Terraform requires that the user uses its special language called HCL, which stands for Hashicorp Configuration Language. Create a folder called terraform-example where the HCL files will live, then change directories to that folder. Terraform providers will need to be defined and installed to use certain types of resources. Providers are easily downloaded and installed with a few lines of HCL and a single command. Create a file called versions.tf where providers will be defined and add the following code:

```
terraform {

  required_providers {
    aws = {
      source = "hashicorp/aws"
    }
provider "aws" {
  region = "us-east-2"
  access_key = "<your_aws_access_key>"
  secret_key = "<your_aws_secret_key>"
}
```

Be sure to replace <your_aws_access_key> and <your_aws_secret_key> with the keys for your account. Now that the required provider is defined, it can be installed by running the command **terraform init**. Ensure that the command is run in the same folder that versions.tf is in. The command should print something like what's below, which lets you know that Terraform is ready to begin creating AWS resources:

```
# terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency
lock file
- Installing hashicorp/aws v3.32.0...
- Installed hashicorp/aws v3.32.0 (signed by HashiCorp)

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running
"terraform plan" to see
any changes that are required for your infrastructure. All
Terraform commands
should now work.
```

```
If you ever set or change modules or backend configuration for
Terraform,
rerun this command to reinitialize your working directory. If
you forget, other
```

Note that a folder has been created alongside versions.tf called .terraform. This folder is where the installed providers are stored to be used for later Terraform processes. Now that the prerequisites to run Terraform are out of the way, the AWS resource definitions can be created. Add a file alongside versions.tf called variables.tf. This file will contain the definition for a single variable that will be passed in on the command line later when resources will be scaled. Add the following to variables.tf:

```
variable "app_count" {
  type = number
  default = 1
}
```

Save and close the file. Create another file called main.tf in the same directory as variables.tf where the resource definitions for the AWS resources will live. Start by adding a data block for AWS availability zones like so:

```
data "aws_availability_zones" "available_zones" {
  state = "available"
}
```

This block will grab availability zones that are available to your account. These will be used for other resource definitions, and to keep a small footprint, only two availability zones will be used. It's best practice to use multiple availability zones when deploying tasks to an AWS ECS Fargate cluster because Fargate will ensure high availability by spreading tasks of the same type as evenly as possible between availability zones. Next, add the resource definition to main.tf with this code:

```
resource "aws_vpc" "default" {
  cidr_block = "10.32.0.0/16"
}
```

Resources that will be created will be defined inside of the VPC. An AWS VPC provides logical isolation of resources from one another. All of the resources that will be defined

will live within the same VPC. Four subnets will be created next. Two will be public and the other two will be private, where each availability zone will have one of each. Add the subnet resource definitions to main.tf:

```
resource "aws_subnet" "public" {
  count                   = 2
  cidr_block              =
cidrsubnet(aws_vpc.default.cidr_block, 8, 2 + count.index)
  availability_zone       =
data.aws_availability_zones.available_zones.names[count.index]
  vpc_id                  = aws_vpc.default.id
  map_public_ip_on_launch = true
}

resource "aws_subnet" "private" {
  count             = 2
  cidr_block        = cidrsubnet(aws_vpc.default.cidr_block, 8,
count.index)
  availability_zone =
data.aws_availability_zones.available_zones.names[count.index]
  vpc_id            = aws_vpc.default.id
}
```

Things that should be public-facing, such as a load balancer, will be added to the public subnet. Other things that don't need to communicate with the internet directly, such as a Hello World service defined inside an ECS cluster, will be added to the private subnet. Define six networking resources with the following blocks of HCL:

```
resource "aws_internet_gateway" "gateway" {
  vpc_id = aws_vpc.default.id
}

resource "aws_route" "internet_access" {
  route_table_id         = aws_vpc.default.main_route_table_id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = aws_internet_gateway.gateway.id
}

resource "aws_eip" "gateway" {
```

```
  count      = 2
  vpc        = true
  depends_on = [aws_internet_gateway.gateway]
}

resource "aws_nat_gateway" "gateway" {
  count         = 2
  subnet_id     = element(aws_subnet.public.*.id, count.index)
  allocation_id = element(aws_eip.gateway.*.id, count.index)
}

resource "aws_route_table" "private" {
  count  = 2
  vpc_id = aws_vpc.default.id

  route {
    cidr_block = "0.0.0.0/0"
    nat_gateway_id = element(aws_nat_gateway.gateway.*.id,
count.index)
  }
}

resource "aws_route_table_association" "private" {
  count          = 2
  subnet_id      = element(aws_subnet.private.*.id,
count.index)
  route_table_id = element(aws_route_table.private.*.id,
count.index)
}
```

These six resources handle networking and communication to and from the internet outside of the VPC. The internet gateway, for example, is what allows communication between the VPC and the internet at all. The NAT gateway allows resources within the VPC to communicate with the internet but will prevent communication to the VPC from outside sources. That is all tied together with the route table association, where the private route table that includes the NAT gateway is added to the private subnets defined earlier. Security groups will need to be added next to allow or reject traffic in a more fine-grained way both from the load balancer and the application service. Add the load balancer security group resource to main.tf like so:

```
resource "aws_security_group" "lb" {
  name        = "example-alb-security-group"
  vpc_id      = aws_vpc.default.id

  ingress {
    protocol    = "tcp"
    from_port   = 80
    to_port     = 80
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

The load balancer's security group will only allow traffic to the load balancer on port 80, as defined by the ingress block within the resource block. Traffic from the load balancer will be allowed to anywhere on any port with any protocol with the settings in the egress block. Add the three resources for the load balancer next with the following code:

```
resource "aws_lb" "default" {
  name            = "example-lb"
  subnets         = aws_subnet.public.*.id
  security_groups = [aws_security_group.lb.id]
}

resource "aws_lb_target_group" "hello_world" {
  name        = "example-target-group"
  port        = 80
  protocol    = "HTTP"
  vpc_id      = aws_vpc.default.id
  target_type = "ip"
}
```

```
resource "aws_lb_listener" "hello_world" {
  load_balancer_arn = aws_lb.default.id
  port              = "80"
  protocol          = "HTTP"

  default_action {
    target_group_arn = aws_lb_target_group.hello_world.id
    type             = "forward"
  }
}
```

The first block defines the load balancer itself and attaches it to the public subnet in each availability zone with the load balancer security group. The target group, when added to the load balancer listener, tells the load balancer to forward incoming traffic on port 80 to wherever the load balancer is attached. In this case, it will be the ECS service defined later. Define the ECS cluster with the block below:

```
resource "aws_ecs_task_definition" "hello_world" {
  family                   = "hello-world-app"
  network_mode             = "awsvpc"
  requires_compatibilities = ["FARGATE"]
  cpu                      = 1024
  memory                   = 2048

  container_definitions = <<DEFINITION
[
  {
    "image": "heroku/nodejs-hello-world",
    "cpu": 1024,
    "memory": 2048,
    "name": "hello-world-app",
    "networkMode": "awsvpc",
    "portMappings": [
      {
        "containerPort": 3000,
        "hostPort": 3000
      }
    ]
```

```
  }
]
DEFINITION
}
```

The task definition defines how the hello world application should be run. This is where it's specified that the platform will be Fargate rather than EC2, so that managing EC2 instances isn't required. This means that CPU and memory for the running task should be specified. The image used is a simple API that returns "Hello World!" and is available as a public docker image. The Docker container exposes the API on port 3000, so that's specified as the host and container ports. The network mode is set to "awsvpc", which tells AWS that an elastic network interface and a private IP address should be assigned to the task when it runs. Create the security group for the ECS service next with the following HCL:

```
resource "aws_security_group" "hello_world_task" {
  name        = "example-task-security-group"
  vpc_id      = aws_vpc.default.id

  ingress {
    protocol        = "tcp"
    from_port       = 3000
    to_port         = 3000
    security_groups = [aws_security_group.lb.id]
  }

  egress {
    protocol    = "-1"
    from_port   = 0
    to_port     = 0
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

The security group for the application task specifies that it should be added to the default VPC and only allow traffic over TCP to port 3000 of the application. The ingress settings also include the security group of the load balancer as that will allow traffic from the network interfaces that are used with that security group. It allows all outbound traffic of any protocol as seen in the egress settings. Finally, add the ECS service and cluster blocks as shown below:

```
resource "aws_ecs_cluster" "main" {
  name = "example-cluster"
}

resource "aws_ecs_service" "hello_world" {
  name            = "hello-world-service"
  cluster         = aws_ecs_cluster.main.id
  task_definition = aws_ecs_task_definition.hello_world.arn
  desired_count   = var.app_count
  launch_type     = "FARGATE"

  network_configuration {
    security_groups = [aws_security_group.hello_world_task.id]
    subnets         = aws_subnet.private.*.id
  }

  load_balancer {
    target_group_arn = aws_lb_target_group.hello_world.id
    container_name   = "hello-world-app"
    container_port   = 3000
  }

  depends_on = [aws_lb_listener.hello_world]
}
```

The ECS service specifies how many tasks of the application should be run with the task_definition and desired_count properties within the cluster. The launch type is Fargate so that no EC2 instance management is required. The tasks will run in the private subnet as specified in the network_configuration block and will be reachable from the outside world through the load balancer as defined in the load_balancer block. Finally, the service shouldn't be created until the load balancer has been, so the load balancer listener is included in the depends_on array.

One final step remains in the Terraform configuration to make the deployed resources easier to test. You'll know that everything is running properly if the application running on ECS returns a blank page with the text "Hello World!". To reach the service, the URL of the load balancer is required. You could find it on the AWS dashboard, but Terraform

can make it easier. Add a file called outputs.tf in the same directory as main.tf, then add the following code:

```
output "load_balancer_ip" {
  value = aws_lb.default.dns_name
}
```

This file will be included in the Terraform configuration when commands are run, and the output will instruct Terraform to print the URL of the load balancer when the plan has been applied. With the entire Terraform configuration complete, run the command terraform plan -out="tfplan" to see what will be created when the configuration is applied. It should look something like this:

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_ecs_cluster.main will be created
  + resource "aws_ecs_cluster" "main" {
      + arn  = (known after apply)
      + id   = (known after apply)

...

      + main_route_table_id              = (known after apply)
      + owner_id                         = (known after apply)
    }

Plan: 23 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + load_balancer_ip = (known after apply)

------------------------------------------------------------------
---------

This plan was saved to: tfplan
```

```
To perform exactly these actions, run the following command to
apply:
    terraform apply "tfplan"
```

If you're satisfied with the plan, apply the configuration to AWS by running terraform apply "tfplan". This step will likely take a few minutes. Once Terraform is done applying the plan, the bottom of the output should look like the text below:

```
Apply complete! Resources: 23 added, 0 changed, 0 destroyed.

The state of your infrastructure has been saved to the path
below. This state is required to modify and destroy your
infrastructure, so keep it safe. To inspect the complete state
use the `terraform show` command.

State path: terraform.tfstate

Outputs:

load_balancer_ip =
"example-lb-1284172108.us-east-2.elb.amazonaws.com"
```