> **Note**
> Use the subnets and security groups of your ECS cluster instead of copying the ones in this example.

Now that we're ready to fire the task on Fargate, let's run the following command:

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

Now, let's check whether the task is running successfully by using the following command:

```
$ ecs-cli ps --cluster cluster-1
Name                    State    Ports            TaskDefinition
cluster-1/8717a149/web  RUNNING  3.80.173.230:80 FARGATE:1
```

As we can see, the task is running on `3.80.173.230:80` as a Fargate task. Let's `curl` this URL to see whether we get a response by using the following command:

```
$ curl 3.80.173.230:80
<html>
<head>
<title>Welcome to nginx!</title>
...
</body>
</html>
```

As we can see, we get the default `nginx` home page.

Now, let's go ahead and delete the task we created by using the following command:

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

As we already know, tasks have a set life cycle, and once they stop, they stop. You cannot start the same task again. Therefore, we must create a **service** to ensure that a certain number of tasks are always running. We'll create a service in the next section.

Scheduling services on ECS

ECS services are similar to Kubernetes **ReplicaSets**. They ensure that a certain number of tasks are always running at a particular time. To schedule a service, we can use the `ecs-cli` command line.

> **Tip**
> Always use services for applications that are long-running, such as web servers. For batch jobs, always use tasks, as we don't want to recreate the job after it ends.

To run the `nginx` web server as a service, we can use the following command:

```
$ ecs-cli compose service up --create-log-groups \
 --cluster cluster-1 --launch-type FARGATE
INFO[0001] Using ECS task definition TaskDefinition="FARGATE:1"
INFO[0002] Auto-enabling ECS Managed Tags
INFO[0013] (service FARGATE) has started 1 tasks: (task 9b48084d).  timestamp="2023-07-03
11:24:42 +0000 UTC"
INFO[0029] Service status desiredCount=1 runningCount=1 serviceName=FARGATE
INFO[0029] (service FARGATE) has reached a steady state.  timestamp="2023-07-03 11:25:00
+0000 UTC"
INFO[0029] (service FARGATE) (deployment ecs-svc/94284856) deployment
completed.  timestamp="2023-07-03 11:25:00 UTC"
INFO[0029] ECS Service has reached a stable state desiredCount=1 runningCount=1
serviceName=FARGATE
INFO[0029] Created an ECS service service=FARGATE taskDefinition="FARGATE:1"
```

Looking at the logs, we can see that the service is trying to ensure that the task's desired count matches the task's running count. If your task is deleted, ECS will replace it with a new one.

Let's list the tasks and see what we get by using the following command:

```
$ ecs-cli ps --cluster cluster-1
Name                     State   Ports              TaskDefinition
cluster-1/9b48084d/web  RUNNING 18.234.123.71:80  FARGATE:1
```

As we can see, the service has created a new task that is running on `18.234.123.71:80`. Let's try to access this URL using the following command:

```
$ curl 18.234.123.71
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

We get the default `nginx` home page in the response. Now, let's try to browse the logs of the task.

Browsing container logs using the ECS CLI

Apart from using Amazon CloudWatch, you can also use the friendly ECS CLI to do this, irrespective of where your logs are stored. This helps us see everything from a single pane of glass.

Run the following command to do so:

```
$ ecs-cli logs --task-id 9b48084d --cluster cluster-1
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
configuration
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/
default.conf
```

```
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/07/03 11:24:57 [notice] 1#1: nginx/1.25.1
2023/07/03 11:24:57 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2023/07/03 11:24:57 [notice] 1#1: OS: Linux 5.10.184-175.731.amzn2.x86_64
2023/07/03 11:24:57 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 65535:65535
2023/07/03 11:24:57 [notice] 1#1: start worker processes
2023/07/03 11:24:57 [notice] 1#1: start worker process 29
2023/07/03 11:24:57 [notice] 1#1: start worker process 30
13.232.8.130 - - [03/Jul/2023:11:30:38 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.81.0"
"-"
```

As we can see, we can browse the logs for the particular task this service is running. Now, let's go ahead and delete the service.

Deleting an ECS service

To delete the service, run the following command:

```
$ ecs-cli compose service down --cluster cluster-1
INFO[0001] Deleted ECS service service=FARGATE
INFO[0001] Service status desiredCount=0 runningCount=1 serviceName=FARGATE
INFO[0006] Service status desiredCount=0 runningCount=0 serviceName=FARGATE
INFO[0006] (service FARGATE) has stopped 1 running tasks: (task
9b48084d11cf49be85141fd9bfe9e1c3).  timestamp="2023-07-03 11:34:10 +0000 UTC"
INFO[0006] ECS Service has reached a stable state desiredCount=0 runningCount=0
serviceName=FARGATE
```

As we can see, the service has been deleted.

Note that even if we create multiple instances of tasks, they run on different IP addresses and can be accessed separately. However, tasks need to be load-balanced, and we need to provide a single endpoint. Let's look at a solution we can use to manage this.

Load balancing containers running on ECS

Load balancing is an essential functionality of multi-instance applications. They help us serve the application on a single endpoint. Therefore, you can have multiple instances of your applications running simultaneously, and the end user doesn't need to worry about which instance they're calling. AWS provides two main load-balancing solutions—**Layer 4** with the **Network Load Balancer** (**NLB**) and **Layer 7** with the **Application Load Balancer** (**ALB**).

> **Tip**
>
> While both load balancers have their use cases, a Layer 7 load balancer provides a significant advantage for HTTP-based applications. It offers advanced traffic management, such as path-based and host-based routing.

So, let's go ahead and create an ALB to frontend our tasks using the following command:

```
$ aws elbv2 create-load-balancer --name ecs-alb --subnets <SUBNET-1> <SUBNET-2> \
--security-groups <SECURITY_GROUP_ID> --region us-east-1
```

The output of the preceding command contains values for `LoadBalancerARN` and `DNSName`. We will need to use them in the subsequent steps, so keep a copy of the output safe.

The next step will be to create a **target group**. The target group defines the group of tasks and the port they will be listening to, and the load balancer will forward traffic to it. Use the following command to define a target group:

```
$ aws elbv2 create-target-group --name target-group --protocol HTTP \
 --port 80 --target-type ip --vpc-id <VPC_ID> --region us-east-1
```

You will get the `targetGroupARN` value in the response. Keep it safe, as we will need it in the next step.

Next, we will need a **listener** running on the load balancer. This should forward traffic from the load balancer to the target group. Use the following command to do so:

```
$ aws elbv2 create-listener  --load-balancer-arn <LOAD_BALANCER_ARN> \
 --protocol HTTP --port 80 \
 --default-actions Type=forward,TargetGroupArn=<TARGET_GROUP_ARN> \
 --region us-east-1
```

You will get the `listenerARN` value in response to this command. Please keep that handy; we will need it in the next step.

Now that we've defined the load balancer, we need to run `ecs-cli compose service up` to deploy our service. We will also provide the target group as a parameter to associate our service with the load balancer.

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

```
$ cd ~/modern-devops/ch7/ECS/loadbalancing/
```

Run the following command to do so:

```
$ ecs-cli compose service up --create-log-groups --cluster cluster-1 \
--launch-type FARGATE --target-group-arn <TARGET_GROUP_ARN> \
--container-name web --container-port 80
```

Now that the service and our task are running on Fargate, we can scale our service to three desired tasks. To do so, run the following command:

```
$ ecs-cli compose service scale 3 --cluster cluster-1
```

Since our service has scaled to three tasks, let's go ahead and hit the load balancer DNS endpoint we captured in the first step. This should provide us with the default `nginx` response. Run the following command to do so:

```
$ curl ecs-alb-1660189891.us-east-1.elb.amazonaws.com
<html>
<head>
<title>Welcome to nginx!</title>
…
</html>
```

As we can see, we get a default `nginx` response from the load balancer. This shows that load balancing is working well!

ECS provides a host of other features, such as horizontal autoscaling, customizable task placement algorithms, and others, but they are beyond the scope of this book. Please read the ECS documentation to learn more about other aspects of the tool. Now, let's look at other popular CaaS products available on the market.

Other CaaS services

Amazon ECS provides a versatile way of managing your container workloads. It works great when you have a smaller, simpler architecture and don't want to add the additional overhead of using a complex container orchestration engine such as Kubernetes.

> **Tip**
> ECS is an excellent tool choice if you run exclusively on AWS and don't have a future multi-cloud or hybrid-cloud strategy. Fargate makes deploying and running your containers easier without worrying about the infrastructure behind the scenes.

ECS is tightly coupled with AWS and its architecture. To solve this problem, we can use managed services within AWS, such as **Elastic Kubernetes Service** (**EKS**). It offers the Kubernetes API to schedule your workloads. This makes managing containers even more versatile as you can easily spin up a Kubernetes cluster and use a standard, open source solution that you can install and run anywhere you like. This does not tie you to a particular vendor. However, EKS is slightly more expensive than ECS and adds a *$0.10* per hour cluster management charge. That is nothing in comparison to the benefits you get out of it.

If you aren't running on AWS, there are options from other providers too. The next of the big three cloud providers is Azure, which offers **Azure Kubernetes Service (AKS)**, a managed Kubernetes solution that can help you get started in minutes. AKS provides a fully managed solution with event-driven elastic provisioning for worker nodes as and when required. It also integrates nicely with **Azure DevOps**, giving you a faster **end-to-end** (**E2E**) development experience. As with AWS, Azure also charges *$0.10* per hour for cluster management.

Google Kubernetes Engine (GKE) is one of the most robust Kubernetes platforms. Since the Kubernetes project came from Google and is the largest contributor to this project in the open source community, GKE is generally quicker to roll out newer versions and is the first to release security patches into the solution. Also, it is one of the most feature-rich with customizable solutions and offers several plugins as a cluster configuration. Therefore, you can choose what to install on Bootstrap and further harden your cluster. However, all these come at a cost, as GKE charges a *$0.10* cluster management charge per hour, just like AWS and Azure.

You can use **Google Cloud Run** if you don't want to use Kubernetes if your architecture is not complicated, and there are only a few containers to manage. Google Cloud Run is a serverless CaaS solution built on the open source **Knative** project. It helps you run your containers without any vendor lock-in. Since it is serverless, you only pay for the number of containers you use and their resource utilization. It is a fully scalable and well-integrated solution with Google Cloud's DevOps and monitoring solutions such as **Cloud Code**, **Cloud Build**, **Cloud Monitoring**, and **Cloud Logging**. The best part is that it is comparable to AWS Fargate and abstracts all infrastructure behind the scenes. So, it's a minimal Ops or NoOps solution.

Now that we've mentioned Knative as an open source CaaS solution, let's discuss it in more detail.

Open source CaaS with Knative

As we've seen, several vendor-specific CaaS services are available on the market. Still, the problem with most of them is that they are tied up to a single cloud provider. Our container deployment specification then becomes vendor-specific and results in vendor lock-in. As modern DevOps engineers, we must ensure that the proposed solution best fits the architecture's needs, and avoiding vendor lock-in is one of the most important requirements.

However, Kubernetes in itself is not serverless. You must have infrastructure defined, and long-running services should have at least a single instance running at a particular time. This makes managing microservices applications a pain and resource-intensive.

But wait! We said that microservices help optimize infrastructure consumption. Yes—that's correct, but they do so within the container space. Imagine that you have a shared cluster of VMs where parts of the application scale with traffic, and each part of the application has its peaks and troughs. Doing this will save a lot of infrastructure by performing this simple multi-tenancy.

However, it also means that you must have at least one instance of each microservice running every time—even if there is zero traffic! Well, that's not the best utilization we have. How about creating instances when you get the first hit and not having any when you don't have traffic? This would save a lot of resources, especially when things are silent. You can have hundreds of microservices making up the application that would not have any instances during an idle period. If you combine it with a managed service that runs Kubernetes and then autoscale your VM instances with traffic, you can have minimal instances during the silent period.

There have been attempts within the open source and cloud-native space to develop an open source, vendor-agnostic, serverless framework for containers. We have Knative for this, which the **Cloud Native Computing Foundation** (**CNCF**) has adopted.

> **Tip**
> The Cloud Run service uses Knative behind the scenes. So, if you use Google Cloud, you can use Cloud Run to use a fully managed serverless offering.

To understand how Knative works, let's look at the Knative architecture.

Knative architecture

The Knative project combines elements of existing CNCF projects such as Kubernetes, **Istio**, **Prometheus**, and **Grafana** and eventing engines such as **Kafka** and **Google Pub/Sub**. Knative runs as a Kubernetes operator using Kubernetes **Custom Resource Definitions** (**CRDs**), which help operators administer Knative using the `kubectl` command line. Knative provides its API for developers, which the `kn` command-line utility can use. The users are provided access through Istio, which, with its traffic management features, is a crucial component of Knative. The following diagram describes this graphically:

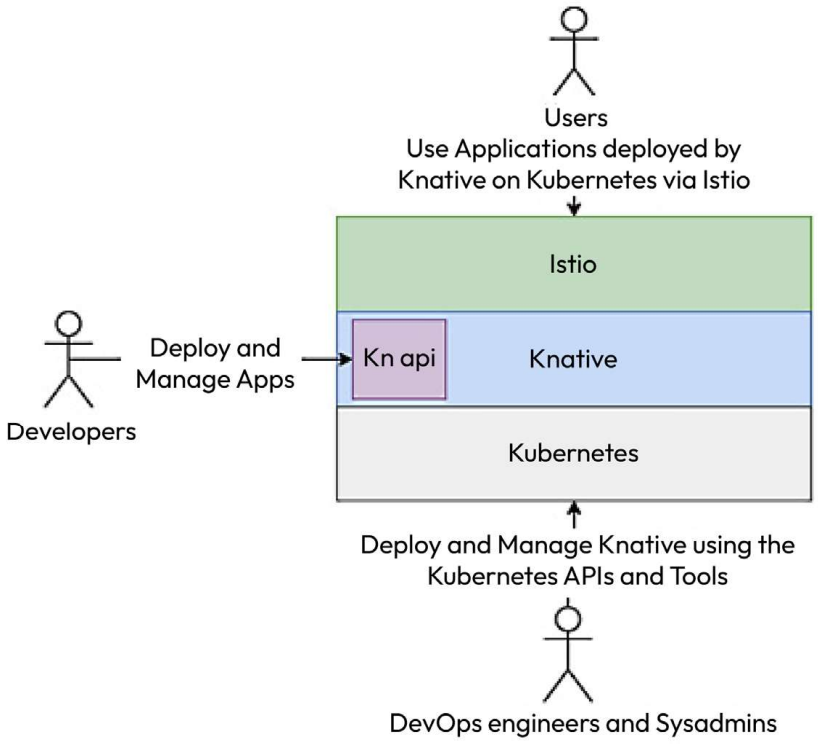

Figure 7.2 – Knative architecture

Knative consists of two main modules—`serving` and `eventing`. While the `serving` module helps us maintain stateless applications using HTTP/S endpoints, the `eventing` module integrates with eventing engines such as Kafka and Google Pub/Sub. As we've discussed mostly HTTP/S traffic, we will scope our discussion to Knative serving for this book.

Knative maintains serving pods, which help route traffic within workload pods and act as proxies using the **Istio Ingress Gateway** component. It provides a virtual endpoint for your service and listens to it. When it discovers a hit on the endpoint, it creates the required Kubernetes components to serve that traffic. Therefore, Knative has the functionality to scale from zero workload pods as it will spin up a pod when it receives traffic for it. The following diagram shows how:

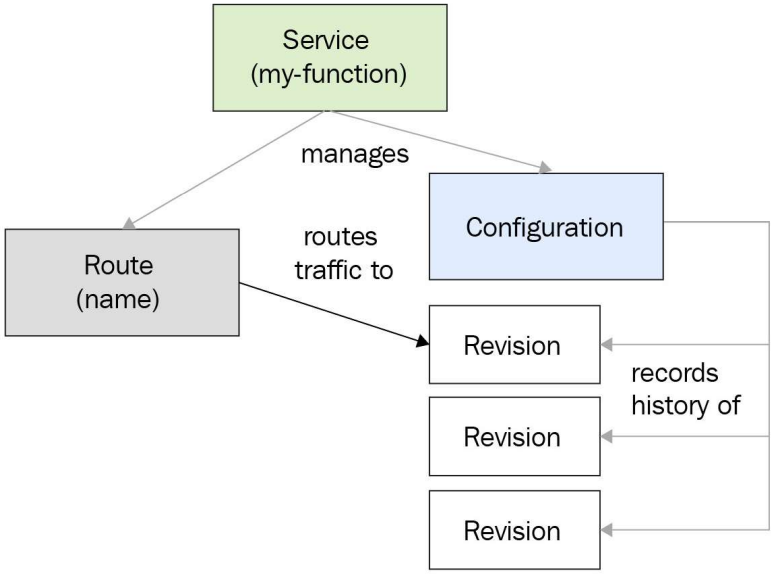

Figure 7.3 – Knative serving architecture

Knative endpoints are made up of three basic parts—`<app-name>`, `<namespace>`, and `<custom-domain>`. While `name` and `namespace` are similar to Kubernetes Services, `custom-domain` is defined by us. It can be a legitimate domain for your organization or a **MagicDNS** solution, such as **sslip.io,** which we will use in our hands-on exercises. If you are using your organization domain, you must create your DNS configuration to resolve the domain to the Istio Ingress Gateway IP addresses.

Now, let's go ahead and install Knative.

For the exercises, we will use GKE. Since GKE is a highly robust Kubernetes cluster, it is a great choice for integrating with Knative. As mentioned previously, Google Cloud provides a free trial of $300 for 90 days. You can sign up at `https://cloud.google.com/free` if you've not done so already.

Spinning up GKE

Once you've signed up and are on your console, you can open the Google Cloud Shell CLI to run the following commands.

You need to enable the GKE API first using the following command:

```
$ gcloud services enable container.googleapis.com
```

To create a two-node autoscaling GKE cluster that scales from `1` node to `5` nodes, run the following command:

```
$ gcloud container clusters create cluster-1 --num-nodes 2 \
--enable-autoscaling --min-nodes 1 --max-nodes 5 --zone us-central1-a
```

And that's it! The cluster is up and running.

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
```

Now that the cluster is up and running, let's go ahead and install Knative.

Installing Knative

We will install the CRDs that define Knative resources as Kubernetes API resources.

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

```
$ cd ~/modern-devops/ch7/knative/
```

Run the following command to install the CRDs:

```
$ kubectl apply -f \
 https://github.com/knative/serving/releases/download/knative-v1.10.2/serving-crds.yaml
```

As we can see, Kubernetes has installed some CRDs. Next, we must install the core components of the Knative `serving` module. Use the following command to do so:

```
$ kubectl apply -f \
 https://github.com/knative/serving/releases/download/knative-v1.10.2/serving-core.yaml
```

Now that the core serving components have been installed, the next step is installing Istio within the Kubernetes cluster. To do so, run the following commands:

```
$ curl -L https://istio.io/downloadIstio | sh -
$ sudo mv istio-*/bin/istioctl /usr/local/bin
$ istioctl install --set profile=demo -y
```

Now that Istio has been installed, we will wait for the Istio Ingress Gateway component to be assigned an external IP address. Run the following command to check this until you get an external IP in the response:

```
$ kubectl -n istio-system get service istio-ingressgateway
NAME                TYPE         EXTERNAL-IP   PORT(S)
istio-ingressgteway LoadBalancer 35.226.198.46 15021,80,443
```

As we can see, we've been assigned an external IP—`35.226.198.46`. We will use this IP for the rest of this exercise.

Now, we will install the Knative Istio controller by using the following command:

```
$ kubectl apply -f \
  https://github.com/knative/net-istio/releases/download/knative-v1.10.1/net-istio.yaml
```

Now that the controller has been installed, we must configure the DNS so that Knative can provide custom endpoints. To do so, we can use the MagicDNS solution known as `sslip.io`, which you can use for experimentation. The MagicDNS solution resolves any endpoint to the IP address present in the subdomain. For example, `35.226.198.46.sslip.io` resolves to `35.226.198.46`.

> **Note**
>
> Do not use MagicDNS in production. It is an experimental DNS service and should only be used for evaluating Knative.

Run the following command to configure the DNS:

```
$ kubectl apply -f \
https://github.com/knative/serving/releases/download/knative-v1.10.2\
/serving-default-domain.yaml
```

As you can see, it provides a batch job that gets fired whenever there is a DNS request.

Now, let's install the **HorizontalPodAutoscaler** (**HPA**) add-on to automatically help us autoscale pods on the cluster with traffic. To do so, run the following command:

```
$ kubectl apply -f \
https://github.com/knative/serving/releases/download/knative-v1.10.2/serving-hpa.yaml
```

That completes our Knative installation.

Now, we need to install and configure the `kn` command-line utility. Use the following commands to do so:

```
$ sudo curl -Lo /usr/local/bin/kn \
https://github.com/knative/client/releases/download/knative-v1.10.0/kn-linux-amd64
$ sudo chmod +x /usr/local/bin/kn
```

In the next section, we'll deploy our first application on Knative.

Deploying a Python Flask application on Knative

To understand Knative, let's try to build and deploy a Flask application that outputs the current timestamp in the response. Let's start by building the app.

Building the Python Flask app

We will have to create a few files to build such an app.

The `app.py` file looks like this:

```
import os
import datetime
from flask import Flask
app = Flask(__name__)
@app.route('/')
def current_time():
  ct = datetime.datetime.now()
  return 'The current time is : {}!\n'.format(ct)
if __name__ == "__main__":
  app.run(debug=True,host='0.0.0.0')
```

We will need the following Dockerfile to build this application:

```
FROM python:3.7-slim
ENV PYTHONUNBUFFERED True
ENV APP_HOME /app
WORKDIR $APP_HOME
COPY . ./
RUN pip install Flask gunicorn
CMD exec gunicorn --bind :$PORT --workers 1 --threads 8 --timeout 0 app:app
```

Now, let's go ahead and build the Docker container using the following command:

```
$ docker build -t <your_dockerhub_user>/py-time .
```

Now that the image is ready, let's push it to Docker Hub by using the following command:

```
$ docker push <your_dockerhub_user>/py-time
```

As we've successfully pushed the image, we can run this on Knative.

Deploying the Python Flask app on Knative

We can use the `kn` command line or create a manifest file to deploy the app. Use the following command to deploy the application:

```
$ kn service create py-time --image <your_dockerhub_user>/py-time
Creating service 'py-time' in namespace 'default':
  9.412s Configuration "py-time" is waiting for a Revision to become ready.
  9.652s Ingress has not yet been reconciled.
  9.847s Ready to serve.
Service 'py-time' created to latest revision 'py-time-00001' is available at URL:
http://py-time.default.35.226.198.46.sslip.io
```

As we can see, Knative has deployed the app and provided a custom endpoint. Let's `curl` the endpoint to see what we get:

```
$ curl http://py-time.default.35.226.198.46.sslip.io
The current time is : 2023-07-03 13:30:20.804790!
```

We get the current time in the response. As we already know, Knative should detect whether there is no traffic coming into the pod and delete it. Let's watch the pods for some time and see what happens:

```
$ kubectl get pod -w
NAME                              READY   STATUS        RESTARTS   AGE
py-time-00001-deployment-jqrbk    2/2     Running       0          5s
py-time-00001-deployment-jqrbk    2/2     Terminating   0          64s
```

As we can see, just after 1 minute of inactivity, Knative starts terminating the pod. Now, that's what we mean by scaling from zero.

To delete the service permanently, we can use the following command:

```
$ kn service delete py-time
```

We've just looked at the imperative way of deploying and managing the application. But what if we want to declare the configuration as we did previously? We can create a CRD manifest with the `Service` resource provided by `apiVersion—serving.knative.dev/v1`.

