APPENDIX

CHAPTER 1: UNDERSTANDING KUBERNETES AND CONTAINERS

ACTIVITY 1.01: CREATING A SIMPLE PAGE COUNT APPLICATION

Solution:

While you can use any programming language and datastore of your choice, we will go with Golang and Redis for this solution.

A PageView Web App

1. Create a **`main.go`** file using the following source code:

```
package main

import (
        "fmt"
        "log"
        "net/http"
)

var pageView int64

func main() {
        http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe("0.0.0.0:8080", nil))
}

func handler(w http.ResponseWriter, r *http.Request) {
        log.Printf("Ping from %s", r.RemoteAddr)
        pageView++
        fmt.Fprintf(w, "Hello, you're visitor #%d !", pageView)
}
```

2. Create a file named **`Dockerfile`** with the following content:

```
FROM alpine:3.10

COPY main /

CMD ["/main"]
```

3. Now, build the executable **pageview-v1** binary using the following command:

```
CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o main
```

4. Use the following command to build the Docker image:

```
docker build -t <username>/pageview:v0.0.1 .
```

5. Now, push it to Docker Hub using this command. Please use your Docker Hub username here:

```
docker push <username>/pageview:v0.0.1
```

Create the **pageview-v1** container and map port 8080 of the host to the same port on the container, as shown here:

```
docker run --name pageview -p 8080:8080 -d <username>/pageview:v0.0.1
```

Use the appropriate **username** and tag.

> **NOTE**
>
> If you are having problems with building your own image, you can use the one that we have provided by using this repository path in the preceding command: **packtworkshops/the-kubernetes-workshop:pageview**.

You should see the following response:

```
6cdcf461356e147cdfc2ef61104600588daddb387dff5f974954e46d35940827
```

6. Access the application multiple times to ensure that the pageview number increases every time. Run the following command repeatedly:

```
curl localhost:8080
```

You should see a response similar to the following:

```
root@ubuntu:~# curl localhost: 8080
Hello, you're visitor #1.
root@ubuntu:~# curl localhost: 8080
Hello, you're visitor #2.
root@ubuntu:~# curl localhost: 8080
Hello, you're visitor #3.
```

Bonus Objective: A Backend Datastore

1. We will use Redis for this solution. Use the following command to fetch and run the Redis container:

```
docker run --name db -d redis
```

The following response indicates that Redis has been downloaded and started successfully:

```
Unable to find image 'redis:latest' locally
latest: Pulling from library/redis
000eee12ec04: Pull complete
5cc53381c195: Pull complete
48bb7bcb5fbf: Pull complete
ef8a890bb1c2: Pull complete
32ada9c6fb0d: Pull complete
76e034b0f296: Pull complete
Digest: sha256:1eedfc017b0cd3e232878ce38bd9328518219802a8ef37fe34f58dcf591688ef
Status: Downloaded newer image for redis:latest
afd6ec76e841fef8ae3cd35dc739f88652019ec9c7a2a088579b03222d101c17
```

Figure 1.32: Starting up Redis

The preceding command creates a container named **`db`**, using the latest **`redis`** image.

2. Now check the status of the container as well as the exposed port:

```
docker ps
```

You should see a response similar to this:

```
CONTAINER ID        IMAGE                     COMMAND                  CREATED             STATUS
          PORTS                    NAMES
afd6ec76e841        redis                     "docker-entrypoint.s…"   24 seconds ago      Up 22 seco
nds       6379/tcp                 db
6cdcf461356e        hweicdl/pageview:v0.0.1   "/main"                  2 minutes ago       Up 2 minut
es        0.0.0.0:8080->8080/tcp   pageview
```

Figure 1.33: Checking our containers

In this case, port **`6379`** is exposed, which we will use for the next part of the solution.

Bonus Objective: Modifying the Web App to Connect to a Backend Datastore

1. To persist the pageview number, we're going to connect to the backend **`db`** (Redis), and then increase the number by 1 each time. The modified version of **`main.go`** should look like this:

```
package main

import (
        "fmt"
        "log"
        "net/http"

        "github.com/go-redis/redis/v7"
)

var dbClient *redis.Client
var key = "pv"

func init() {
        dbClient = redis.NewClient(&redis.Options{
                Addr:     "db:6379",
        })
}

func main() {
        http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe("0.0.0.0:8080", nil))
}

func handler(w http.ResponseWriter, r *http.Request) {
        log.Printf("Ping from %s", r.RemoteAddr)
        pageView, err := dbClient.Incr(key).Result()
        if err != nil {
                panic(err)
        }
        fmt.Fprintf(w, "Hello, you're visitor #%v.\n", pageView)
}
```

   Here are a few things worth noting:

The database connection address is specified as **db:6379**. The **db** alias will be dynamically interpreted to its IP when the Redis container is linked to the **pageview-v2** container during runtime.

We are using **dbClient.Incr(key)** so that the pageview number increases by 1 on the **db** server-side, and the **db** server can deal with racing conditions when several clients try to write at the same moment.

2. Proceed with compiling the binary and building a container out of it, just like we did with **pageview-v1**. Use the following commands:

```
CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o main
docker build -t <username>/pageview:v0.0.2 .
docker push <username>/pageview:v0.0.2
```

3. Remove the **pageview-v1** container first, and then launch the **pageview-v2** container using these commands one after the other:

```
docker rm -f pageview
docker run --name pageview --link db:db -p 8080:8080 -d <username>/
pageview:v0.0.2
```

> **NOTE**
>
> If you are having problems with building your own image, you can use the one that we have provided by using this repository path in the preceding command: **packtworkshops/the-kubernetes-workshop:pageview-v2**.

You should see the following response:

```
acdb5f271d08a294215147371c9b113d417d93727e338059d1a66e1b4e53fc55
```

4. Let's check whether the correct containers are running by listing the containers:

```
docker ps
```

You should see a response similar to the following:

```
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS
          PORTS                    NAMES
acdb5f271d08        hweicdl/pageview:v0.0.2        "/main"                  11 seconds ago      Up 10 seco
nds       0.0.0.0:8080->8080/tcp   pageview
afd6ec76e841        redis                          "docker-entrypoint.s…"   14 minutes ago      Up 14 minu
tes       6379/tcp                 db
```

Figure 1.34: Getting a list of all of the containers

5. Access the **`pageview-v2`** multiple times, and check whether the number increases by 1 each time. Use this command repeatedly:

```
curl localhost:8080
```

You should see the following response:

```
root@ubuntu:~# curl localhost: 8080
Hello, you're visitor #1.
root@ubuntu:~# curl localhost: 8080
Hello, you're visitor #2.
root@ubuntu:~# curl localhost: 8080
Hello, you're visitor #3.
```

6. Now, let's kill and restart the **`pageview-v2`** container, and see whether the pageview number can still be calculated properly. First, let's restart the container by using the following commands one after the other:

```
docker rm -f pageview
docker run --name pageview --link db:db -p 8080:8080 -d <username>/
pageview:v0.0.2
```

You should see a response similar to the following:

```
1e04fed99b7faaf37d30245975b3b15baf4a1bf711ddc9dfb7eb29095f4b8582
```

7. Now, try accessing our application:

```
curl localhost:8080
```

You should see the following response:

```
Hello, you're visitor #4.
```

The result shows that it's the fourth visit instead of the first. In this case, we accessed our application three times before terminating it. The count then started at 4 when we restarted it. Therefore, we can conclude that the state of the application was persisted.

CHAPTER 2: AN OVERVIEW OF KUBERNETES

ACTIVITY 2.01: RUNNING THE PAGEVIEW APP IN KUBERNETES

Solution:

In this activity, we are going to migrate the Pageview app created in the previous chapter to Kubernetes.

Connecting the Pageview App to the Redis Datastore Using a Service

1. For the frontend, create a file named **`k8s-pageview-deploy.yaml`** with the following content:

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

> **NOTE**
>
> In the **`image`** field, use the same location that you used in the previous chapter. If for some reason, you don't have the image and the repository path from the previous chapter, you can use this YAML config as-is in order to use the pre-made images that are provided as part of the solution.

2. For the Redis datastore backend, create a file named **`k8s-redis-deploy.yaml`** with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: k8s-redis
spec:
  replicas: 1
  selector:
    matchLabels:
      tier: backend
  template:
    metadata:
      labels:
        tier: backend
    spec:
      containers:
      - name: k8s-redis
        image: redis
```

3. To define a Service for internal traffic routing, create a file named **`k8s-redis-internal-service.yaml`** with the following content:

```
kind: Service
apiVersion: v1
metadata:
  name: db
spec:
  selector:
    tier: backend
  ports:
  - port: 6379
    targetPort: 6379
```

4. Use the following commands one after the other to apply the three YAML manifests we've created so far:

```
kubectl apply -f k8s-redis-deploy.yaml
kubectl apply -f k8s-redis-internal-service.yaml
kubectl apply -f k8s-pageview-deploy.yaml
```

You should see the following response:

```
k8suser@ubuntu:~$ kubectl apply -f k8s-redis-deploy.yaml
deployment.apps/k8s-redis created
k8suser@ubuntu:~$ kubectl apply -f k8s-redis-internal-service.yaml
service/db created
k8suser@ubuntu:~$ kubectl apply -f k8s-pageview-deploy.yaml
deployment.apps/k8s-pageview created
```

Figure 2.31: Creating the required Deployments and Service

5. Now, let's check whether the Deployments and the Service have been created successfully:

```
kubectl get deploy,service
```

You should see the following output:

```
NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/k8s-pageview    1/1     1            1           6s
deployment.apps/k8s-redis       1/1     1            1           6s

NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/db           ClusterIP   10.103.167.238   <none>        6379/TCP   6s
```

Figure 2.32: Getting the list of Deployments and Services

6. For the NodePort Service, create a file named **`k8s-pageview-external-service.yaml`** with the following content:

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

7. Let's apply the spec we created in the previous step:

```
kubectl apply -f k8s-pageview-external-service.yaml
```

You should see the following output:

```
service/k8s-pageview created
```

8. Let's check the URL that we will use to access our Service:

```
minikube service k8s-pageview
```

You should see a response similar to the following:

```
k8suser@ubuntu:~$ minikube service k8s-pageview
|-----------|--------------|-------------|-----------------------------|
| NAMESPACE |     NAME     | TARGET PORT |             URL             |
|-----------|--------------|-------------|-----------------------------|
| default   | k8s-pageview |             | http://192.168.99.100:31953 |
|-----------|--------------|-------------|-----------------------------|
```

Figure 2.33: Getting the URL and port to access the NodePort Service

9. Try accessing the external access endpoint of the Pageview app:

```
curl <service endpoint>:<node port>
```

Please use the URL from the previous step that you get in your case. You should see a response similar to the following:

```
k8suser@ubuntu:~$ curl http://192.168.99.100:31953
Hello, you're the visitor #1.
k8suser@ubuntu:~$ curl http://192.168.99.100:31953
Hello, you're the visitor #2.
k8suser@ubuntu:~$ curl http://192.168.99.100:31953
Hello, you're the visitor #3.
```

Figure 2.34: Accessing the application and confirming that the visitor count increases

Running the Pageview App in Multiple Replicas

Now that we have our Pageview app connected to a backend datastore, let's scale up the number of replicas of the frontend:

1. Now, make the following edits to **`k8s-pageview-deploy.yaml`** to increase the replicas to three:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: k8s-pageview
spec:
```

```
    # replicas: 1
    replicas: 3
    selector:
      matchLabels:
        tier: frontend
    template:
      metadata:
        labels:
          tier: frontend
      spec:
        containers:
        - name: k8s-pageview
          image: packtworkshops/the-kubernetes-workshop:pageview
```

As you can see, we have increased the number of **`replicas`** to **`3`**.

2. Apply the changes and bring the edited spec into effect:

```
kubectl apply -f k8s-pageview-deploy.yaml
```

You should see the following response:

```
deployment.apps/k8s-pageview configured
```

3. Check the number of pods that we have now:

```
kubectl get pod
```

You should see an output similar to the following:

```
NAME                           READY   STATUS    RESTARTS   AGE
k8s-pageview-74bb5d4dfd-2h7p6  1/1     Running   0          7s
k8s-pageview-74bb5d4dfd-lvtqk  1/1     Running   0          33m
k8s-pageview-74bb5d4dfd-x4rgn  1/1     Running   0          8s
k8s-redis-769fb5f9b7-92zpd     1/1     Running   0          33m
```

Figure 2.35: Getting the list of pods

We can see that we have three pageview pods running.

4. Let's confirm once again that our application is running:

```
curl <service endpoint>:<node port>
```

You should see a response similar to the following:

```
k8suser@ubuntu:~$ curl http://192.168.99.100:31953
Hello, you're the visitor #4.
k8suser@ubuntu:~$ curl http://192.168.99.100:31953
Hello, you're the visitor #5.
k8suser@ubuntu:~$ curl http://192.168.99.100:31953
Hello, you're the visitor #6.
```

Figure 2.36: Repeatedly accessing the application

5. Let's check the logs of each pod:

```
kubectl logs <pod name>
```

Run this command for each of the three pods running the frontend of our application. You should see a response similar to the following:

```
k8suser@ubuntu:~$ kubectl logs k8s-pageview-74bb5d4dfd-lvtqk
2019/12/14 20:42:13 Ping from 172.17.0.1:42504
2019/12/14 20:42:16 Ping from 172.17.0.1:42518
2019/12/14 20:42:17 Ping from 172.17.0.1:42524
k8suser@ubuntu:~$ kubectl logs k8s-pageview-74bb5d4dfd-2h7p6
2019/12/14 21:12:11 Ping from 172.17.0.1:49726
2019/12/14 21:12:12 Ping from 172.17.0.1:49738
k8suser@ubuntu:~$ kubectl logs k8s-pageview-74bb5d4dfd-x4rgn
2019/12/14 21:12:12 Ping from 172.17.0.1:49732
```

Figure 2.37: Getting the logs of the pod replicas running the application

The result may vary, but it will be somewhat similar to what you can see in the previous screenshot – some pods get more requests than others as the distribution is not perfectly even for a small sample size.

6. We will create a Bash script to simulate killing pods continuously. For this script, create a file named **`kill-pods.sh`** with the following content:

```
#!/usr/bin/env bash

# Sleep 1 second in each cycle.
SLEEP_INTERVAL=1

while true; do
    # Always keep one Pod running.
```

```
        running_pod_kept=
        # In each cycle, get the running Pods, and fetch their names.
        kubectl get pod -l tier=frontend --no-headers | grep Running
            | awk '{print $1}' | while read pod_name; do
            # Keep the 1st Pod running.
            if [[ -z $running_pod_kept ]]; then
                running_pod_kept=yes
                echo "Keeping Pod $pod_name running"
            # Delete all other running Pods.
            else
                echo "Killing Pod $pod_name"
                kubectl delete pod $pod_name
            fi
        done
        sleep $SLEEP_INTERVAL
    done
```

7. Similarly, we can also create a Bash script to simulate repeatedly accessing our application. For this, create a file named **`access-pageview.sh`** with the following content:

```
    #!/usr/bin/env bash
    # Sleep 1 second in each cycle.
    SLEEP_INTERVAL=1

    while true; do
        curl http://192.168.99.100:31953
        sleep $SLEEP_INTERVAL
    done
```

> **NOTE**
>
> The highlighted URL (in this example, we have **`http://192.168.99.100:31953`**) needs to be replaced with the **`<service endpoint>:<node port>`** that you get for your environment.

8. Now, open two terminal windows. In the first one, let's call the script to kill the pods:

```
./kill-pods.sh
```

You should see an output similar to the following:

```
k8suser@ubuntu:~$ ./kill-pods.sh
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 2.38: Killing pods via a script

In this output, we can see that the script kills a pod every 1 second.

9. In the other terminal window, call the script to repeatedly access our application:

```
./access-pageview.sh
```

You should see an output similar to the following:

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

Figure 2.39: Repeatedly accessing the application via the script

From the preceding output, we can tell that the availability of the Pageview app isn't impacted – we can still visit it without running into any errors, even though we have a script that's constantly killing pods.

Press *Ctrl* + *C* in both terminal windows to terminate the respective scripts.

CHAPTER 3: KUBECTL – THE KUBERNETES COMMAND CENTER

ACTIVITY 3.01: EDITING A LIVE DEPLOYMENT FOR A REAL-LIFE APPLICATION

Solution:

This section shows a sample solution regarding how to deploy a sample real-life application and make it accessible to the public. While this solution can be adapted for any platform (cloud or local) and any application, for reference, we are using the application that we provided with Minikube. Perform the following steps to complete this activity:

1. Use the following command to get the definition of the application:

```
curl https://raw.githubusercontent.com/PacktWorkshops/Kubernetes-
Workshop/master/Chapter03/Activity03.01/sample-application.yaml
--output sample-application.yaml
```

2. Use the following command to deploy this YAML manifest file to your current cluster:

```
kubectl apply -f sample-application.yaml
```

You can expect an output similar to the following:

```
deployment.apps/redis-back created
service/redis-back created
deployment.apps/melonvote-front created
service/melonvote-front created
```

This indicates that you have created the related Kubernetes objects successfully.

3. You can use the following command to get the statuses of the current pods and monitor their Deployment statuses:

```
kubectl get pod
```

You can expect an output similar to the following:

```
kubeserve-6995cffd5f-4q8r4          0/1     ContainerCreating   0          4m5s
kubeserve-6fcbcf845b-hgh6q          1/1     Running             0          16h
kubeserve-6fcbcf845b-klm7s          1/1     Running             0          16h
kubeserve-6fcbcf845b-sjjzg          1/1     Running             0          16h
melonvote-front-85c8b7cf8d-p8bns    0/1     ContainerCreating   0          5m13s
redis-back-559c848b4c-s94x9         0/1     ContainerCreating   0          5m13s
```

Figure 3.17: Using the kubectl get command to check pods

This indicates that the pods are running in the default namespace.

4. Use the following command to get all the services that have been created in the default namespace:

```
kubectl get svc
```

You can expect an output similar to the following:

```
NAME              TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes        ClusterIP      10.0.0.1       <none>        443/TCP        27d
melonvote-front   LoadBalancer   10.0.243.12    40.68.95.73   80:32651/TCP   90s
redis-back        ClusterIP      10.0.133.234   <none>        6379/TCP       90s
```

Figure 3.18: Using the kubectl get command to check the services

If this has been performed on any cloud providers that support load balancers, it would have provisioned an external IP address so that users could access the frontend application. However, when setting the **`LoadBalancer`** type on Minikube, it makes the service accessible through the **`minikube service`** command.

5. Use the following command to get the IP and port so that we can access our application:

```
minikube service melonvote-front
```

You should see the following output:

```
|-----------|-----------------|-------------|-----------------------------|
| NAMESPACE |      NAME       | TARGET PORT |             URL             |
|-----------|-----------------|-------------|-----------------------------|
| default   | melonvote-front |             | http://192.168.99.100:30228 |
|-----------|-----------------|-------------|-----------------------------|
```

Figure 3.19: Using the minikube service command to check the melonvote-front service

Note the URL shown in the preceding image. If you're using a cloud provider instead of Minikube for this activity, you can also use the following command:

```
kubectl get svc melonvote-front
```

You can expect an output similar to the following:

```
NAME              TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
melonvote-front   LoadBalancer   10.0.243.12   40.68.95.73   80:32651/TCP   10m
```

Figure 3.20: Using the kubectl get command to check the melonvote-front service

6. Get the list of Deployments:

```
kubectl get deploy
```

You can expect an output similar to the following:

```
NAME              READY    UP-TO-DATE    AVAILABLE    AGE
aci-helloworld    1/1      1             1            27d
kubeserve         1/1      1             1            6h27m
redis-back        1/1      1             1            6h27m
```

7. Now, let's edit the **`melonvote-front`** Deployment:

```
kubectl edit deploy melonvote-front
```

You can expect an output similar to the following:

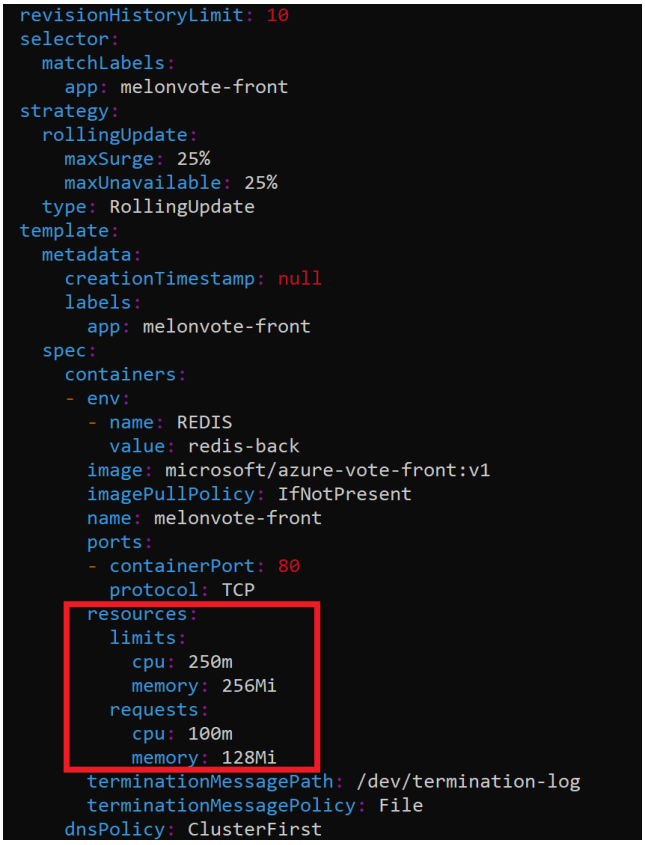

Figure 3.21: Using the kubectl edit command to check a specific Deployment

Now, you can edit the resource **`requests`** and **`limits`** section, as highlighted in the preceding image.

8. Update the resource **`limits`** and **`requests`** with the following values:

```
resources:
  limits:
    cpu: 500m
    memory: 512Mi
  requests:
    cpu: 200m
    memory: 256Mi
```

Figure 3.22: Editing the Deployment

9. Save and quit your text editor after making these changes. You should see an output similar to the following:

```
deployment.extensions/melonvote-front edited
```

10. Copy the URL shown in *Figure 3.19* and paste it into the address bar of your web browser. You'll be able to test your application by checking the voting app:

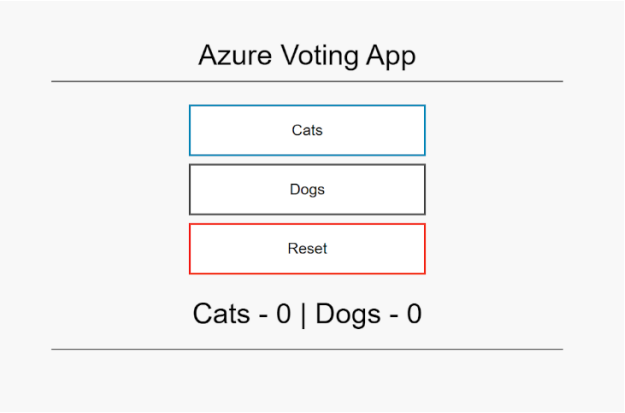

Figure 3.23: UI of the voting app

CHAPTER 4: HOW TO COMMUNICATE WITH KUBERNETES (API SERVER)

ACTIVITY 4.01: CREATING A DEPLOYMENT USING A SERVICEACCOUNT IDENTITY

Solution:

First, we will need to create the namespace, the service principle, and the RoleBinding. For that, we will need to use our current credentials that were created by Minikube as it has the cluster admin permissions:

1. Create the namespace:

```
kubectl create namespace activity-example
```

   You should get the following response:

```
namespace/activity-example created
```

2. Create a ServiceAccount in the **`activity-example`** namespace:

```
kubectl create serviceaccount activity-sa -n activity-example
```

   You should get the following response:

```
serviceacoount/activity-sa created
```

3. Create a RoleBinding for the ServiceAccount and assign it a cluster admin ClusterRole:

```
kubectl create rolebinding activity-sa-clusteradmin \
  --clusterrole=cluster-admin \
  --serviceaccount=activity-example:activity-sa \
  --namespace=activity-example
```

   You should get the following response:

```
rolebinding.rbac.authorization.k8s.io/activity-sa-clusteradm
in created
```

4. For our curl command to work, we will need the ServiceAccount token and the CA certificate. We can get the ServiceAccount token and the certificate from the Secret associated with the ServiceAccount:

```
kubectl get secrets -n activity-example
```

You should get the following response:

```
NAME                       TYPE                                  DATA   AGE
activity-sa-token-jzdkv    kubernetes.io/service-account-token   3      3m49s
default-token-7l266        kubernetes.io/service-account-token   3      3m56s
```

Figure 4.60: Getting a list of secrets

Notice how the name of the Secret starts with the same name as the ServiceAccount. In this case, it is **`activity-sa-token-jsmtr`**.

5. We can check the content of this secret by displaying the output in YAML form:

```
kubectl get secret activity-sa-token-jzdkv -n activity-example -o
yaml
```

In this command, use the name of the ServiceAccount obtained from the previous step. We are interested in the values of **`ca.cert`** and **`token`** in the following output:

```
apiVersion: v1
data:
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUV5akNDQXJLZ0F3SUJBZ0lSQU9uVlRjd0
  namespace: YWN0aXZpdHktZXhhbXBsZQ==
  token: ZXlKaGJHY2lPaUpTVXpJMU5pSXNJbXRwWkNJNklqaHVTa3BGUlVWTFJXaEJkV3RvU0VsVVIzSmx
kind: Secret
metadata:
  annotations:
    kubernetes.io/service-account.name: activity-sa
    kubernetes.io/service-account.uid: 865d3bb3-bf78-44b9-82a5-ee1839f19ee3
  creationTimestamp: "2020-05-17T17:29:46Z"
  name: activity-sa-token-jzdkv
  namespace: activity-example
  resourceVersion: "5186788"
  selfLink: /api/v1/namespaces/activity-example/secrets/activity-sa-token-jzdkv
  uid: 42477ea4-ee6a-4b93-ad97-98601474aa00
type: kubernetes.io/service-account-token
```

Figure 4.61: Checking the content of the secret

However, those values are Base64-encoded. In order to use it in our curl command, we will need to decode it. We can use the Base64 tool, which is installed by default on most Linux distributions and macOS.

6. To decode the values of **`ca.crt`** and get the token from the output of the previous step, we can use the following command:

```
echo "<value_of_ca.crt>" | base64 --decode > ~/ca.crt
```

   This will write the decoded value to a file named **`ca.crt`**.

7. Do the same for the value of **`token`**:

```
echo "<value_of_token>" | base64 --decode > ~/token
```

   This will write the decoded value to a file named **`token`**.

8. We can also keep **`token`** and **`ca.crt`** stored in a variable to use them in an easier way:

```
CACERT=~/ca.crt
TOKEN=$(cat ~/token)
```

9. So now, the only thing left is the API server endpoint, which can be taken from our kubeconfig file. We can also easily find it with **`kubectl config view`**:

```
kubectl config view -o jsonpath="{.clusters[?(@.name==\"minikube\")].
cluster.server}" ; echo
```

   In this command, we are using **`jsonpath`** to get just the API server endpoint, instead of the full kubeconfig file. So, you should get the API server address similar to the following:

```
https://10.210.254.169:8443
```

   Use the API server URL address that you get in your case.

10. Since we are creating a Deployment from YAML data, we can easily get it by using **`kubectl create`** with **`--dry-run=client`**:

```
kubectl create deployment activity-nginx --image=nginx
--dry-run=client -o yaml
```

You should get the following response:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: activity-nginx
  name: activity-nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: activity-nginx
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: activity-nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        resources: {}
status: {}
```

Figure 4.62: Getting the required YAML spec using dry-run

11. So, now that we have all the arguments ready, let's go ahead and create the Deployment using our **`curl`** command:

```
curl -X POST https://10.210.254.169:8443/apis/apps/v1/namespaces/
activity-
    example/deployments --cacert $CACERT -H "Authorization:
      Bearer $TOKEN" -H 'Content-Type: application/yaml' --data '
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
```

```
      run: activity-nginx
  name: activity-nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      run: activity-nginx
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: activity-nginx
    spec:
      containers:
      - image: nginx
        name: activity-nginx
        resources: {}
status: {}
'
```

You should get a response that begins something like the following:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    run: activity-nginx
  name: activity-nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      run: activity-nginx
```

Figure 4.63: Creating the activity-nginx Deployment

Note that this is a truncated version of the output screenshot. You will get a much larger version.

12. So, now that we have created our Deployment, we can move on to the next requirement to list the pods in our Deployment and confirm that they are in a running state from the status conditions:

```
curl -X GET https://10.210.254.169:8443/api/v1/namespaces/activity-example/pods --cacert $CACERT -H "Authorization: Bearer $TOKEN"
```

You should get the following result:


```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/namespaces/activity-example/pods",
    "resourceVersion": "53388"
  },
  "items": [
    {
      "metadata": {
        "name": "activity-nginx-84d75f9495-pgf64",
        "generateName": "activity-nginx-84d75f9495-",
        "namespace": "activity-example",
        "selfLink": "/api/v1/namespaces/activity-example/pod
s/activity-nginx-84d75f9495-pgf64",
        "uid": "d57dfbb7-a437-4366-8cdc-2dc24adef0d3",
        "resourceVersion": "53001",
        "creationTimestamp": "2019-12-03T21:13:58Z",
        "labels": {
          "pod-template-hash": "84d75f9495",
          "run": "activity-nginx"
        },
        "ownerReferences": [
          {
            "apiVersion": "apps/v1",
            "kind": "ReplicaSet",
```


Figure 4.64: A list of pods in JSON

13. Finally, we can remove our activity namespace with the following code:

```
curl -X DELETE https://10.210.254.169:8443/api/v1/namespaces/
activity-example --cacert $CACERT -H "Authorization: Bearer $TOKEN"
```

You should get the following response:


```
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "activity-example",
    "selfLink": "/api/v1/namespaces/activity-example",
    "uid": "4e255760-06b6-4023-8b26-4bd2ffa103ab",
    "resourceVersion": "53821",
    "creationTimestamp": "2019-12-03T19:51:57Z",
    "deletionTimestamp": "2019-12-03T21:25:12Z"
  },
  "spec": {
    "finalizers": [
      "kubernetes"
    ]
  },
  "status": {
    "phase": "Terminating"
  }
```


Figure 4.65: Deleting the activity-example namespace

You can see in the **`status`** section that the namespace is in the **`Terminating`** phase. So, it is being deleted.

CHAPTER 5: PODS

ACTIVITY 5.01: DEPLOYING AN APPLICATION IN A POD

Solution:

1. Run the following command to create a new **`pods-activity`** namespace:

```
kubectl create namespace pods-activity
```

   You should see the following response:

```
namespace/pods-activity created
```

2. Create **`activity-pod-configuration.yaml`** with the following pod configuration:

```
apiVersion: v1
kind: Pod
metadata:
  name: application-pod
  namespace: pods-activity
spec:
  restartPolicy: OnFailure
  containers:
    - name: custom-application-container
      image: packtworkshops/the-kubernetes-workshop:custom-
        application-for-pods-chapter
      readinessProbe:
        exec:
          command:
            - cat
            - /tmp/healthy
        initialDelaySeconds: 20
        periodSeconds: 10
```

3. Run the following command to create the pod using **`activity-pod-configuration.yaml`**:

```
kubectl create -f activity-pod-configuration.yaml
```

You should see the following response:

```
pod/application-pod created
```

4. Now, let's run the following commands to monitor how the pod is working:

```
kubectl --namespace pods-activity get pod application-pod
```

You should see the following response:

```
NAME                READY       STATUS      RESTARTS     AGE
application-pod     0/1         Running     0            15s
```

5. We can see that the pod has been created and is in the **`Running`** state. However, it's not **`READY`** yet. Let's wait for a few seconds and run the same command again:

```
kubectl --namespace pods-activity get pod application-pod
```

You should see the following response:

```
NAME                READY       STATUS      RESTARTS     AGE
application-pod     1/1         Running     0            37s
```

Now we can see that the pod is running.

CHAPTER 6: LABELS AND ANNOTATIONS

ACTIVITY 6.01: CREATING PODS WITH LABELS/ANNOTATIONS AND GROUPING THEM AS PER GIVEN CRITERIA

Solution:

1. Run the following command to create a new **`namespace`** called **`metadata activity`**:

```
$ kubectl create namespace metadata-activity
```

You should see the following response:

```
namespace/metadata-activity created
```

2. Next, we need to write pod configurations for all three pods as per the **`metadata`** requirements. We will handle this in three steps. First, create a file called **`arbitrary-product-application.yaml`** with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: arbitrary-product-application
  namespace: metadata-activity
  labels:
    environment: production
    team: product-development
    critical: "false"
  annotations:
      team-link: "https://jira-link/team-link-1"
spec:
  containers:
  - name: application-container
    image: nginx
```

As you can see in the highlighted section of the pod definition, we have created labels to mark the environment, team, and whether or not a pod is critical. The annotation is just a hypothetical link.

3. Create another file called **`infra-libraries-application.yaml`** with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: infra-libraries-application
  namespace: metadata-activity
  labels:
    environment: production
    team: infra-libraries
    critical: "true"
  annotations:
      team-link: "https://jira-link/team-link-2"
spec:
  containers:
  - name: application-container
    image: nginx
```

4. Create a third file called **`infra-libraries-application-staging.yaml`** with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: infra-libraries-application-staging
  namespace: metadata-activity
  labels:
    environment: staging
    team: infra-libraries
  annotations:
      team-link: "https://jira-link/team-link-2"
spec:
  containers:
  - name: application-container
    image: nginx
```

5. Now, we need to create all three pods. First, create the **`arbitrary-product-application`** pod using the following command:

```
$ kubectl create -f arbitrary-product-application.yaml
```

You should see the following response:

```
pod/arbitrary-product-application created
```

6. Now, create the **`infra-libraries-application.yaml`** pod:

```
$ kubectl create -f infra-libraries-application.yaml
```

You should see the following response:

```
pod/infra-libraries-application created
```

7. Finally, create the **`infra-libraries-application-staging.yaml`** pod:

```
$ kubectl create -f infra-libraries-application-staging.yaml
```

You should see the following response:

```
pod/infra-libraries-application-staging created
```

8. Run the following command to verify that all the pods have been created with the correct metadata:

```
$ kubectl --namespace metadata-activity describe pod arbitrary-
product-application
```

You should see the following response:

```
Name:         arbitrary-product-application
Namespace:    metadata-activity
Priority:     0
Node:         minikube/10.0.2.15
Start Time:   Fri, 18 Oct 2019 02:48:13 +0200
Labels:       critical=false
              environment=production
              team=product-development
Annotations:  team-link: https://jira-link/team-link-1
Status:       Running
IP:           172.17.0.12
IPs:
  IP:  172.17.0.12
Containers:
  application-container:
    Container ID:   docker://8da148615189f96c9bb126f6d524dde27aaba217baf89578f5be91bf38eb8dc3
    Image:          nginx
    Image ID:       docker-pullable://nginx@sha256:77ebc94e0cec30b20f9056bac1066b09fbdc049401b71850
922c63fc0cc1762e
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Fri, 18 Oct 2019 02:48:16 +0200
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-5tdkj (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-5tdkj:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-5tdkj
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age        From               Message
  ----    ------     ----       ----               -------
  Normal  Scheduled  <unknown>  default-scheduler  Successfully assigned metadata-activity/arbitrar
y-product-application to minikube
  Normal  Pulling    6m50s      kubelet, minikube  Pulling image "nginx"
  Normal  Pulled     6m48s      kubelet, minikube  Successfully pulled image "nginx"
  Normal  Created    6m48s      kubelet, minikube  Created container application-container
  Normal  Started    6m48s      kubelet, minikube  Started container application-container
```

Figure 6.20: Describing the arbitrary-product-application pod

In the highlighted section of the output, we can see that the metadata has been added as required. For the other two pods as well, you can similarly verify the metadata.

9. Run the following command to group all the pods that run in a production environment and are critical:

```
$ kubectl --namespace metadata-activity get pods -l
'environment=production,critical=true'
```

You should see the following list of pods:

```
NAME                          READY   STATUS        RESTARTS    AGE
infra-libraries-application   1/1     Running       0           12m
```

10. Run the following command to group all the pods that are not critical among all environments:

```
$ kubectl --namespace metadata-activity get pods -l 'critical!=true'
```

You should see the following list of pods:

```
NAME                                 READY   STATUS   RESTARTS AGE
arbitrary-product-application        1/1     Running  0        14m
infra-libraries-application-staging  1/1     Running  0        14m
```

CHAPTER 7: KUBERNETES CONTROLLERS

ACTIVITY 7.01: CREATING A DEPLOYMENT RUNNING AN APPLICATION

Solution:

1. Run the following command to create a new namespace called **`metadata-activity`**:

```
kubectl create namespace controllers-activity
```

   You should see the following response:

```
namespace/controllers-activity created
```

2. Create an **`activity-deployment.yaml`** file with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: activity-deployment
  namespace: controllers-activity
  labels:
    app: nginx
spec:
  replicas: 6
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: "50%"
      maxSurge: "50%"
  selector:
    matchLabels:
      chapter: controllers
      activity: first
  template:
    metadata:
      labels:
        chapter: controllers
        activity: first
```

```
      spec:
        containers:
        - name: nginx-container
          image: nginx
```

3. Create the Deployment using the **`kubectl apply`** command because we want to change the Deployment later:

```
kubectl apply -f activity-deployment.yaml
```

You should see the following response:

```
deployment.apps/activity-deployment created
```

4. Run the following command to verify that the Deployment has actually created six replicas of the Pod:

```
kubectl --namespace controllers-activity get pods
```

You should see output similar to the following:

```
NAME                                   READY   STATUS    RESTARTS   AGE
activity-deployment-54b9c6ff99-45shk   1/1     Running   0          40s
activity-deployment-54b9c6ff99-cl2hc   1/1     Running   0          40s
activity-deployment-54b9c6ff99-g6t7v   1/1     Running   0          40s
activity-deployment-54b9c6ff99-h2vb2   1/1     Running   0          40s
activity-deployment-54b9c6ff99-njnzc   1/1     Running   0          40s
activity-deployment-54b9c6ff99-z2fxg   1/1     Running   0          40s
```

Figure 7.15: Six Pods have been created

5. Modify the content of the **`activity-deployment.yaml`** file to change the **`replicas`** field to **`10`** and apply the new configuration using the following command:

```
kubectl apply -f activity-deployment.yaml
```

You should see the following response:

```
deployment.apps/activity-deployment configured
```

6. Check the Pods to make sure that the new Pods have been created as expected:

```
kubectl --namespace controllers-activity get pods
```

You should see output similar to the following:

