What's the problem? Actually, it's not a problem but a feature! Docker containers are isolated from the host operating system and other containers, not only at the filesystem level but also in terms of networking. So while the container really is listening on port 5000, that is only on a port *inside* the container, which isn't accessible on the host OS. If you want to expose a port from the container on the host OS, you have to do it via the `-p` flag.

First, hit Ctrl-C to shut down the `training/webapp` container: note that it's C this time, not D, and it's Ctrl regardless of OS, as you're shutting down a process, rather than exiting an interactive prompt. Now rerun the container but this time with the `-p` flag as follows:

```
$ docker run -p 5000:5000 training/webapp
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Adding `-p 5000:5000` to the command tells Docker to expose port 5000 inside the container on port 5000 of the host OS. In another terminal on your host OS, you should now be able to see the web app working:

```
$ curl localhost:5000
Hello world!
```

CLEANING UP CONTAINERS

Every time you run `docker run` and exit, you are leaving behind containers, which take up disk space. You may wish to clean them up with the `docker rm <CONTAINER_ID>` command, where `CONTAINER_ID` is the ID of the container from the `docker ps` output. Alternatively, you could include the `--rm` flag in your `docker run` command to have Docker automatically clean up when you exit the container.

A Crash Course on Kubernetes

Kubernetes is an orchestration tool for Docker, which means it's a platform for running and managing Docker containers on your servers, including scheduling (picking which servers should run a given container workload),

auto healing (automatically redeploying containers that failed), auto scaling (scaling the number of containers up and down in response to load), load balancing (distributing traffic across containers), and much more.

Under the hood, Kubernetes consists of two main pieces:

Control plane