We will create the following manifest file, called `py-time-deploy.yaml`, for this:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: py-time
```

```
spec:
  template:
    spec:
      containers:
        - image: <your_dockerhub_user>/py-time
```

As we've created this file, we will use the `kubectl` CLI to apply it. It makes deployment consistent with Kubernetes.

> **Note**
>
> Though it is a `service` resource, don't confuse this with the typical Kubernetes `Service` resource. It is a custom resource provided by `apiVersion` `serving.knative.dev/v1`. That is why `apiVersion` is very important.

Let's go ahead and run the following command to do so:

```
$ kubectl apply -f py-time-deploy.yaml
service.serving.knative.dev/py-time created
```

With that, the service has been created. To get the service's endpoint, we will have to query the `ksvc` resource using `kubectl`. Run the following command to do so:

```
$ kubectl get ksvc py-time
NAME      URL
py-time   http://py-time.default.35.226.198.46.sslip.io
```

The URL is the custom endpoint we have to target. Let's `curl` the custom endpoint using the following command:

```
$ curl http://py-time.default.35.226.198.46.sslip.io
The current time is : 2023-07-03 13:30:23.345223!
```

We get the same response this time as well! So, if you want to keep using `kubectl` for managing Knative, you can easily do so.

Knative helps scale applications based on the load it receives—automatic horizontal scaling. Let's run load testing on our application to see that in action.

Load testing your app on Knative

We will use the `hey` utility to perform load testing. Since your application has already been deployed, run the following command to do the load test:

```
$ hey -z 30s -c 500 http://py-time.default.35.226.198.46.sslip.io
```

Once the command has executed, run the following command to get the currently running instances of the `py-time` pods:

```
$ kubectl get pod
NAME                              READY STATUS RESTARTS   AGE
py-time-00001-deployment-52vjv 2/2   Running   0        44s
py-time-00001-deployment-bhhvm 2/2   Running   0        44s
py-time-00001-deployment-h6qr5 2/2   Running   0        42s
py-time-00001-deployment-h92jp 2/2   Running   0        40s
py-time-00001-deployment-p27gl 2/2   Running   0        88s
py-time-00001-deployment-tdwrh 2/2   Running   0        38s
py-time-00001-deployment-zsgcg 2/2   Running   0        42s
```

As we can see, Knative has created seven instances of the `py-time` pod. This is horizontal autoscaling in action.

Now, let's look at the cluster nodes by running the following command:

```
$ kubectl get nodes
NAME                                        STATUS   AGE
gke-cluster-1-default-pool-353b3ed4-js71   Ready   3m17s
gke-cluster-1-default-pool-353b3ed4-mx83   Ready   106m
gke-cluster-1-default-pool-353b3ed4-vf7q   Ready   106m
```

As we can see, GKE has created another node in the node pool because of the extra burst of traffic it received. This is phenomenal, as we have the Kubernetes API to do what we want. We have automatically horizontally autoscaled our pods. We have also automatically horizontally autoscaled our cluster worker nodes. This means we have a fully automated solution for running containers without worrying about the management nuances! That is open source serverless in action for you!

Summary

This chapter covered CaaS and serverless CaaS services. These help us manage container applications with ease without worrying about the underlying infrastructure and managing them. We used Amazon's ECS as an example and deep-dived into it. Then, we briefly discussed other solutions that are available on the market.

Finally, we looked at Knative, an open source serverless solution for containers that run on top of Kubernetes and use many other open source CNCF projects.

In the next chapter, we will delve into IaC with Terraform.

Questions

1. ECS allows us to deploy to which of the following? (Choose two)

   A. EC2

   B. AWS Lambda

C. Fargate

D. Amazon Lightsail

2. ECS uses Kubernetes in the background. (True/False)
3. We should always use services in ECS instead of tasks for batch jobs. (True/False)
4. We should always use Fargate for batch jobs as it runs for a short period, and we only pay for the resources that are consumed during that time. (True/False)
5. Which of the following are CaaS services that implement the Kubernetes API? (Choose three)

   A. GKE

   B. AKS

   C. EKS

   D. ECS

6. Google Cloud Run is a serverless offering that uses Knative behind the scenes. (True/False)
7. Which one of the following is offered as a Knative module? (Choose two)

   A. Serving

   B. Eventing

   C. Computing

   D. Containers

Answers

1. A, C
2. False
3. False
4. True
5. A, B, C
6. True
7. A, B

Part 3: Managing Config and Infrastructure

This part takes a deep dive into infrastructure and configuration management in the public cloud, exploring various tools that enable infrastructure automation, configuration management, and immutable infrastructure.

This part has the following chapters:

- *Chapter 8*, *Infrastructure as Code (IaC) with Terraform*
- *Chapter 9*, *Configuration Management with Ansible*
- *Chapter 10*, *Immutable Infrastructure with Packer*

8

Infrastructure as Code (IaC) with Terraform

Cloud computing is one of the primary factors of DevOps enablement today. The initial apprehensions about the cloud are a thing of the past. With an army of security and compliance experts manning cloud platforms 24x7, organizations are now trusting the *public cloud* like never before. Along with cloud computing, another buzzword has taken the industry by storm – **Infrastructure as Code** (**IaC**). This chapter will focus on IaC with **Terraform**, and by the end of this chapter, you will understand the concept and have enough hands-on experience with Terraform to get you started on your journey.

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

- Introduction to IaC
- Setting up Terraform and Azure providers
- Understanding Terraform workflows and creating your first resource using Terraform
- Terraform modules
- Terraform state and backends
- Terraform workspaces
- Terraform outputs, state, console, and graphs

Technical requirements

For this chapter, you can use any machine to run Terraform. Terraform supports many platforms, including Windows, Linux, and macOS.

You will need an active Azure subscription to follow the exercises. Currently, Azure is offering a free trial for 30 days with $200 worth of free credits; you can sign up at `https://azure.microsoft.com/en-in/free`.

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, and `cd` into the `ch8` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
  modern-devops
$ cd modern-devops/ch8
```

So, let's get started!

Introduction to IaC

IaC is the concept of using code to define infrastructure. While most people can visualize infrastructure as something tangible, virtual infrastructure is already commonplace and has existed for around two decades. Cloud providers provide a web-based console through which you can manage your infrastructure intuitively. But the process is not repeatable or recorded.

If you spin up a set of infrastructure components using the console in one environment and want to replicate it in another, it is a duplication of effort. To solve this problem, cloud platforms provide APIs to manipulate resources within the cloud and some command-line tools that can help trigger the APIs. You can start writing scripts using commands to create the infrastructure and parameterize them to use the same scripts in another environment. Well, that solves the problem, right?

Not really! Writing scripts is an imperative way of managing infrastructure. Though you can still call it IaC, its problem is that it does not effectively manage infrastructure changes. Let me give you a few examples:

- What would happen if you needed to modify something already in the script? Changing the script somewhere in the middle and rerunning the entire thing may create havoc with your infrastructure. Imperative management of infrastructure is not idempotent. So, managing changes becomes a problem.
- What if someone manually changes the script-managed infrastructure using the console? Will your script be able to detect it correctly? What if you want to change the same thing using a script? It will soon start to get messy.
- With the advent of hybrid cloud architecture, most organizations use multiple cloud platforms for their needs. When you are in such a situation, managing multiple clouds with imperative scripts soon becomes a problem. Different clouds have different ways of interacting with their APIs and have distinct command-line tools.

The solution to all these problems is a declarative IaC solution such as Terraform. HashiCorp's Terraform is the most popular IaC tool available on the market. It helps you automate and manage your infrastructure using code and can run on various platforms. As it is declarative, you just need to define what you need (the desired end state) instead of describing how to achieve it. It has the following features:

- It supports multiple cloud platforms via providers and exposes a single declarative **HashiCorp Configuration Language** (**HCL**)-based interface to interact with it. Therefore, it allows you to manage various cloud platforms using a similar language and syntax. So, having a few Terraform experts within your team can handle all your IaC needs.
- It tracks the state of the resources it manages using state files and supports local and remote backends to store and manage them. That helps in making the Terraform configuration idempotent. So, if someone manually changes a Terraform-managed resource, Terraform can detect the difference in the next run and prompt corrective action to bring it to the defined configuration. The admin can then absorb the change or resolve any conflicts before applying it.
- It enables GitOps in infrastructure management. With Terraform, you can have the infrastructure configuration alongside application code, making versioning, managing, and releasing infrastructure the same as managing code. You can also include code scanning and gating using pull requests so that someone can review and approve the changes to higher environments before you apply them. A great power indeed!

Terraform has multiple offerings – **open source**, **cloud**, and **enterprise**. The open source offering is a simple **command-line interface** (**CLI**)-based tool that you can download on any supported **operating system** (**OS**) and use. The cloud and enterprise offerings are more of a wrapper on top of the open source one. They provide a web-based GUI and advanced features such as **policy as code** with **Sentinel**, **cost analysis**, **private modules**, **GitOps**, and **CI/CD pipelines**.

This chapter will discuss the open source offering and its core functions.