```
NAME                                   READY   STATUS    RESTARTS   AGE
activity-deployment-54b9c6ff99-45shk   1/1     Running   0          4m39s
activity-deployment-54b9c6ff99-57kls   1/1     Running   0          18s
activity-deployment-54b9c6ff99-cl2hc   1/1     Running   0          4m39s
activity-deployment-54b9c6ff99-dswsb   1/1     Running   0          18s
activity-deployment-54b9c6ff99-g6t7v   1/1     Running   0          4m39s
activity-deployment-54b9c6ff99-h2vb2   1/1     Running   0          4m39s
activity-deployment-54b9c6ff99-njnzc   1/1     Running   0          4m39s
activity-deployment-54b9c6ff99-vl2md   1/1     Running   0          18s
activity-deployment-54b9c6ff99-z2fxg   1/1     Running   0          4m39s
activity-deployment-54b9c6ff99-zp5zj   1/1     Running   0          18s
```

Figure 7.16: Four new Pods have been created by the Deployment

As highlighted in the output shown in this screenshot, we can see that 4 of the 10 Pods were created later on when we scaled up the number of replicas from **6** to **10**.

7. For the next step, modify the content of the **`activity-deployment.yaml`** file to change the **`replicas`** field to **5** and apply the new configuration using the following command:

```
kubectl apply -f activity-deployment.yaml
```

You should see the following response:

```
deployment.apps/activity-deployment configured
```

8. Check the Pods to make sure that the new Pods have been created as expected:

```
kubectl --namespace controllers-activity get pods
```

You should see the following output:

```
NAME                                   READY   STATUS    RESTARTS   AGE
activity-deployment-54b9c6ff99-45shk   1/1     Running   0          9m14s
activity-deployment-54b9c6ff99-cl2hc   1/1     Running   0          9m14s
activity-deployment-54b9c6ff99-g6t7v   1/1     Running   0          9m14s
activity-deployment-54b9c6ff99-h2vb2   1/1     Running   0          9m14s
activity-deployment-54b9c6ff99-njnzc   1/1     Running   0          9m14s
```

Figure 7.17: Only five pods are running now

As we can see in this output, the Deployment has been successfully scaled down from 10 to 5 replicas.

CHAPTER 8: SERVICE DISCOVERY

ACTIVITY 8.01: CREATING A SERVICE TO EXPOSE THE APPLICATION RUNNING ON A POD

Solution:

Follow these steps to complete the solution for this activity:

1. Run the following command to create a new namespace called **`services-activity`**:

```
kubectl create namespace services-activity
```

   You should see the following response:

```
namespace/services-activity created
```

2. Create a file named **`backend-application.yaml`** with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-application
  namespace: services-activity
  labels:
    application: backend
spec:
  replicas: 4
  strategy:
    type: Recreate
  selector:
    matchLabels:
      application: backend
  template:
    metadata:
      labels:
        application: backend
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

3. Create the Deployment using the **`kubectl create`** command:

```
kubectl create -f backend-application.yaml
```

You should see the following response:

```
deployment.apps/backend-application created
```

4. Run the following command to verify that four Pods have been created for the backend application:

```
kubectl --namespace services-activity get pods
```

You should see the following response:

```
NAME                                  READY   STATUS    RESTARTS   AGE
backend-application-95f75bb84-5x9wf   1/1     Running   0          13s
backend-application-95f75bb84-9b528   1/1     Running   0          13s
backend-application-95f75bb84-wd4xb   1/1     Running   0          13s
backend-application-95f75bb84-xv5gh   1/1     Running   0          13s
```

Figure 8.19: Getting all the Pods

5. Create a file named **`backend-service.yaml`** with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
  namespace: services-activity
spec:
  type: ClusterIP
  ports:
    - targetPort: 80
      port: 80
  selector:
    application: backend
```

6. Run the following command to create the backend Service:

```
kubectl create -f backend-service.yaml
```

You should see the following response:

```
service/backend-service created
```

7. Run the following command to check the cluster IP of the Service:

```
kubectl --namespace services-activity get service backend-service
```

You should see the following response:

```
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
backend-service   ClusterIP   10.99.220.18    <none>        80/TCP    101s
```

Figure 8.20: Getting the Cluster IP from the Service

8. Let's make sure we can access the backend Service from inside the cluster:

```
minikube ssh
curl 10.99.220.18
```

You should get the following output:

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

$ curl 10.99.220.18
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Figure 8.21: Sending a curl request from inside the cluster

Now, we know that the backend application has been exposed at IP address **`10.99.220.18`** and port number **`80`** inside the cluster. We will use this information to add environment variables to the frontend application Pods.

9. Run the following command to exit the SSH session inside minikube:

```
exit
```

10. Create a file named **`frontend-application.yaml`** with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-application
  namespace: services-activity
  labels:
    application: frontend
spec:
  replicas: 5
  strategy:
    type: Recreate
  selector:
    matchLabels:
      application: frontend
  template:
    metadata:
      labels:
        application: frontend
    spec:
      containers:
      - name: apache-container
        image: httpd:2.4
        env:
        - name: BACKEND_SERVICE_IP
          value: "10.99.220.18"
        - name: BACKEND_SERVICE_PORT
          value: "80"
```

11. Run the following command to create the frontend application Deployment:

```
kubectl create -f frontend-application.yaml
```

    You should get the following response:

```
deployment.apps/frontend-application created
```

12. Run the following command to verify that five of the frontend application Pods have been created:

```
kubectl --namespace services-activity get pods
```

    You should get the following response:

```
NAME                                    READY   STATUS    RESTARTS   AGE
backend-application-95f75bb84-5x9wf     1/1     Running   0          13m
backend-application-95f75bb84-9b528     1/1     Running   0          13m
backend-application-95f75bb84-wd4xb     1/1     Running   0          13m
backend-application-95f75bb84-xv5gh     1/1     Running   0          13m
frontend-application-595c89d6c5-44sdh   1/1     Running   0          26s
frontend-application-595c89d6c5-67f5r   1/1     Running   0          26s
frontend-application-595c89d6c5-hdkjk   1/1     Running   0          26s
frontend-application-595c89d6c5-hn9hl   1/1     Running   0          26s
frontend-application-595c89d6c5-zm8ql   1/1     Running   0          26s
```

Figure 8.22: Getting all the Pods with the frontend application as well

    Here, we can see that along with the four existing backend Pods, five new frontend Pods have also been created.

13. Next, run the following command to verify that these Pods have the correct environment variables set:

```
kubectl --namespace services-activity describe pod frontend-
application-595c89d6c5-hn9hl
```

    Pick any of the frontend application's Pod names from the output of the previous step.

You should get the following response:

```
Name:           frontend-95cc76679-kpf9z
Namespace:      services-activity
Priority:       0
Node:           minikube/192.168.99.100
Start Time:     Sun, 08 Mar 2020 18:58:53 +0100
Labels:         application=frontend
                pod-template-hash=95cc76679
Annotations:    <none>
Status:         Running
IP:             172.17.0.11
Controlled By:  ReplicaSet/frontend-95cc76679
Containers:
  apache-container:
    Container ID:   docker://0bcba2900b0bf8dfb7555be219bf8d86c9589221572e1be9832682c17fe7d498
    Image:          httpd:2.4
    Image ID:       docker-pullable://httpd@sha256:946c54069130dbf136903fe658fe7d113bd8db8004de31282e20b262a3e1
06fb
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Sun, 08 Mar 2020 18:58:54 +0100
    Ready:          True
    Restart Count:  0
    Environment:
      BACKEND_SERVICE_IP:    10.99.220.18
      BACKEND_SERVICE_PORT:  80
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-xrtps (ro)
```

Figure 8.23: Verifying the environment variables

In the highlighted section of the preceding output, we can see that the two environment variables, **`BACKEND_SERVICE_IP`** and **`BACKEND_SERVICE_PORT`**, have the desired values.

14. Next, create a file named **`frontend-service.yaml`** with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
  namespace: services-activity
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      nodePort: 31000
```

```
    selector:
      application: frontend
```

15. Run the following command to create the frontend Service:

```
kubectl create -f frontend-service.yaml
```

    You should get the following response:

```
service/frontend-service created
```

16. Run the following command to check the status of the Service:

```
kubectl --namespace services-activity get service frontend-service
```

    You should get the following response:

```
NAME               TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
frontend-service   NodePort   10.100.9.135   <none>        80:31000/TCP   35s
```

Figure 8.24: Verifying that the NodePort Service has been created

17. Next, we need to verify that the frontend service is accessible from outside the Kubernetes cluster. For that, we need to know the IP address of the host. Run the following command to find out the IP address of the minikube host:

```
minikube ip
```

    You should get the following output:

```
192.168.99.100
```

18. Run the following command using **`curl`** to access the frontend application:

```
curl 192.168.99.100:31000
```

    You should see the following output:

```
<html><body><h1>It works!</h1></body></html>
```

19. We can also enter the same address in a browser's address bar to see the following output:

Figure 8.25: Verifying the frontend application in the browser

CHAPTER 9: STORING AND READING DATA ON DISK

ACTIVITY 9.01: CREATING A POD THAT USES A DYNAMICALLY PROVISIONED PERSISTENTVOLUME

Solution:

1. Run the following command to create a new namespace, **`volumes-activity`**:

```
kubectl create namespace volumes-activity
```

   You should see the following output:

```
namespace/volumes-activity created
```

   This output indicates that the namespace was successfully created.

2. Create a file called **`sc-activity-local.yaml`** with the following content:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: activity-local
provisioner: k8s.io/minikube-hostpath
```

   In this configuration for **`StorageClass`**, we have used the local **`k8s.io/minikube-hostpath`** provisioner that comes with minikube.

3. Run the following command to create the storage class using the preceding configuration:

```
kubectl create -f sc-activity-local.yaml
```

   Note that we haven't used a **`--namespace`** flag when creating the storage class. That's because the storage classes are cluster-scoped. So, this storage class will be visible to all namespaces.

   You should see the following output:

```
storageclass.storage.k8s.io/activity-local created
```

4. Run the following command to check the status of the newly created storage class:

```
kubectl get sc activity-local
```

Note that **sc** is an accepted shortened name for **StorageClass**. You should see the following output:

```
NAME              PROVISIONER                 AGE
activity-local    k8s.io/minikube-hostpath    11s
```

5. Next, let's create a file called **pvc-activity-local-claim.yaml** with the following content:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: activity-local-claim
  namespace: volumes-activity
spec:
  storageClassName: activity-local
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

In this configuration, we have specified the storage class name to be **activity-local**, which is the storage class we created in the last step.

6. Run the following command to create a PVC using the preceding configuration:

```
kubectl create -f pvc-activity-local-claim.yaml
```

You should see the following output:

```
persistentvoumeclaim/activity-local-claim created
```

7. Run the following command to check the status of the claim we just created:

```
kubectl --namespace volumes-activity get pvc activity-local-claim
```

You should see the following output:

```
NAME                   STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS
 AGE
activity-local-claim   Bound    pvc-219ecbf6-3fb4-4b72-b1aa-80d94f872598   100Mi      RWO            activity-local
 21s
```

Figure 9.14: Checking the status of the PVC

We can see that the PVC is bound to a Volume that was dynamically created.

8. Run the following command to check the list of PVs in this namespace:

```
kubectl --namespace volumes-activity get pv
```

You should see the following output:

```
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
                STORAGECLASS     REASON   AGE
pvc-219ecbf6-3fb4-4b72-b1aa-80d94f872598   100Mi      RWO            Delete           Bound    volumes-activity/activ
ity-local-claim   activity-local            34s
```

Figure 9.15: Checking the list of PVs

We can see that we have a PV that has an **`activity-local`** storage class and is bound to the **`activity-local-claim`** claim.

9. Next, create a file called **`activity-pod.yaml`** with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: activity-pod
  namespace: volumes-activity
spec:
  containers:
  - image: ubuntu
    name: ubuntu-1
    command: ['/bin/bash', '-ec', 'echo "Data written by container-1"
> /data/application/data.txt; sleep 3600']
    volumeMounts:
    - mountPath: /data/application
      name: pvc-volume
  - image: ubuntu
    name: ubuntu-2
    command: ['/bin/bash', '-ec', 'sleep 30; cat /data/application/
data.txt; sleep 3600']
    volumeMounts:
    - mountPath: /data/application
      name: pvc-volume
  volumes:
  - name: pvc-volume
    persistentVolumeClaim:
      claimName: activity-local-claim
```

As we can see, we have used the claim named **activity-local-claim** as the volume and mounted it on both the containers at **/data/application**. The first container writes some data to a file in the PV and the second container waits for 30 seconds and then reads out the content of the file written by the former container. Both of the containers then sleep for 1 hour. This is an optional configuration. We can also skip the sleeping step at the end.

10. Run the following command to create the pod using the preceding configuration:

```
kubectl create -f activity-pod.yaml
```

You should see the following output:

```
pod/activity-pod created
```

11. Run the following command to get the status of the pod we just created to confirm that it is still running:

```
kubectl --namespace volumes-activity get pod activity-pod
```

You should see the following output:

```
NAME           READY    STATUS     RESTARTS   AGE
activity-pod   2/2      Running    0          12s
```

12. Next, let's wait for at least 30 seconds and then check the logs of the **ubuntu-2** container on the pod:

```
kubectl --namespace volumes-activity logs activity-pod -c ubuntu-2
```

You should see the following output:

```
Data written by container-1
```

This output verifies that the **ubuntu-2** container can read the data written by the **ubuntu-1** container in the PV directory.

CHAPTER 10: CONFIGMAPS AND SECRETS

ACTIVITY 10.01: USING A CONFIGMAP AND SECRET TO PROMOTE AN APPLICATION THROUGH DIFFERENT STAGES

Solution:

First, we will create objects in the **`my-app-test`** namespace, as shown in the following steps:

1. First, create the test namespace:

```
kubectl create namespace my-app-test
```

   You should get a response like this:

```
namespace/my-app-test created
```

2. Create a folder for the test namespace:

```
mkdir -p activity-configmap-secret/test
cd activity-configmap-secret/test
```

3. Create the **`application-data.properties`** file. You can use any text editor to create this file:

```
external-system-location=https://testvendor.example.com
external-system-basic-auth-username=user123
```

4. Create the ConfigMap in the test namespace using this command:

```
kubectl create configmap my-app-data --from-file=./application-data.
properties --namespace my-app-test
```

   You should see the following output:

```
configmap/my-app-data created
```

5. Create a file with the password, as shown here:

```
cat > application-secure.properties << EOF
external-system-basic-auth-password=password123
EOF
```

6. Create a Secret in the test namespace using the following command:

```
kubectl create secret generic my-app-secret --from-file=application-
secure.properties --namespace my-app-test
```

You should see a response like this:

```
secret/my-app-secret created
```

7. Create a YAML file named **`activity-solution-pod-spec.yaml`** with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-secrets-activity-pod
spec:
  containers:
    - name: configmap-secrets-activity-container
      image: k8s.gcr.io/busybox
      command: [ «/bin/sh", "-c", "cat /etc/app-
        data/application-data.properties; cat /etc/secure-
          data/application-secure.properties;" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/app-data
      - name: secret-volume
        mountPath: /etc/secure-data
  volumes:
    - name: config-volume
      configMap:
        name: my-app-data
    - name: secret-volume
      secret:
        secretName: my-app-secret
```

8. Run the Pod using the YAML specification in the previous step. It should load both the ConfigMap and Secret onto the Pod:

```
kubectl create -f activity-solution-pod-spec.yaml --namespace my-app-
test
```

When the Pod is successfully created, you should see a response like this:

```
pod/configmap-secrets-activity-pod created
```

9. View the logs to see whether the required data is loaded onto the Pod:

```
kubectl logs -f configmap-secrets-activity-pod --namespace my-app-test
```

The logs should show the username and password, as shown in this output:

```
external-system-location=https://testvendor.example.com
external-system-basic-auth-username=user123
external-system-basic-auth-password=password123
```

Figure 10.20: Getting logs for the test environment

10. Next, we will create objects in the **`my-app-prod`** namespace, as mentioned in the following steps. First, create the test namespace:

```
kubectl create namespace my-app-prod
```

You should see an output like this:

```
namespace/my-app-prod created
```

11. Create a folder for the **`prod`** namespace using the following commands:

```
mkdir -p activity-configmap-secret/prod
cd activity-configmap-secret/prod
```

12. Create the **`application-data.properties`** file with the required data:

```
cat > application-data.properties << EOF
external-system-location=https://vendor.example.com
external-system-basic-auth-username=activityapplicationuser
EOF
```

13. Create a ConfigMap in the **`my-app-prod`** namespace, as follows:

```
kubectl create configmap my-app-data --from-file=./application-data.
properties --namespace my-app-prod
```

You should expect the following response:

```
configmap/my-app-data created
```

14. Create the file that contains the password using the following command:

```
cat > application-secure.properties << EOF
external-system-basic-auth-password=A#4b*(1=B88%tFr3
EOF
```

15. Create a Secret in the **`my-app-prod`** namespace:

```
kubectl create secret generic my-app-secret --from-file=application-
secure.properties --namespace my-app-prod
```

    You should see a response like this:

```
secret/my-app-secret created
```

16. Create and run the Pod using the following command:

```
kubectl create -f activity-solution-pod-spec.yaml --namespace my-app-
prod
```

    You should see an output like this:

```
pod/configmap-secrets-activity-pod created
```

17. View the logs to check whether the required information is loaded onto our Pod:

```
kubectl logs -f configmap-secrets-activity-pod --namespace my-app-prod
```

    You should be able to see the production configuration for the Pod, as shown here:

```
external-system-location=https://vendor.example.com
external-system-basic-auth-username=activityapplicationuser
external-system-basic-auth-password=A#4b*(1=B88%tFr3
```

Figure 10.21: Getting logs for the production environment

CHAPTER 11: BUILD YOUR OWN HA CLUSTER

ACTIVITY 11.01: TESTING THE RESILIENCE OF A HIGHLY AVAILABLE CLUSTER

Solution:

First, please ensure that you have the cluster infrastructure set up as shown in *Exercise 11.01*, *Setting Up Our Kubernetes Cluster*. Then, follow these steps:

1. First, let's scale the deployment to 10 replicas:

```
kubectl scale deployment -n kubernetes-dashboard kubernetes-dashboard
--replicas 10
```

   You should see the following response:

```
deployment.extensions/kubernetes-dashboard scaled
```

2. Now it is bringing up more Pods. We can confirm this by using the following command:

```
kubectl get pod -n kubernetes-dashboard
```

   You should see this response:

```
NAME                                         READY   STATUS    RESTARTS   AGE
kubernetes-dashboard-5c8f9556c4-b6sbs        1/1     Running   0          76s
kubernetes-dashboard-5c8f9556c4-kk67w        1/1     Running   0          76s
kubernetes-dashboard-5c8f9556c4-ljscz        1/1     Running   0          11m
kubernetes-dashboard-5c8f9556c4-mwz2g        1/1     Running   0          77s
kubernetes-dashboard-5c8f9556c4-ntchz        1/1     Running   0          76s
kubernetes-dashboard-5c8f9556c4-p6fth        1/1     Running   0          77s
kubernetes-dashboard-5c8f9556c4-p75ms        1/1     Running   0          76s
kubernetes-dashboard-5c8f9556c4-rbndl        1/1     Running   0          77s
kubernetes-dashboard-5c8f9556c4-sqkvk        1/1     Running   0          76s
kubernetes-dashboard-5c8f9556c4-w6skn        1/1     Running   0          76s
kubernetes-metrics-scraper-86456cdd8f-mx7qw  1/1     Running   0          11m
```

Figure 11.16: Verifying that the number of Pods has scaled

3. Okay, now for the fun part. Let's start deleting Pods randomly. We can do that using a script. Create a file named **`delete_pods.sh`** with the following content:

```
#!/bin/bash
while [ true ]; do

NUM=$[ ( $RANDOM % 10 ) ]
kubectl get pod -n kubernetes-dashboard | awk -v num=$NUM
  '(NR==num && NR > 1) {print $1}' | xargs kubectl delete pod -n
```

```
        kubernetes-dashboard
    sleep "$NUM"s

    done
```

This script will pick a random number between 1 and 10 and delete that Pod.

4. Now, let's run our script:

```
bash delete_pods.sh
```

You should see a response similar to this:

```
pod "kubernetes-dashboard-5c8f9556c4-rbndl" deleted
pod "kubernetes-dashboard-5c8f9556c4-ntchz" deleted
```

5. Try accessing the application at the following URL:

```
http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/
services/https:kubernetes-dashboard:/proxy/
```

We should still be able to get a response from our application:

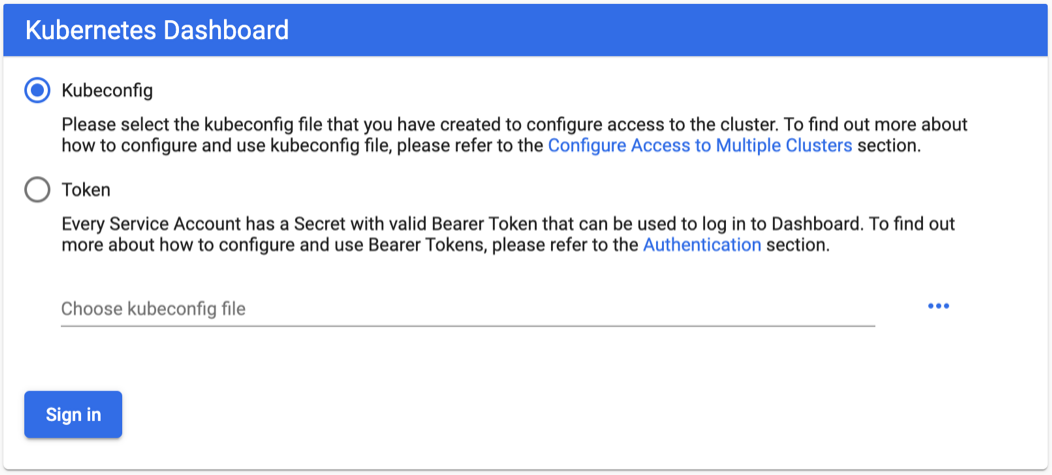

Figure 11.17: Kubernetes Dashboard prompt for entering a token

6. It looks like our application is still online. What happens though if we accidentally (or on purpose) delete a node? Go to the AWS console and navigate to the EC2 section:

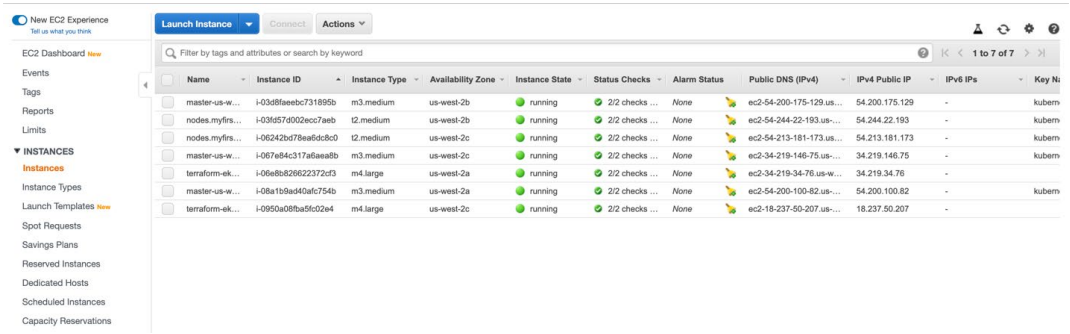

Figure 11.18: List of EC2 instances on the AWS dashboard

7. All of the nodes that are part of our cluster have **`myfirstcluster`** as part of their name. Let's pick one of these nodes and terminate it using the **`Actions`** menu:

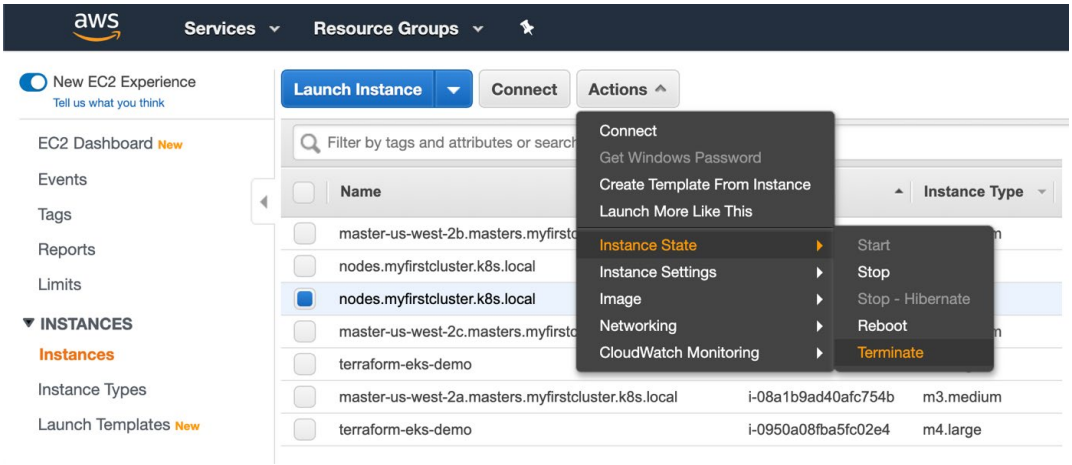

Figure 11.19: Terminating a node

8. Click **`Yes, Terminate`** to acknowledge your proposed mistake:

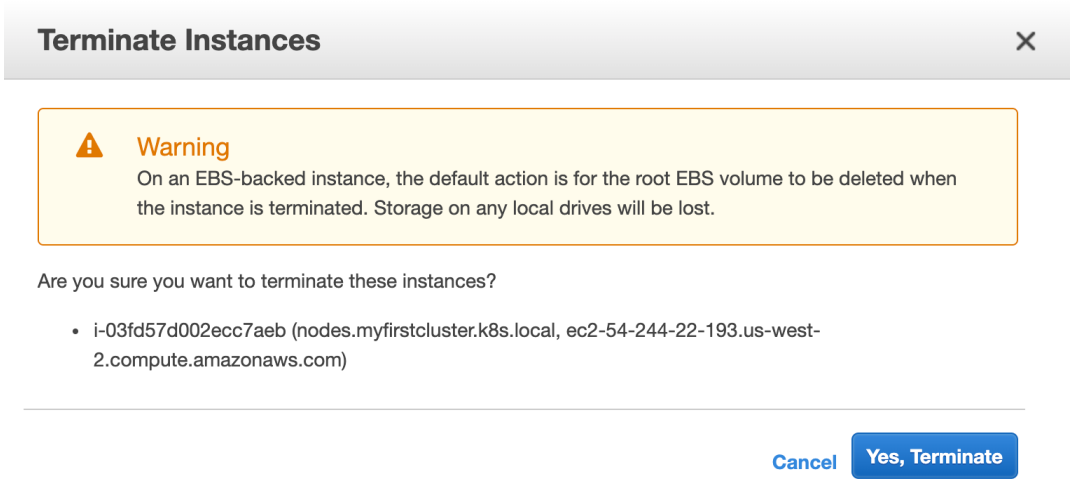

Figure 11.20: Confirming the termination of a node

9. Let's go check to see whether the application is online by checking the list of Pods:

```
kubectl get pod -n kubernetes-dashboard
```

You should see the following response:

```
NAME                                    READY   STATUS              RESTARTS   AGE
kubernetes-dashboard-5c8f9556c4-dfrmf   1/1     Running             0          10s
kubernetes-dashboard-5c8f9556c4-gmmjk   1/1     Running             0          9s
kubernetes-dashboard-5c8f9556c4-jhn2p   1/1     Running             0          10s
kubernetes-dashboard-5c8f9556c4-jqrsr   0/1     ContainerCreating   0          9s
kubernetes-dashboard-5c8f9556c4-kk67w   1/1     Running             0          37m
kubernetes-dashboard-5c8f9556c4-mwz2g   1/1     Running             0          37m
kubernetes-dashboard-5c8f9556c4-ncl2m   1/1     Running             0          10s
kubernetes-dashboard-5c8f9556c4-p6fth   1/1     Running             0          37m
kubernetes-dashboard-5c8f9556c4-p75ms   1/1     Running             0          37m
kubernetes-dashboard-5c8f9556c4-w6skn   1/1     Running             0          37m
kubernetes-metrics-scraper-86456cdd8f-xpz9s   1/1     Running             0          10s
```

Figure 11.21: Kubernetes Dashboard prompt for entering a token

As you can see in the preceding screenshot, the Pods are already being restarted on nodes that are still online. In addition, the AWS Autoscaling group that kops created for us will automatically spin up a new node to replace the one we terminated.

10. Now, let's add an additional node. To do that, we need to edit InstanceGroups. First, get a list of InstanceGroups:

```
kops get ig
```

You should see a response similar to this:

```
Using cluster from kubectl context: myfirstcluster.k8s.local

NAME                    ROLE    MACHINETYPE     MIN     MAX     ZONES
master-us-west-2a       Master  m3.medium       1       1       us-west-2a
master-us-west-2b       Master  m3.medium       1       1       us-west-2b
master-us-west-2c       Master  m3.medium       1       1       us-west-2c
nodes                   Node    t2.medium       2       2       us-west-2a,us-west-2b,us-west-2c
```

Figure 11.22: List of instance groups

11. We want to edit the **`nodes`** instance group. Run the following command:

```
kops edit ig nodes
```

This will open up the spec of the instance group in a text editor, as shown in the following screenshot:

```
apiVersion: kops.k8s.io/v1alpha2
kind: InstanceGroup
metadata:
  creationTimestamp: "2020-02-11T03:50:57Z"
  labels:
    kops.k8s.io/cluster: myfirstcluster.k8s.local
  name: nodes
spec:
  image: kope.io/k8s-1.15-debian-stretch-amd64-hvm-ebs-2020-01-17
  machineType: t2.medium
  maxSize: 3
  minSize: 3
  nodeLabels:
    kops.k8s.io/instancegroup: nodes
  role: Node
  subnets:
  - us-west-2a
  - us-west-2b
  - us-west-2c
```

Figure 11.23: List of instance groups

In this spec, change the value of **`maxSize`** and **`minSize`** to **`3`** and then save it.

12. To apply this change, run the following command:

```
kops update cluster --yes
```

You should see this response:

```
Using cluster from kubectl context: myfirstcluster.k8s.local

I0210 21:03:03.299035   12701 apply_cluster.go:556] Gossip DNS: skipping DNS validation
I0210 21:03:05.097410   12701 executor.go:103] Tasks: 0 done / 105 total; 47 can run
I0210 21:03:07.694834   12701 executor.go:103] Tasks: 47 done / 105 total; 26 can run
I0210 21:03:10.388278   12701 executor.go:103] Tasks: 73 done / 105 total; 24 can run
I0210 21:03:11.847549   12701 executor.go:103] Tasks: 97 done / 105 total; 5 can run
I0210 21:03:12.471843   12701 executor.go:103] Tasks: 102 done / 105 total; 3 can run
I0210 21:03:12.799184   12701 executor.go:103] Tasks: 105 done / 105 total; 0 can run
I0210 21:03:13.193759   12701 update_cluster.go:294] Exporting kubecfg for cluster
kops has set your kubectl context to myfirstcluster.k8s.local

Cluster changes have been applied to the cloud.


Changes may require instances to restart: kops rolling-update cluster
```

Figure 11.24: Applying updates to our cluster

13. In the previous screenshot, you can see that kops suggests that we restart the cluster, which is done in the form of a rolling update. This is not an absolute necessity for our scenario. You will learn more about rolling updates in *Chapter 18*, *Upgrading Your Cluster without Downtime*. For now, you can simply run the **`rolling-update`** command:

```
kops rolling-update cluster --yes
```

You should see output similar to this:

```
Using cluster from kubectl context: myfirstcluster.k8s.local

NAME                    STATUS  NEEDUPDATE      READY   MIN     MAX     NODES
master-us-west-2a       Ready   0               1       1       1       1
master-us-west-2b       Ready   0               1       1       1       1
master-us-west-2c       Ready   0               1       1       1       1
nodes                   Ready   0               3       3       3       3

No rolling-update required.
```

Figure 11.25: Applying rolling-update to our cluster

From the output, we can see that we have three nodes up and running in our cluster.

CHAPTER 12: YOUR APPLICATION AND HA

ACTIVITY 12.01: EXPANDING THE STATE MANAGEMENT OF OUR APPLICATION

Solution:

Before starting these steps, make sure you have run through *Exercise 12.04*, *Deploying an Application with State Management* and still have all the resources from that exercise. We need to modify a few files in order to get this solution to work:

1. We need to add the following to the **`main.tf`** Terraform file that we used to spin up the cluster:

```
resource "aws_security_group" "redis_sg" {
  name = "redis-sg"
  vpc_id = aws_vpc.demo.id
  ingress {
    from_port = 6379
    protocol = "TCP"
    to_port = 6379
    security_groups = [aws_security_group.demo-node.id]
    }
  }

resource "aws_elasticache_subnet_group" "redis_subnet" {
  name = "redis-cluster-subnet-group"
  subnet_ids = aws_subnet.demo.*.id

  }

resource "aws_elasticache_cluster" "redis_cluster" {
  cluster_id = "redis-cluster"
  engine = "redis"
  node_type = "cache.m4.large"
  num_cache_nodes = 1
  parameter_group_name = "default.redis3.2"
  engine_version = "3.2.10"
  port = 6379
  security_group_ids = [aws_security_group.redis_sg.id]
  subnet_group_name = aws_elasticache_subnet_group.redis_subnet.name
  }
```

```
output "redis_address" {
  value = "${aws_elasticache_cluster.redis_cluster.cache_nodes[0].
address}:6379"
  }
```

2. Let's apply and approve the changes:

```
terraform apply
```

Enter the correct value of the region if prompted again. You should see the following output:

```
provider.aws.region
  The region where AWS operations will take place. Examples
  are us-east-1, us-west-2, etc.

  Enter a value: us-west-2

data.aws_region.current: Refreshing state...
data.aws_availability_zones.available: Refreshing state...
aws_iam_role.demo-cluster: Refreshing state... [id=terraform-eks-demo-cluster]
aws_iam_role.demo-node: Refreshing state... [id=terraform-eks-demo-node]
aws_vpc.demo: Refreshing state... [id=vpc-007e74ade3a6dbac3]
aws_iam_role_policy_attachment.demo-cluster-AmazonEKSClusterPolicy: Refreshing s
tate... [id=terraform-eks-demo-cluster-20200125195509671500000005]
aws_iam_role_policy_attachment.demo-cluster-AmazonEKSServicePolicy: Refreshing s
tate... [id=terraform-eks-demo-cluster-20200125195509660600000002]
aws_iam_instance_profile.demo-node: Refreshing state... [id=terraform-eks-demo]
aws_iam_role_policy_attachment.demo-node-AmazonEC2ContainerRegistryReadOnly: Ref
reshing state... [id=terraform-eks-demo-node-20200125195509661900000004]
aws_iam_role_policy_attachment.demo-node-AmazonEKS_CNI_Policy: Refreshing state.
.. [id=terraform-eks-demo-node-20200125195509660700000003]
aws_iam_role_policy_attachment.demo-node-AmazonEKSWorkerNodePolicy: Refreshing s
tate... [id=terraform-eks-demo-node-20200125195509660600000001]
```

Figure 12.21: Recreating our cluster with ElastiCache for state management

This should add an ElastiCache Redis cluster for us to use. The previous command produces a long output. In that output, find the **`redis_address`** field near the end. It should look like this:

```
redis_address = redis-cluster.jbnten.0001.usw2.cache.amazonaws.com:6379
```

Figure 12.22: redis_address from the Terraform output screen

Then, we need to modify our application to connect to this instance instead of our in-cluster Redis instance.

3. In the **`containers`** section of the Deployment spec of the manifest from *Exercise 12.04*, *Deploying an Application with State Management*, add an environment variable named **`REDIS_SVC_ADDR`** with the value that you obtained from the previous step:

```
            env:
              - name: REDIS_SVC_ADDR
                value: "use-your-value"
```

4. Apply the changes:

```
kubectl apply -f with_redis_activity.yaml
```

You should see the following response:

```
deployment.apps/kubernetes-test-ha-application-with-redis-deployment created
service/kubernetes-test-ha-application-with-redis created
```

Figure 12.23: Recreating the objects for our application

5. Now, let's test the change:

```
kubectl port-forward svc/kubernetes-test-ha-application-with-redis
8080:80
```

You should see the following response:

```
Forwarding from 127.0.0.1:8080 -> 8080
```

6. Now, in another terminal session, let's try accessing our application multiple times:

```
curl localhost:8080/get-number
```

You should see the number increasing, as follows:

```
Last login: Sat Jan 25 20:14:49 on ttys000
curl %
zarnold@Zachs-MacBook-Pro activity_solution % curl localhost:8080/get-number
zarnold@Zachs-MacBook-Pro activity_solution % curl localhost:8080/get-number
{number: 1}%
zarnold@Zachs-MacBook-Pro activity_solution % curl localhost:8080/get-number
{number: 2}%
zarnold@Zachs-MacBook-Pro activity_solution % curl localhost:8080/get-number
{number: 3}%
zarnold@Zachs-MacBook-Pro activity_solution % curl localhost:8080/get-number
{number: 4}%
zarnold@Zachs-MacBook-Pro activity_solution % curl localhost:8080/get-number
{number: 5}%
zarnold@Zachs-MacBook-Pro activity_solution %
```

Figure 12.24: Getting a predictable output from our application with an ElastiCache backend

CHAPTER 13: RUNTIME AND NETWORK SECURITY IN KUBERNETES

ACTIVITY 13.01: SECURING OUR APP

Solution:

In this activity, we will modify the RBAC policies from *Exercise 13.01*, *Creating a Kubernetes RBAC ClusterRole*, to allow required traffic:

1. Ensure that you have a cluster infrastructure and all the objects from *Exercise 13.01*, *Creating a Kubernetes RBAC ClusterRole*.
2. Create a file named **`pod_security_policy.yaml`** with the following content:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: psp-example
  namespace: default
spec:
  privileged: false
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
    - '*'
```

3. Now, apply this policy using the following command:

```
kubectl apply -f pod_security_policy.yaml
```

You should see the following response:

```
podsecuritypolicy.policy/psp-example created
```

4. Create a file named **`network_policy.yaml`** with the following content:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector: {}
  ingress:
    - {}
  egress:
    - {}
  policyTypes:
    - Ingress
    - Egress
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy-redis
  namespace: default
spec:
  podSelector: {}
  ingress:
    - {}
  egress:
    - {}
  policyTypes:
    - Ingress
    - Egress
```

> **NOTE**
>
> There is a workaround in this file since kubectl proxy opens a different port on your machine to communicate with the cluster. Since this port is unknown ahead of time, it is set to **`all`**.

5. Now, apply this policy using the following command:

```
kubectl apply -f network_policy.yaml
```

You should see the following response:

```
networkpolicy.networking.k8s.io/test-network-policy created
```

6. If you have the application from *Exercise 13.02*, *Creating a NetworkPolicy* still deployed in your cluster, you can move on to the next step. Otherwise, rerun *steps 5* and *6* from that exercise.

7. Now, let's test our application:

```
curl localhost:8001/api/v1/namespaces/default/services/kubernetes-
test-ha-application-with-redis:/proxy/get-number
```

You should see the following response:

```
zarnold@zachs-mbp kubernetes % curl -H 'Host: counter.com' a3960d10c980e40f99887]
ea068f41b7b-1447612395.us-east-1.elb.amazonaws.com/get-number
zarnold@zachs-mbp kubernetes % curl -H 'Host: counter.com' a3960d10c980e40f99887]
ea068f41b7b-1447612395.us-east-1.elb.amazonaws.com/get-number
{number: 1}%                                                                    ]
zarnold@zachs-mbp kubernetes % curl -H 'Host: counter.com' a3960d10c980e40f99887
ea068f41b7b-1447612395.us-east-1.elb.amazonaws.com/get-number
{number: 2}%                                                                    ]
zarnold@zachs-mbp kubernetes % curl -H 'Host: counter.com' a3960d10c980e40f99887
ea068f41b7b-1447612395.us-east-1.elb.amazonaws.com/get-number
{number: 3}%                                                                    ]
zarnold@zachs-mbp kubernetes % curl -H 'Host: counter.com' a3960d10c980e40f99887
ea068f41b7b-1447612395.us-east-1.elb.amazonaws.com/get-number
{number: 4}%                                                                    ]
zarnold@zachs-mbp kubernetes % curl -H 'Host: counter.com' a3960d10c980e40f99887
ea068f41b7b-1447612395.us-east-1.elb.amazonaws.com/get-number
{number: 5}%
```

Figure 13.15: Testing our application

CHAPTER 14: RUNNING STATEFUL COMPONENTS IN KUBERNETES

ACTIVITY 14.01: CHART-IFYING OUR STATEFULSET DEPLOYMENT

Solution:

Before you begin with these steps, please ensure that your cluster infrastructure is set up as instructed in *step 1* of *Exercise 14.01*, *Deploying a Counter App with a MySQL Backend*.

1. Let's start by creating the Helm chart:

```
helm create counter-mysql
```

You should see this response:

```
Creating counter-mysql
```

2. Next, change directory into it:

```
cd counter-mysql
```

Now let's clean up the working directory a bit here:

```
rm ./templates/*.yaml && rm ./templates/NOTES.txt && rm -r ./
templates/tests/
```

3. Now we need to download the **`with_mysql.yaml`** file from the GitHub repository for this chapter:

```
curl https://raw.githubusercontent.com/PacktWorkshops/Kubernetes-
Workshop/master/Chapter14/Exercise14.01/with_mysql.yaml > ./
templates/with_mysql.yaml
```

You should see the following response:

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2337  100  2337    0     0   8200      0 --:--:-- --:--:-- --:--:--  8200
```

Figure 14.15: Downloading the combined manifest for our application

4. Now we need to split this into different files, so separate them using **`---`** into different files. You can name the files whatever you want; Helm will glob them together. Note that you should remove **`with_mysql.yaml`** after you are done copying out of it:

```
tree .
```

The following file structure gives an example of how you can split them:

```
.
├── Chart.yaml
├── charts
├── templates
│   ├── _helpers.tpl
│   ├── app_deployment.yaml
│   ├── app_service.yaml
│   ├── mysql_secret.yaml
│   ├── mysql_service.yaml
│   └── mysql_statefulset.yaml
└── values.yaml

2 directories, 8 files
```

Figure 14.16: Directory structure for our Helm chart

5. Now we need to templatize our MySQL version deployment. Clear out all the contents of **`values.yaml`** and replace it with the following:

```
mysql:
  version: 5.7
```

6. Now, we should change the manifest for the MySQL StatefulSet. Let's change the **`mysql_statefulset.yaml`** file to take this value as an argument to the **`image`** field in the manifest as shown here:

```
spec:
  containers:
    - name: mysql
      image: mysql:{{ .Values.mysql.version }}
```

7. Now let's install this chart by running the following command:

```
helm install --generate-name -f values.yaml .
```

You should see a response similar to the following:

```
NAME: chart-1591835373
LAST DEPLOYED: Wed Jun 10 20:29:35 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Figure 14.17: Installing our Helm chart with a generated name

8. Let's set up port forwarding so that we can access our application:

```
kubectl port-forward service/kubernetes-test-ha-application-with-
mysql 8080:80
```

You should see this response:

```
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

9. Now, open another terminal window to access our application. Run the following command a few times:

```
curl localhost:8080/get-number
```

You should see this response:

```
{number: 1}
[zarnold@zachs-mbp counter-mysql % curl localhost:8080/get-number
{number: 2}
[zarnold@zachs-mbp counter-mysql % curl localhost:8080/get-number
{number: 3}
[zarnold@zachs-mbp counter-mysql % curl localhost:8080/get-number
{number: 4}
[zarnold@zachs-mbp counter-mysql % curl localhost:8080/get-number
{number: 5}
[zarnold@zachs-mbp counter-mysql % curl localhost:8080/get-number
{number: 6}
[zarnold@zachs-mbp counter-mysql % curl localhost:8080/get-number
{number: 7}
[zarnold@zachs-mbp counter-mysql % curl localhost:8080/get-number
{number: 8}
[zarnold@zachs-mbp counter-mysql % curl localhost:8080/get-number
{number: 9}
[zarnold@zachs-mbp counter-mysql % curl localhost:8080/get-number
{number: 10}
[zarnold@zachs-mbp counter-mysql % curl localhost:8080/get-number
{number: 11}
[zarnold@zachs-mbp counter-mysql % curl localhost:8080/get-number
{number: 12}
```

Figure 14.18: Testing our application

10. Now to clean up, let's run the following command to find out what is currently deployed:

```
helm ls
```

You should see a response similar to the following:

```
NAME                    NAMESPACE       REVISION        UPDATED
                STATUS          CHART                   APP VERSION
chart-1592789369        default         1               2020-06-21 21:29:30.7551
07 -0400 EDT    deployed        counter-mysql-0.1.0     1.16.0
```

Figure 14.19: Listing all the installed Helm charts

From this output, you can determine the name of the chart that you need to delete for cleanup. For the author's instance, the name of the chart is **`chart-1592789369`**.

11. Now run the following command to delete the chart, using the name for your instance:

```
helm uninstall chart-1592789369
```

You should see a response similar to the following:

```
release "chart-1592789369" uninstalled
```

12. Finally, because this is a StatefulSet, we need to clean up any PersistentVolumes before deleting our cluster out of AWS or we'll be paying for them for quite some time!

```
kubectl delete pv --all
```

You should see a response similar to this:

```
persistentvolume "pvc-b20b473b-73a9-4d9c-befc-c0be15c44011" deleted
```

Don't forget to clean up your cloud resources using the **`terraform destroy`** command.

CHAPTER 15: MONITORING AND AUTOSCALING IN KUBERNETES

ACTIVITY 15.01: AUTOSCALING OUR CLUSTER USING CLUSTERAUTOSCALER

Solution:

Before you begin, please make sure you have your EKS cluster set up and that you have successfully completed all the exercises in this chapter:

1. Assuming that your infrastructure is set up and you have all the required resources from the exercises in this chapter, we simply need to run another load test, but this time for much longer. Run the following command to begin the load test:

```
docker run --rm skandyla/wrk -t10 -c1000 -d86400 -H 'Host: counter.
com'  http://YOUR_HOSTNAME/get-number
```

   Use the Ingress endpoint that you obtain in *step 5* of *Exercise 15.02*, *Scaling Workloads in Kubernetes*. Note that this may take between 15 minutes to an hour of sustained load. You should see the following output, indicating that the load test is running:

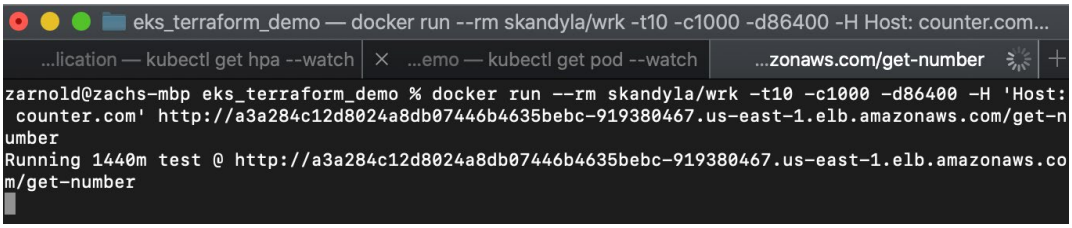

Figure 15.34: Running the load test

2. Run the following command to observe the number of worker nodes:

```
kubectl get node
```

You should see the following response:

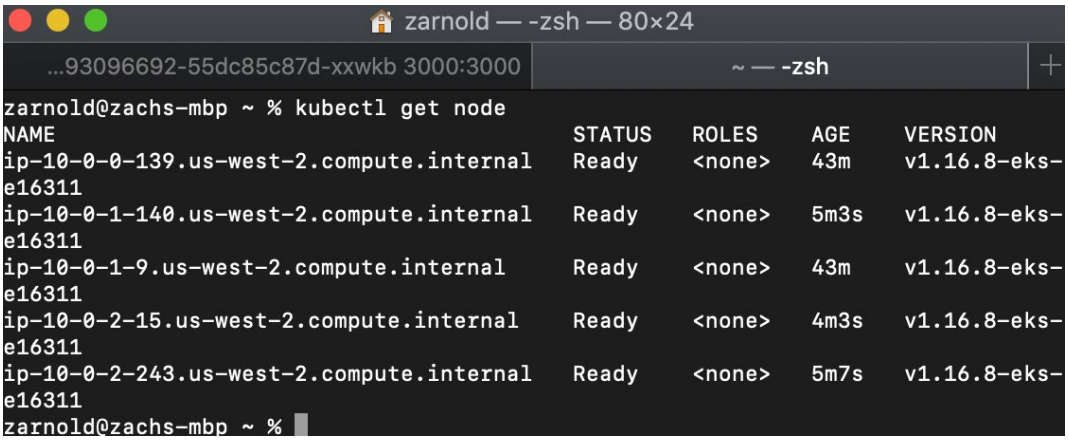

Figure 15.35: Observing the nodes through the kubectl command

Note that there are now five nodes running.

3. At this stage, you can even observe the nodes in the Grafana dashboard:

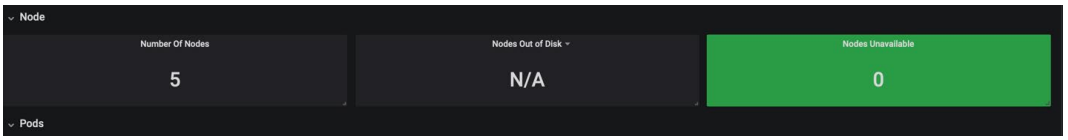

Figure 15.36: Observing the nodes in the Grafana dashboard

Don't forget to clean up your cloud resources using the **`terraform destroy`** command to stop AWS billing you after you are done with the activity.

CHAPTER 16: KUBERNETES ADMISSION CONTROLLERS

ACTIVITY 16.01: CREATING A MUTATING WEBHOOK THAT ADDS AN ANNOTATION TO A POD

Solution:

1. If you are following the exercises in the chapter, please make sure that you delete the existing **`webhooks`** namespace:

```
kubectl delete ns webhooks
```

You should see the following response:

```
namespace "webhooks" deleted
```

2. Create a namespace with the name **`webhooks`**:

```
kubectl create ns webhooks
```

You should see the following response:

```
namespace/webhooks created
```

3. Generate a self-signed certificate:

```
openssl req -nodes -new -x509 -keyout controller_ca.key -out
controller_ca.crt -subj "/CN=Mutating Admission Controller Webhook
CA"
```

You should see the following response:

```
$openssl req -nodes -new -x509 -keyout controller_ca.key -out controller_ca.crt -subj "/CN=Mutating Admiss
ion Controller Webhook CA"
Generating a 2048 bit RSA private key
.....................+++
.....................................................................................+++
writing new private key to 'controller_ca.key'
-----
```

Figure 16.19: Generating a self-signed certificate

4. Generate a private/public key pair and sign it with a CA certificate. This can be done with the following two commands:

```
openssl genrsa -out tls.key 2048


openssl req -new -key tls.key -subj "/CN=webhook-server.webhooks.svc"
\
    | openssl x509 -req -CA controller_ca.crt -CAkey controller_
ca.key -CAcreateserial -out tls.crt
```

You will get the following response:

```
$openssl genrsa -out tls.key 2048
Generating RSA private key, 2048 bit long modulus
.....................................................................................+++
......+++
e is 65537 (0x10001)
$openssl req -new -key tls.key -subj "/CN=webhook-server.webhooks.svc" \
>     | openssl x509 -req -CA controller_ca.crt -CAkey controller_ca.key -CAcreateserial -out tls.crt
Signature ok
subject=/CN=webhook-server.webhooks.svc
Getting CA Private Key
$
$
```

Figure 16.20: Signing a private/public key pair with our certificate

5. Create a secret that holds the private/public key pair:

```
kubectl -n webhooks create secret tls webhook-server-tls --cert "tls.
crt" --key "tls.key"
```

You should see the following response:

```
secret/webhook-server-tls created
```

6. Now, create a file named **`mutatingcontroller.go`** with the code for the webhook. A reference code for the webhook is provided at this link: https://packt.live/3bCS16D.

7. To build an image, create a file named **`Dockerfile`** with the following content:

```
FROM golang:alpine as builder

LABEL version="1.0" \
      maintainer=»name@example.com»

RUN mkdir /go/src/app && apk update && apk add --no-cache git && go
get -u github.com/golang/dep/cmd/dep
COPY ./*.go ./Gopkg.toml /go/src/app/

WORKDIR /go/src/app

RUN dep ensure -v
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .

FROM registry.fedoraproject.org/fedora-minimal:32
RUN groupadd appgroup && useradd appuser -G appgroup
COPY --from=builder /go/src/app/main /app/
WORKDIR /app
```

```
USER appuser
CMD ["./main"]
```

This example is using the Docker multi-stage build. In the first stage, the code is compiled. In the second stage, we are getting the compiled code and building a container that contains just the binary.

> **NOTE**
>
> This will work even if you do not use a multi-stage build and instead go with a simple build, as demonstrated in *Chapter 1*, *Understanding Kubernetes and Containers*. If you want to know more about multi-stage builds, refer to this link: https://docs.docker.com/develop/develop-images/multistage-build/.

8. Now, we can build the container using the following command:

```
docker build -t webhooks:0.0.1 .
```

You should see the following output:

```
$docker build -t webhooks:0.0.1 .
Sending build context to Docker daemon  14.55MB
Step 1/13 : FROM golang:alpine as builder
 ---> 6b21b4c6e7a3
Step 2/13 : LABEL version="1.0"       maintainer="fmasood@redhat.com"
 ---> Using cache
 ---> 971aaf4f1199
Step 3/13 : RUN mkdir /go/src/app && apk update && apk add --no-cache git && go get -u github.com/golang
/dep/cmd/dep
 ---> Using cache
 ---> f66d101a2383
Step 4/13 : COPY ./*.go ./Gopkg.toml /go/src/app/
 ---> Using cache
 ---> 4a5f9950037b
Step 5/13 : WORKDIR /go/src/app
 ---> Using cache
 ---> bdb1d1beecde
Step 6/13 : RUN dep ensure -v
 ---> Using cache
 ---> c73b0b343bc8
Step 7/13 : RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
 ---> Using cache
 ---> 742dc2d704e7
Step 8/13 : FROM registry.fedoraproject.org/fedora-minimal:32
 ---> b092bf43d972
Step 9/13 : RUN groupadd appgroup && useradd appuser -G appgroup
 ---> Using cache
 ---> 82b1875a9b6e
Step 10/13 : COPY --from=builder /go/src/app/main /app/
 ---> Using cache
 ---> 4a1c5a2706c2
Step 11/13 : WORKDIR /app
 ---> Using cache
 ---> 1d9d0f0331bf
Step 12/13 : USER appuser
 ---> Using cache
 ---> cc4f52a31401
Step 13/13 : CMD ["./main"]
 ---> Using cache
 ---> 9a623b63e59f
Successfully built 9a623b63e59f
Successfully tagged webhooks:0.0.1
$
```

Figure 16.21: Building the Docker image for our webhook

9. Now push the packaged container to a public repository. We have used **quay.io**, which allows you to create a free public repository. The push is a two-step process: the first step is to tag the local container, and the second step is to push it up. Use the following two commands, one after the other:

```
docker tag webhooks:0.0.1 YOUR_REPO_NAME/webhooks:0.0.1

docker push YOUR_REPO_NAME/webhooks:0.0.1
```

Change these commands as per your repository. You should see the following response:

```
c83a8affbb23: Pushed
bbfc0444dbab: Pushed
951a82a6f24a: Pushed
[DEPRECATION NOTICE] registry v2 schema1 support will be removed in an upcoming release. Please contact
admins of the quay.io registry NOW to avoid future disruption.
```

Figure 16.22: Pushing the Docker image to a public repository

10. Now we need to deploy the webhook in the **`webhooks`** namespace. For that, first, define a Deployment by creating a file, named **`mutating-server.yaml`**, with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webhook-server
  labels:
    app: webhook-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webhook-server
  template:
    metadata:
      labels:
        app: webhook-server
    spec:
      containers:
      - name: server
        image: packtworkshops/the-kubernetes-workshop:webhook
        imagePullPolicy: Always
        ports:
```

```
        - containerPort: 8443
          name: webhook-api
        volumeMounts:
        - name: webhook-tls-certs
          mountPath: /etc/secrets/tls
          readOnly: true
      volumes:
      - name: webhook-tls-certs
        secret:
          secretName: webhook-server-tls
```

> **NOTE**
>
> If you are using the image you have built and uploaded to your own repository as per the previous steps, remember to update the **`image`** field in this specification. Otherwise, you can keep it the same, and the Pod will fetch the reference image that we have provided.

There are two important sections in this definition. The first is the image location and the second is the Kubernetes secret, which contains the SSL certificates for the webhook server.

11. Deploy the webhook server with this definition using the following command:

```
kubectl create -f mutating-server.yaml -n webhooks
```

You should see the following response:

```
deployment.apps/webhook-server created
```

12. You might need to wait a bit and check whether the webhook Pods have been created. Keep checking the status of the Pods:

```
kubectl get pods -n webhooks -w
```

You should see the following response:

```
NAME                              READY   STATUS              RESTARTS   AGE
webhook-server-68b8d6b987-fbv95   0/1     ContainerCreating   0          12s
webhook-server-68b8d6b987-fbv95   1/1     Running             0          15s
^C$
```

Figure 16.23: Checking the status of our webhook server

Note that the **`-w`** flag continuously watches the Pods. You can end the watch using *Ctrl* + *C* when all the Pods are ready.

13. Expose the deployed webhook server via a Kubernetes service. Create a file named **`mutating-serversvc.yaml`** using the following definition:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: webhook-server
  name: webhook-server
  namespace: webhooks
spec:
  ports:
  - port: 443
    protocol: TCP
    targetPort: 8443
  selector:
    app: webhook-server
  sessionAffinity: None
  type: ClusterIP
```

Note that the webhook service has to be running on port **`443`**, as this is the standard port used for SSL connections.