The control plane is responsible for managing the Kubernetes cluster. It is the "brains" of the operation, responsible for storing the state of the cluster, monitoring containers, and coordinating actions across the cluster. It also runs the API server, which provides an API you can use from command-line tools (e.g., `kubectl`), web UIs (e.g., the Kubernetes Dashboard), and IaC tools (e.g., Terraform) to control what's happening in the cluster.

Worker nodes

The worker nodes are the servers used to actually run your containers. The worker nodes are entirely managed by the control plane, which tells each worker node what containers it should run.

Kubernetes is open source, and one of its strengths is that you can run it anywhere: in any public cloud (e.g., AWS, Azure, Google Cloud), in your own datacenter, and even on your own developer workstation. A little later in this chapter, I'll show you how you can run Kubernetes in the cloud (in AWS), but for now, let's start small and run it locally. This is easy to do if you installed a relatively recent version of Docker Desktop, as it has the ability to fire up a Kubernetes cluster locally with just a few clicks.

If you open Docker Desktop's preferences on your computer, you should see Kubernetes in the nav, as shown in Figure 7-7.

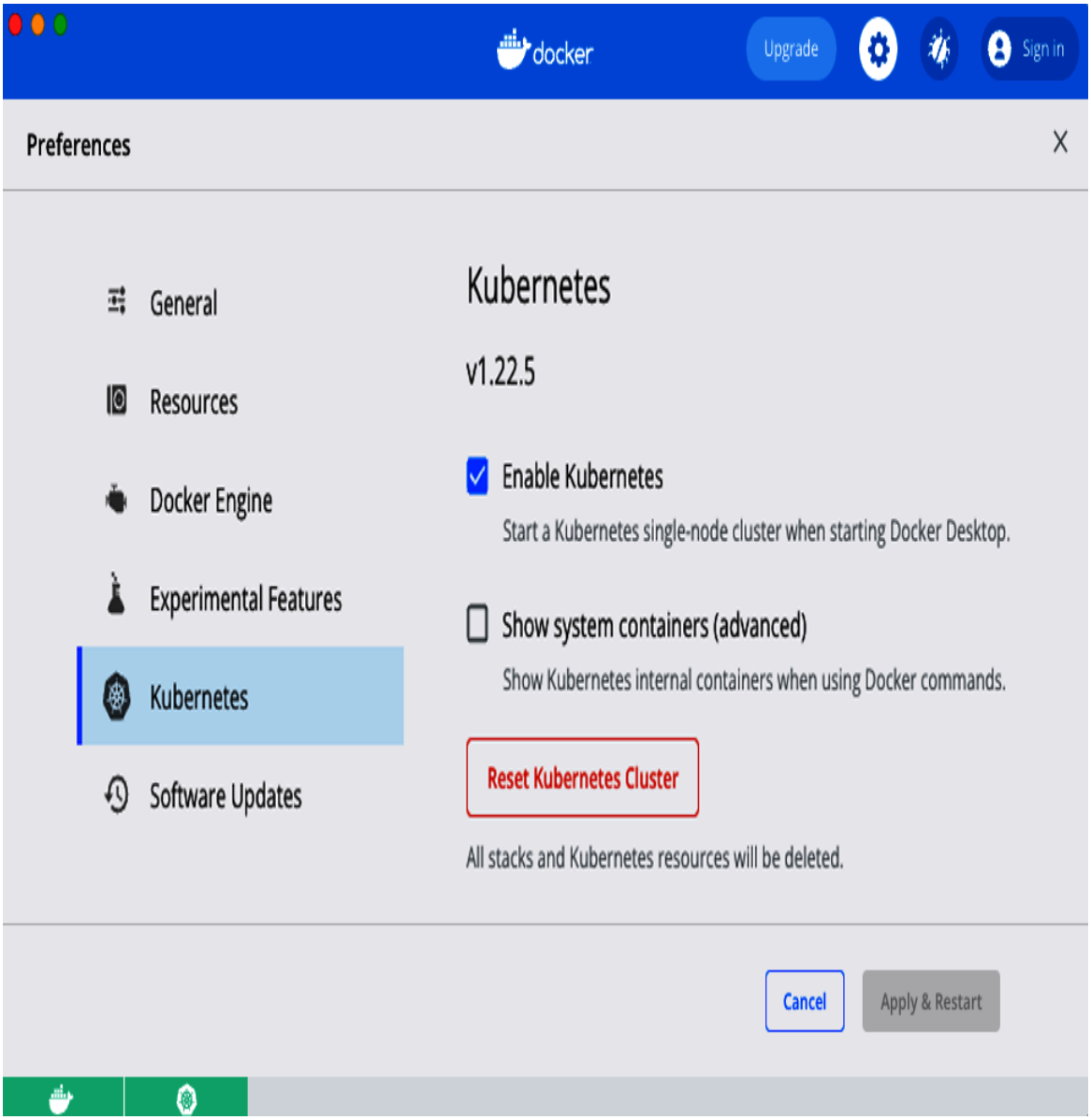

Figure 7-7. Enable Kubernetes on Docker Desktop.

If it's not enabled already, check the Enable Kubernetes checkbox, click Apply & Restart, and wait a few minutes for that to complete. In the meantime, follow the instructions on the Kubernetes website to install `kubectl`, which is the command-line tool for interacting with Kubernetes.

To use `kubectl`, you must first update its configuration file, which lives in *$HOME/.kube/config* (that is, the *.kube* folder of your home directory), to tell it what Kubernetes cluster to connect to. Conveniently, when you enable Kubernetes in Docker Desktop, it updates this config file for you, adding a

`docker-desktop` entry to it, so all you need to do is tell `kubectl` to use this configuration as follows:

```
$ kubectl config use-context docker-desktop
Switched to context "docker-desktop".
```

Now you can check if your Kubernetes cluster is working with the `get nodes` command:

```
$ kubectl get nodes
NAME             STATUS   ROLES                  AGE   VERSION
docker-desktop   Ready    control-plane,master   95m   v1.22.5
```

The `get nodes` command shows you information about all the nodes in your cluster. Since you're running Kubernetes locally, your computer is the only node, and it's running both the control plane and acting as a worker node. You're now ready to run some Docker containers!

To deploy something in Kubernetes, you create Kubernetes *objects*, which are persistent entities you write to the Kubernetes cluster (via the API server) that record your intent: e.g., your intent to have specific Docker images running. The cluster runs a *reconciliation loop*, which continuously checks the objects you stored in it and works to make the state of the cluster match your intent.

There are many different types of Kubernetes objects available. For the examples in this book, let's use the following two objects:

Kubernetes Deployment

> A *Kubernetes Deployment* is a declarative way to manage an application in Kubernetes. You declare what Docker images to run, how many copies of them to run (called *replicas*), a variety of settings for those images (e.g., CPU, memory, port numbers, environment variables), and the strategy to roll out updates to those images, and the Kubernetes Deployment will then work to ensure that the requirements you declared are always met. For example, if you specified you wanted three replicas, but one of the worker nodes went down so only two

replicas are left, the Deployment will automatically spin up a third replica on one of the other worker nodes.

Kubernetes Service

A *Kubernetes Service* is a way to expose a web app running in Kubernetes as a networked service. For example, you can use a Kubernetes Service to configure a load balancer that exposes a public endpoint and distributes traffic from that endpoint across the replicas in a Kubernetes Deployment.

The idiomatic way to interact with Kubernetes is to create YAML files describing what you want—e.g., one YAML file that defines the Kubernetes Deployment and another one that defines the Kubernetes Service—and to use the `kubectl apply` command to submit those objects to the cluster. However, using raw YAML has drawbacks, such as a lack of support for code reuse (e.g., variables, modules), abstraction (e.g., loops, if-statements), clear standards on how to store and manage the YAML files (e.g., to track changes to the cluster over time), and so on. Therefore, many Kubernetes users turn to alternatives, such as Helm or Terraform. Since this is a book on Terraform, I'm going to show you how to create a Terraform module called `k8s-app` (K8S is an acronym for Kubernetes in the same way that I18N is an acronym for internationalization) that deploys an app in Kubernetes using a Kubernetes Deployment and Kubernetes Service.

Create a new module in the *modules/services/k8s-app* folder. Within that folder, create a *variables.tf* file that defines the module's API via the following input variables:

```
variable "name" {
  description = "The name to use for all resources created by
this module"
  type        = string
}

variable "image" {
  description = "The Docker image to run"
  type        = string
```

```
}

variable "container_port" {
  description = "The port the Docker image listens on"
  type        = number
}

variable "replicas" {
  description = "How many replicas to run"
  type        = number
}

variable "environment_variables" {
  description = "Environment variables to set for the app"
  type        = map(string)
  default     = {}
}
```

This should give you just about all the inputs you need for creating the Kubernetes Deployment and Service. Next, add a *main.tf* file, and at the top, add the `required_providers` block to it with the Kubernetes provider:

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = "~> 2.0"
    }
  }
}
```

Hey, a new provider, neat! OK, let's make use of that provider to create a Kubernetes Deployment by using the `kubernetes_deployment` resource:

```
resource "kubernetes_deployment" "app" {
}
```

There are quite a few settings to configure within the `kubernetes_deployment` resource, so let's go through them one at a time. First, you need to configure the `metadata` block:

```
resource "kubernetes_deployment" "app" {
  metadata {
    name = var.name
  }
}
```

Every Kubernetes object includes metadata that can be used to identify and target that object in API calls. In the preceding code, I'm setting the Deployment name to the `name` input variable.

The rest of the configuration for the `kubernetes_deployment` resource goes into the `spec` block:

```
resource "kubernetes_deployment" "app" {
  metadata {
    name = var.name
  }

  spec {
  }
}
```

The first item to put into the `spec` block is to specify the number of replicas to create:

```
  spec {
    replicas = var.replicas
  }
```

Next, define the `template` block:

```
  spec {
    replicas = var.replicas

    template {
    }
  }
```

In Kubernetes, instead of deploying one container at a time, you deploy *Pods*, which are groups of containers that are meant to be deployed together. For example, you could have a Pod with one container to run a web app (e.g., the Python app you saw earlier) and another container that gathers metrics on the web app and sends them to a central service (e.g., Datadog). The `template` block is where you define the *Pod Template*, which specifies what container(s) to run, the ports to use, environment variables to set, and so on.

One important ingredient in the Pod Template will be the labels to apply to the Pod. You'll need to reuse these labels in several places—e.g., the Kubernetes Service uses labels to identify the Pods that need to be load balanced—so let's define those labels in a local variable called `pod_labels`:

```
locals {
  pod_labels = {
    app = var.name
  }
}
```

And now use `pod_labels` in the `metadata` block of the Pod Template:

```
  spec {
    replicas = var.replicas

    template {
      metadata {
        labels = local.pod_labels
      }
    }
  }
```

Next, add a `spec` block inside of `template`:

```
  spec {
    replicas = var.replicas

    template {
      metadata {
```

```
      labels = local.pod_labels
    }

    spec {
      container {
        name  = var.name
        image = var.image

        port {
          container_port = var.container_port
        }

        dynamic "env" {
          for_each = var.environment_variables
          content {
            name  = env.key
            value = env.value
          }
        }
      }
    }
  }
}
```

There's a lot here, so let's go through it one piece at a time:

`container`

Inside the `spec` block, you can define one or more `container` blocks to specify which Docker containers to run in this Pod. To keep this example simple, there's just one `container` block in the Pod. The rest of these items are all within this `container` block.

