminal upon successful execution.

21. In the host Terminal, verify the status of the Deployment by running the following command in the host session:

```
kubectl get deploy
```

You should see the following response:



```
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
k8s-for-beginners    1/1     1            1           11m
```

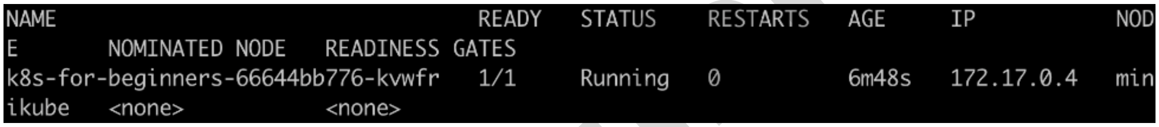
Figure 2.66: Getting the list of Deployments

Now, everything looks healthy as the Deployment shows **`1/1`** under the **`READY`** column, which means that the pod is online.

22. Similarly, verify the status of the pod:

```
kubectl get pod -o wide
```

You should get an output similar to the following:



```
NAME                               READY   STATUS    RESTARTS   AGE     IP           NOD
E       NOMINATED NODE   READINESS GATES
k8s-for-beginners-66644bb776-kvwfr   1/1     Running   0          6m48s   172.17.0.4   min
ikube   <none>           <none>
```

Figure 2.67: Getting more information about the pod

We can see **`Running`** under **`STATUS`** and that it's been assigned to the **`minikube`** node.

In this exercise, we traced each phase of a pod's life cycle by breaking the Kubernetes components and then recovering them one by one. Now, based on the observations we made about this exercise, we have better clarity regarding the answers to the questions that were raised before this exercise:

- **Steps 12 – 16**: We saw that in the case of a Deployment, a controller manager is responsible for requesting the creation of pods.
- **Steps 17 – 19**: The scheduler is responsible for choosing a node to place in the pod. It assigns the node by setting a pod's **`nodeName`** spec to the desired node. Associating a pod to a node, at this moment, merely happened at the level of the API object.
- **Steps 20 – 22**: The kubelet actually brings up the containers to get our pod running.

Throughout a pod's life cycle, Kubernetes components cooperate by updating a pod's spec properly. The API server serves as the key component that accepts pod update requests, as well as to report pod changes to interested parties.

In the following activity, we will bring together the skills we learned in the chapter to find out how we can migrate from a container-based environment to a Kubernetes environment in order to run our application.

Activity 2.01: Running the Pageview App in Kubernetes

In *Activity 1.01*, *Creating a Simple Page Count Application*, in the previous chapter, we built a web application called Pageview and connected it to a Redis backend datastore. So, here is a question: without making any changes to the source code, can we migrate the Docker-based application to Kubernetes and enjoy Kubernetes' benefits immediately? Try it out in this activity with the guidelines given.

This activity is divided into two parts: in the first part, we will create a simple pod with our application that is exposed to traffic outside the cluster by a Service and connected to a Redis datastore running as another pod. In the second part, we will scale the application to three replicas.

Connecting the Pageview App to a Redis Datastore Using a Service

Similar to the **`--link`** option in Docker, Kubernetes provides a Service that serves as an abstraction layer to expose one application (let's say, a series of pods tagged with the same set of labels) that can be accessed internally or externally. For example, as we discussed in this chapter, a frontend app can be exposed via a NodePort Service so that it can be accessed by external users. In addition to that, in this activity, we need to define an internal Service in order to expose the backend application to the frontend application. Follow these steps:

1. In *Activity 1.01*, *Creating a Simple Page Count Application*, we built two Docker images – one for the frontend Pageview web app and another for the backend Redis datastore. You can use the skills we learned in this chapter to migrate them into Kubernetes YAMLs.

2. Two pods (each managed by a Deployment) for the application is not enough. We also have to create the Service YAML to link them together.

   Ensure that the **`targetPort`** field in the manifest is consistent with the exposed port that was defined in the Redis image, which was **`6379`** in this case. In terms of the **`port`** field, theoretically, it can be any port, as long as it's consistent with the one specified in the Pageview application.

   The other thing worth mentioning here is the **`name`** field of the pod for Redis datastore. It's the symbol that's used in the source code of the Pageview app to reference the Redis datastore.

Now, you should have three YAMLs – two pods and a Service. Apply them using **`kubectl -f <yaml file name>`**, and then use **`kubectl get deploy,service`** to ensure that they're created successfully.

3. At this stage, the Pageview app should function well since it's connected to the Redis app via the Service. However, the Service only works as the internal connector to ensure they can talk to each other inside the cluster.

   To access the Pageview app externally, we need to define a NodePort Service. Unlike the internal Service, we need to explicitly specify the **`type`** as **`NodePort`**.
4. Apply the external Service YAML using **`kubectl -f <yaml file name>`**.
5. Run **`minikube service <external service name>`** to fetch the Service URL.
6. Access the URL multiple times to ensure that the Pageview number gets increased by one each time.