14. Create the service with this definition using the following command:

```
kubectl create -f mutating-serversvc.yaml -n webhooks
```

You should see the following response:

```
service/webhook-server created
```

15. Create a Base64-encoded version of the CA certificate. The first command is to convert the certificate into a PEM format. And the second one is to convert the PEM certificate into Base64:

```
openssl x509 -inform PEM -in controller_ca.crt > controller_ca.crt.
pem

openssl base64 -in controller_ca.crt.pem -out controller_ca-base64.
crt.pem
```

The terminal shows no response once these commands have been successfully executed. You can inspect the file using your favorite text editor, or the **cat** command, as follows:

```
cat controller_ca-base64.crt.pem
```

The file contents should be similar to the following figure. Please note that the TLS certificates you generate will not look exactly like what is shown here:

```
$cat controller_ca-base64.crt.pem
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUM0akNDQWNvQ0NRRHNPMFph
Q0swMDVUQU5CZ2txaGtpRzl3MEJBUXNGQURBek1URXdMd1lEVlFRRERDaE4KZFhS
aGRHbHVaeUJCWkcxcGMzTnBiMjRnUTI5dWRISnZiR3hsY2lCWFpXSm9iMjlySUVO
Qk1CNFhEVEU1TURneQpNakEwTWpBeU0xb1hEVEU1TURreU1UQTBNakF5TTFvd016
RXhNQzhHQTFVRUF3d29UWFYwWVhScGJtY2dRV1J0CmFYTnphVzl1SUVOdmJuUnli
MnhzWlhJZ1YyVmlhRzl2YXlCRFFUQ0NBU0l3RFFZSktvWklodmNOQVFFQkJRUQK
Z2dFUEFEQ0NBUW9DZ2dFQkFLbnVhT1pIcml2TDNIZ3oxbHhrVVdnczY0ei9DVFRV
OTRjOGhLTkdNdHdvMGt6SwpxWXR4NnQ5MTIxNWc0Q0dsbE0zM0pEamljd21XRUJW
dkNDNWFYbEtDc1d6STlHS1ZtVEFESW80RFpDSXhvaHBNClR5TkhPZUxTbDJlSllY
c2V2cEowRnRBSlRJZ0ZkRm52UUlYYUNYUkJyYVdNV3JpUWZnZGsvaThyQzJveWsw
blEKRElQb3ljVEFBcmlSZ0FvZnBTek9nZnVqc3ZJeEViaDYvMnNRaHY5M25icFFR
TWNHKzNkWHVpWjZweTFTRDArYQplSThXRUFoMDQ0MXpaWGRDOEI5OHlndldCWWdo
QUN3dUIzdHFKeEdjbzdDZ011NkVWQXZVTFNvSlA3a0VQS3VTCnRtbHYvUm16Vm5P
NWx5NkI1Y1BjOGRKYk1WKzROZVNObngxNmc0MENBd0VBQVRBTkJna3Foa2lHOXcw
QkFRc0YKQUFPQ0FRRUFpVC96Q0F1dVR6OFhsSlduczIwWGIwSmJGcE4vZDZKN3hU
VGp5S0k3SU9rRW9zaUZaSDg3ZDB3NgpYd3JBTnZiR2NIdlhOZHRRTkY3S3ArZ1k1
RzFSeFV3V3BEUnpjdVlrOEVWWnJCMDNhM2JDVXl4ejZmRkUxanlUCmcvMmdKSFhV
d005Yy92LzhOZ2NNa2ZtdFlLeGErREV6anQ3V2oxbDFUYlVFM3NCK1ZFalVYYWFt
Z3pyZEJFRHQKQmhVejhVZE1sUUVONVNEVWtHNlJzTkVlMWJ1dml0cHlYVGdOb0tB
ZWVQWkU5Rk5NME5lSmJVbVVacWsyejhTNgpDUVRjOUtSU0piR203bGUrV1ptYnBy
NzJGcG9GbmZhbXdHUDJVODIxOTQ2TWVROXFiRGVwSXNFSURlTXlUeEdOClcxZWdr
eFVodVdSdTVGbmZkd2ZYT1ZlaEhiMDlwdz09Ci0tLS0tRU5EIENFUlRJRklDQVRF
LS0tLS0K
$
```

Figure 16.24: Contents of the Base64-encoded CA certificate

16. Now we need to remove the blank lines from our Base64-encoded certificate. For that, you can use the following command:

```
cat controller_ca-base64.crt.pem | tr -d '\n' > onelinecert.pem
```

This command shows no response. You can see the new file by issuing the following command:

```
cat onelinecert.pem
```

The contents of the file should look like this:

```
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUM0akNDQWNvQ0NRRHNPMFphQ0swMDVUQU5CZ2txaGtpRzl3MEJBUXNGQURBek1URXdMd1lEVlFRRER
DaE4KZFhSaGRHbHVaeUJCWkcxcGMzTnBiMjRnUTI5dWRISnZiR3hsY2lCWFpXSm9iMjlySUVOQk1CNFhEVEU1TURneQpNakEwTWpBeU0xb1hEVEU1TURreU
1UQTBNakF5TTFvd016RXhNQzhHQTFVRUF3d29UWFYwWVhScGJtY2dRV1J0CmFYTnphVzl1SUVOdmJuUnliMnhzWlhJZ1YyVmlhRzl2YXlCRFFUQ0NBU0l3R
FFZSktvWklodmNOQVFFQkJRQUQKZ2dFUEFEQ0NBUW9DZ2dFQkFLbnVhT1pIcml2TDNIZ3oxbHhrVVdnczY0ei9DVFRVOTRjOGhLTkdNdHdvMGt6SwpxWXR4
NnQ5MTIxNWc0Q0dsbE0zM0pEamljd21XRUJWdkNDNWFYbEtDc1d6STlHS1ZtVEFESW80RFpDSXhvaHBNClR5TkhPZUxTbDJlSllYc2V2cEowRnRBSlRJZ0Z
kRm52UU1YYUNYUkJyYVdNV3JpUWZnZGsvaThyQzJveWswblEKRElQb3ljVEFBcmlSZ0FvZnBTek9nZnVqc3ZJeEViaDYvMnNRaHY5M25icFFRTWNHKzNkWH
VpWjZweTFTRDArYQplSThXRUFoMDQ0MXpaWGRDOEI5OHlndldCWWdoQUN3dUIzdHFKeEdjbzdDZ011NkVWQXZVTFNvSlA3a0VQS3VTCnRtbHYvUm16Vm5PN
Wx5NkI1Y1BjOGRKYk1WKzROZVNObngxNmc0MENBd0VBQVRBTkJna3Foa2lHOXcwQkFRc0YKQUFPQ0FRRUFpVC96Q0F1dVR6OFhsSlduczIwWGIwSmJGcE4v
ZDZKN3hUVGp5S0k3SU9rRW9zaUZaSDg3ZDB3NgpYd3JBTnZiR2NIdlhOZHRRTkY3S3ArZ1k1RzFSeFV3V3BEUnpjdVlrOEVWWnJCMDNhM2JDVXl4ejZmRkU
xanlUCmcvMmdKSFhVd005Yy92LzhOZ2NNa2ZtdFlLeGErREV6anQ3V2oxbDFUY1VFM3NCK1ZFalVYYWFtZ3pyZEJFRHQKQmhVejhVZE1sUUVONVNEVWtHNl
JzTkVlMWJ1dml0cHlYVGdOb0tBZWVQWkU5Rk5NME5lSmJVbVVacWsyejhTNgpDUVRjOUtSU0piR203bGUrV1ptYnByNzJGcG9GbmZhbXdHUDJVODIxOTQ2T
WVROXFiRGVwSXNFSURlTXlUeEdOClcxZWdreFVodVdSdTVGbmZkd2ZYT1ZlaEhiMDlwdz09Ci0tLS0tRU5EIENFUlRJRklDQVRFLS0tLS0K$
```

Figure 16.25: Base64-encoded CA certificate with the line breaks removed

Now we have the Base64-encoded certificate with no blank lines. For the next step, we will copy the value that you get in this output, being careful not to copy the **$** (which would be `%`, in the case of Zsh) at the end of the value. Paste this value in place of **`CA_BASE64_PEM`** (a placeholder for **`caBundle`**) in **`mutating-config.yaml`**, which will be created in the next step.

17. Create a **`MutatingWebhookConfiguration`** object using the following specification to configure the Kubernetes API server to call our webhook. Save this file as **`mutating-config.yaml`**:

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  name: pod-annotation-webhook
webhooks:
  - name: webhook-server.webhooks.svc
    clientConfig:
      service:
        name: webhook-server
        namespace: webhooks
        path: «/mutate»
      caBundle: "CA_BASE64_PEM"    #Retain the quotes when you
        copy the caBundle here. Please read the note below on
        how to add specific values here.
    rules:
      - operations: [ «CREATE" ]
        apiGroups: [""]
        apiVersions: ["v1"]
        resources: [«pods»]
```

> **NOTE**
>
> The **`CA_BASE64_PEM`** placeholder will be replaced with the contents of **`onelinecert.pem`** from the previous step. Be careful not to copy any line breaks. Remember that this is the value of the CA certificate with blank lines from the Base64-encoded value obtained in *step 15*.

18. Create the **`MutatingWebHookConfiguration`** object using the following command:

```
kubectl create -f mutating-config.yaml -n webhooks
```

    You should see the following response:

```
mutatingwebhookconfiguration.admissionregistration.k8s.io/
pod-annotation-webhook created
```

19. Confirm that the webhook server Pod is running:

```
kubectl get pods -n webhooks | grep webhook-server
```

    You should see this response:

```
webhook-server-6f7957bbf5-pbbcn  1/1   Running    1   18h
```

    So, we have confirmation that our webhook is running. Now, let's test whether it is working as expected in the following steps.

20. Create a simple Pod with no annotations. Use the following Pod specification and create a file named **`target-pod.yaml`**:

```
apiVersion: v1
kind: Pod
metadata:
  name: mutating-pod-example
spec:
  containers:
    - name: cmutating-pod-example-container
      image: k8s.gcr.io/busybox
      command: [ «/bin/sh", "-c", "while :; do echo '.'; sleep
        5 ; done" ]
```

    Note that there is no annotation defined in this specification.

21. Use the following command to create the Pod:

```
kubectl create -f target-pod.yaml -n webhooks
```

    You will see the following response:

```
pod/mutating-pod-example created
```

22. Issue the following command to describe the Pod to check whether the annotation has been inserted:

```
kubectl describe pod mutating-pod-example -n webhooks
```

    You should get the following output:

```
Name:               mutating-pod-example
Namespace:          webhooks
Priority:           0
PriorityClassName:  <none>
Node:               minikube/192.168.247.150
Start Time:         Thu, 22 Aug 2019 19:49:45 +1000
Labels:             <none>
Annotations:        podModified: true
Status:             Running
```

Figure 16.26: Checking the mutations on our sample Pod

This output has been truncated for brevity. As you can see, the webhook code is inserting the annotation with the name as **`podModified`** and the value as **`true`**, which is what we had coded.

We can now intercept the API traffic and change the object as per our requirements. This is useful for various purposes such as audits, policy enforcement, quotas, standards, and much more.

ACTIVITY 16.02: CREATING A VALIDATING WEBHOOK THAT CHECKS FOR A LABEL IN A POD

Solution:

1. To prevent various objects from interfering with each other, please make sure that you delete the existing **`webhooks`** namespace:

```
kubectl delete ns webhooks
```

   You should see the following response:

```
namespace "webhooks" deleted
```

2. Now recreate the **`webhooks`** namespace:

```
kubectl create ns webhooks
```

   You should see the following response:

```
namespace/webhooks created
```

3. Generate a self-signed certificate using this command:

```
openssl req -nodes -new -x509 -keyout controller_ca.key -out
controller_ca.crt -subj "/CN=Mutating Admission Controller Webhook
CA"
```

   You will see the following response:

```
$openssl req -nodes -new -x509 -keyout controller_ca.key -out controller_ca.crt -subj "/CN=Mutating Admiss
ion Controller Webhook CA"
Generating a 2048 bit RSA private key
.....................+++
.....................................................................................+++
writing new private key to 'controller_ca.key'
-----
```

Figure 16.27: Generating a self-signed certificate

4. Generate a private/public key pair and sign it with the CA certificate using the following two commands, one after the other:

```
openssl genrsa -out tls.key 2048


openssl req -new -key tls.key -subj "/CN=webhook-server.webhooks.svc"
\
    | openssl x509 -req -CA controller_ca.crt -CAkey controller_
ca.key -CAcreateserial -out tls.crt
```

You should see the following response:

```
$openssl genrsa -out tls.key 2048
Generating RSA private key, 2048 bit long modulus
.....................................................................................+++
......+++
e is 65537 (0x10001)
$openssl req -new -key tls.key -subj "/CN=webhook-server.webhooks.svc" \
>     | openssl x509 -req -CA controller_ca.crt -CAkey controller_ca.key -CAcreateserial -out tls.crt
Signature ok
subject=/CN=webhook-server.webhooks.svc
Getting CA Private Key
$
$
```

Figure 16.28: Signing the private/public key pair with our certificate

5. Create a secret that holds the private/public key pair:

```
kubectl -n webhooks create secret tls webhook-server-tls \
--cert "tls.crt" \
--key "tls.key"
```

You should see the following response:

```
secret/webhook-server-tls created
```

6. Create a file, named **`validatecontroller.go`**, with the code for the webhook server. The reference code is provided at this link: https://packt.live/3bAlHB4.

7. Now, create the **`Dockerfile`** for containerizing this application. The reference **`Dockerfile`** shown here uses a multi-stage build, but you can also use a simple build:

```
FROM golang:alpine as builder

LABEL version="1.0" \
      maintainer=»name@example.com»

RUN mkdir /go/src/app && apk update && apk add --no-cache git && go
get -u github.com/golang/dep/cmd/dep
COPY ./*.go ./Gopkg.toml /go/src/app/

WORKDIR /go/src/app

RUN dep ensure -v
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
```

```
FROM registry.fedoraproject.org/fedora-minimal:32
RUN groupadd appgroup && useradd appuser -G appgroup
COPY --from=builder /go/src/app/main /app/
WORKDIR /app
USER appuser
CMD ["./main"]
```

8. Using the **`Dockerfile`** shown in the previous step, which is available in the code repository, build the container using the **`docker build`** command, as follows:

```
docker build -t webhooks:0.0.1 .
```

You should see the following output:

```
$docker build -t webhooks:0.0.1 .
Sending build context to Docker daemon  14.55MB
Step 1/13 : FROM golang:alpine as builder
 ---> 6b21b4c6e7a3
Step 2/13 : LABEL version="1.0"       maintainer="fmasood@redhat.com"
 ---> Using cache
 ---> 971aaf4f1199
Step 3/13 : RUN mkdir /go/src/app && apk update && apk add --no-cache git && go get -u github.com/golang
/dep/cmd/dep
 ---> Using cache
 ---> f66d101a2383
Step 4/13 : COPY ./*.go ./Gopkg.toml /go/src/app/
 ---> Using cache
 ---> 4a5f9950037b
Step 5/13 : WORKDIR /go/src/app
 ---> Using cache
 ---> bdb1d1beecde
Step 6/13 : RUN dep ensure -v
 ---> Using cache
 ---> c73b0b343bc8
Step 7/13 : RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
 ---> Using cache
 ---> 742dc2d704e7
Step 8/13 : FROM registry.fedoraproject.org/fedora-minimal:32
 ---> b092bf43d972
Step 9/13 : RUN groupadd appgroup && useradd appuser -G appgroup
 ---> Using cache
 ---> 82b1875a9b6e
Step 10/13 : COPY --from=builder /go/src/app/main /app/
 ---> Using cache
 ---> 4a1c5a2706c2
Step 11/13 : WORKDIR /app
 ---> Using cache
 ---> 1d9d0f0331bf
Step 12/13 : USER appuser
 ---> Using cache
 ---> cc4f52a31401
Step 13/13 : CMD ["./main"]
 ---> Using cache
 ---> 9a623b63e59f
Successfully built 9a623b63e59f
Successfully tagged webhooks:0.0.1
$
```

Figure 16.29: Building the Docker image

9. Now, push the packaged container to a public repository. The push is a two-step process – the first step is to tag the local container and the second step is to push your local container to a central repository. Use the following two commands in the given order:

```
docker tag webhooks:0.0.1 YOUR_REPO_NAME/webhooks:0.0.1

docker push YOUR_REPO_NAME/webhooks:0.0.1
```

You should see the following response:

```
bbfc0444dbab: Pushed
951a82a6f24a: Pushed
[DEPRECATION NOTICE] registry v2 schema1 support will be removed in an upcoming release. Please contact
admins of the quay.io registry NOW to avoid future disruption.
```

Figure 16.30: Pushing the Docker image to a public repository

10. Next, we need to define a Deployment to deploy the webhook in the **`webhooks`** namespace. Create a file named **`validating-server.yaml`** with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webhook-server
  labels:
    app: webhook-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webhook-server
  template:
    metadata:
      labels:
        app: webhook-server
    spec:
      containers:
      - name: server
        image: packtworkshops/the-kubernetes-workshop:webhook
        imagePullPolicy: Always
        ports:
        - containerPort: 8443
```

```
            name: webhook-api
          volumeMounts:
          - name: webhook-tls-certs
            mountPath: /etc/secrets/tls
            readOnly: true
        volumes:
        - name: webhook-tls-certs
          secret:
            secretName: webhook-server-tls
```

Note that there is one important section in this definition that is different from the previous activity – the Kubernetes secret, which contains the SSL certificates, for the webhook server.

11. Deploy the webhook server by using the Deployment defined in the previous step:

```
kubectl create -f validating-server.yaml -n webhooks
```

You should see the following response:

```
deployment.apps/webhook-server created
```

12. You might need to wait a bit and check whether the webhook Pods have been created. Keep checking the status of the Pods:

```
kubectl get pods -n webhooks -w
```

You should see the following response:

```
NAME                              READY   STATUS              RESTARTS   AGE
webhook-server-68b8d6b987-fbv95   0/1     ContainerCreating   0          12s
webhook-server-68b8d6b987-fbv95   1/1     Running             0          15s
^C$
```

Figure 16.31: Checking whether the webhook Pod is online

Note that the **-w** flag continuously watches the Pods. You can end the watch when all the Pods are ready.