`name`

The name to use for the container. I've set this to the `name` input variable.

`image`

The Docker image to run in the container. I've set this to the `image` input variable.

port

The ports to expose in the container. To keep the code simple, I'm assuming the container only needs to listen on one port, set to the `container_port` input variable.

env

The environment variables to expose to the container. I'm using a `dynamic` block with `for_each` (two concepts you may remember from Chapter 5) to set this to the variables in the `environment_variables` input variable.

OK, that wraps up the Pod Template. There's just one thing left to add to the `kubernetes_deployment` resource—a `selector` block:

```
spec {
  replicas = var.replicas

  template {
    metadata {
      labels = local.pod_labels
    }

    spec {
      container {
        name  = var.name
        image = var.image

        port {
          container_port = var.container_port
        }

        dynamic "env" {
          for_each = var.environment_variables
          content {
            name  = env.key
            value = env.value
          }
        }
      }
    }
```

```
    }

    selector {
      match_labels = local.pod_labels
    }
  }
```

The `selector` block tells the Kubernetes Deployment what to target. By setting it to `pod_labels`, you are telling it to manage deployments for the Pod Template you just defined. Why doesn't the Deployment just assume that the Pod Template defined within that Deployment is the one you want to target? Well, Kubernetes tries to be an extremely flexible and decoupled system: e.g., it's possible to define a Deployment for Pods that are defined separately, so you always need to specify a `selector` to tell the Deployment what to target.

That wraps up the `kubernetes_deployment` resource. The next step is to use the `kubernetes_service` resource to create a Kubernetes Service:

```
resource "kubernetes_service" "app" {
  metadata {
    name = var.name
  }

  spec {
    type = "LoadBalancer"
    port {
      port        = 80
      target_port = var.container_port
      protocol    = "TCP"
    }
    selector = local.pod_labels
  }
}
```

Let's go through these parameters:

`metadata`

> Just as with the Deployment object, the Service object uses metadata to identify and target that object in API calls. In the preceding code, I've

set the Service name to the `name` input variable.

`type`

I've configured this Service as type `LoadBalancer`, which, depending on how your Kubernetes cluster is configured, will deploy a different type of load balancer: e.g., in AWS, with EKS, you might get an Elastic Load Balancer, whereas in Google Cloud, with GKE, you might get a Cloud Load Balancer.

`port`

I'm configuring the load balancer to route traffic on port 80 (the default port for HTTP) to the port the container is listening on.

`selector`

Just as with the Deployment object, the Service object uses a selector to specify what that Service should be targeting. By setting the selector to `pod_labels`, the Service and the Deployment will both operate on the same Pods.

The final step is to expose the Service endpoint (the load balancer hostname) as an output variable in *outputs.tf*:

```
locals {
  status = kubernetes_service.app.status
}

output "service_endpoint" {
  value = try(
    "http://${local.status[0]["load_balancer"][0]["ingress"][0]
["hostname"]}",
    "(error parsing hostname from status)"
  )
  description = "The K8S Service endpoint"
}
```

This convoluted code needs a bit of explanation. The `kubernetes_service` resource has an output attribute called `status` that returns the latest status of the Service. I've stored this attribute in a local variable called `status`. For a Service of type LoadBalancer, `status` will contain a complicated object that looks something like this:

```
[
  {
    load_balancer = [
      {
        ingress = [
          {
            hostname = "<HOSTNAME>"
          }
        ]
      }
    ]
  }
]
```

Buried within this deeply nested object is the hostname for the load balancer that you want. This is why the `service_endpoint` output variable needs to use a complicated sequence of array lookups (e.g., `[0]`) and map lookups (e.g., `["load_balancer"]`) to extract the hostname. But what happens if the `status` attribute returned by the `kubernetes_service` resource happens to look a little different? In that case, any of those array and map lookups could fail, leading to a confusing error.

To handle this error gracefully, I've wrapped the entire expression in a function called `try`. The `try` function has the following syntax:

```
try(ARG1, ARG2, ..., ARGN)
```

This function evaluates all the arguments you pass to it and returns the first argument that doesn't produce any errors. Therefore, the `service_endpoint` output variable will either end up with a hostname in it (the first argument) or, if reading the hostname caused an error, the

variable will instead say “error parsing hostname from status” (the second argument).