Terraform open source is divided into two main parts – **Terraform Core** and **Terraform providers**, as seen in the following diagram:

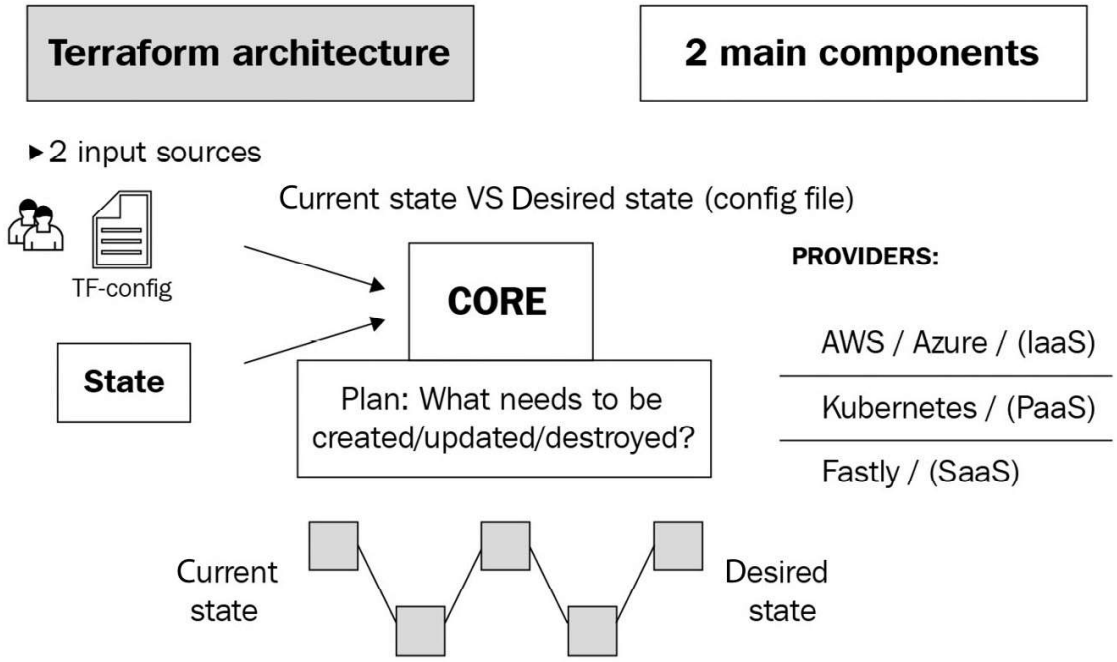

Figure 8.1 – Terraform architecture

Let's look at the functions of both components:

- **Terraform Core** is the CLI that we will use to interact with Terraform. It takes two main inputs – your Terraform configuration files and the existing state. It then takes the difference in configuration and applies it.
- **Terraform providers** are plugins that Terraform uses to interact with cloud providers. The providers translate the Terraform configuration into the respective cloud's REST API calls so that Terraform can manage its associated infrastructure. For example, if you want Terraform to manage AWS infrastructure, you must use the Terraform AWS provider.

Now let's see how we can install open source Terraform.

Installing Terraform

Installing Terraform is simple; go to `https://www.terraform.io/downloads.html` and follow the instructions for your platform. Most of it will require you to download a binary and move it to your system path.

Since we've been using Ubuntu throughout this book, I will show the installation on Ubuntu. Use the following commands to use the `apt` package manager to install Terraform:

```
$ wget -O- https://apt.releases.hashicorp.com/gpg | \
  sudo gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg
```

```
$ echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \
https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \
sudo tee /etc/apt/sources.list.d/hashicorp.list
$ sudo apt update && sudo apt install terraform
```

Check whether Terraform has been installed successfully with the following command:

```
$ terraform version
Terraform v1.5.2
```

It shows that Terraform has been installed successfully. Terraform uses Terraform providers to interact with cloud providers, so let's look at those in the next section.

Terraform providers

Terraform has a decentralized architecture. While the Terraform CLI contains Terraform's core functionality and provides all functionalities not related to any specific cloud provider, Terraform providers provide the interface between the Terraform CLI and the cloud providers themselves. This decentralized approach has allowed public cloud vendors to offer their Terraform providers so that their customers can use Terraform to manage infrastructure in their cloud. Such is Terraform's popularity that it has now become an essential requirement for every public cloud provider to offer a Terraform provider.

We will interact with Azure for this chapter's entirety and use the Azure Terraform provider for our activity.

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

```
$ cd ~/modern-devops/ch8/terraform-exercise/
```

Before we go ahead and configure the provider, we need to understand how Terraform needs to authenticate and authorize with the Azure APIs.

Authentication and authorization with Azure

The simplest way to authenticate and authorize with Azure is to log in to your account using the Azure CLI. When you use the Azure provider within your Terraform file, it will automatically act as your account and do whatever it needs to. Now, this sounds dangerous. Admins generally have a lot of access, and having a tool that acts as an admin might not be a great idea. What if you want to plug Terraform into your CI/CD pipelines? Well, there is another way to do it – by using **Azure service principals**. Azure service principals allow you to access the required features without using a named user account. You can then apply the **principle of least privilege** to the service principal and provide only the necessary access.

Before configuring the service principal, let's install the Azure CLI on our machine. To do so, run the following command:

```
$ curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
```

The preceding command will download a shell script and execute it using `bash`. The script will then automatically download and configure the Azure CLI. To confirm whether the Azure CLI is installed successfully, run the following command:

```
$ az --version
azure-cli                         2.49.0
```

We see that the Azure CLI is correctly installed on the system. Now, let's go ahead and configure the service principal.

To configure the Azure service principal, follow these steps.

Log in to Azure using the following command and follow all the steps the command prompts. You must browse to a specified URL and enter the given code. Once you've logged in, you will get a JSON response that will include some details, something like the following:

```
$ az login
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter
the code XXXXXXXXX to authenticate:
[
  {
    "id": "00000000-0000-0000-0000-0000000000000",
    ...
  }
]
```

Make a note of the `id` attribute, which is the subscription ID, and if you have more than one subscription, you can use the following to set it to the correct one:

```
$ export SUBSCRIPTION_ID="<SUBSCRIPTION_ID>"
$ az account set --subscription="$SUBSCRIPTION_ID"
```

Use the following command to create a **service principal** with the `contributor` role to allow Terraform to manage the subscription's infrastructure.

> **Tip**
>
> Follow the principle of least privilege while granting access to the service principal. Do not give privileges thinking you might need them in the future. If any future access is required, you can grant it later.

We use `contributor` access for simplicity, but finer-grained access is possible and should be used:

```
$ az ad sp create-for-rbac --role="Contributor" \
 --scopes="/subscriptions/$SUBSCRIPTION_ID"
Creating 'Contributor' role assignment under scope '/subscriptions/<SUBSCRIPTION_ID>'
The output includes credentials that you must protect. Ensure you do not include these
credentials in your code or check the credentials into your source control (for more
information, see https://aka.ms/azadsp-cli):
{
  "appId": "00000000-0000-0000-0000-0000000000000",
  "displayName": "azure-cli-2023-07-02-09-13-40",
  "password": "00000000000.xx-00000000000000000",
  "tenant": "00000000-0000-0000-0000-0000000000000"
}
```

We've successfully created the **service principal**. The response JSON consists of `appId`, `password`, and `tenant`. We will need these to configure Terraform to use the service principal. In the next section, let's define the Azure Terraform provider with the details.

Using the Azure Terraform provider

Before we define the Azure Terraform provider, let's understand what makes a Terraform root module. The Terraform root module is just a working directory within your filesystem containing one or more `.tf` files that help you define your configuration and are where you would typically run your Terraform commands.

Terraform scans all your `.tf` files, combines them, and processes them internally as one. Therefore, you can have one or more `.tf` files that you can split according to your needs. While there are no defined standards for naming `.tf` files, most conventions use `main.tf` as the main Terraform file where they define resources, a `vars.tf` file for defining variables, and `outputs.tf` for defining outputs.

For this discussion, let's create a `main.tf` file within our working directory and add a `provider` configuration like the following:

```
terraform {
  required_providers {
    azurerm = {
      source  = "azurerm"
      version = "=3.55.0"
    }
  }
}
provider "azurerm" {
  subscription_id = var.subscription_id
  client_id       = var.client_id
  client_secret   = var.client_secret
  tenant_id       = var.tenant_id
  features {}
}
```

The preceding file contains two blocks. The `terraform` block contains the `required_providers` block, which declares the `version` constraint for the `azurerm` provider. The `provider` block declares an `azurerm` provider, which requires four parameters.

> **Tip**
>
> Always constrain the provider version, as providers are released without notice, and if you don't include the version number, something that works on your machine might not work on someone else's machine or the CI/CD pipeline. Using a version constraint avoids breaking changes and keeps you in control.

