Now, as we are within the `gke-node-1dhh` node, let's curl `localhost:32618` using the following command:

```
$ curl localhost:32618
Hi there! This page was last visited on 2023-06-26, 08:37:50.
```

And we get a response back! You can SSH into any node and curl the endpoint and should get a similar response.

To exit from the node and get back to the Cloud Shell prompt, run the following command:

```
$ exit
Connection to 35.202.82.74 closed.
```

And you are back at the Cloud Shell prompt.

> **Tip**
>
> A `NodePort Service` resource is an intermediate kind of resource. This means that while it forms an essential building block of providing external services, it is not used on its own most of the time. When you are running on the cloud, you can use `LoadBalancer Service` resources instead. Even for an on-premises setup, it makes sense not to use `NodePort` for every `Service` resource and instead use `Ingress` resources.

Now, let's look at the `LoadBalancer Service` resource used extensively to expose your Kubernetes workloads externally.

LoadBalancer Service resources

`LoadBalancer Service` resources help expose your pods on a single load-balanced endpoint. These `Service` resources can only be used within cloud platforms and platforms that provide Kubernetes controllers with access to spin up external network resources. A `LoadBalancer` Service practically spins up a `NodePort Service` resource and then requests the Cloud API to spin up a load balancer in front of the node ports. That way, it provides a single endpoint to access your `Service` resource from the external world.

Spinning up a `LoadBalancer Service` resource is simple—just set the type to `LoadBalancer`.

Let's expose the Flask application as a load balancer using the following manifest—`flask-loadbalancer.yaml`:

```
...
spec:
  type: LoadBalancer
...
```

Now, let's apply the manifest using the following command:

```
$ kubectl apply -f flask-loadbalancer.yaml
```

Let's get the `Service` resource to notice the changes using the following command:

```
$ kubectl get svc flask-app
NAME      TYPE         CLUSTER-IP   EXTERNAL-IP PORT(S)
flask-app LoadBalancer 10.3.240.246 34.71.95.96 5000:32618
```

The `Service` resource type is now `LoadBalancer`. As you can see, it now contains an external IP along with the cluster IP.

You can then curl on the external IP on port `5000` using the following command:

```
$ curl 34.71.95.96:5000
Hi there! This page was last visited on 2023-06-26, 08:37:50.
```

And you get the same response as before. Your `Service` resource is now running externally.

> **Tip**
>
> `LoadBalancer` `Service` resources tend to be expensive as every new resource spins up a network load balancer within your cloud provider. If you have HTTP-based workloads, use `Ingress` resources instead of `LoadBalancer` to save on resource costs and optimize traffic as they spin up an application load balancer instead.

While Kubernetes Services form the basic building block of exposing your container applications internally and externally, Kubernetes also provides `Ingress` resources for additional fine-grained control over traffic. Let's have a look at this in the next section.

Ingress resources

Imagine you have a beautiful front entrance to your restaurant where customers come in. They walk through this main entrance to reach different parts of your restaurant, such as the dining area or the bar. This entrance is like your "ingress."

In Kubernetes, an Ingress is like that front entrance. It helps manage external access to the Services inside your cluster. Instead of exposing each Service individually, you can use an Ingress to decide how people from the outside can reach different parts of your application.

In simple terms, a Kubernetes Service is like a central delivery point for your application's different parts, and an Ingress is like a front entrance that helps people from the outside find and access those parts easily.

`Ingress` resources act as reverse proxies into Kubernetes. You don't need a load balancer for every application you run within your estate, as load balancers normally forward traffic and don't require high levels of computing power. Therefore, spinning up a load balancer for everything does not make sense.

Therefore, Kubernetes provides a way of routing external traffic into your cluster via `Ingress` resources. These resources help you subdivide traffic according to multiple conditions. Some of these are set out here:

- Based on the URL path
- Based on the hostname
- A combination of the two

The following diagram illustrates how `Ingress` resources work:

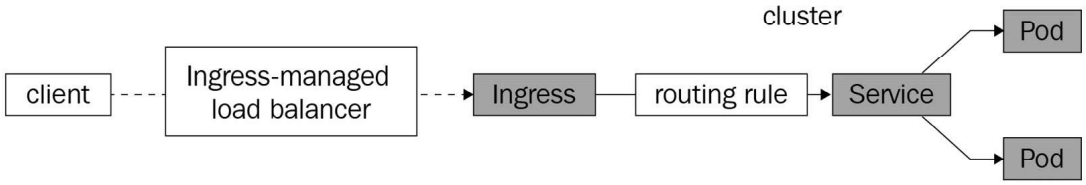

Figure 6.5 – Kubernetes Ingress resources

`Ingress` resources require an ingress controller to work. While most cloud providers have a controller installed, you must install an ingress controller on-premises or in a self-managed Kubernetes cluster. For more details on installing an ingress controller, refer to `https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/`. You can install more than one ingress controller, but you will have to annotate your manifests to denote explicitly which controller the `Ingress` resource should use.

For this chapter, we will use the **nginx ingress controller** (`https://github.com/kubernetes/ingress-nginx`), which is actively maintained by the Kubernetes open source community. The reason we use this instead of the native GKE ingress controller is that we want to make our setup as cloud-agnostic as possible. The nginx ingress controller is also feature-packed and runs the same irrespective of the environment. So, if we want to migrate to another cloud provider, we will retain all the features we had with `Ingress` resources before and do an exact like-for-like migration.

To understand how the nginx ingress controller works on GKE (or any other cloud), let's look at the following diagram:

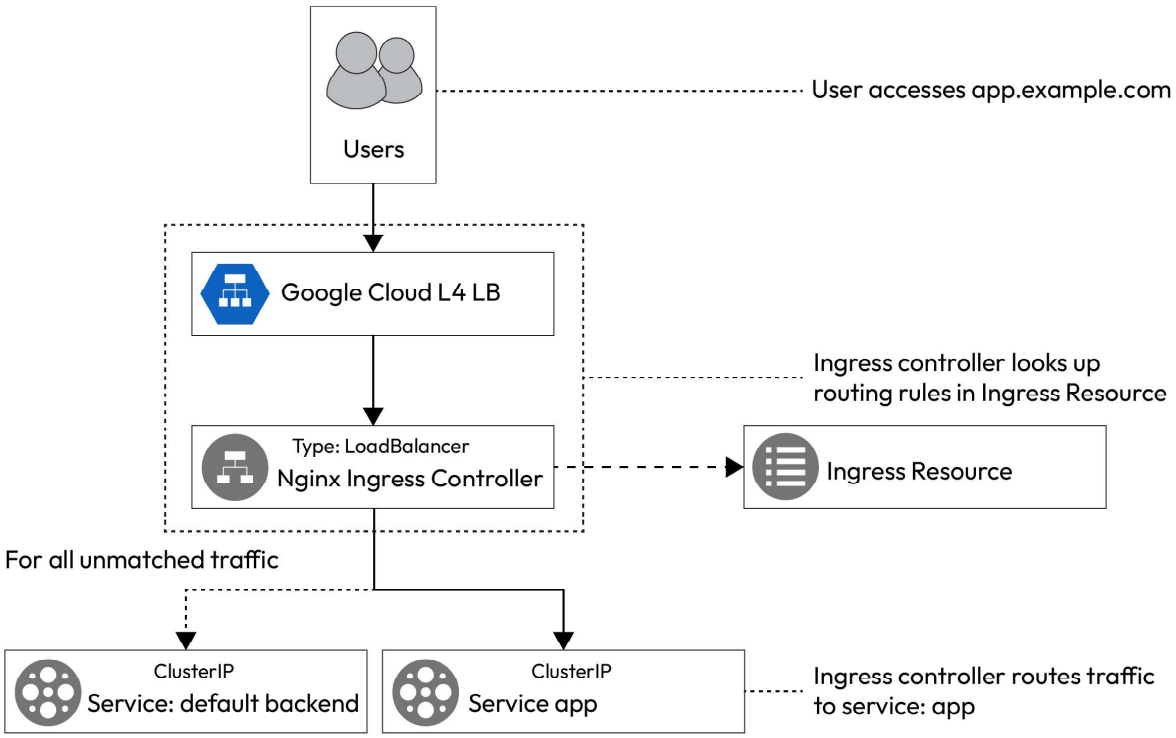

Figure 6.6 – nginx ingress controller on GKE

The client connects to the `Ingress` resource via an ingress-managed load balancer, and the traffic moves to the ingress controllers that act as the load balancer's backend. The ingress controllers then route the traffic to the correct `Service` resource via routing rules defined on the `Ingress` resource.

Now, let's go ahead and install the `nginx` ingress controller using the following command:

```
$ kubectl apply -f \
https://raw.githubusercontent.com/kubernetes/ingress-nginx\
/controller-v1.8.0/deploy/static/provider/cloud/deploy.yaml
```

This will boot up several resources under the `ingress-nginx` namespace. Most notable is the `ingress-nginx-controller` `Deployment`, which is exposed via the `ingress-nginx-controller` `LoadBalancer` `Service`.