13. Now, we have to expose the deployed webhook server via the Kubernetes service. Create a file named **`validating-serversvc.yaml`** with the following content:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: webhook-server
  name: webhook-server
  namespace: webhooks
spec:
  ports:
  - port: 443
    protocol: TCP
    targetPort: 8443
  selector:
    app: webhook-server
  sessionAffinity: None
  type: ClusterIP
```

Note that the webhook service has to be running on port **`443`**.

14. Use the definition from the previous step to create the service using the following command:

```
kubectl create -n webhooks -f validating-serversvc.yaml
```

You should see the following output:

```
service/webhook-server created
```

15. Create a Base64-encoded version of the CA certificate. Use the following commands, one after the other:

```
openssl x509 -inform PEM -in controller_ca.crt > controller_ca.crt.
pem

openssl base64 -in controller_ca.crt.pem -out controller_ca-base64.
crt.pem
```

The first command is to convert the certificate into a PEM format. And the second one is to convert the PEM certificate into Base64. These commands show no response. You can inspect the file using the following command:

```
cat controller_ca-base64.crt.pem
```

The file contents should be similar to the following figure:

Figure 16.32: The contents of the Base64-encoded CA certificate

Please note that the TLS certificates you will generate will not look exactly like what is shown here.

16. Use the following two commands to clean up the blank lines from our CA certificate and add the contents to a new file:

```
cat controller_ca-base64.crt.pem | tr -d '\n' > onelinecert.pem


cat onelinecert.pem
```

The first command shows no response, and the second one prints out the contents of **onelinecert.pem**. You should see the following response:

```
$cat controller_ca-base64.crt.pem | tr -d '\n' > onelinecert.txt
$cat onelinecert.txt
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUM0akNDQWNvQ0NRRHNPMFphQ0swMDVUQU5CZ2txaGtpRzl3MEJBUXNGQURBek1URXd
Md1lEVlFRRERDaE4KZFhSaGRHbHVaeUJCWkcxcGMzTnBiMjRnUTI5dWRISnZiR3hsY2lCWFpXSm9iMjlySUVOQk1CNFhEVEU1TURneQpNak
EwTWpBeU0xb1hEVEU1TURreU1UQTBNakF5TTFvd016RXhNQzhHQTFVRUF3d29UWFYwWVhScGJtY2dRV1J0CmFYTnphVzl1SUVOdmJuUnliM
nhzWlhJZ1YyVmlhRzl2YXlCRFFUQ0NBU0l3RFFZSktvWklodmNOQVFFQkJRQUQKZ2dFUEFEQ0NBUW9DZ2dFQkFLbnVhT1pIcml2TDNIZ3ox
bHhrVVdnczY0ei9DVFRVOTRjOGhLTkdNdHdvMGt6SwpxWXR4NnQ5MTIxNWc0Q0dsbE0zM0pEamljd21XRUJWdkNDNWFYbEtDc1d6STlHS1Z
tVEFESW80RFpDSXhvaHBNClR5TkhPZUxTbDJlS1llYc2V2cEowRnRBSlRJZ0ZkRm52UUlYYUNYUkJyYVdNV3JpUWZnZGsvaThyQzJveWswbl
EKRElQb3ljVEFBcmlSZ0FvZnBTek9nZnVqc3ZJeEViaDYvMnNRaHY5M25icFFRTWNHKzNkWHVpWjZweTFTRDArYQplSThXRUFoMDQ0MXpaW
GRDOEI5OHlndldCWWdoQUN3dUIzdHFKeEdjbzdDZ011NkVWQXZVTFNvSlA3a0VQS3VTCnRtbHYvUm16Vm5PNWx5NkI1Y1BjOGRKYk1WKzRO
ZVNObngxNmc0MENBd0VBQVRBTkJna3Foa2lHOXcwQkFRc0YKQUFPQ0FRRUFpVC96Q0F1dVR6OFhsSlduczIwWGIwSmJGcE4vZDZKN3hUVGp
5S0k3SU9rRW9zaUZaSDg3ZDB3NgpYd3JBTnZiR2NIdlhOZHRRTkY3S3ArZ1k1RzFSeFV3V3BEUnpjdVlrOEVWWnJCMDNhM2JDVXl4ejZmRk
UxanlUCmcvMmdKSFhVd005Yy92LzhOZ2NNa2ZtdFlLeGErREV6anQ3V2oxbDFUYlVFM3NCK1ZFalVYYWFtZ3pyZEJFRHQKQmhVejhVZE1sU
UVONVNEVWtHNlJzTkVlMWJ1dml0cHlYVGdOb0tBZWVQWkU5Rk5NME5lSmJVbVVacWsyejhTNgpDUVRjOUtSU0piR203bGUrV1ptYnByNzJG
cG9GbmZhbXdHUDJVODIxOTQ2TWVROXFiRGVwSXNFSURlTXlUeEdOClcxZWdreFVodVdSdTVGbmZkd2ZYT1ZlaEhiMDlwdz09Ci0tLS0tRU5
FIENFUlRJRklDQVRFLS0tLS0K$
```

Figure 16.33: Base64-encoded CA certificate with the line breaks removed

Now, we have the Base64-encoded certificate with no blank lines. For the next step, we will copy the value that you get in this output, being careful not to copy the **$** (which would be **%**, in the case of Zsh) at the end of the value. Paste this value in place of **CA_BASE64_PEM** (a placeholder for **caBundle**) in **validating-config.yaml**, which will be created in the next step.

17. Create a file, named **validation-config.yaml**, using the following **ValidatingWebHookConfiguration** specification to configure the Kubernetes API server and call our webhook:

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration
metadata:
  name: pod-label-verify-webhook
webhooks:
  - name: webhook-server.webhooks.svc
    clientConfig:
      service:
        name: webhook-server
        namespace: webhooks
        path: «/validate»
      caBundle: "CA_BASE64_PEM"    #Retain the quotes when you
        copy the caBundle here. Please read the note below on
        how to add specific values here.
```

```
      rules:
        - operations: [ «CREATE" ]
          apiGroups: [""]
          apiVersions: ["v1"]
          resources: [«pods»]
```

> **NOTE**
>
> The **`CA_BASE64_PEM`** placeholder will be replaced with the contents of **`onelinecert.pem`** from the previous step. Be careful not to copy any line breaks. Remember that this is the value of the CA certificate with blank lines from the Base64-encoded value obtained in *step 14*.

18. Create the **`ValidatingWebHookConfiguration`** object using the following command:

```
kubectl create -f validation-config.yaml -n webhooks
```

```
validatingwebhookconfiguration.admissionregistration.k8s.io/pod-label-verify-webhook created
```

Figure 16.34: Creating the ValidatingWebhookConfiguration object

19. Create a simple Pod with no labels. Use the following Pod specification to create a file named **`target-validating-pod.yaml`**:

```
apiVersion: v1
kind: Pod
metadata:
  name: validating-pod-example
spec:
  containers:
    - name: validating-pod-example-container
      image: k8s.gcr.io/busybox
      command: [ «/bin/sh", "-c", "while :; do echo '.'; sleep
        5 ; done" ]
```

20. Create the Pod defined in the previous step:

```
kubectl create -f target-validating-pod.yaml -n webhooks
```

You should see the following response:

```
Error from server: error when creating "target-validating-pod.yaml":
 admission webhook "webhook-server.webhooks.svc" denied the request:
 labels not found
```

Figure 16.35: Pod getting rejected by the validating webhook

Note that the Pod creation is rejected as there is a **`labels`** section defined in the Pod specification in the previous step.

21. Now, modify the definition in *step 19* to add the **`teamName`** label. This is the key of the label that our validating webhook will try to validate. Your specification should look like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: validating-pod-example
  labels:
    teamName: kubeteam
spec:
  containers:
    - name: validating-pod-example-container
      image: k8s.gcr.io/busybox
      command: [ «/bin/sh", "-c", "while :; do echo '.'; sleep
        5 ; done" ]
```

22. Try creating the Pod again:

```
kubectl create -f target-validating-pod.yaml -n webhooks
```

You will see the following response:

```
pod/validating-pod-example created
```

Note that the Pod is created this time.

23. Issue the following command to describe the Pod to check the label:

```
kubectl describe pod validating-pod-example -n webhooks
```

You should see the following output:

```
Name:               validating-pod-example
Namespace:          webhooks
Priority:           0
PriorityClassName:  <none>
Node:               minikube/192.168.247.150
Start Time:         Sat, 24 Aug 2019 00:05:18 +1000
Labels:             teamName=kubeteam
```

Figure 16.36: Checking the annotations of our Pod

Note that we have truncated the output for brevity. As you can see, the Pod was created because we had added the required label to it.

Therefore, we have learned how to intercept the API traffic and allow/reject the object (or Pod, in this case) creation as per our requirements.

CHAPTER 17: ADVANCED SCHEDULING IN KUBERNETES

ACTIVITY 17.01: CONFIGURING A KUBERNETES SCHEDULER TO SCHEDULE PODS

Solution:

1. Create a new namespace:

```
kubectl create ns scheduleractivity
```

   This should give the following response:

```
namespace/scheduleractivity created
```

2. Create a file named **`pod-priority-api.yaml`** with the following contents:

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: api-pod-priority
value: 9999
globalDefault: false
description: "This priority class should be used for API
  pods."
```

3. Create a Pod priority using the priority class that we defined in the previous step:

```
kubectl create -f pod-priority-api.yaml -n scheduleractivity
```

   This will give the following response:

```
priorityclass.scheduling.k8s.io/api-pod-priority created
```

4. Create a file named **`api-pod-deploy.yaml`** using the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-pod
  labels:
    app: api-pod
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api-pod
```

```
    template:
      metadata:
        labels:
          app: api-pod
      spec:
        priorityClassName: api-pod-priority
        containers:
        - name: api-pod
          image: k8s.gcr.io/busybox
          imagePullPolicy: Always
          command: [ «/bin/sh", "-c", "while :; do echo 'this is
            API pod'; sleep 3 ; done" ]
          resources:
            limits:
              cpu: "0.1"
              memory: «100Mi»
            requests:
              cpu: "0.1"
              memory: «100Mi»
```

5. Create the Deployment defined in the previous step to get the API Pod running:

```
kubectl create -f api-pod-deploy.yaml -n scheduleractivity
```

You will see the following response:

```
deployment.apps/api-pod created
```

6. Verify that the API Pods are running:

```
kubectl get pods -n scheduleractivity
```

You should get the following response

```
NAME                          READY   STATUS    RESTARTS   AGE
api-pod-77466d47d8-vwclx      1/1     Running   1          17h
```

Figure 17.23: Getting the list of Pods

7. Create a file named **`gui-pod-deploy.yaml`** with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gui-pod
  labels:
```

```
      app: gui-pod
  spec:
    replicas: 1
    selector:
      matchLabels:
        app: gui-pod
    template:
      metadata:
        labels:
          app: gui-pod
      spec:
        affinity:
          podAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                - key: app
                operator: In
                  values:
                  - api-pod
              topologyKey: kubernetes.io/hostname

        containers:
        - name: gui-pod
          image: k8s.gcr.io/busybox
          imagePullPolicy: Always
          command: [ «/bin/sh", "-c", "while :; do echo 'this is
            GUI pod'; sleep 5 ; done" ]
          resources:
            limits:
              cpu: "0.1"
              memory: «100Mi»
            requests:
              cpu: "0.1"
              memory: «100Mi»
```

8. Create the GUI Pod using the Deployment that we defined in the previous step:

```
kubectl create -f gui-pod-deploy.yaml -n scheduleractivity
```

You will see the following response:

```
deployment.apps/gui-pod created
```

9. Verify that both the API and GUI Pods are running:

```
kubectl get pods -n scheduleractivity
```

You will see the following list of Pods:

```
NAME                       READY   STATUS    RESTARTS   AGE
api-pod-77466d47d8-vwclx   1/1     Running   1          17h
gui-pod-7f49d768dd-xg7rn   1/1     Running   0          6s
```

Figure 17.24: Getting the list of Pods

10. Scale up the number of replicas for the API Pod to **2**:

```
kubectl scale deployment/api-pod --replicas=2 -n scheduleractivity
```

You should see the following response:

```
deployment.extensions/api-pod scaled
```

11. Scale up the number of replicas for the GUI Pod to **2**:

```
kubectl scale deployment/gui-pod --replicas=2 -n scheduleractivity
```

You will see the following response:

```
deployment.extensions/gui-pod scaled
```

12. Verify that two Pods each for API and GUI are running:

```
kubectl get pods -n scheduleractivity
```

You should see the following list of Pods:

```
NAME                       READY   STATUS    RESTARTS   AGE
api-pod-77466d47d8-tfskq   1/1     Running   0          93s
api-pod-77466d47d8-vwclx   1/1     Running   1          17h
gui-pod-7f49d768dd-tpknx   1/1     Running   0          22s
gui-pod-7f49d768dd-xg7rn   1/1     Running   0          2m29s
```

Figure 17.25: Getting the list of pods

13. Create a file named **`pod-priority-realtime.yaml`** with the following content:

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: realtime-pod-priority
value: 100000
globalDefault: false
description: "This priority class should be used for pods with
  the highest of priority."
```

14. Create the Pod priority using the priority class definition from the previous step:

```
kubectl create -f pod-priority-realtime.yaml -n scheduleractivity
```

You should see the following response:

```
priorityclass.scheduling.k8s.io.realtime-pod-priority created
```

15. Create a file named **`realtime-pod-deploy.yaml`** with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: realtime-pod
  labels:
    app: realtime-pod
spec:
  replicas: 1
  selector:
    matchLabels:
      app: realtime-pod
  template:
    metadata:
      labels:
        app: realtime-pod
    spec:
      priorityClassName: realtime-pod-priority
      containers:
      - name: realtime-pod
        image: k8s.gcr.io/busybox
        imagePullPolicy: Always
```

```
          command: [ «/bin/sh", "-c", "while :; do echo 'this is
            REALTIME pod'; sleep 1 ; done" ]
          resources:
            limits:
              cpu: "0.1"
              memory: «100Mi»
            requests:
              cpu: "0.1"
              memory: «100Mi»
```

16. Create this Deployment to get the real-time Pods running:

```
kubectl create -f realtime-pod-deploy.yaml -n scheduleractivity
```

    You will see the following response:

```
deployment.apps/realtime-pod created
```

17. Verify that the Pods are running:

```
kubectl get pods -n scheduleractivity
```

    You should see the following list of Pods:

```
NAME                            READY   STATUS    RESTARTS   AGE
api-pod-77466d47d8-tfskq        1/1     Running   0          6m1s
api-pod-77466d47d8-vwclx        1/1     Running   1          17h
gui-pod-7f49d768dd-tpknx        1/1     Running   0          4m50s
gui-pod-7f49d768dd-xg7rn        1/1     Running   0          6m57s
realtime-pod-fd55d8c88-g6pc7    1/1     Running   0          53s
```

Figure 17.26: Getting the list of Pods

18. For the real-time Pod, scale up the number of replicas to **10**:

```
kubectl scale deployment/realtime-pod --replicas=10 -n
scheduleractivity
```

    You should see the following response:

```
deployment.extensions/realtime-pod scaled
```

19. Now, let's check the current status of Pods:

```
kubectl get pods -n scheduleractivity
```

You should see a response that's something like this:

```
NAME                              READY   STATUS    RESTARTS   AGE
api-pod-c644d44b8-cvjz4           1/1     Running   0          91s
api-pod-c644d44b8-nhk99           1/1     Running   0          2m18s
gui-pod-6c494b5888-54vxp          0/1     Pending   0          69s
gui-pod-6c494b5888-lzcbh          0/1     Pending   0          69s
realtime-pod-59d4c8b768-2q8lr     1/1     Running   0          70s
realtime-pod-59d4c8b768-89nwj     1/1     Running   0          69s
realtime-pod-59d4c8b768-bxfmx     1/1     Running   0          70s
realtime-pod-59d4c8b768-cprk8     1/1     Running   0          70s
realtime-pod-59d4c8b768-dgnvr     1/1     Running   0          8m33s
realtime-pod-59d4c8b768-h94h9     1/1     Running   0          70s
realtime-pod-59d4c8b768-ng7g8     1/1     Running   0          70s
realtime-pod-59d4c8b768-njpjs     1/1     Running   0          69s
realtime-pod-59d4c8b768-rfrtq     1/1     Running   0          70s
realtime-pod-59d4c8b768-z7rbd     1/1     Running   0          70s
```

Figure 17.27: Getting the list of Pods

As you can see in the preceding image, all API Pods are running. You may observe that for your case, some of the API or GUI Pods have moved to the **`Pending`** state. If you do not observe this on your machine, try increasing the number of real-time Pod replicas by a factor of 5.

20. Use the following command to increase the number of real-time Pod replicas to **`12`**:

```
kubectl scale deployment/realtime-pod --replicas=12 -n
scheduleractivity
```

Note that we are trying to demonstrate that as the cluster is limited by resources, some of the lower-priority Pods get evicted. Here, we are scaling the real-time Pods to **`12`**. You may need to experiment a bit to see how many Pods you need to hit the resource limits of your environment. You should see the following response:

```
deployment.extensions/realtime-pod scaled
```

21. Check whether the real-time Pods are running and the API and GUI Pods are in the **`Pending`** state:

```
kubectl get pods -n scheduleractivity
```

You should see the following list of Pods:

```
NAME                            READY   STATUS    RESTARTS   AGE
api-pod-c644d44b8-f5xq2         0/1     Pending   0          56s
api-pod-c644d44b8-wztg6         0/1     Pending   0          56s
gui-pod-6c494b5888-54vxp        0/1     Pending   0          4m2s
gui-pod-6c494b5888-lzcbh        0/1     Pending   0          4m2s
realtime-pod-59d4c8b768-2q8lr   1/1     Running   0          4m3s
realtime-pod-59d4c8b768-89nwj   1/1     Running   0          4m2s
realtime-pod-59d4c8b768-9vdpb   1/1     Running   0          56s
realtime-pod-59d4c8b768-bxfmx   1/1     Running   0          4m3s
realtime-pod-59d4c8b768-cprk8   1/1     Running   0          4m3s
realtime-pod-59d4c8b768-dgnvr   1/1     Running   0          11m
realtime-pod-59d4c8b768-dlznz   1/1     Running   0          56s
realtime-pod-59d4c8b768-h94h9   1/1     Running   0          4m3s
realtime-pod-59d4c8b768-ng7g8   1/1     Running   0          4m3s
realtime-pod-59d4c8b768-njpjs   1/1     Running   0          4m2s
realtime-pod-59d4c8b768-rfrtq   1/1     Running   0          4m3s
realtime-pod-59d4c8b768-z7rbd   1/1     Running   0          4m3s
```

Figure 17.28: Getting the list of Pods

22. Bring the number of replicas of the real-time Pod back to **1**:

```
kubectl scale deployment/realtime-pod --replicas=1 -n
scheduleractivity
```

You should see the following response:

```
deployment.extensions/realtime-pod scaled
```

This should free up resources to schedule our GUI and API Pods.

23. Check whether the Pods are running:

```
kubectl get pods -n scheduleractivity
```

You should see the following list of Pods.

```
NAME                            READY   STATUS    RESTARTS   AGE
api-pod-c644d44b8-f5xq2         1/1     Running   0          2m16s
api-pod-c644d44b8-wztg6         1/1     Running   0          2m16s
gui-pod-6c494b5888-54vxp        1/1     Running   0          5m22s
gui-pod-6c494b5888-lzcbh        1/1     Running   0          5m22s
realtime-pod-59d4c8b768-dgnvr   1/1     Running   0          12m
```

Figure 17.29: Getting the list of Pods

CHAPTER 18: UPGRADING YOUR CLUSTER WITHOUT DOWNTIME

ACTIVITY 18.01: UPGRADING THE KUBERNETES PLATFORM FROM VERSION 1.15.7 TO 1.15.10

Solution:

In this activity, we will perform a full upgrade to the cluster, similar to the upgrade we performed through all the exercises in the chapter:

1. Check the version of kops and ensure that it is **`1.15`**:

```
kops version
```

You should get this response:

```
Version 1.15.0 (git-9992b4055)
```

2. Use the **`kops upgrade cluster`** command to see what update is available:

```
kops upgrade cluster ${NAME}
```

You should see a response similar to the following:

```
I0315 01:03:16.106957    3832 upgrade_cluster.go:216] Custom image (cos-cloud/cos
-stable-65-10323-99-0) has been provided for Instance Group "master-australia-sou
theast1-a"; not updating image
I0315 01:03:16.107009    3832 upgrade_cluster.go:216] Custom image (cos-cloud/cos
-stable-65-10323-99-0) has been provided for Instance Group "master-australia-sou
theast1-b"; not updating image
I0315 01:03:16.107026    3832 upgrade_cluster.go:216] Custom image (cos-cloud/cos
-stable-65-10323-99-0) has been provided for Instance Group "master-australia-sou
theast1-c"; not updating image
I0315 01:03:16.107047    3832 upgrade_cluster.go:216] Custom image (cos-cloud/cos
-stable-65-10323-99-0) has been provided for Instance Group "nodes"; not updating
 image
ITEM    PROPERTY                OLD     NEW
Cluster KubernetesVersion       1.15.7  1.15.10

Must specify --yes to perform upgrade
```

Figure 18.35: Checking the available upgrade for our cluster

You can see from the preceding screenshot that the **`OLD`** version is **`1.15.7`**, which is our current version, and an update is available to the **`NEW`** version (**`1.15.10`**), which is our target version.