You might have noticed that we have declared several variables within the preceding file instead of directly inputting the values. There are two main reasons for that – we want to make our template as generic as possible to promote reuse. So, suppose we want to apply a similar configuration in another subscription or use another service principal; we should be able to change it by changing the variable values. Secondly, we don't want to check `client_id` and `client_secret` in source control. It is a bad practice as we expose our service principal to users beyond those who need to know about it.

> **Tip**
>
> Never store sensitive data in source control. Instead, use a `tfvars` file to manage sensitive information and keep it in a secret management system such as HashiCorp's Vault.

Okay, so as we've defined the provider resource and the attribute values are sourced from variables, the next step would be to declare variables. Let's have a look at that now.

Terraform variables

To declare variables, we will need to create a `vars.tf` file with the following data:

```
variable "subscription_id" {
  type        = string
  description = "The azure subscription id"
}
variable "app_id" {
  type        = string
  description = "The azure service principal appId"
}
variable "password" {
  type        = string
  description = "The azure service principal password"
  sensitive   = true
}
variable "tenant" {
  type        = string
  description = "The azure tenant id"
}
```

So, we've defined four variables here using `variable` blocks. Variable blocks typically have a `type` and a `description`. The `type` attribute defines the data type of the variable we declare and defaults to the `string` data type. It can be a primitive data type such as `string`, `number`, or `bool`, or a complex data structure such as `list`, `set`, `map`, `object`, or `tuple`. We will look at types in detail when we use them later in the exercises. The `description` attribute provides more information regarding the variable so users can refer to it for better understanding.

> **Tip**
> Always set the `description` attribute right from the beginning, as it is user-friendly and promotes the reuse of your template.

The `client_secret` variable also contains a third attribute called `sensitive`, a Boolean attribute set to `true`. When the `sensitive` attribute is `true`, the Terraform CLI does not display it in the screen's output. This attribute is highly recommended for sensitive variables such as passwords and secrets.

> **Tip**
> Always declare a sensitive variable as `sensitive`. This is because if you use Terraform within your CI/CD pipelines, unprivileged users might access sensitive information by looking at the logs.

Apart from the other three, an attribute called `default` will help you specify default variable values. The default values help you provide the best possible value for a variable, which your users can override if necessary.

> **Tip**
> Always use default values where possible, as they allow you to provide users with soft guidance about your enterprise standard and save them time.

The next step would be to provide variable values. Let's have a look at that.

Providing variable values

There are a few ways to provide variable values within Terraform:

- **Via the console using `-var` flags**: You can use multiple `-var` flags with the `variable_name=variable_value` string to supply the values.
- **Via a variable definition file (the `.tfvars` file)**: You can use a file containing the list of variables and their values ending with an extension of `.tfvars` (if you prefer HCL) or `.tfvars.json` (if you prefer JSON) via the command line with the `-var-file` flag.

- **Via default variable definition files**: If you don't want to supply the variable definition file via the command line, you can create a file with the name `terraform.tfvars` or end it with an extension of `.auto.tfvars` within the Terraform workspace. Terraform will automatically scan these files and take the values from there.
- **Environment variables**: If you don't want to use a file or pass the information via the command line, you can use environment variables to supply it. You must create environment variables with the `TF_VAR_<var-name>` structure containing the variable value.
- **Default**: When you run a Terraform plan without providing values to variables in any other way, the Terraform CLI will prompt for the values, and you must manually enter them.

If multiple methods are used to provide the same variable's value, the first method in the preceding list has the highest precedence for a specific variable. It overrides anything that is defined in the methods listed later.

We will use the `terraform.tfvars` file for this activity and provide the values for the variables.

Add the following data to the `terraform.tfvars` file:

```
subscription_id = "<SUBSCRIPTION_ID>"
app_id          = "<SERVICE_PRINCIPAL_APP_ID>"
password        = "<SERVICE_PRINCIPAL_PASSWORD>"
tenant          = "<TENANT_ID>"
```

If you are checking the Terraform configuration into source control, add the file to the ignore list to avoid accidentally checking it in.

If you use Git, adding the following to the `.gitignore` file will suffice:

```
*.tfvars
.terraform*
```

Now, let's go ahead and look at the Terraform workflow to progress further.

Terraform workflow

The Terraform workflow typically consists of the following:

- `init`: Initializes the Terraform **workspace** and **backend** (more on them later) and downloads all required providers. You can run the `init` command multiple times during your build, as it does not change your workspace or state.
- `plan`: It runs a speculative plan on the requested resources. This command typically connects with the cloud provider and then checks whether the objects managed by Terraform exist within the cloud provider and whether they have the same configuration as defined in the Terraform template. It then shows the delta in the plan output that an admin can review and change the

configuration if unsatisfied. If satisfied, they can apply the plan to commit the changes to the cloud platform. The `plan` command does not make any changes to the current infrastructure.

- `apply`: This applies the delta configuration to the cloud platform. When you use `apply` by itself, it runs the `plan` command first and asks for confirmation. If you supply a plan to it, it applies the plan directly. You can also use `apply` without running the plan using the `-auto-approve` flag.
- `destroy`: The `destroy` command destroys the entire infrastructure Terraform manages. It is, therefore, not a very popular command and is rarely used in a production environment. That does not mean that the `destroy` command is not helpful. Suppose you are spinning up a development infrastructure for temporary purposes and don't need it later. In that case, destroying everything you created using this command takes a few minutes.

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

```
$ cd ~/modern-devops/ch8/terraform-exercise
```

Now, let's look at these in detail with hands-on exercises.

terraform init

To initialize a Terraform workspace, run the following command:

```
$ terraform init
Initializing the backend...
Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "3.63.0"...
- Installing hashicorp/azurerm v3.63.0...
- Installed hashicorp/azurerm v3.63.0 (signed by HashiCorp)
Terraform has created a lock file, .terraform.lock.hcl, to record the provider selections
it made previously. Include this file in your version control repository so that Terraform
can guarantee to make the same selections by default when you run terraform init in the
future.
Terraform has been successfully initialized!
```

As the Terraform workspace has been initialized, we can create an **Azure resource group** to start working with the cloud.

Creating the first resource – Azure resource group

We must use the `azurerm_resource_group` resource within the `main.tf` file to create an Azure resource group. Add the following to your `main.tf` file to do so:

```
resource "azurerm_resource_group" "rg" {
  name     = var.rg_name
  location = var.rg_location
}
```

As we've used two variables, we've got to declare those, so add the following to the `vars.tf` file:

```
variable "rg_name" {
  type        = string
  description = "The resource group name"
}
variable "rg_location" {
  type        = string
  description = "The resource group location"
}
```

Then, we need to add the resource group name and location to the `terraform.tfvars` file. Therefore, add the following to the `terraform.tfvars` file:

```
rg_name=terraform-exercise
rg_location="West Europe"
```

So, now we're ready to run a plan, but before we do so, let's use `terraform fmt` to format our files into the canonical standard.

terraform fmt

The `terraform fmt` command formats the `.tf` files into a canonical standard. Use the following command to format your files:

```
$ terraform fmt
terraform.tfvars
vars.tf
```

The command lists the files that it formatted. The next step is to validate your configuration.

terraform validate

The `terraform validate` command validates the current configuration and checks whether there are any syntax errors. To validate your configuration, run the following:

```
$ terraform validate
Success! The configuration is valid.
```

The success output denotes that our configuration is valid. If there were any errors, it would have highlighted them in the validated output.

> **Tip**
>
> Always run `fmt` and `validate` before every Terraform plan. It saves you a ton of planning time and helps you keep your configuration in good shape.

As the configuration is valid, we are ready to run a plan.

terraform plan

To run a Terraform plan, use the following command:

```
$ terraform plan
Terraform used the selected providers to generate the following execution plan. Resource
actions are indicated with the following symbols:  + create
Terraform will perform the following actions:
  # azurerm_resource_group.rg will be created
  + resource "azurerm_resource_group" "rg" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "terraform-exercise"
    }
Plan: 1 to add, 0 to change, 0 to destroy.
Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to
take exactly these actions if you run terraform apply now.
```

The `plan` output tells us that if we run `terraform apply` immediately, it will create a single `terraform_exercise` resource group. It also outputs a note that since we did not save this plan, the subsequent application is not guaranteed to result in the same action. Meanwhile, things might have changed; therefore, Terraform will rerun `plan` and prompt us for `yes` when applying. Thus, you should save the plan to a file if you don't want surprises.

