# How to Deploy a Dockerised Application on AWS ECS With Terraform

Go from creating a simple Node app to having it containerized, load-balanced, and deployed

Andrew Bestbier Follow
Mar 18, 2020 · 7 min read ★

Photo by Pixabay from Pexels

In this post, I will guide you through the process of deploying a Node app on AWS ECS with Terraform.

## Guide Overview

We will follow these steps:

1. Create a simple Node app and run it locally.
2. Dockerize the Node app.
3. Create an image repository on AWS ECR and push the image.
4. Create an AWS ECS cluster.
5. Create an AWS ECS task.
6. Create an AWS ECS service.
7. Create a load balancer.

The technologies used in this guide are:

- Amazon ECS — a fully managed container orchestration service
- Amazon ECR — a fully-managed Docker container registry
- Terraform — an open-source infrastructure as code tool

## Prerequisites

- An AWS account
- Node installed
- Docker installed and some experience using it
- Terraform installed

## Step 1. Create a Simple Node App

First, run the following commands to create and navigate to our application's directory:

```
$ mkdir node-docker-ecs
$ cd node-docker-ecs
```

Next, create an `npm` project:

```
$ npm init --y
```

Install Express:

```
$ npm install express
```

Create an `index.js` file with the following code:

```
1  const express = require('express')
2  const app = express()
3  const port = 3000
4
5  app.get('/', (req, res) => res.send('Hello World!'))
6
7  app.listen(port, () => console.log(`Example app listening on port ${port}!`))
```

index.js hosted with ❤ by GitHub — view raw

The app can then run with this command:

```
$ node index.js
```

You should see your app at `http://localhost:3000/` :


## Step 2. Dockerize the Node App

If you are new to Docker, I highly recommend this course by Stephen Grider or the official getting started guide. I promise that they are well worth your time.

Create a `Dockerfile` in your project directory and populate it with the following code:

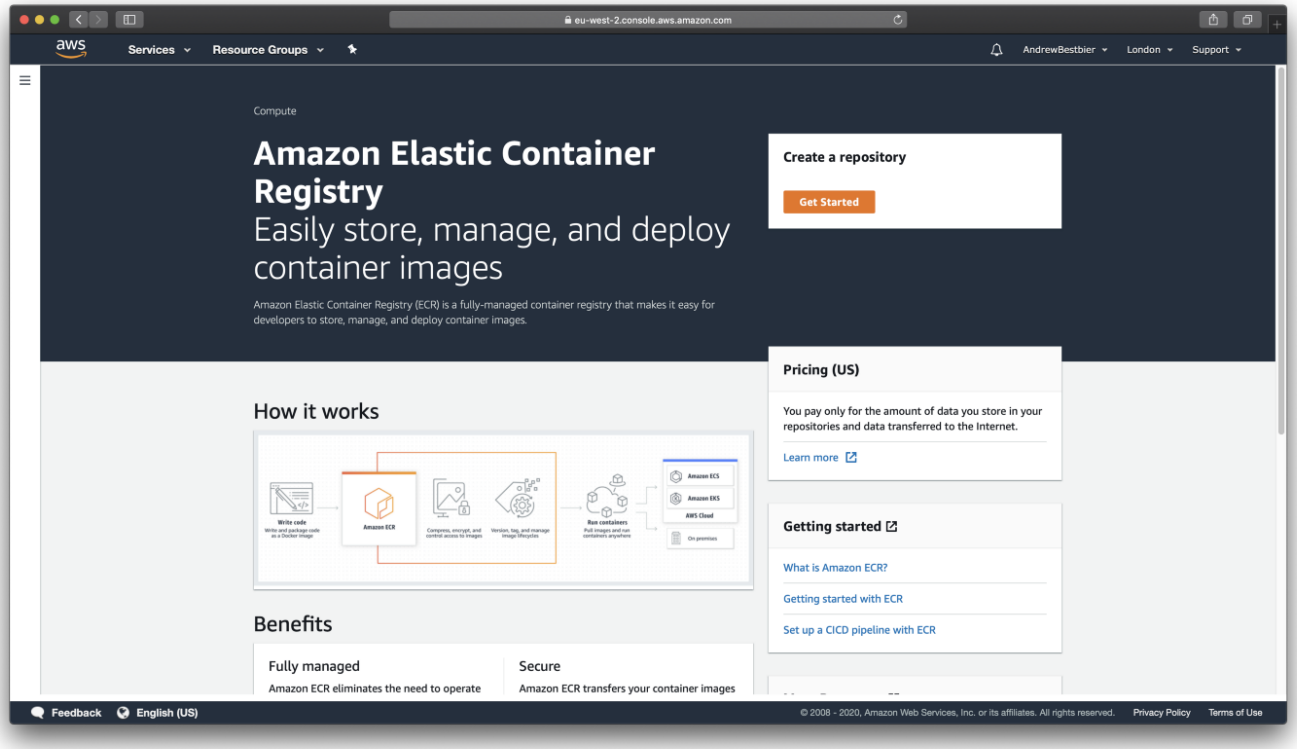
```
# Use an official Node runtime as a parent image
FROM node:12.7.0-alpine

# Set the working directory to /app
WORKDIR '/app'

# Copy package.json to the working directory
COPY package.json .

# Install any needed packages specified in package.json
RUN yarn

# Copying the rest of the code to the working directory
COPY . .
```

```
15
16  # Make port 3000 available to the world outside this container
17  EXPOSE 3000
18
19  # Run index.js when the container launches
20  CMD ["node", "index.js"]
```