Let's now expose the `flask-app` `Service` via an `Ingress` resource, but before we do that, we will have to expose the `flask-app` `Service` on a `ClusterIP` instead, so let's apply the relevant manifest using the following command:

```
$ kubectl apply -f flask-clusterip.yaml
```

The next step is to define an `Ingress` resource. Remember that as GKE is running on a public cloud, it has the ingress controllers installed and running. So, we can simply go and create an ingress manifest—`flask-basic-ingress.yaml`:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: flask-app
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  defaultBackend:
    service:
      name: flask-app
      port:
        number: 5000
```

This resource defines a default backend that passes all traffic to the `flask-app` pod, so it is counter-productive, but let's look at it for simplicity.

Apply the manifest using the following command:

```
$ kubectl apply -f flask-basic-ingress.yaml
```

Now, let's list the `Ingress` resources using the following command:

```
$ kubectl get ingress flask-app
NAME        CLASS    HOSTS   ADDRESS   PORTS   AGE
flask-app   <none>   *                 80      40s
```

We can see that the `flask-app` `Ingress` resource is now listed with `HOSTS *`. That means that this would listen on all hosts on all addresses. So, anything that does not match other Ingress rules will be routed here. As mentioned, we need the `nginx-ingress-controller` Service external IP address to invoke all Services exposed via Ingress. To get the external IP of the `nginx-ingress-controller` Service, run the following command:

```
$ kubectl get svc ingress-nginx-controller -n ingress-nginx
NAME                       TYPE         EXTERNAL-IP
ingress-nginx-controller LoadBalancer 34.120.27.34
```

We see an external IP allocated to it, which we will use further.

> **Important note**
>
> Remember that Ingress rules take a while to propagate across the cluster, so if you receive an error initially when you curl the endpoint, wait for 5 minutes, and you should get the response back.

Let's curl this IP and see what we get using the following command:

```
$ curl 34.120.27.34
Hi there! This page was last visited on 2023-06-26, 09:28:26.
```

Now, let's clean up the `Ingress` resource using the following command:

```
$ kubectl delete ingress flask-app
```

The simple Ingress rule is counterproductive as it routes all traffic to a single `Service` resource. The idea of Ingress is to use a single load balancer to route traffic to multiple targets. Let's look at two ways to do this—**path-based** and **name-based** routing.

Path-based routing

Let's consider an application with two versions, `v1` and `v2`, and you want both to co-exist on a single endpoint. You can use **path-based routing** for such a scenario.

Let's create the two application versions first using the imperative method by running the following commands:

```
$ kubectl run nginx-v1 --image=bharamicrosystems/nginx:v1
$ kubectl run nginx-v2 --image=bharamicrosystems/nginx:v2
```

Now, expose the two pods as `ClusterIP Service` resources using the following commands:

```
$ kubectl expose pod nginx-v1 --port=80
$ kubectl expose pod nginx-v2 --port=80
```

We will then create an `Ingress` resource using the following manifest file, `nginx-app-path-ingress.yaml`, which will expose two endpoints—`<external-ip>/v1`, which routes to the `v1 Service` resource, and `<external-ip>/v2`, which routes to the `v2 Service` resource:

```
...
spec:
  rules:
  - http:
      paths:
      - path: /v1
        pathType: Prefix
        backend:
          service:
            name: nginx-v1
            port:
              number: 80
      - path: /v2
        pathType: Prefix
        backend:
          service:
            name: nginx-v2
```

```
            port:
              number: 80
```

The Ingress manifest contains several rules. The `http` rule has two paths—`/v1` and `/v2`, having the `pathType` value set to `Prefix`. Therefore, any traffic arriving on a URL that starts with `/v1` is routed to the `nginx-v1 Service` resource on port `80`, and traffic arriving on `/v2` is routed to the `nginx-v2 Service` resource on port `80`.

Let's apply the manifest by using the following command:

```
$ kubectl apply -f nginx-app-path-ingress.yaml
```

Now, let's list the `Ingress` resources by running the following command:

```
$ kubectl get ingress nginx-app -w
NAME        CLASS    HOSTS   ADDRESS          PORTS   AGE
nginx-app   <none>   *       34.120.27.34     80      114s
```

Now that we have the external IP, we can `curl` both endpoints to see what we get using the following commands:

```
$ curl 34.120.27.34/v1/
This is version 1
$ curl 34.120.27.34/v2/
This is version 2
```

Sometimes, a path-based route is not always feasible, as you might not want your users to remember the path of multiple applications. However, you can still run multiple applications using a single Ingress endpoint—that is, by using **name-based routing**.

Name-based routing

Name-based or **FQDN-based routing** relies on the `host` header we pass while making an HTTP request. The `Ingress` resource can route based on the header. For example, if we want to access the `v1 Service` resource, we can use `v1.example.com`, and for the `v2 Service` resource, we can use the `v2.example.com` URL.

Let's now have a look at the `nginx-app-host-ingress.yaml` manifest to understand this concept further:

```
...
spec:
  rules:
  - host: v1.example.com
    http:
      paths:
      - path: "/"
        pathType: Prefix
        backend:
```

```
          service:
            name: nginx-v1
            port:
              number: 80
  - host: v2.example.com
    http:
      paths:
      - path: "/"
        pathType: Prefix
        backend:
          service:
            name: nginx-v2
            port:
              number: 80
```

The manifest now contains multiple hosts—`v1.example.com` routing to `nginx-v1`, and `v2.example.com` routing to `nginx-v2`.

Now, let's apply this manifest and get the Ingress using the following commands:

```
$ kubectl apply -f nginx-app-host-ingress.yaml
$ kubectl get ingress
NAME       HOSTS                          ADDRESS       PORTS
nginx-app  v1.example.com,v2.example.com  34.120.27.34  80
```

This time, we can see that two hosts are defined, `v1.example.com` and `v2.example.com`, running on the same address. Before we hit those endpoints, we need to make an entry on the `/etc/hosts` file to allow our machine to resolve `v1.example.com` and `v2.example.com` to the Ingress address.

Edit the `/etc/hosts` file and add the following entry at the end:

```
<Ingress_External_IP> v1.example.com v2.example.com
```

Now, let's `curl` both endpoints and see what we get:

```
$ curl v1.example.com
This is version 1
$ curl v2.example.com
This is version 2
```

And, as we can see, the name-based routing is working correctly! You can create a more dynamic setup by combining multiple hosts and path-based routing.

`Service`, `Ingress`, `Pod`, `Deployment`, and `ReplicaSet` resources help us to maintain a set number of replicas within Kubernetes and help to serve them under a single endpoint. As you may have noticed, they are linked together using a combination of `labels` and `matchLabels` attributes. The following diagram will help you visualize this:

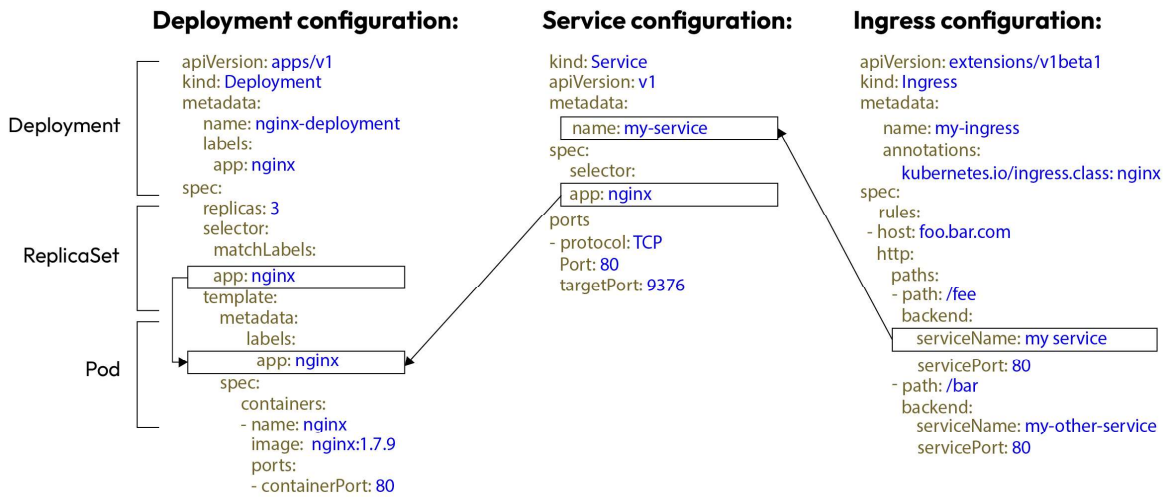

Figure 6.7 – Linking Deployment, Service, and Ingress

Till now, we have been scaling our pods manually, but a better way would be to autoscale the replicas based on resource utilization and traffic. Kubernetes provides a resource called `HorizontalPodAutoscaler` to handle that requirement.

Horizontal Pod autoscaling

Imagine you're the manager of a snack bar at a park. On a sunny day, lots of people come to enjoy the park, and they all want snacks. Now, you have a few workers at your snack bar who make and serve the snacks.

Horizontal Pod autoscaling in Kubernetes is like having magical helpers who adjust the number of snack makers (pods) based on how many people want snacks (traffic).

Here's how it works:

- **Average days**: You might only need one or two snack makers on regular days with fewer people. In Kubernetes terms, you have a few pods running your application.
- **Busy days**: But when it's a sunny weekend, and everyone rushes to the park, more people want snacks. Your magical helpers (Horizontal Pod autoscaling) notice the increase in demand. They say, "*We need more snack makers!*" So, more snack makers (pods) are added automatically to handle the rush.
- **Scaling down**: Once the sun sets and the crowd leaves, you don't need as many snack makers anymore. Your magical helpers see the decrease in demand and say, "*We can have fewer snack makers now.*" So, extra snack makers (pods) are removed, saving resources.

- **Automatic adjustment**: These magical helpers monitor the crowd and adjust the number of snack makers (pods) in real time. When the demand goes up, they deploy more. When it goes down, they remove some.

In the same way, Kubernetes Horizontal pod autoscaling watches how busy your application is. If there's more traffic (more people wanting your app), it automatically adds more pods. If things quieten down, it scales down the number of pods. This helps your app handle varied traffic without you manually doing everything.

So, Horizontal pod autoscaling is like having magical assistants that ensure your application has the correct number of workers (pods) to handle the crowd (traffic) efficiently.

`HorizontalPodAutoscaler` is a Kubernetes resource that helps you to update replicas within your `ReplicaSet` resources based on defined factors, the most common being CPU and memory.

To understand this better, let's create an `nginx` Deployment, and this time, we will set the resource limits within the pod. Resource limits are a vital element that enables `HorizontalPodAutoscaler` resources to function. It relies on the percentage utilization of the limits to decide when to spin up a new replica. We will use the following `nginx-autoscale-deployment.yaml` manifest under `~/modern-devops/ch6/deployments` for this exercise:

```
...
spec:
  replicas: 1
  template:
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          limits:
            cpu: 200m
            memory: 200Mi
...
```

Use the following command to perform a new deployment:

```
$ kubectl apply -f nginx-autoscale-deployment.yaml
```

Let's expose this deployment with a `LoadBalancer` `Service` resource and get the external IP:

```
$ kubectl expose deployment nginx --port 80 --type LoadBalancer
$ kubectl get svc nginx
NAME  TYPE         CLUSTER-IP   EXTERNAL-IP   PORT(S)
nginx LoadBalancer 10.3.243.225 34.123.234.57 80:30099/TCP
```

Now, let's autoscale this deployment. The `Deployment` resource needs at least 1 pod replica and can have a maximum of 5 pod replicas while maintaining an average CPU utilization of `25%`. Use the following command to create a `HorizontalPodAutoscaler` resource:

```
$ kubectl autoscale deployment nginx --cpu-percent=25 --min=1 --max=5
```

Now that we have the `HorizontalPodAutoscaler` resource created, we can load test the application using the `hey` load testing utility preinstalled in Google Cloud Shell. But before you fire the load test, open a duplicate shell session and watch the `Deployment` resource using the following command:

```
$ kubectl get deployment nginx -w
```

Open another duplicate shell session and watch the `HorizontalPodAutoscaler` resource using the following command:

```
$ kubectl get hpa nginx -w
```

Now, in the original window, run the following command to fire a load test:

```
$ hey -z 120s -c 100 http://34.123.234.57
```

It will start a load test for 2 minutes, with 10 concurrent users continuously hammering the `Service`. You will see the following output if you open the window where you're watching the `HorizontalPodAutoscaler` resource. As soon as we start firing the load test, the average utilization reaches `46%`. The `HorizontalPodAutoscaler` resource waits for some time, then it increases the replicas, first to `2`, then to `4`, and finally to `5`. When the test is complete, the utilization drops quickly to `27%`, `25%`, and finally, `0%`. When the utilization goes to `0%`, the `HorizontalPodAutoscaler` resource spins down the replicas from `5` to `1` gradually:

```
$ kubectl get hpa nginx -w
NAME    REFERENCE          TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
nginx   deployment/nginx   <unknown>/25%   1         5         1          32s
nginx   deployment/nginx   46%/25%         1         5         1          71s
nginx   deployment/nginx   46%/25%         1         5         2          92s
nginx   deployment/nginx   92%/25%         1         5         4          2m2s
nginx   deployment/nginx   66%/25%         1         5         5          2m32s
nginx   deployment/nginx   57%/25%         1         5         5          2m41s
nginx   deployment/nginx   27%/25%         1         5         5          3m11s
nginx   deployment/nginx   23%/25%         1         5         5          3m41s
nginx   deployment/nginx   0%/25%          1         5         4          4m23s
nginx   deployment/nginx   0%/25%          1         5         2          5m53s
nginx   deployment/nginx   0%/25%          1         5         1          6m30s
```

Likewise, we will see the replicas of the `Deployment` changing when the `HorizontalPodAutoscaler` resource actions the changes:

```
$ kubectl get deployment nginx -w
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   1/1     1            1           18s
nginx   1/2     1            1           77s
nginx   2/2     2            2           79s
nginx   2/4     2            2           107s
nginx   3/4     4            3           108s
nginx   4/4     4            4           109s
nginx   4/5     4            4           2m17s
nginx   5/5     5            5           2m19s
nginx   4/4     4            4           4m23s
nginx   2/2     2            2           5m53s
nginx   1/1     1            1           6m30s
```

Besides CPU and memory, you can use other parameters to scale your workloads, such as network traffic. You can also use external metrics such as latency and other factors that you can use to decide when to scale your traffic.

> **Tip**
>
> While you should use the `HorizontalPodAutoscaler` resource with CPU and memory, you should also consider scaling on external metrics such as response time and network latency. That will ensure better reliability as they directly impact customer experience and are crucial to your business.

Till now, we have been dealing with stateless workloads. However, pragmatically speaking, some applications need to save the state. Let's look at some considerations for managing stateful applications.

Managing stateful applications

Imagine you're a librarian in a magical library. You have a bunch of enchanted books that store valuable knowledge. Each book has a unique story and is kept in a specific spot on the shelf. These books are like your "stateful applications," and managing them requires extra care.

Managing stateful applications in the world of technology is like taking care of these magical books in your library.

Here's how it works:

- **Stateful books**: Some books in your library are "stateful." This means they hold vital information that changes over time, such as bookmarks or notes from readers.
- **Fixed locations**: Just as each book has a specific place on the shelf, stateful applications must also be in particular locations. They might need to be on certain machines or use specific storage to keep their data safe.
- **Maintaining inventory**: You must remember where each book is placed. Similarly, managing stateful applications means remembering their exact locations and configurations.
- **Careful handling**: When someone borrows a stateful book, you must ensure they return it in good condition. With stateful applications, you must handle updates and changes carefully to avoid losing important data.
- **Backup spells**: Sometimes, you cast a spell to create a copy of a book, just in case something happens to the original. With stateful applications, you back up your data to restore it if anything goes wrong.
- **Moving with caution**: If you need to rearrange the library, you move books one at a time so that nothing gets lost. Similarly, with stateful applications, if you need to move them between machines or storage, it's done cautiously to avoid data loss.

In the world of technology, managing stateful applications means taking extra care of applications that hold important data. You ensure they're placed in the right spots, handle updates carefully, and create backups to keep valuable information safe, just like how you protect your enchanted books in the magical library!

`Deployment` resources are beneficial for stateless workloads, as they do not need to add any state considerations while updating `ReplicaSet` resources, but they cannot work effectively with stateful workloads. To manage such workloads, you can use a `StatefulSet` resource.

StatefulSet resources

`StatefulSet` resources help manage stateful applications. They are similar to `Deployment` resources, but unlike a `Deployment` resource, they also keep track of state and require `Volume` and `Service` resources to operate. `StatefulSet` resources maintain a sticky identity for each pod. This means that the volume mounted on one pod cannot be used by the other. In a `StatefulSet` resource, Kubernetes orders pods by numbering them instead of generating a random hash. Pods within a `StatefulSet` resource are also rolled out and scaled-in in order. If a particular pod goes down and is recreated, the same volume is mounted to the pod.

The following diagram illustrates a `StatefulSet` resource:

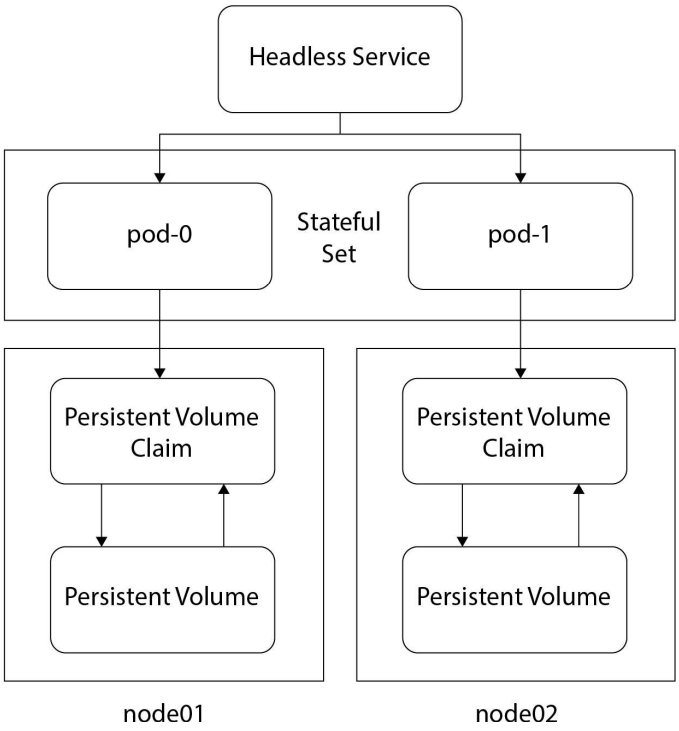

Figure 6.8 – StatefulSet resource

A `StatefulSet` resource has a stable and unique network identifier, therefore, it requires a headless `Service` resource. Headless Services are `Service` resources that do not have a cluster IP. Instead, the Kubernetes DNS resolves the `Service` resource's FQDN directly to the pods.

As a `StatefulSet` resource is supposed to persist data, it requires Persistent Volumes to operate. Therefore, let's look at how to manage volumes using Kubernetes.

Managing Persistent Volumes

Persistent Volumes are Kubernetes resources that deal with storage. They can help you manage and mount **hard disks**, **SSDs**, **filestores**, and other block and network storage entities. You can provision Persistent Volumes manually or use dynamic provisioning within Kubernetes. When you use dynamic provisioning, Kubernetes will request the cloud provider via the cloud controller manager to provide the required storage. Let's look at both methods to understand how they work.

Static provisioning

Static provisioning is the traditional method of provisioning volumes. It requires someone (typically an administrator) to manually provision a disk and create a `PersistentVolume` resource using the disk information. The developer can then use the `PersistentVolume` resource within their `StatefulSet` resource, as in the following diagram:

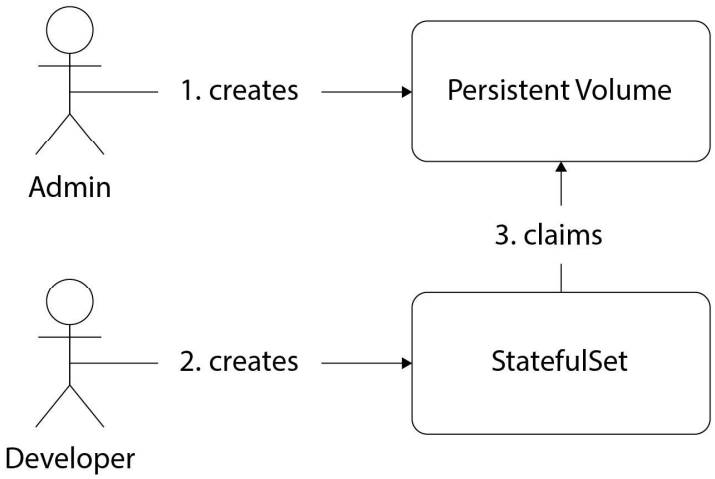

Figure 6.9 – Static provisioning

Let's now look at a static provisioning example.

To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch6/statefulsets/
```