With that, we have successfully run the Pageview application in Kubernetes. But what if the Pageview app is down? Although Kubernetes can create a replacement pod automatically, there is still downtime between when the failure is detected and when the new pod is ready.

A common solution is to increase the replica number of the application so that the whole application is available as long as there is at least one replica running.

Running the Pageview App in Multiple Replicas

The Pageview app can certainly work with a single replica. However, in a production environment, high availability is essential and is achieved by maintaining multiple replicas across nodes to avoid single points of failure. (This will be covered in detail in upcoming chapters.)

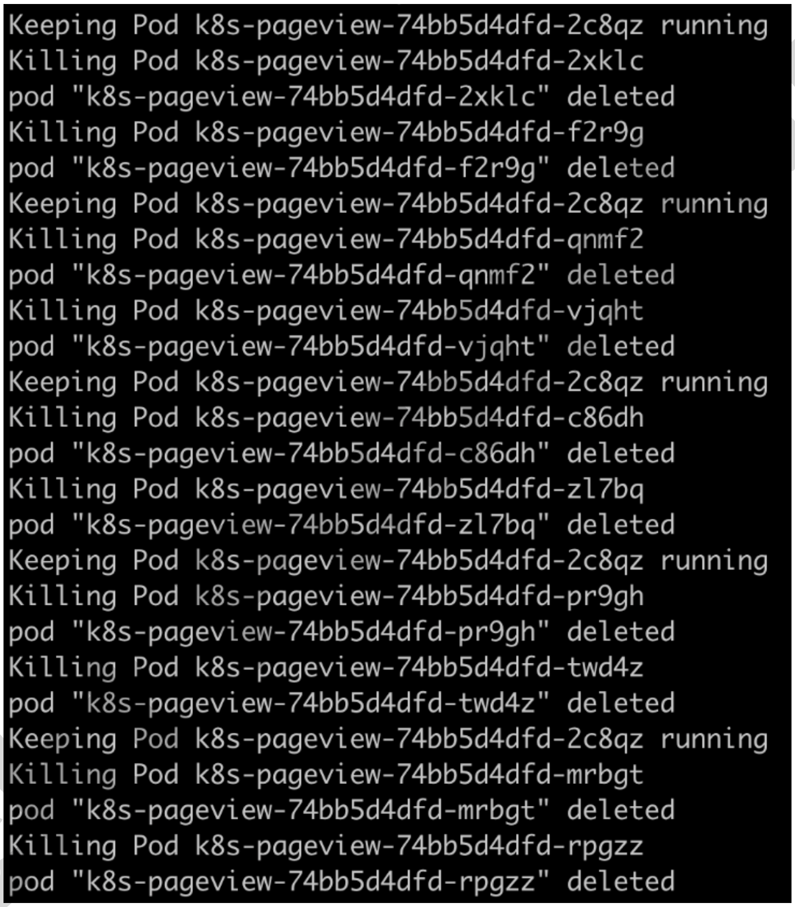
In Kubernetes, to ensure the high availability of an application, we can simply increase the replica number. Follow these steps to do so:

1. Modify the Pageview YAML to change **`replicas`** to **`3`**.
2. Apply these changes by running **`kubectl apply -f <pageview app yaml>`**.
3. By running **`kubectl get pod`**, you should be able to see three Pageview pods running.
4. Access the URL shown in the output of the **`minikube service`** command multiple times.

   Check the logs of each pod to see whether the requests are handled evenly among the three pods.

5. Now, let's verify the high availability of the Pageview app. Terminate any arbitrary pods continuously while keeping one healthy pod. You can achieve this manually or automatically by writing a script. Alternatively, you can open another Terminal and check whether the Pageview app is always accessible.