> **Tip**
>
> Always save `terraform plan` output to a file and use the file to apply the changes. This is to avoid any last-minute surprises with things that might have changed in the background and `apply` not doing what it is intended to do, especially when your plan is reviewed as a part of your process.

So, let's go ahead and save the plan to a file first using the following command:

```
$ terraform plan -out rg_terraform_exercise.tfplan
```

This time, the plan is saved to a file called `rg_terraform_exercise.tfplan`. We can use this file to apply the changes subsequently.

terraform apply

To apply the changes using the `plan` file, run the following command:

```
$ terraform apply "rg_terraform_exercise.tfplan"
azurerm_resource_group.rg: Creating...
```

```
azurerm_resource_group.rg: Creation complete after 2s [id=/subscriptions/id/
resourceGroups/terraform-exercise]
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

And that's it! Terraform has applied the configuration. Let's use the Azure CLI to verify whether the resource group is created.

Run the following command to list all resource groups within your subscription:

```
$ az group list
...
    "id": "/subscriptions/id/resourceGroups/terraform-exercise",
    "location": "westeurope",
    "name": "terraform-exercise",
...
```

We see that our resource group is created and within the list.

There might be instances when `apply` is partially successful. In that case, Terraform will automatically taint resources it believes weren't created successfully. Such resources will be recreated automatically in the next run. If you want to taint a resource for recreation manually, you can use the `terraform taint` command:

```
$ terraform taint <resource>
```

Suppose we want to destroy the resource group as we no longer need it. We can use `terraform destroy` for that.

terraform destroy

To destroy the resource group, we can run a speculative plan first. It is always a best practice to run a speculative plan to confirm that what we need to destroy is within the output to have no surprises later. Terraform, like Linux, does not have an undo button.

To run a speculative destroy plan, use the following command:

```
$ terraform plan -destroy
Terraform used the selected providers to generate the following execution plan. Resource
actions are indicated with the following symbols:
  - destroy
Terraform will perform the following actions:
  # azurerm_resource_group.rg will be destroyed
  - resource "azurerm_resource_group" "rg" {
      - id = "/subscriptions/id/resourceGroups/terraform-exercise" -> null
      - location = "westeurope" -> null
      - name = "terraform-exercise" -> null
      - tags = {} -> null
    }
Plan: 0 to add, 0 to change, 1 to destroy.
```

As we see, as the resource group was the only resource managed by Terraform, it has listed that as the resource that will be destroyed. There are two ways of destroying the resource: using `terraform destroy` on its own or saving the speculative plan using the `out` parameter and running `terraform apply` on the destroy plan.

Let's use the first method for now.

Run the following command to destroy all resources managed by Terraform:

```
$ terraform destroy
Terraform will perform the following actions:
  # azurerm_resource_group.rg will be destroyed
Do you really want to destroy all resources?
Terraform will destroy all your managed infrastructure, as shown above. There is no undo.
Only yes will be accepted to confirm.
  Enter a value:
```

Now, this time, Terraform reruns `plan` and prompts for a value. It will only accept `yes` as confirmation. So, you can review the output, type `yes`, and hit *Enter* to confirm:

```
  Enter a value: yes
azurerm_resource_group.rg: Destroying... [id=/subscriptions/id/resourceGroups/terraform-
exercise]
azurerm_resource_group.rg: Still destroying... [id=/subscriptions/id/resourceGroups/
terraform-exercise, 10s elapsed]
azurerm_resource_group.rg: Destruction complete after 1m20s
```

And it has now destroyed the resource group.

We've looked at a basic root module and explored the Terraform workflow. The basic root module helps us create and manage resources but lacks a very important feature – reusability. Terraform provides us with modules to allow reusability for common templates. Let's look at it in the next section.

Terraform modules

Terraform modules are reusable, repeatable templates. They allow abstraction in provisioning infrastructure, which is much needed if your usage grows beyond just some proof of concept. HashiCorp visualizes modules as designed by experts who know the enterprise standard and used by developers who want to apply the enterprise standard infrastructure in their projects. That way, things are standard across the organization. It saves developers time and avoids duplication of efforts. Modules can be versioned and distributed using a **module repository** or through your version control. That provides infrastructure admins with an ample amount of power and control at the same time.

As we created a resource group in the last section, let's modularize that in the next exercise. To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch8/terraform-modules/
```

Within this directory, we have the following directory structure:

```
.
├── main.tf
├── modules
│   └── resource_group
│       ├── main.tf
│       └── vars.tf
├── terraform.tfvars
└── vars.tf
```

As we can see, we have the `main.tf`, `terraform.tfvars`, and `vars.tf` files in the root directory like before. However, we have included an additional `modules` directory, which contains a `resource_group` subdirectory that contains a `main.tf` file and a `vars.tf` file. Let's look at both.

`modules/resource_group/main.tf` looks like the following:

```
resource "azurerm_resource_group" "rg" {
  name     = var.name
  location = var.location
}
```

It only contains an `azurerm_resource_group` resource with a name and location sourced from the `name` and `location` variables defined in the following `modules/resource_group/vars.tf` file:

```
variable "name" {
  type        = string
  description = "The resource group name"
}
variable "location" {
  type        = string
  description = "The resource group location"
}
```

In the root module, which is the current directory, we've modified the `main.tf` file to look like the following:

```
terraform {
  required_providers {
    ...
  }
}
provider "azurerm" {
  ...
}
module "rg" {
  source  = "./modules/resource_group"
  name     = var.rg_name
```

```
  location = var.rg_location
}
```

As we can see, instead of defining the resource directly in this file, we have defined a module called `rg`, whose `source` is `./modules/resource_group`. Note that we pass the value for the variables defined for the module, that is, `name`, and `location`, from the variables defined at the root level, that is, `var.rg_name` and `var.rg_location`.

Now, let's go ahead and see what happens when we initialize and apply this configuration.

Run the following command to initialize the Terraform workspace:

```
$ terraform init
Initializing the backend...
Initializing modules...
- rg in modules/resource_group
Initializing provider plugins...
...
Terraform has been successfully initialized!
```

As we can see, Terraform has detected the new module and initialized it during init.

> **Tip**
> Whenever you define a new module, you must always reinitialize Terraform.

Now, let's go ahead and run a plan using the following command:

```
$ terraform plan
Terraform will perform the following actions:
  # module.rg.azurerm_resource_group.rg will be created
  + resource "azurerm_resource_group" "rg" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "terraform-exercise"
    }
Plan: 1 to add, 0 to change, 0 to destroy.
```

As we can see, it will create the resource group. However, this is now a part of the module addressed `module.rg.azurerm_resource_group.rg`. To apply the plan, let's run the following command:

```
$ terraform apply
module.rg.azurerm_resource_group.rg: Creating...
module.rg.azurerm_resource_group.rg: Creation complete after 4s [id=/subscriptions/id/
resourceGroups/terraform-exercise]
```

And the resource group has been created! To destroy the resource group, let's run the following command:

```
$ terraform destroy
```

By using modules, you can simplify infrastructure creation and management, enhance collaboration among teams, and establish a consistent approach to deploying resources in a scalable and maintainable manner.

> **Tip**
> Use Terraform modules to encapsulate and reuse infrastructure configurations, promoting modularity and reusability.

Until now, we've seen Terraform creating and destroying resources, but how does Terraform know what it had created before and what it needs to destroy? Well, it uses a **state file** for that. Let's have a look.

Managing Terraform state

Terraform uses a state file to track what it has deployed and what resources it is managing. The state file is essential as it records all the infrastructure Terraform maintains. If you lose it, Terraform will lose track of what it has done so far and start treating resources as new and needing to be created again. Therefore, you should protect your state as code.

Terraform stores state in backends. By default, Terraform stores the state file as `terraform.tfstate` within the `workspace` directory, which is called the local backend. However, that is not the best way of managing the state. There are a couple of reasons why you should not store state in a local system:

- Multiple admins cannot work on the same infrastructure if the state file is stored within someone's local directory
- Local workstations are not backed up; therefore, the risk of losing the state file is high even if you have a single admin doing the job

You might argue that we can resolve these problems by checking the state file into source control with the `.tf` files. Don't do that! State files are plaintext, and if your infrastructure configuration contains sensitive information such as passwords, anyone can see it. Therefore, you need to store a state file securely. Also, storing state files in source control does not provide state locking, resulting in conflicts if multiple people are simultaneously modifying the state file.