So, we first need to create a disk within the cloud platform. Since we're using Google Cloud, let's proceed and use the `gcloud` commands to do so.

Use the following command to create a persistent zonal disk. Ensure that you use the same zone as your Kubernetes cluster. As we are using the `us-central1-a` zone for the Kubernetes cluster, we will use the same in the following command:

```
$ gcloud compute disks create nginx-manual \
--size 50GB --type pd-ssd --zone us-central1-a
Created [https://www.googleapis.com/compute/v1/projects/<project_id>/zones/us-central1-a/
disks/nginx-manual].
NAME          ZONE           SIZE_GB  TYPE    STATUS
nginx-manual  us-central1-a  50       pd-ssd  READY
```

As the disk is now ready, we can then create a `PersistentVolume` resource from it.

The manifest file, `nginx-manual-pv.yaml`, looks like this:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nginx-manual-pv
  labels:
    usage: nginx-manual-disk
spec:
  capacity:
    storage: 50G
  accessModes:
    - ReadWriteOnce
  gcePersistentDisk:
    pdName: nginx-manual
    fsType: ext4
```

The `spec` section contains `capacity`, `accessModes`, and the kind of disk it needs to provision. You can specify one or more access modes to a PersistentVolumes:

- `ReadWriteOnce`: Only one pod can read and write to the disk at a time; therefore, you cannot mount such a volume to multiple pods
- `ReadOnlyMany`: Multiple pods can read from the same volume simultaneously, but no pod can write to the disk
- `ReadWriteMany`: Multiple pods can read and write to the same volume at once

> **Tip**
>
> Not all kinds of storage support all access modes. You need to decide the volume type during the initial requirement analysis and architectural assessment phase.

OK—let's now go and apply the manifest to provision the `PersistentVolume` resource using the following command:

```
$ kubectl apply -f nginx-manual-pv.yaml
```

Let's now check whether the Persistent Volume is available by using the following command:

```
$ kubectl get pv
NAME             CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS
nginx-manual-pv  50G       RWO           Retain          Available
```

As the Persistent Volume is now available, we must create a headless `Service` resource to help maintain network identity in the `StatefulSet` resource. The following `nginx-manual-service.yaml` manifest describes it:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-manual
  labels:
    app: nginx-manual
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx-manual
```

It is very similar to the regular `Service` resource, except that we have specified `clusterIP` as `None`.

Now, let's go and apply the manifest using the following command:

```
$ kubectl apply -f nginx-manual-service.yaml
```

As the `Service` resource is created, we can create a `StatefulSet` resource that uses the created `PersistentVolume` and `Service` resources. The `StatefulSet` resource manifest, `nginx-manual-statefulset.yaml`, looks like this:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx-manual
spec:
  selector:
    matchLabels:
      app: nginx-manual
  serviceName: "nginx-manual"
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx-manual
    spec:
      containers:
      - name: nginx
        image: nginx
        volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
```

```
      name: html
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 40Gi
    selector:
      matchLabels:
        usage: nginx-manual-disk
```

The manifest contains various sections. While most are similar to the `Deployment` resource manifest, this requires a `volume` definition and a separate `volumeClaimTemplates` section. The `volumeClaimTemplates` section consists of the `accessModes`, `resources`, and `selector` sections. The `selector` section defines the `matchLabels` attribute, which helps to select a particular `PersistentVolume` resource. In this case, it selects the `PersistentVolume` resource we defined previously. It also contains the `serviceName` attribute that defines the headless `Service` resource it will use.

Now, let's go ahead and apply the manifest using the following command:

```
$ kubectl apply -f nginx-manual-statefulset.yaml
```

Now, let's inspect a few elements to see where we are. The `StatefulSet` resource creates a `PersistentVolumeClaim` resource to claim the `PersistentVolume` resource we created before.

Get the `PersistentVolumeClaim` resource using the following command:

```
$ kubectl get pvc
NAME                 STATUS VOLUME           CAPACITY  ACCESS MODES
html-nginx-manual-0  Bound  nginx-manual-pv  50G       RWO
```

As we can see, the `StatefulSet` resource has created a `PersistentVolumeClaim` resource called `html-nginx-manual-0` that is bound to the `nginx-manual-pv` `PersistentVolume` resource. Therefore, manual provisioning has worked correctly.

If we query the `PersistentVolume` resource using the following command, we will see that the status is now showing as `Bound`:

```
$ kubectl get pv
NAME             CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS
nginx-manual-pv  50G       RWO           Retain          Bound
```

Now, let's have a look at the pods using the following command:

```
$ kubectl get pod
NAME             READY   STATUS    RESTARTS   AGE
nginx-manual-0   1/1     Running   0          14s
```

As we see, the `StatefulSet` resource has created a pod and appended it with a serial number instead of a random hash. It wants to maintain ordering between the pods and mount the same volumes to the pods they previously mounted.

Now, let's open a shell into the pod and create a file within the `/usr/share/nginx/html` directory using the following commands:

```
$ kubectl exec -it nginx-manual-0 -- /bin/bash
root@nginx-manual-0:/# cd /usr/share/nginx/html/
root@nginx-manual-0:/usr/share/nginx/html# echo 'Hello, world' > index.html
root@nginx-manual-0:/usr/share/nginx/html# exit
```

Great! So, let's go ahead and delete the pod and see whether we can get the file in the same location again using the following commands:

```
$ kubectl delete pod nginx-manual-0
$ kubectl get pod
NAME             READY   STATUS              RESTARTS   AGE
nginx-manual-0   1/1     Running             0          3s
$ kubectl exec -it nginx-manual-0 -- /bin/bash
root@nginx-manual-0:/# cd /usr/share/nginx/html/ && cat index.html
Hello, world
root@nginx-manual-0:/usr/share/nginx/html# exit
```

And, as we can see, the file still exists, even after we deleted the pod.

Static provisioning isn't one of the best ways of doing things, as you must manually keep track and provision volumes. That involves a lot of manual activities and may be error-prone. Some organizations that want to keep a line between Dev and Ops can use this technique. Kubernetes allows this provision. However, for more DevOps-friendly organizations, **dynamic provisioning** is a better way of doing it.

Dynamic provisioning

Dynamic provisioning is when Kubernetes provides storage resources for you by interacting with the cloud provider. When we provisioned the disk manually, we interacted with the cloud APIs using the `gcloud` command line. What if your organization decides to move to some other cloud provider later? That would break many existing scripts, and you would have to rewrite the storage provisioning steps. Kubernetes is inherently portable and platform-independent. You can provision resources in the same way on any cloud platform.

But then, different cloud providers have different storage offerings. How would Kubernetes know what kind of storage it needs to provision? Well, Kubernetes uses `StorageClass` resources for that. `StorageClass` resources are Kubernetes resources that define the type of storage they need to provide when someone uses it.

The following diagram illustrates dynamic provisioning:

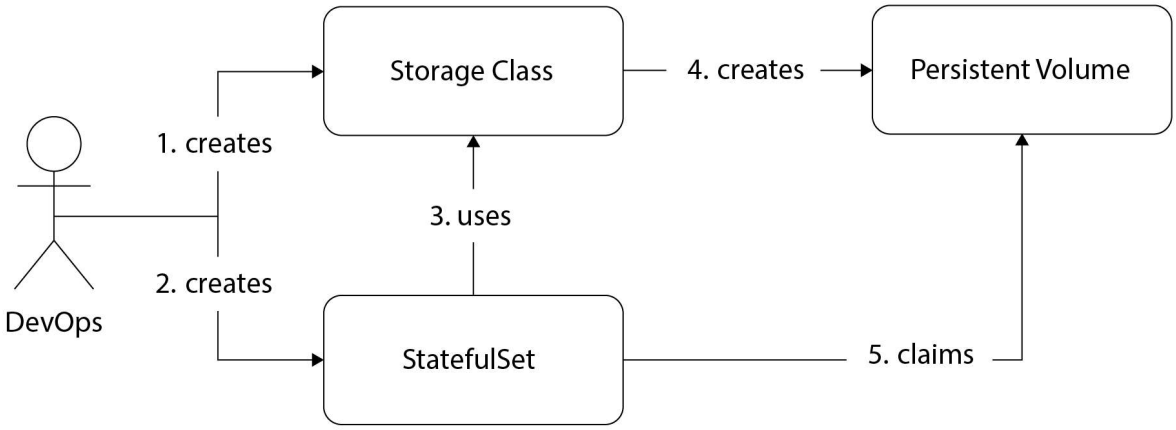

Figure 6.10 – Dynamic provisioning

Let's see an example storage class manifest, `fast-storage-class.yaml`, that provisions an SSD within GCP:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

The `StorageClass` resource contains a provisioner and any parameters the provisioner requires. You may have noticed that I have kept the name `fast` instead of `gce-ssd` or similar. That is because we want to keep the names as generic as possible.

> **Tip**
>
> Keep generic storage class names such as `fast`, `standard`, `block`, and `shared`, and avoid names specific to the cloud platform. Because storage class names are used in Persistent Volume claims, if you migrate to another cloud provider, you may end up changing a lot of manifests just to avoid confusion.

Let's go ahead and apply the manifest using the following command:

```
$ kubectl apply -f fast-storage-class.yaml
```

As the `StorageClass` resource is created, let's use it to provision an `nginx` `StatefulSet` resource dynamically.

We need to create a `Service` resource manifest, `nginx-dynamic-service.yaml`, first:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-dynamic
  labels:
    app: nginx-dynamic
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx-dynamic
```

The manifest is very similar to the manual `Service` resource. Let's go ahead and apply it using the following command:

```
$ kubectl apply -f nginx-dynamic-service.yaml
```

Now, let's look at the `StatefulSet` resource manifest, `nginx-dynamic-statefulset.yaml`:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx-dynamic
spec:
...
  serviceName: "nginx-dynamic"
  template:
    spec:
      containers:
      - name: nginx
        image: nginx
        volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
...
  volumeClaimTemplates:
  - metadata:
      name: html
    spec:
      storageClassName: "fast"
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 40Gi
```

The manifest is similar to the manual one, but this one contains the `storageClassName` attribute in the `volumeClaimTemplates` section and lacks the `selector` section, as we are dynamically provisioning the storage. Use the following command to apply the manifest:

```
$ kubectl apply -f nginx-dynamic-statefulset.yaml
```

As the `StatefulSet` resource is created, let's go ahead and check the `PersistentVolumeClaim` and `PersistentVolume` resources using the following commands:

```
$ kubectl get pvc
NAME                   STATUS  VOLUME    CAPACITY  ACCESS MODES  STORAGECLASS
html-nginx-dynamic-0   Bound   pvc-6b78  40Gi      RWO           fast
$ kubectl get pv
NAME      CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  CLAIM
pvc-6b78  40Gi      RWO           Delete          Bound   default/html-nginx-dynamic-0
```

And we can see that the claim is bound to a Persistent Volume that is dynamically provisioned. Now, let's proceed and run the following command to do similar tests with this `StatefulSet` resource.

Let's create a file in the `nginx-dynamic-0` pod using the following command:

```
$ kubectl exec -it nginx-dynamic-0 -- bash
root@nginx-dynamic-0:/# cd /usr/share/nginx/html/
root@nginx-dynamic-0:/usr/share/nginx/html# echo 'Hello, dynamic world' > index.html
root@nginx-dynamic-0:/usr/share/nginx/html# exit
```

Now, delete the pod and open a shell session again to check whether the file exists by using the following commands:

```
$ kubectl delete pod nginx-dynamic-0
$ kubectl get pod nginx-dynamic-0
NAME              READY   STATUS    RESTARTS   AGE
nginx-dynamic-0   1/1     Running   0          13s
$ kubectl exec -it nginx-dynamic-0 -- bash
root@nginx-dynamic-0:/# cd /usr/share/nginx/html/
root@nginx-dynamic-0:/usr/share/nginx/html# cat index.html
Hello, dynamic world
root@nginx-dynamic-0:/usr/share/nginx/html# exit
```

And as we can see, the file exists in the volume, even if the pod was deleted. That is dynamic provisioning in action for you!

You will have observed that we have used the `kubectl` command multiple times throughout this chapter. When you perform activities throughout the day, using shortcuts and best practices wherever you can makes sense. Let's look at some best practices while using `kubectl`.

Kubernetes command-line best practices, tips, and tricks

For seasoned Kubernetes developers and administrators, `kubectl` is a command they run most of the time. The following steps will simplify your life, save you a ton of time, let you focus on more essential activities, and set you apart from the rest.

Using aliases

Most system administrators use aliases for an excellent reason—they save valuable time. Aliases in Linux are different names for commands, and they are mostly used to shorten the most frequently used commands; for example, `ls -l` becomes `ll`.

You can use the following aliases with `kubectl` to make your life easier.

k for kubectl

Yes—that's right. By using the following alias, you can use `k` instead of typing `kubectl`:

```
$ alias k='kubectl'
$ k get node
NAME                 STATUS   ROLES    AGE     VERSION
kind-control-plane   Ready    master   5m7s    v1.26.1
kind-worker          Ready    <none>   4m33s   v1.26.1
```

That will save a lot of time and hassle.

Using kubectl --dry-run

`kubectl --dry-run` helps you to generate YAML manifests from imperative commands and saves you a lot of typing time. You can write an imperative command to generate a resource and append that with a `--dry-run=client -o yaml` string to generate a YAML manifest from the imperative command. The command does not create the resource within the cluster, but instead just outputs the manifest. The following command will generate a `Pod` manifest using `--dry-run`:

```
$ kubectl run nginx --image=nginx --dry-run=client -o yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
  dnsPolicy: ClusterFirst
```

```
    restartPolicy: Always
  status: {}
```

And you now have the skeleton YAML file that you can edit according to your liking.

Now, imagine typing this command multiple times during the day! At some point, it becomes tiring. Why not shorten it by using the following alias?

```
$ alias kdr='kubectl --dry-run=client -o yaml'
```

You can then use the alias to generate other manifests.

To generate a `Deployment` resource manifest, use the following command:

```
$ kdr create deployment nginx --image=nginx
```

You can use the dry run to generate almost all resources from imperative commands. However, some resources do not have an imperative command, such as a `DaemonSet` resource. You can generate a manifest for the closest resource and modify it for such resources. A `DaemonSet` manifest is very similar to a `Deployment` manifest, so you can generate a `Deployment` manifest and change it to match the `DameonSet` manifest.

Now, let's look at some of the most frequently used `kubectl` commands and their possible aliases.

kubectl apply and delete aliases

If you use manifests, you will use the `kubectl apply` and `kubectl delete` commands most of the time within your cluster, so it makes sense to use the following aliases:

```
$ alias kap='kubectl apply -f'
$ alias kad='kubectl delete -f'
```

You can then use them to apply or delete resources using the following commands:

```
$ kap nginx-deployment.yaml
$ kad nginx-deployment.yaml
```

While troubleshooting containers, most of us use `busybox`. Let's see how to optimize it.

Troubleshooting containers with busybox using an alias

We use the following commands to open a `busybox` session:

```
$ kubectl run busybox-test --image=busybox -it --rm --restart=Never -- <cmd>
```

Now, opening several `busybox` sessions during the day can be tiring. How about minimizing the overhead by using the following alias?