3. Once you verify the changes from the command in the previous step, run the same command with the **`--yes`** flag:

```
kops upgrade cluster --yes
```

You should see a response similar to the following:

```
I0315 01:04:22.847381    3842 upgrade_cluster.go:216] Custom image (cos-cloud/cos
-stable-65-10323-99-0) has been provided for Instance Group "master-australia-sou
theast1-a"; not updating image
I0315 01:04:22.847411    3842 upgrade_cluster.go:216] Custom image (cos-cloud/cos
-stable-65-10323-99-0) has been provided for Instance Group "master-australia-sou
theast1-b"; not updating image
I0315 01:04:22.847422    3842 upgrade_cluster.go:216] Custom image (cos-cloud/cos
-stable-65-10323-99-0) has been provided for Instance Group "master-australia-sou
theast1-c"; not updating image
I0315 01:04:22.847440    3842 upgrade_cluster.go:216] Custom image (cos-cloud/cos
-stable-65-10323-99-0) has been provided for Instance Group "nodes"; not updating
 image
ITEM    PROPERTY                OLD     NEW
Cluster KubernetesVersion       1.15.7  1.15.10

Updates applied to configuration.
You can now apply these changes, using `kops update cluster myfirstcluster.k8s.lo
cal`
```

Figure 18.36: Updating the kops configuration of our cluster

4. This command will update the cloud and cluster resources as per the kops desired state set in the previous step:

```
kops update cluster ${NAME} --yes
```

You should see a response similar to this:

```
I0315 01:05:17.315372    3859 apply_cluster.go:556] Gossip DNS: skipping DNS vali
dation
W0315 01:05:17.380021    3859 external_access.go:36] TODO: Harmonize gcemodel Ext
ernalAccessModelBuilder with awsmodel
W0315 01:05:17.380071    3859 firewall.go:35] TODO: Harmonize gcemodel with awsmo
del for firewall - GCE model is way too open
W0315 01:05:17.380110    3859 firewall.go:64] Adding overlay network for X -> nod
e rule - HACK
W0315 01:05:17.380147    3859 firewall.go:118] Adding overlay network for X -> ma
ster rule - HACK
W0315 01:05:17.698190    3859 storageacl.go:71] we need to split master / node ro
les
W0315 01:05:17.698238    3859 storageacl.go:86] adding bucket level write ACL to
gs://faisal-kube to support etcd backup
W0315 01:05:17.698444    3859 autoscalinggroup.go:106] enabling storage-rw for et
cd backups
W0315 01:05:17.698551    3859 autoscalinggroup.go:106] enabling storage-rw for et
cd backups
ModelBuilder with awsmodel
W0315 01:05:17.380071    3859 firewall.go:35] TODO: Harmonize gcemodel with awsmodel for fir
```

Figure 18.37: Updating our infrastructure to meet the requirements of our upgrade

5. Get a list of the instance groups for your worker nodes. Note that the name of the instance group for our worker nodes is **`nodes`**:

```
kops get instancegroups
```

You should get an output similar to the following:

```
Using cluster from kubectl context: myfirstcluster.k8s.local

NAME                            ROLE    MACHINETYPE     MIN     MAX     ZONES
master-australia-southeast1-a   Master  n1-standard-1   1       1       australia-southeast1
-a
master-australia-southeast1-b   Master  n1-standard-1   1       1       australia-southeast1
-b
master-australia-southeast1-c   Master  n1-standard-1   1       1       australia-southeast1
-c
nodes                           Node    n1-standard-2   3       3       australia-southeast1
-a,australia-southeast1-b,australia-southeast1-c
```

Figure 18.38: Getting a list of the instance groups

6. Upgrade the first master node to the new version:

```
kops rolling-update cluster ${NAME} --instance-group master-
australia-southeast1-a --yes
```

Use the appropriate name of the instance group for your case. This command may take time based on your node configuration. Be patient and watch the logs to see whether there are any errors. After some time, you will see a successful message:

```
I0315 01:13:38.099246    3885 instancegroups.go:275] Cluster did not pass validation,
will try again in "30s" until duration "15m0s" expires: machine "https://www.googleapi
s.com/compute/beta/projects/kube-test-258704/zones/australia-southeast1-a/instances/ma
ster-australia-southeast1-a-q2pw" has not yet joined cluster.
I0315 01:14:04.521005    3885 gce_cloud.go:273] Scanning zones: [australia-southeast1-
b australia-southeast1-c australia-southeast1-a]
I0315 01:14:08.347291    3885 instancegroups.go:278] Cluster validated.
I0315 01:14:08.347343    3885 rollingupdate.go:184] Rolling update completed for clust
er "myfirstcluster.k8s.local"!
```

Figure 18.39: Applying a rolling update to the instance group running the first master node

7. Verify that the node is upgraded to the target version, which is **`1.15.10`**, in our case:

```
kubectl get nodes
```

You should see a response similar to the following:

```
NAME                                STATUS   ROLES    AGE     VERSION
master-australia-southeast1-a-q2pw  Ready    master   2m54s   v1.15.10
master-australia-southeast1-b-4jll  Ready    master   18m     v1.15.7
master-australia-southeast1-c-0ndl  Ready    master   18m     v1.15.7
nodes-6htd                          Ready    node     18m     v1.15.7
nodes-7lx0                          Ready    node     18m     v1.15.7
nodes-wjth                          Ready    node     18m     v1.15.7
```

Figure 18.40: Checking the version of Kubernetes on the first master node

You can see that the first master node is on version **`1.15.10`**.

8. Apply the previous two steps to all the master nodes:

9. Verify that the nodes are ready:

```
kubectl get nodes
```

You should see a response similar to the following:

```
NAME                                   STATUS   ROLES    AGE   VERSION
master-australia-southeast1-a-q2pw     Ready    master   40m   v1.15.10
master-australia-southeast1-b-4jll     Ready    master   26m   v1.15.10
master-australia-southeast1-c-0ndl     Ready    master   18m   v1.15.10
nodes-6htd                             Ready    node     55m   v1.15.7
nodes-7lx0                             Ready    node     55m   v1.15.7
nodes-wjth                             Ready    node     55m   v1.15.7
```

Figure 18.41: Checking the version of Kubernetes on all the master nodes

10. Now, we need to upgrade the worker nodes. To keep things simple, we only have one instance group for the worker nodes, named **`nodes`**. Run **`kops rolling-update`** for the **`nodes`** instance group without the **`--yes`** flag:

```
kops rolling-update cluster ${NAME} --node-interval 3m --instance-
group nodes --post-drain-delay 3m --logtostderr --v 1
```

You should see the following response:

```
I0315 03:04:31.740608    5159 gce_cloud.go:273] Scanning zones: [australia-southeast1-b australia-southeas
t1-c australia-southeast1-a]
NAME    STATUS          NEEDUPDATE      READY   MIN     MAX     NODES
nodes   NeedsUpdate     1               0       1       1       1
nodes   NeedsUpdate     1               0       1       1       1
nodes   NeedsUpdate     1               0       1       1       1
```

Figure 18.42: Performing a dry run of the rolling update on the worker nodes

This will provide you with a summary of what will be updated with this command.

11. Now, run the upgrade by using the same command from the previous step but with a **`--yes`** flag. This tells kops to actually perform the upgrade:

```
kops rolling-update cluster ${NAME} --node-interval 3m --instance-
group nodes --post-drain-delay 3m --yes
```

After a while, you will see that the cluster upgrade is finished with a success message, as shown:

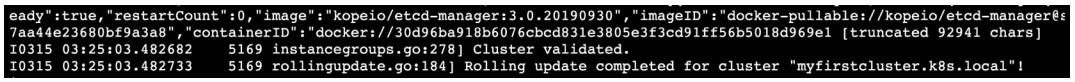

Figure 18.43: Applying the rolling update to our worker nodes

12. Verify that the worker node is updated to the target version, which is **`1.15.10`** for this activity:

```
kubectl get nodes
```

You should see a response as follows:

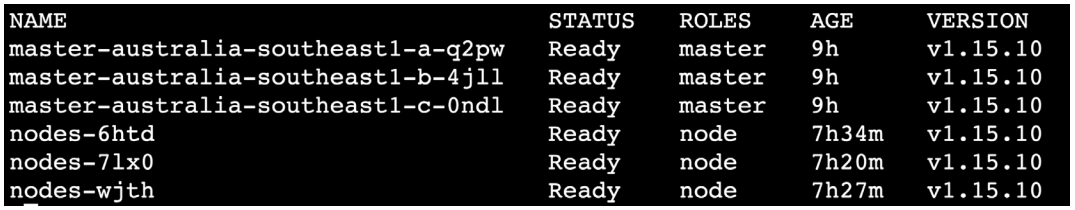

Figure 18.44: Checking the version of Kubernetes on all the nodes

As we can see in the preceding image, all of our master and worker nodes are running Kubernetes version **`1.15.10`**.

13. Verify that the pods are in a running state:

```
kubectl get pods -n upgrade-demo
```

You should see a response as follows:

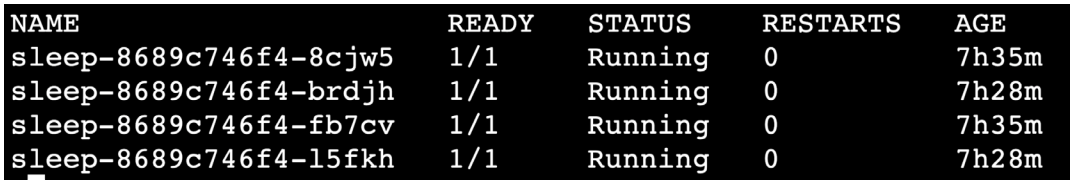

Figure 18.45: Checking whether all the worker pods are running successfully

Thus, all of our pods are running successfully.

CHAPTER 19: CUSTOM RESOURCE DEFINITIONS IN KUBERNETES

ACTIVITY 19.01: CRD AND CUSTOM CONTROLLER IN ACTION

Solution:

1. We will start with a clean slate. First, delete the existing namespace:

```
kubectl delete ns crddemo
```

You should see the following response:

```
namespace "crddemo" deleted
```

2. Create a new namespace called **`crddemo`**:

```
kubectl create ns crddemo
```

You should see the following response:

```
namespace/crddemo created
```

3. Delete the CRD if it exists from the previous exercise and you have not created a new namespace:

```
kubectl get crd
kubectl delete crd podlifecycleconfigs.controllers.kube.book.au
```

You should see the following response:

```
$kubectl get crd
NAME                                          CREATED AT
podlifecycleconfigs.controllers.kube.book.au   2019-08-31T09:23:06Z
$kubectl delete crd podlifecycleconfigs.controllers.kube.book.au
customresourcedefinition.apiextensions.k8s.io "podlifecycleconfigs.controllers.kube.book.au" deleted
```

Figure 19.11: Deleting the existing CRD

4. Create a file named **`pod-normaliser-crd.yaml`** with the following specs:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: podlifecycleconfigs.controllers.kube.book.au
spec:
  group: controllers.kube.book.au
  version: v1
  scope: Namespaced
  names:
```

```
        kind: PodLifecycleConfig
        plural: podlifecycleconfigs
        singular: podlifecycleconfig
      #1.15 preserveUnknownFields: false
      validation:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                namespaceName:
                  type: string
                podLiveForThisMinutes:
                  type: integer
```

5. Create a CRD using the file created in the previous step:

```
kubectl create -f pod-normaliser-crd.yaml
```

You should see the following response:

```
$kubectl create -f pod-normaliser-crd.yaml
customresourcedefinition.apiextensions.k8s.io/podlifecycleconfigs.controllers.kube.book.au created
```

Figure 19.12: Creating our CRD

6. Verify that it has been created:

```
kubectl get crd
```

You should see the following response:

```
$kubectl get crd
NAME                                              CREATED AT
podlifecycleconfigs.controllers.kube.book.au      2019-09-23T05:23:02Z
$
```

Figure 19.13: Verifying that our CRD was created

7. Now, we will create our CR. Create a file named **`pod-normaliser.yaml`**:

```
apiVersion: "controllers.kube.book.au/v1"
kind: PodLifecycleConfig
metadata:
  name: demo-pod-lifecycle
```

```
    # namespace: "crddemo"
  spec:
    namespaceName: crddemo
    podLiveForThisMinutes: 1
```

8. Create the CR defined in the previous step:

```
kubectl create -f pod-normaliser.yaml -n crddemo
```

You will see the following response:

```
$kubectl create -f pod-normaliser.yaml -n crddemo
podlifecycleconfig.controllers.kube.book.au/demo-pod-lifecycle created
$
```

Figure 19.14: Creating our CR

This CR will live in the **`crddemo`** namespace and will allow Pods to live for only a minute.

9. Create a custom Role that will have access to get all the CRs we have defined. Use the file named **`pod-normaliser-crd-roles.yaml`** using the following content:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: podlifecycleconfig-crd-view
  namespace: crddemo
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
    rbac.authorization.k8s.io/aggregate-to-view: "true"
rules:
- apiGroups: ["controllers.kube.book.au"]
  resources: ["podlifecycleconfigs"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list", "delete"]
```

Note the **`rules`** section, where we are defining what would be accessible to someone who has been granted this **`Role`**.

10. Use the definition from the previous step to create a Pod using the following command:

```
kubectl create -f pod-normaliser-crd-roles.yaml -n crddemo
```

It should give the following response:

```
$kubectl create -f pod-normaliser-crd-roles.yaml -n crddemo
role.rbac.authorization.k8s.io/podlifecycleconfig-crd-view created
```

Figure 19.15: Creating a custom Role to access our CRs

11. Once the role has been created, let's associate this role with a service account. We need to associate the newly created role with the **`default`** ServiceAccount from the **`crddemo`** namespace by creating a RoleBinding. Create a file named **`pod-normaliser-role-binding.yaml`** using the following content:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: podlifecycleconfig-crd-view-binding
  namespace: crddemo
subjects:
- kind: ServiceAccount
  name: default
  namespace: crddemo
roleRef:
  kind: Role
  name: podlifecycleconfig-crd-view
  apiGroup: ""
  # rbac.authorization.k8s.io
```

Note that under the **`subjects`** section, we are referencing the **`default`** ServiceAccount, and in the **`roleRef`** section, we are referencing the newly created role in the previous step.

12. Create a RoleBinding object using the definition from the previous step:

```
kubectl create -f pod-normaliser-role-binding.yaml -n crddemo
```

You should see the following response:

```
$kubectl create -f pod-normaliser-role-binding.yaml -n crddemo
rolebinding.rbac.authorization.k8s.io/podlifecycleconfig-crd-view-binding created
```

Figure 19.16: Creating our RoleBinding

> **NOTE**
>
> You may choose to skip the part where we build the Docker image. We have provided a working Docker image for the controller. You can use the image tagged as **`crd`** at this link: https://hub.docker.com/repository/docker/packtworkshops/the-kubernetes-workshop/tags.

13. Now, we need to build and deploy the custom controller. We have provided a **`Dockerfile`**, along with the required code in the repository that will build the container for you:

```
cd controller && docker build -t crd:0.0.1 .
```

    Don't miss the `.` (dot) character at the end of this command. This will give a long output, and upon successful completion, should end in a similar way to the following screenshot:

Figure 19.17: Building our Docker container

14. Tag and push the image to a public container registry of your choice. Make sure you replace the registry name as per your settings. Going with our choice of Docker repository, enter the following two commands, one after the other:

```
docker tag crd:0.0.1 <YOUR DOCKER REGISTRY>/crd:0.0.1

docker push <YOUR DOCKER REGISTRY>/crd:0.0.1
```

You should see the following response:

```
$docker tag crd:0.0.1 quay.io/masood_faisal/crd:0.0.1
$docker push quay.io/masood_faisal/crd:0.0.1
The push refers to repository [quay.io/masood_faisal/crd]
add865ea6aa3: Pushed
bbfc0444dbab: Layer already exists
951a82a6f24a: Layer already exists
[DEPRECATION NOTICE] registry v2 schema1 support will be removed in an upcoming release. Please contact admins of the quay.io reg
0.0.1: digest: sha256:cdb752ab23ff0baad1e06db8482007512ff7b93debc7f7f48e68adc1c1f6420e size: 4205
```

Figure 19.18: Pushing our Docker image

15. Now, let's define a Pod to run our custom controller. Create a file named **`crd-deploy.yaml`** using the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: crd-server
  labels:
    app: crd-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: crd-server
  template:
    metadata:
      labels:
        app: crd-server
    spec:
      containers:
      - name: server
        image: packtworkshops/the-kubernetes-workshop:crd
        env:
        - name: namespaceToWatch
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: podName
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        imagePullPolicy: Always
```

> **NOTE**
>
> If you are using a Docker image that you have created and pushed to your repository, then please replace the **`image`** field in the preceding Pod spec.

16. Deploy the custom controller to Kubernetes using the following command:

```
kubectl create -f crd-deploy.yaml -n crddemo
```

    You should see the following response:

```
deployment.apps/crd-server created
```

17. Make sure that the custom controller Pod is up and running:

```
kubectl get pods -n crddemo -w
```

    You should see the following list of Pods:

```
$kubectl get pods -n crddemo -w
NAME                          READY   STATUS              RESTARTS   AGE
crd-server-794644797d-9hcj7   0/1     ContainerCreating   0          13s
crd-server-794644797d-9hcj7   1/1     Running             0          14s
```

Figure 19.19: Verifying that our custom controller is online

    Press *Ctrl* + *C* to terminate the watch when the Pod is running.

18. Check the controller pod logs. Remember to use the Pod name from the output of the previous step:

```
kubectl logs -f crd-server-794644797d-9hcj7 -n crddemo
```

    You should see the following log:

```
2019/09/23 08:04:10 A Custom Resource has been Added
2019/09/23 08:04:10 The CRD is for namespace crddemo with pod active time as 1
```

Figure 19.20: Checking the logs on our custom controller Pod

    Notice that the log says **`A Custom Resource has been Added`**. If you refer to the **`AddFunc`** code mentioned in the **`types.go`** file, you will find that that the log statement was in the **`AddFunc`** section of the code. What this means is that the CR defined in *step 7* has been found by the controller.

    Remember that we are allowing Pods to run only for a minute. Let's test this by creating a simple Pod to see whether it gets killed.

19. Create the following pod specification, which is just a loop to keep the Pod working. Create a file named **`target-pod.yaml`** with the following definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: long-running-pod-example
spec:
  containers:
    - name: mutating-pod-example-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "while :; do echo '.'; sleep
        5 ; done" ]
```

20. Next, create a Pod using the definition from the previous step:

```
kubectl create -f target-pod.yaml -n crddemo
```

    You should see the following response:

```
pod/long-running-pod-example created
```

21. Now, let's watch the Pod for some time to see whether it is killed after a minute:

```
kubectl get pods -w -n crddemo
```

    You should see an output similar to the following:

```
$kubectl get pods -w -n crddemo
NAME                          READY   STATUS    RESTARTS   AGE
crd-server-77ffcff74b-wdk9j   1/1     Running   0          3m19s
long-running-pod-example      0/1     Pending   0          0s
long-running-pod-example      0/1     Pending   0          0s
long-running-pod-example      0/1     ContainerCreating   0          0s
long-running-pod-example      1/1     Running             0          4s
long-running-pod-example      1/1     Terminating         0          61s
long-running-pod-example      1/1     Terminating         0          61s
```

Figure 19.21: Observing our sample Pod

Remember that our CR set the value for the **`podLiveForThisMinutes`** field as **`1`**. You can see, in this screenshot, that **`long-running-pod`** is terminated automatically after approximately 1 minute.