OK, that wraps up the `k8s-app` module. To use it, add a new example in *examples/kubernetes-local*, and create a *main.tf* file in it with the following contents:

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000
}
```

This configures the module to deploy the `training/webapp` Docker image you ran earlier, with two replicas listening on port 5000, and to name all the Kubernetes objects (based on their `metadata`) “simple-webapp”. To have this module deploy into your local Kubernetes cluster, add the following `provider` block:

```
provider "kubernetes" {
  config_path    = "~/.kube/config"
  config_context = "docker-desktop"
}
```

This code tells the Kubernetes provider to authenticate to your local Kubernetes cluster by using the `docker-desktop` context from your `kubectl` config. Run `terraform apply` to see how it works:

```
$ terraform apply

(...)

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

service_endpoint = "http://localhost"
```

Give the app a few seconds to boot and then try out that `service_endpoint`:

```
$ curl http://localhost
Hello world!
```

Success!

That said, this looks nearly identical to the output of the `docker run` command, so was all that extra work worth it? Well, let's look under the hood to see what's going on. You can use `kubectl` to explore your cluster. First, run the `get deployments` command:

```
$ kubectl get deployments
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
simple-webapp   2/2     2            2           3m21s
```

You can see your Kubernetes Deployment, named `simple-webapp`, as that was the name in the `metadata` block. This Deployment is reporting that 2/2 Pods (the two replicas) are ready. To see those Pods, run the `get pods` command:

```
$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
simple-webapp-d45b496fd-7d447   1/1     Running   0
2m36s
simple-webapp-d45b496fd-vl6j7   1/1     Running   0
2m36s
```

So that's one difference from `docker run` already: there are multiple containers running here, not just one. Moreover, those containers are being actively monitored and managed. For example, if one crashed, a replacement will be deployed automatically. You can see this in action by running the `docker ps` command:

```
$ docker ps
CONTAINER ID   IMAGE             COMMAND           CREATED
STATUS
```

```
b60f5147954a   training/webapp   "python app.py"   3 seconds ago
Up 2 seconds
c350ec648185   training/webapp   "python app.py"   12 minutes ago
Up 12 minutes
```

Grab the `CONTAINER ID` of one of those containers, and use the `docker kill` command to shut it down:

```
$ docker kill b60f5147954a
```

If you run `docker ps` again very quickly, you'll see just one container left running:

```
$ docker ps
CONTAINER ID   IMAGE             COMMAND           CREATED
STATUS
c350ec648185   training/webapp   "python app.py"   12 minutes ago
Up 12 minutes
```

But just a few seconds later, the Kubernetes Deployment will have detected that there is only one replica instead of the requested two, and it'll launch a replacement container automatically:

```
$ docker ps
CONTAINER ID   IMAGE             COMMAND           CREATED
STATUS
56a216b8a829   training/webapp   "python app.py"   1 second ago
Up 5 seconds
c350ec648185   training/webapp   "python app.py"   12 minutes ago
Up 12 minutes
```

So Kubernetes is ensuring that you always have the expected number of replicas running. Moreover, it is also running a load balancer to distribute traffic across those replicas, which you can see by running the `kubectl get services` command:

```
$ kubectl get services
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP
PORT(S)        AGE
kubernetes      ClusterIP      10.96.0.1      <none>
```

```
443/TCP         4h26m
simple-webapp   LoadBalancer   10.110.25.79   localhost
80:30234/TCP    4m58s
```

The first service in the list is Kubernetes itself, which you can ignore. The second is the Service you created, also with the name `simple-webapp` (based on the `metadata` block). This service runs a load balancer for your app: you can see the IP it's accessible at (`localhost`) and the port it's listening on (80).

Kubernetes Deployments also provide automatic rollout of updates. A fun trick with the `training/webapp` Docker image is that if you set the environment variable `PROVIDER` to some value, it'll use that value instead of the word *world* in the text "Hello, world!" Update *examples/kubernetes-local/main.tf* to set this environment variable as follows:

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Terraform"
  }
}
```

Run `apply` one more time:

```
$ terraform apply

(...)

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.

Outputs:

service_endpoint = "http://localhost"
```