```
$ alias kbb='kubectl run busybox-test --image=busybox -it --rm --restart=Never --'
```

We can then open a shell session to a new `busybox` pod using the following command:

```
$ kbb sh
/ #
```

Now, that is much cleaner and easier. Likewise, you can also create aliases of other commands that you use frequently. Here's an example:

```
$ alias kgp='kubectl get pods'
$ alias kgn='kubectl get nodes'
$ alias kgs='kubectl get svc'
$ alias kdb='kubectl describe'
$ alias kl='kubectl logs'
$ alias ke='kubectl exec -it'
```

And so on, according to your needs. You may also be used to autocompletion within `bash`, where your commands autocomplete when you press *Tab* after typing a few words. `kubectl` also provides autocompletion of commands, but not by default. Let's now look at how to enable `kubectl` autocompletion within `bash`.

Using kubectl bash autocompletion

To enable `kubectl bash` autocompletion, use the following command:

```
$ echo "source <(kubectl completion bash)" >> ~/.bashrc
```

The command adds the `kubectl` completion `bash` command as a source to your `.bashrc` file. So, the next time you log in to your shell, you should be able to use `kubectl` autocomplete. That will save you a ton of time when typing commands.

Summary

We began this chapter by managing pods with `Deployment` and `ReplicaSet` resources and discussed some critical Kubernetes deployment strategies. We then looked into Kubernetes service discovery and models and understood why we required a separate entity to expose containers to the internal or external world. We then looked at different `Service` resources and where to use them. We talked about `Ingress` resources and how to use them to create reverse proxies for our container workloads. We then delved into Horizontal Pod autoscaling and used multiple metrics to scale our pods automatically.

We looked at state considerations and learned about static and dynamic storage provisioning using `PersistentVolume`, `PersistentVolumeClaim`, and `StorageClass` resources, and talked about some best practices surrounding them. We looked at `StatefulSet` resources as essential resources that help you schedule and manage stateful containers. Finally, we looked at some best practices, tips, and tricks surrounding the `kubectl` command line and how to use them effectively.

The topics covered in this and the previous chapter are just the core of Kubernetes. Kubernetes is a vast tool with enough functionality to write an entire book, so these chapters only give you the gist of what it is all about. Please feel free to read about the resources in detail in the Kubernetes official documentation at `https://kubernetes.io`.

In the next chapter, we will delve into the world of the cloud and look at **Container-as-a-Service** (**CaaS**) and serverless offerings for containers.

Questions

1. A Kubernetes Deployment deletes an old `ReplicaSet` resource when the image is updated. (True/False)
2. What are the primary deployment strategies supported by Kubernetes? (Choose two)

   A. Recreate

   B. Rolling update

   C. Ramped slow rollout

   D. Best-effort controlled rollout
3. Which types of resources can you use to expose containers externally? (Choose three)

   A. `ClusterIP Service`

   B. `NodePort Service`

   C. `LoadBalancer Service`

   D. `Ingress`
4. It is a best practice to start with a `ClusterIP` Service and change the Service type later if needed. (True/False)
5. `Deployment` resources are suitable for stateful workloads. (True/False)
6. Which kinds of workloads can you run with Ingresses?

   A. HTTP

   B. TCP

   C. FTP

   D. SMTP

7. Which resources would you define for dynamic volume provisioning? (Choose two)

   A. `StorageClass`

   B. `PersistentVolumeClaim`

   C. `PersistentVolume`

   D. `StatefulSet`

8. To make your horizontal scaling more meaningful, what parameters should you use to scale your pods? (Choose three)

   A. CPU

   B. Memory

   C. External metrics, such as response time

   D. Packets per second (PPS)

9. What are the forms of routing within an `Ingress` resource? (Choose two)

   A. Simple

   B. Path-based

   C. Name-based

   D. Complex

Answers

1. False. An image Deployment just scales the old `ReplicaSet` resource to `0`.
2. A and B
3. B, C, and D
4. True
5. False. Use `StatefulSet` resources instead.
6. A
7. A and B
8. A, B, and C
9. B and C

7

Containers as a Service (CaaS) and Serverless Computing for Containers

In the last two chapters, we covered Kubernetes and how it helps manage your containers seamlessly. Now, let's look at other ways of automating and managing container deployments—**Containers as a Service** (**CaaS**) and **serverless computing for containers**. CaaS provides container-based virtualization that abstracts away all management behind the scenes and helps you manage your containers without worrying about the underlying infrastructure and orchestration.

For simple deployments and less complex applications, CaaS can be a savior. Serverless computing is a broad term that encompasses applications that can be run without us having to worry about the infrastructure behind the scenes. It has an additional benefit that you can focus purely on the application. We will discuss CaaS technologies such as **Amazon Elastic Container Service** (**Amazon ECS**) with **Amazon Web Services Fargate** (**AWS Fargate**) in detail and briefly discuss other cloud-based CaaS offerings such as **Azure Kubernetes Services** (**AKS**), **Google Kubernetes Engine** (**GKE**), and **Google Cloud Run**. We will then delve into the popular open source serverless CaaS solution known as **Knative**.

In this chapter, we're going to cover the following main topics:

- The need for serverless offerings
- Amazon ECS with **Elastic Compute Cloud** (**EC2**) and Fargate
- Other CaaS services
- Open source CaaS with Knative

Technical requirements

You will need an active AWS subscription for this chapter's exercises. AWS is the market's most popular, feature-rich cloud platform. Currently, AWS is offering a free tier for some products. You can sign up for this at `https://aws.amazon.com/free`. This chapter uses some paid services, but we will try to minimize how many we use as much as possible during the exercises.

You will also need to clone the following GitHub repository for some of the exercises:

`https://github.com/PacktPublishing/Modern-DevOps-Practices-2e`

Run the following command to clone the repository into your home directory. Then, `cd` into the `ch7` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
  modern-devops
$ cd modern-devops/ch7
```

As the repository contains files with placeholder strings, you must replace the `<your_dockerhub_user>` string with your actual Docker Hub user. Use the following commands to substitute the placeholders:

```
$ find ./ -type f -exec sed -i -e \
's/<your_dockerhub_user>/<your actual docker hub user>/g' {} \;
```

So, let's get started!

The need for serverless offerings

Numerous organizations, so far, have been focusing a lot on infrastructure provisioning and management. They optimize the number of resources, machines, and infrastructure surrounding the applications they build. However, they should focus on what they do best—software development. Unless your organization wants to invest heavily in an expensive infrastructure team to do a lot of heavy lifting behind the scenes, you'd be better off concentrating on writing and building quality applications rather than focusing on where and how to run and optimize them.

Serverless offerings come as a reprieve for this problem. Instead of concentrating on how to host your infrastructure to run your applications, you can declare what you want to run, and the serverless offering manages it for you. This has become a boon for small enterprises that do not have the budget to invest heavily in infrastructure and want to get started quickly without wasting too much time standing up and maintaining infrastructure to run applications.

Serverless offerings also offer automatic placement and scaling for container and application workloads. You can spin from 0 to 100 instances in minutes, if not seconds. The best part is that you pay for what you use in some services rather than what you allocate.

This chapter will concentrate on a very popular AWS container management offering called **ECS** and AWS's container serverless offering, **AWS Fargate**. We will then briefly examine offerings from other cloud platforms and, finally, the open source container-based serverless solution known as **Knative**.

Now, let's go ahead and look at Amazon ECS.

Amazon ECS with EC2 and Fargate

Amazon ECS is a container orchestration platform that AWS offers. It is simple to use and manage, uses Docker behind the scenes, and can deploy your workloads to **Amazon EC2**, a **virtual machine** (**VM**)-based solution, or **AWS Fargate**, a serverless offering.

It is a highly scalable solution that deploys containers in seconds. It makes hosting, running, stopping, and starting your containers easy. Just as Kubernetes offers **pods**, ECS offers **tasks** that help you run your container workloads. A task can contain one or more containers grouped according to a logical relationship. You can also group one or more tasks into **services**. Services are similar to Kubernetes controllers, which manage tasks and can ensure that the required number of replicas of your tasks are running in the right place at the right time. ECS uses simple API calls to provide many functionalities, such as creating, updating, reading, and deleting tasks and services.

ECS also allows you to place your containers according to multiple placement strategies while keeping **high availability** (**HA**) and resource optimization in mind. You can tweak the placement algorithm according to your priority—cost, availability, or a mix of both. So, you can use ECS to run one-time batch workloads or long-running microservices, all using a simple-to-use API interface.

ECS architecture

Before we explore the ECS architecture, it is important to understand some common AWS terminologies to follow it. Let's look at some AWS resources:

- **AWS Regions**: An AWS Region is a geographical region where AWS provides its services. It is normally a city or a metropolitan region but can sometimes span multiple cities. It comprises multiple **Availability Zones** (**AZs**). Some examples of AWS Regions are `us-east-1`, `us-west-1`, `ap-southeast-1`, `eu-central-1`, and so on.
- **AWS AZs**: AWS AZs are data centers within an AWS Region connected with low-latency, high-bandwidth networks. Most resources run within AZs. Examples of AZs are `us-east-1a`, `us-east-1b`, and so on.
- **AWS virtual private cloud (VPC)**: An AWS VPC is an isolated network resource you create within AWS. You associate a dedicated private IP address range to it from which the rest of your resources, such as EC2 instances, can derive their IP addresses. An AWS VPC spans an AWS Region.
- **Subnet**: A subnet, as the name suggests, is a subnetwork within the VPC. You must subdivide the IP address ranges you provided to the VPC and associate them with subnets. Resources normally reside within subnets, and each subnet spans an AZ.

- **Route table**: An AWS route table routes traffic within the VPC subnets and to the internet. Every AWS subnet is associated with a route table through **subnet route table associations**.
- **Internet gateways**: An internet gateway allows connection to and from the internet to your AWS subnets.
- **Identity Access Management (IAM)**: AWS IAM helps you control access to resources by users and other AWS resources. They help you implement **role-based access control** (**RBAC**) and the **principle of least privilege** (**PoLP**).
- **Amazon EC2**: EC2 allows you to spin up VMs within subnets, also known as instances.
- **AWS Auto Scaling groups (ASGs)**: An AWS ASG works with Amazon EC2 to provide HA and scalability to your instances. It monitors your EC2 instances and ensures that a defined number of healthy instances are always running. It also takes care of autoscaling your instances with increasing load in your machines to allow for handling more traffic. It uses the **instance profile** and **launch configuration** to decide on the properties of new EC2 instances it spins up.
- **Amazon CloudWatch**: Amazon CloudWatch is a monitoring and observability service. It allows you to collect, track, and monitor metrics, log files, and set alarms to take automated actions on specific conditions. CloudWatch helps understand application performance, health, and resource utilization.