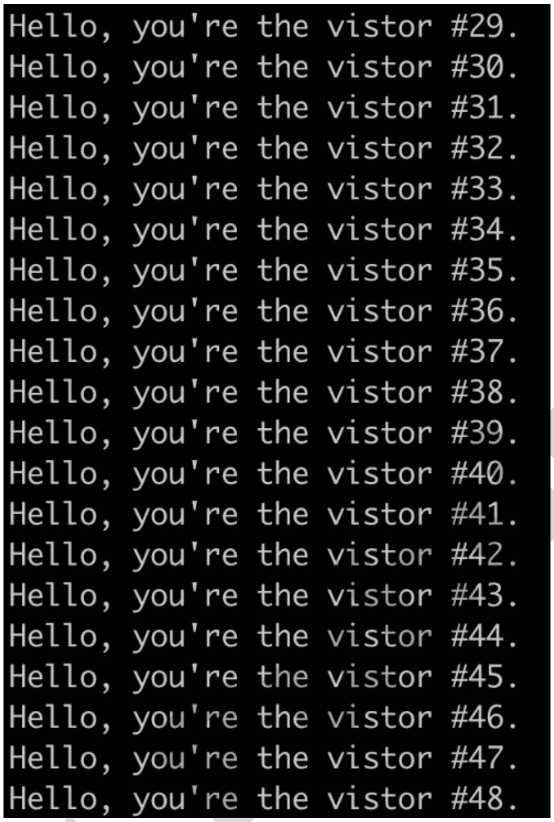
If you opt for writing scripts to terminate the pods, you will see results similar to the following:



```
Keeping Pod k8s-pageview-74bb5d4dfd-2c8qz running
Killing Pod k8s-pageview-74bb5d4dfd-2xklc
pod "k8s-pageview-74bb5d4dfd-2xklc" deleted
Killing Pod k8s-pageview-74bb5d4dfd-f2r9g
pod "k8s-pageview-74bb5d4dfd-f2r9g" deleted
Keeping Pod k8s-pageview-74bb5d4dfd-2c8qz running
Killing Pod k8s-pageview-74bb5d4dfd-qnmf2
pod "k8s-pageview-74bb5d4dfd-qnmf2" deleted
Killing Pod k8s-pageview-74bb5d4dfd-vjqht
pod "k8s-pageview-74bb5d4dfd-vjqht" deleted
Keeping Pod k8s-pageview-74bb5d4dfd-2c8qz running
Killing Pod k8s-pageview-74bb5d4dfd-c86dh
pod "k8s-pageview-74bb5d4dfd-c86dh" deleted
Killing Pod k8s-pageview-74bb5d4dfd-zl7bq
pod "k8s-pageview-74bb5d4dfd-zl7bq" deleted
Keeping Pod k8s-pageview-74bb5d4dfd-2c8qz running
Killing Pod k8s-pageview-74bb5d4dfd-pr9gh
pod "k8s-pageview-74bb5d4dfd-pr9gh" deleted
Killing Pod k8s-pageview-74bb5d4dfd-twd4z
pod "k8s-pageview-74bb5d4dfd-twd4z" deleted
Keeping Pod k8s-pageview-74bb5d4dfd-2c8qz running
Killing Pod k8s-pageview-74bb5d4dfd-mrbgt
pod "k8s-pageview-74bb5d4dfd-mrbgt" deleted
Killing Pod k8s-pageview-74bb5d4dfd-rpgzz
pod "k8s-pageview-74bb5d4dfd-rpgzz" deleted
```

Figure 2.68: Killing pods via a script

Assuming that you take a similar approach and write a script to check whether the application is online, you should see an output similar to the following:



```
Hello, you're the vistor #29.
Hello, you're the vistor #30.
Hello, you're the vistor #31.
Hello, you're the vistor #32.
Hello, you're the vistor #33.
Hello, you're the vistor #34.
Hello, you're the vistor #35.
Hello, you're the vistor #36.
Hello, you're the vistor #37.
Hello, you're the vistor #38.
Hello, you're the vistor #39.
Hello, you're the vistor #40.
Hello, you're the vistor #41.
Hello, you're the vistor #42.
Hello, you're the vistor #43.
Hello, you're the vistor #44.
Hello, you're the vistor #45.
Hello, you're the vistor #46.
Hello, you're the vistor #47.
Hello, you're the vistor #48.
```

Figure 2.69: Repeatedly accessing the application via the script

> **Note**
>
> The solution to this activity can be found in the appendix at the end of this book.

A Glimpse into the Advantages of Kubernetes for Multi-Node Clusters

You can only truly appreciate the advantages of Kubernetes after seeing it in the context of a multi-node cluster. This chapter, like many of the other chapters in this book, uses a single-node cluster (Minikube environment) to demonstrate the features that Kubernetes provides. However, in a real-world production environment, Kubernetes is deployed with multiple workers and master nodes. Only then can you ensure that a fault in a single node won't impact the general availability of the application. And reliability is just one of the many benefits that a multi-node Kubernetes cluster can bring to us.

But wait – isn't it true that we can implement applications and deploy them in a highly available manner *without using Kubernetes*? That's true, but that usually comes with a lot of management hassle, both in terms of managing the application as well as the infrastructure. For example, during the initial Deployment, you may have to intervene manually to ensure that all redundant containers are not running on the same machine. In the case of a node failure, you will have to not only ensure that a new replica is respawned properly, but to guarantee high availability, you also need to ensure that the new one doesn't land on the nodes that are already running existing replicas. This can be achieved either by using a DevOps tool or injecting logic on the application side. However, either way is very complex. Kubernetes provides a unified platform that we can use to wire apps to proper nodes by describing the high-availability features we want using Kubernetes primitives (API objects). This pattern frees the minds of application developers, as they only need to consider how to build their applications. Features that are required for high availability, such as failure detection and recovery, are taken care of by Kubernetes under the hood.

Summary

In this chapter, we used Minikube to provision a single-node Kubernetes cluster and gave a high-level overview of Kubernetes' core components, as well as its key design rationale. After that, we migrated an existing Docker container to Kubernetes and explored some basic Kubernetes API objects, such as pods, Services, and Deployments. Lastly, we intentionally broke a Kubernetes cluster and restored it one component at a time, which allowed us to understand how the different Kubernetes components work together to get a pod up and running on a node.

Throughout this chapter, we have used Kubectl to manage our cluster. We provided a quick introduction to this tool, but in the following chapter, we will take a closer look at this powerful tool and explore the various ways in which we can use it.