Dockerfile hosted with ❤️ by GitHub view raw

## Step 3. Push the Node App to AWS ECR

Now it is time we pushed our container to a container registry service — in this case, we will use AWS ECR:




Instead of using the AWS UI, we will use terraform to create our repository. In your directory create a file called `main.tf`. Populate your file with the following commented code:

```
1  provider "aws" {
2    version = "~> 2.0"
```

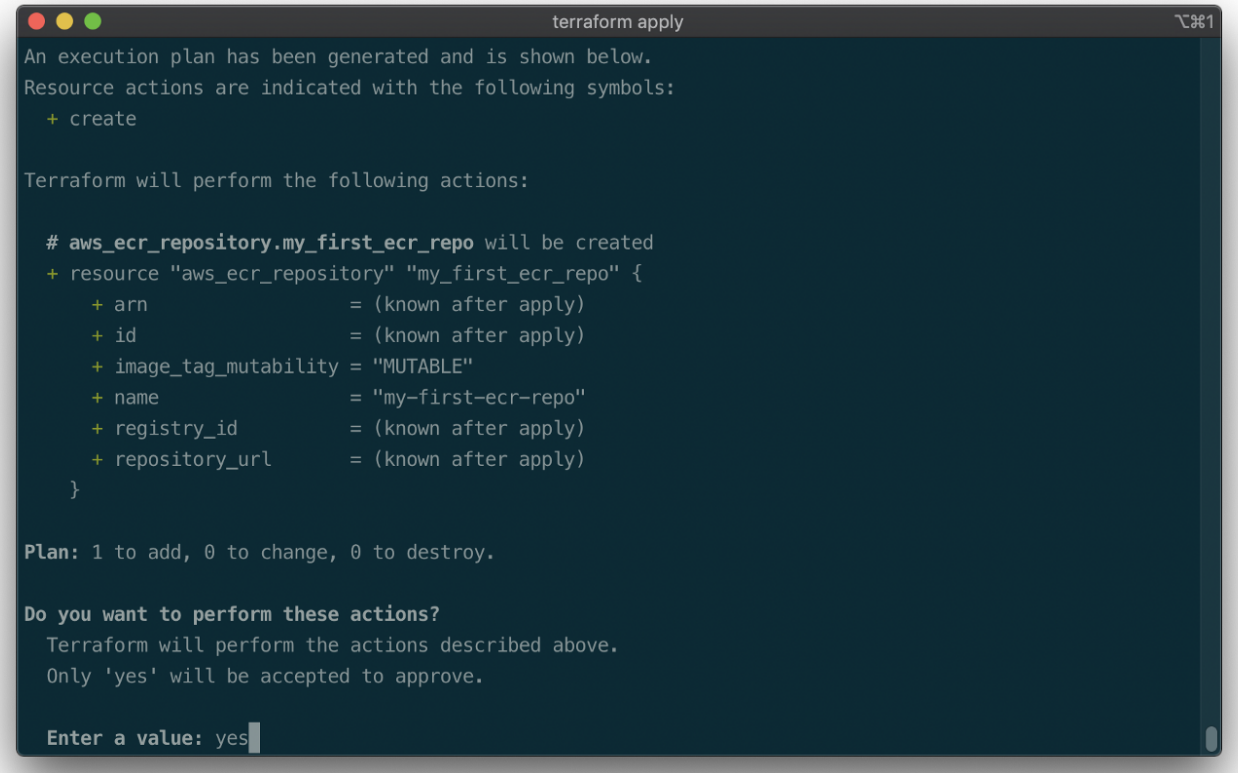
```
  region  = "eu-west-2" # Setting my region to London. Use your own region here
}

resource "aws_ecr_repository" "my_first_ecr_repo" {
  name = "my-first-ecr-repo" # Naming my repository
}
```