ECS is a cloud-based regional service. When you spin up an ECS cluster, the instances span multiple AZs, where you can schedule your tasks and services using simple manifests. ECS manifests are very similar to `docker-compose` YAML manifests, where we specify which tasks to run and which tasks comprise a service.

You can run ECS within an existing VPC. We can schedule tasks in either Amazon EC2 or AWS Fargate.

Your ECS cluster can have one or more EC2 instances attached to it. You also have the option to attach an existing EC2 instance to a cluster by installing the ECS node agent within your EC2 instance. The agent sends information about your containers' state and tasks to the ECS scheduler. It then interacts with the container runtime to schedule containers within the node. They are similar to `kubelet` in the Kubernetes ecosystem. If you run your containers within EC2 instances, you pay for the number of EC2 instances you allocate to the cluster.

If you plan to use Fargate, the infrastructure is wholly abstracted from you, and you must specify the amount of CPU and memory your container is set to consume. You pay for the CPU and memory your container consumes rather than the resources you allocate to the machines.

> **Tip**
>
> Although you only pay for the resources you consume in Fargate, it is more expensive than running your tasks on EC2, especially when running long-running services such as a web server. A rule of thumb is to run long-running online tasks within EC2 and batch tasks with Fargate. That will give you the best cost optimization.

When we schedule a task, AWS spins up the container on a managed EC2 or Fargate server by pulling the required container image from a **container registry**. Every **task** has an **elastic network interface** (**ENI**) attached to it. Multiple tasks are grouped as a **service**, and the service ensures that all the required tasks run at once.

Amazon ECS uses a **task scheduler** to schedule containers on your cluster. It places your containers in an appropriate node of your cluster based on placement logic, availability, and cost requirements. The scheduler also ensures that the desired number of tasks run on the node at a given time.

The following diagram explains the ECS cluster architecture beautifully:

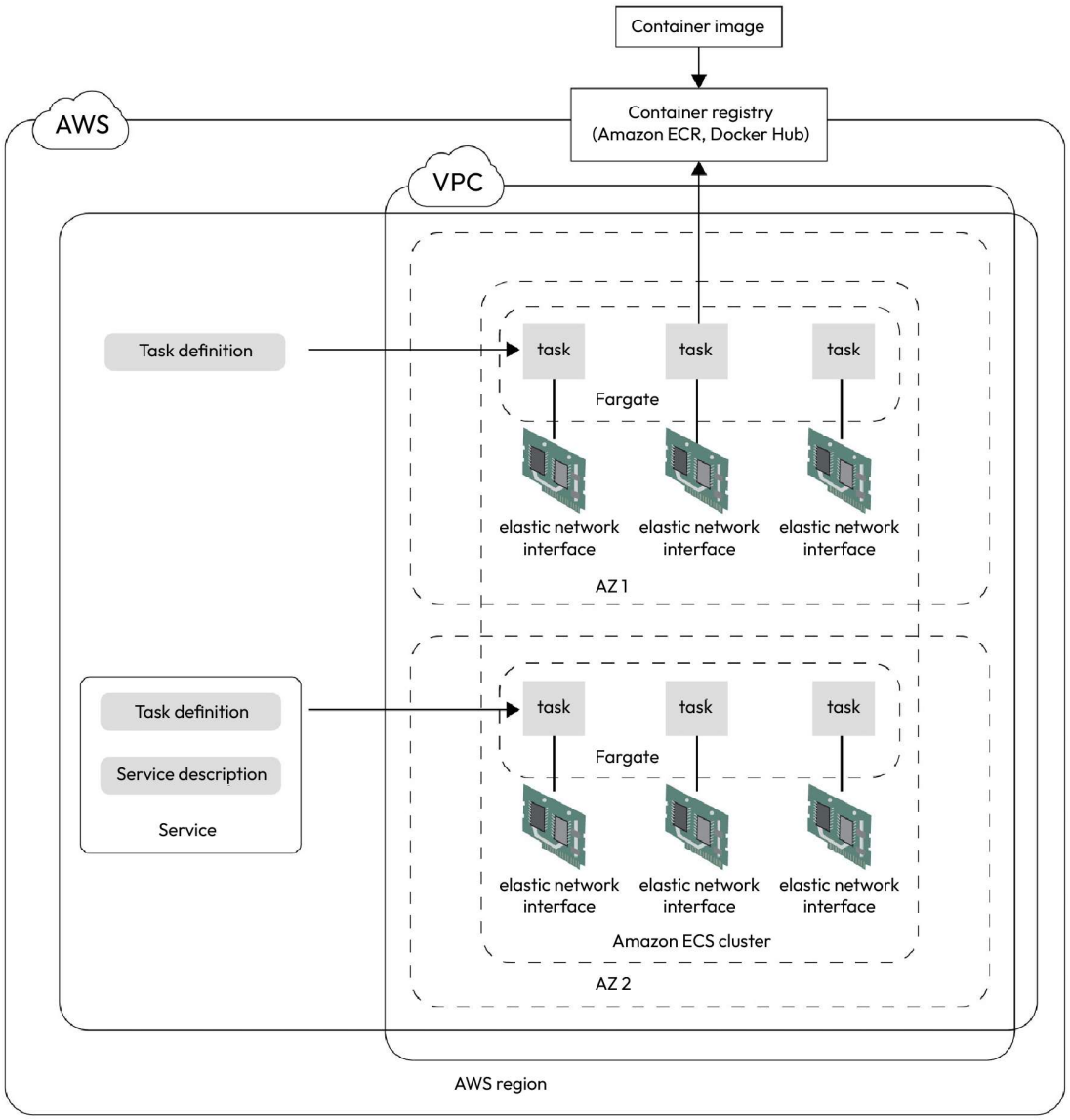

Figure 7.1 – ECS architecture

Amazon provides the ECS **command-line interface** (**CLI**) for interacting with the ECS cluster. It is a simple command-line tool that you can use to administer an ECS cluster and create and manage tasks and services on the ECS cluster.

Now, let's go ahead and install the ECS CLI.

Installing the AWS and ECS CLIs

The AWS CLI is available as a `deb` package within the public `apt` repositories. To install it, run the following commands:

```
$ sudo apt update && sudo apt install awscli -y
$ aws --version
aws-cli/1.22.34 Python/3.10.6 Linux/5.19.0-1028-aws botocore/1.23.34
```

Installing the ECS CLI in the Linux ecosystem is simple. We need to download the binary and move to the system path using the following command:

```
$ sudo curl -Lo /usr/local/bin/ecs-cli \
https://amazon-ecs-cli.s3.amazonaws.com/ecs-cli-linux-amd64-latest
$ sudo chmod +x /usr/local/bin/ecs-cli
```

Run the following command to check whether `ecs-cli` has been installed correctly:

```
$ ecs-cli --version
ecs-cli version 1.21.0 (bb0b8f0)
```

As we can see, `ecs-cli` has been successfully installed on our system.

The next step is to allow `ecs-cli` to connect with your AWS API. You need to export your AWS CLI environment variables for this. Run the following commands to do so:

```
$ export AWS_SECRET_ACCESS_KEY=...
$ export AWS_ACCESS_KEY_ID=...
$ export AWS_DEFAULT_REGION=...
```

Once we've set the environment variables, `ecs-cli` will use them to authenticate with the AWS API. In the next section, we'll spin up an ECS cluster using the ECS CLI.

Spinning up an ECS cluster

We can use the ECS CLI commands to spin up an ECS cluster. You can run your containers in EC2 and Fargate, so first, we will create a cluster that runs EC2 instances. Then, we will add Fargate tasks within the cluster.

To connect with your EC2 instances, you need to generate a key pair within AWS. To do so, run the following command:

```
$ aws ec2 create-key-pair --key-name ecs-keypair
```

The output of this command will provide the key pair in a JSON file. Extract the JSON file's key material and save that in a separate file called `ecs-keypair.pem`. Remember to replace the `\n` characters with a new line when you save the file.

Once we've generated the key pair, we can use the following command to create an ECS cluster using the ECS CLI:

```
$ ecs-cli up --keypair ecs-keypair --instance-type t2.micro \
--size 2 --cluster cluster-1 --capability-iam
INFO[0002] Using recommended Amazon Linux 2 AMI with ECS Agent 1.72.0 and Docker version
20.10.23
INFO[0003] Created cluster cluster=cluster-1 region=us-east-1
INFO[0004] Waiting for your cluster resources to be created...
INFO[0130] Cloudformation stack status stackStatus=CREATE_IN_PROGRESS
VPC created: vpc-0448321d209bf75e2
Security Group created: sg-0e30839477f1c9881
Subnet created: subnet-02200afa6716866fa
Subnet created: subnet-099582f6b0d04e419
Cluster creation succeeded.
```

When we issue this command, in the background, AWS spins up a stack of resources using CloudFormation. CloudFormation is AWS's **Infrastructure-as-Code** (**IaC**) solution that helps you deploy infrastructure on AWS through reusable templates. The CloudFormation template consists of several resources such as a VPC, a security group, a subnet within the VPC, a route table, a route, a subnet route table association, an internet gateway, an IAM role, an instance profile, a launch configuration, an ASG, a VPC gateway attachment, and the cluster itself. The ASG contains two EC2 instances running and serving the cluster. Keep a copy of the output; we will need the details later during the exercises.