> **Tip**
> Never store state files in source control. Use a `.gitignore` file entry to bypass the `terraform.tfstate` file.

The best place to store your Terraform state is on remote cloud storage. Terraform provides a remote backend to store state remotely. There are multiple types of remote backends you can use. When writing

this book, **Azure RM**, **Consul**, **cos**, **gcs**, **http**, **Kubernetes**, **oss**, **pg**, **S3**, and **Remote** were available backends. **Remote** is an enhanced backend type that allows running Terraform `plan` and `apply` within the backend, and only Terraform Cloud and Enterprise support it.

> **Tip**
> While choosing the state storage solution, you should prefer storage with state locking. That will allow multiple people to manipulate the resources without stepping on each other's shoes and causing conflict, as once a state file is locked, others cannot acquire it until the lock is released.

As we're using Azure, we can use Azure Storage to store our state. The advantages are three-fold:

- Your state file is centralized. You can have multiple admins working together and managing the same infrastructure.
- The store is encrypted at rest.
- You get automatic backup, redundancy, and high availability.

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

```
$ cd ~/modern-devops/ch8/terraform-backend/
```

Let's now use the `azurerm` backend and use Azure Storage to persist our Terraform state.

Using the Azure Storage backend

As we will end up in a chicken-or-egg situation if we use Terraform to build a backend to store its state, we will have to configure this bit without using Terraform.

Therefore, let's use the `az` command to configure the storage account in a different resource group that Terraform will not manage.

Creating Azure Storage resources

Let's start by defining a few variables:

- `$ RESOURCE_GROUP=tfstate`
- `$ STORAGE_ACCOUNT_NAME=tfstate$RANDOM`
- `$ CONTAINER_NAME=tfstate`

Create a resource group first using the following command:

```
$ az group create --name $RESOURCE_GROUP --location westeurope
```

Now, let's go ahead and create a storage account within the resource group using the following command:

```
$ az storage account create --resource-group $RESOURCE_GROUP \
--name $STORAGE_ACCOUNT_NAME --sku Standard_LRS \
--encryption-services BLOB
```

The next step is to fetch the account key using the following command:

```
$ ACCOUNT_KEY=$(az storage account keys list \
--resource-group tfstate --account-name $STORAGE_ACCOUNT_NAME \
--query '[0].value' -o tsv)
```

Now, we can go ahead and create a Blob Storage container using the following command:

```
$ az storage container create --name $CONTAINER_NAME \
--account-name $STORAGE_ACCOUNT_NAME --account-key $ACCOUNT_KEY
```

If we receive a `created` response, the storage account is created and ready for use. Now, we can go and define the backend configuration file in Terraform.

Creating a backend configuration in Terraform

Before we create the backend, we will need the `STORAGE_ACCOUNT_NAME` value. To get this, run the following command:

```
$ echo $STORAGE_ACCOUNT_NAME
tfstate28099
```

To create the backend configuration in Terraform, create a file called `backend.tf` within the workspace:

```
terraform {
  backend "azurerm" {
    resource_group_name   = "tfstate"
    storage_account_name  = "tfstate28099"
    container_name        = "tfstate"
    key                   = "example.tfstate"
  }
}
```

In the backend configuration, we've defined the `resource_group_name` backend where the Blob Storage instance exists – `storage_account_name`, `container_name`, and `key`. The `key` attribute specifies the filename that we will use to define the state of this configuration. There might be multiple projects that you are managing using Terraform, and all of them will need separate state files. Therefore, the `key` attribute defines the state file's name that we will use for our project. That allows multiple Terraform projects to use the same Azure Blob Storage to store the state.

> **Tip**
>
> Always use the name of the project as the name of the key. For example, if your project name is `foo`, name the key `foo.tfstate`. That will prevent potential conflicts with others and also allow you to locate your state file quickly.

To initialize the Terraform workspace with the new backend configuration, run the following command:

```
$ terraform init
Initializing the backend...
Backend configuration changed!
Terraform has detected that the configuration specified for the backend has changed.
Terraform will now check for existing state in the backends.
Successfully configured the backend azurerm! Terraform will automatically use this backend
unless the backend configuration changes.
```

When we initialize that, Terraform detects that the backend has changed and checks whether anything is available in the existing backend. If it finds something, it asks whether we want to migrate the current state to the new backend. If it does not, it automatically switches to the new backend, as we see here.

Now, let's go ahead and use the `terraform plan` command to run a plan:

```
$ terraform plan
Acquiring state lock. This may take a few moments...
Terraform will perform the following actions:
  # azurerm_resource_group.rg will be created
  + resource "azurerm_resource_group" "rg" {
    ...
    }
Plan: 1 to add, 0 to change, 0 to destroy.
```

So, as we see, `terraform plan` tells us that it will create a new resource group called `terraform-exercise`. Let's apply the configuration, and this time with an `auto-approve` flag so that the plan does not run again, and Terraform immediately applies the changes using the following command:

```
$ terraform apply -auto-approve
Acquiring state lock. This may take a few moments...
azurerm_resource_group.rg: Creating...
azurerm_resource_group.rg: Creation complete after 2s [id=/subscriptions/id/
resourceGroups/terraform-exercise]
Releasing state lock. This may take a few moments...
```

We now have the resource created successfully.

Now, let's go to Azure Blob Storage and see whether we have a `tfstate` file there, as shown in the following screenshot:

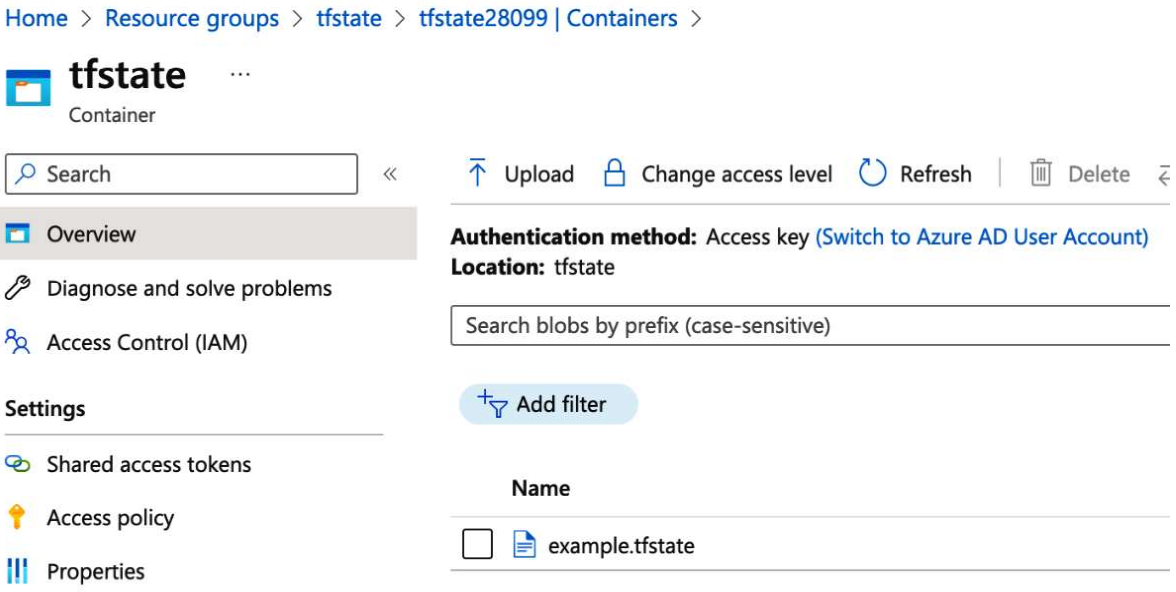

Figure 8.2 – Terraform state

As we see, we have a file called `example.tfstate` within the blob container. That is how remote storage works, and now anyone with access to the Blob Storage instance can use the Terraform configuration and make changes.

So far, we've been managing resources using the default workspace, but what if there are multiple environments that you need to control using the same configuration? Well, Terraform offers workspaces for those scenarios.

Terraform workspaces

Software development requires multiple environments. You develop software within your workspace, deploy it into the development environment, unit test it, and then promote the tested code to a test environment. Your QA team will test the code extensively in the test environment, and once all test cases pass, you can promote your code to production.

That means you must maintain a similar infrastructure in all environments. With an IaC tool such as Terraform, infrastructure is represented as code, and we must manage our code to fit multiple environments. But Terraform isn't just code; it also contains state files, and we must maintain state files for every environment.