Now that our cluster is up, we will spin up our first task.

Creating task definitions

ECS tasks are similar to Kubernetes pods. They are the basic building blocks of ECS and comprise one or more related containers. Task definitions are the blueprints for ECS tasks and define what the ECS task should look like. They are very similar to `docker-compose` files and are written in YAML format. ECS also uses all versions of `docker-compose` to allow us to define tasks. They help you define containers and their images, resource requirements, where they should run (EC2 or Fargate), volume and port mappings, and other networking requirements.

> **Tip**
>
> Using the `docker-compose` manifest to spin up tasks and services is a great idea, as it will help you align your configuration with an open standard.

A task is a finite process and only runs once. Even if it's a long-running process, such as a web server, the task still runs once as it waits for the long-running process to end (which runs indefinitely in theory). The task's life cycle follows the **Pending** -> **Running** -> **Stopped** states. So, when you schedule your task, the task enters the **Pending** state, attempting to pull the image from the container registry. Then, it tries to start the container. Once the container has started, it enters the **Running** state. When the container has completed executing or errored out, it ends up in the **Stopped** state. A container with startup errors directly transitions from the **Pending** state to the **Stopped** state.

Now, let's go ahead and deploy an `nginx` web server task within the ECS cluster we just created.

To access the resources for this section, `cd` into the following directory:

```
$ cd ~/modern-devops/ch7/ECS/tasks/EC2/
```

We'll use `docker-compose` task definitions here. So, let's start by defining the following `docker-compose.yml` file:

```
version: '3'
services:
  web:
    image: nginx
    ports:
      - "80:80"
    logging:
      driver: awslogs
      options:
        awslogs-group: /aws/webserver
        awslogs-region: us-east-1
        awslogs-stream-prefix: ecs
```

The YAML file defines a `web` container with an `nginx` image with host port `80` mapped to container port `80`. It uses the `awslogs` logging driver, which streams logs into Amazon CloudWatch. It will stream the logs to the `/aws/webserver` log group in the `us-east-1` region with the `ecs` stream prefix.

The task definition also includes the resource definition—that is, the amount of resources we want to reserve for our task. Therefore, we will have to define the following `ecs-params.yaml` file:

```
version: 1
task_definition:
  services:
    web:
      cpu_shares: 100
      mem_limit: 524288000
```

This YAML file defines `cpu_shares` in millicores and `mem_limit` in bytes for the container we plan to fire. Now, let's look at scheduling this task as an EC2 task.

Scheduling EC2 tasks on ECS

Let's use `ecs-cli` to apply the configuration and schedule our task using the following command:

```
$ ecs-cli compose up --create-log-groups --cluster cluster-1 --launch-type EC2
```

Now that the task has been scheduled and the container is running, let's list all the tasks to get the container's details and find out where it is running. To do so, run the following command:

```
$ ecs-cli ps --cluster cluster-1
Name                    State    Ports                TaskDefinition
cluster-1/fee1cf28/web  RUNNING  34.237.218.7:80->80  EC2:1
```

As we can see, the web container is running on `cluster-1` on `34.237.218.7:80`. Now, use the following command to curl this endpoint to see what we get:

```
$ curl 34.237.218.7:80
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

Here, we get the default `nginx` home page! We've successfully scheduled a container on ECS using the EC2 launch type. You might want to duplicate this task to handle more traffic. This is known as horizontal scaling. We'll see how in the next section.

Scaling tasks

We can easily scale tasks using `ecs-cli`. Use the following command to scale the tasks to `2`:

```
$ ecs-cli compose scale 2 --cluster cluster-1 --launch-type EC2
```

Now, use the following command to check whether two containers are running on the cluster:

```
$ ecs-cli ps --cluster cluster-1
Name                    State    Ports                  TaskDefinition
cluster-1/b43bdec7/web  RUNNING  54.90.208.183:80->80   EC2:1
cluster-1/fee1cf28/web  RUNNING  34.237.218.7:80->80    EC2:1
```

As we can see, two containers are running on the cluster. Now, let's query CloudWatch to get the logs of the containers.

Querying container logs from CloudWatch

To query logs from CloudWatch, we must list the log streams using the following command:

```
$ aws logs describe-log-streams --log-group-name /aws/webserver \
--log-stream-name-prefix ecs | grep logStreamName
  "logStreamName": "ecs/web/b43bdec7",
  "logStreamName": "ecs/web/fee1cf28",
```

As we can see, there are two log streams for this—one for each task. `logStreamName` follows the convention `<log_stream_prefix>/<task_name>/<task_id>`. So, to get the logs for `ecs/b43bdec7/web`, run the following command:

```
$ aws logs get-log-events --log-group-name/aws/webserver \
--log-stream ecs/web/b43bdec7
```

Here, you will see a stream of logs in JSON format in the response. Now, let's look at how we can stop running tasks.

Stopping tasks

`ecs-cli` uses the friendly `docker-compose` syntax for everything. Use the following command to stop the tasks in the cluster:

```
$ ecs-cli compose down --cluster cluster-1
```

Let's list the containers to see whether the tasks have stopped by using the following command:

```
$ ecs-cli ps --cluster cluster-1
INFO[0001] Stopping container... container=cluster-1/b43bdec7/web
INFO[0001] Stopping container... container=cluster-1/fee1cf28/web
INFO[0008] Stopped container... container=cluster-1/b43bdec7/web
desiredStatus=STOPPED lastStatus=STOPPED taskDefinition="EC2:1"
INFO[0008] Stopped container... container=cluster-1/fee1cf28/web
desiredStatus=STOPPED lastStatus=STOPPED taskDefinition="EC2:1"
```

As we can see, both containers have stopped.

Running tasks on EC2 is not a serverless way of doing things. You still have to provision and manage the EC2 instances, and although ECS manages workloads on the cluster, you still have to pay for the amount of resources you've provisioned in the form of EC2 instances. AWS offers Fargate as a serverless solution where you pay per resource consumption. Let's look at how we can create the same task as a Fargate task.

Scheduling Fargate tasks on ECS

Scheduling tasks on Fargate is very similar to EC2. Here, we need to specify the `launch-type` value as `FARGATE`.

To schedule the same task on Fargate, run the following command:

```
$ ecs-cli compose up --create-log-groups --cluster cluster-1 --launch-type FARGATE
FATA[0001] ClientException: Fargate only supports network mode 'awsvpc'.
```

Oops! We have a problem! Well, it's complaining about the network type. For a Fargate task, we must supply the network type as `awsvpc` instead of the default bridge network. The `awsvpc` network is an overlay network that implements the **Container Network Interface** (**CNI**). To understand more about Docker networking, please refer to *Chapter 1*, *The Modern Way of DevOps*. For now, let's go ahead and configure the `awsvpc` network type. But before that, the Fargate task requires a few configurations.

To access the resources for this section, `cd` into the following directory:

```
$ cd ~/modern-devops/ch7/ECS/tasks/FARGATE/
```

First, we'll have to assume a task execution role for the ECS agent to authenticate with the AWS API and interact with Fargate.

To do so, create the following `task-execution-assume-role.json` file:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "ecs-tasks.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Then, use the following command to assume the task execution role:

```
$ aws iam --region us-east-1 create-role --role-name ecsTaskExecutionRole \
 --assume-role-policy-document file://task-execution-assume-role.json
```

ECS provides a default role policy called `AmazonECSTaskExecutionRolePolicy`, which contains various permissions that help you interact with CloudWatch and **Elastic Container Registry** (**ECR**). The following JSON code outlines the permission that the policy has:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```

```
                "ecr:GetAuthorizationToken",
                "ecr:BatchCheckLayerAvailability",
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

We have to assign this role policy to the `ecsTaskExecution` role we assumed previously by using the following command:

```
$ aws iam attach-role-policy \
--policy-arn arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy \
--role-name ecsTaskExecutionRole
```

Once we've assigned the policy to the `ecsTaskExecution` role, we need to source the ID of both subnets and the security group of the ECS cluster when we created it. You can find those details in the command-line output from when we created the cluster. We will use these details in the following `ecs-params.yml` file:

```
version: 1
task_definition:
  task_execution_role: ecsTaskExecutionRole
  ecs_network_mode: awsvpc
  task_size:
    mem_limit: 0.5GB
    cpu_limit: 256
run_params:
  network_configuration:
    awsvpc_configuration:
      subnets:
        - "subnet-088b52c91a6f40fd7"
        - "subnet-032cd63290da67271"
      security_groups:
        - "sg-097206175813aa7e7"
      assign_public_ip: ENABLED
```

The `ecs-params.yml` file consists of `task_execution_role`, which we created, and `ecs_network_mode` set to `awsvpc`, as Fargate requires. We've defined `task_size` to have `0.5GB` of memory and `256` millicores of CPU. So, since Fargate is a serverless solution, we only pay for the CPU cores and memory we consume. The `run_params` section consists of `network_configuration`, which contains `awsvpc_configuration`. Here, we specify both subnets created when we created the ECS cluster. We must also specify `security_groups`, which we created with the ECS cluster.