tutorial.tf hosted with ❤️ by GitHub view raw

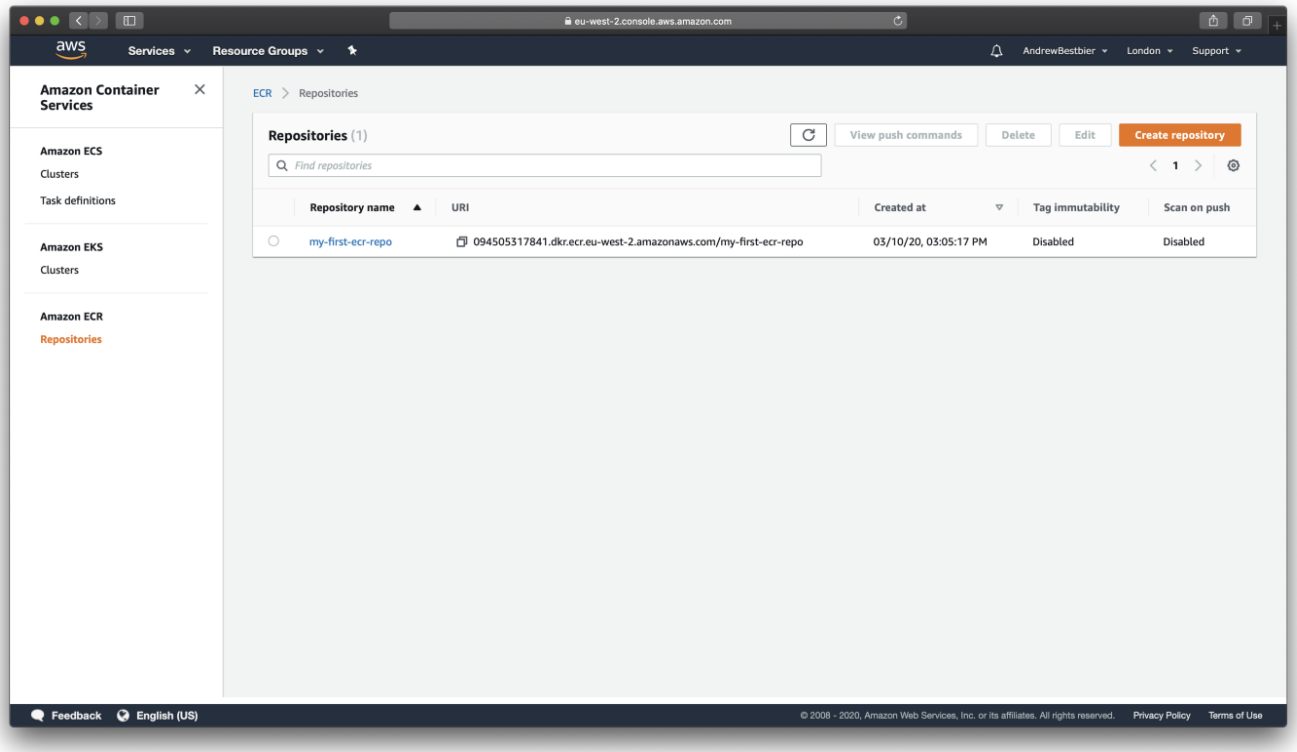
Next, in your terminal, type:

```
terraform init
terraform apply
```

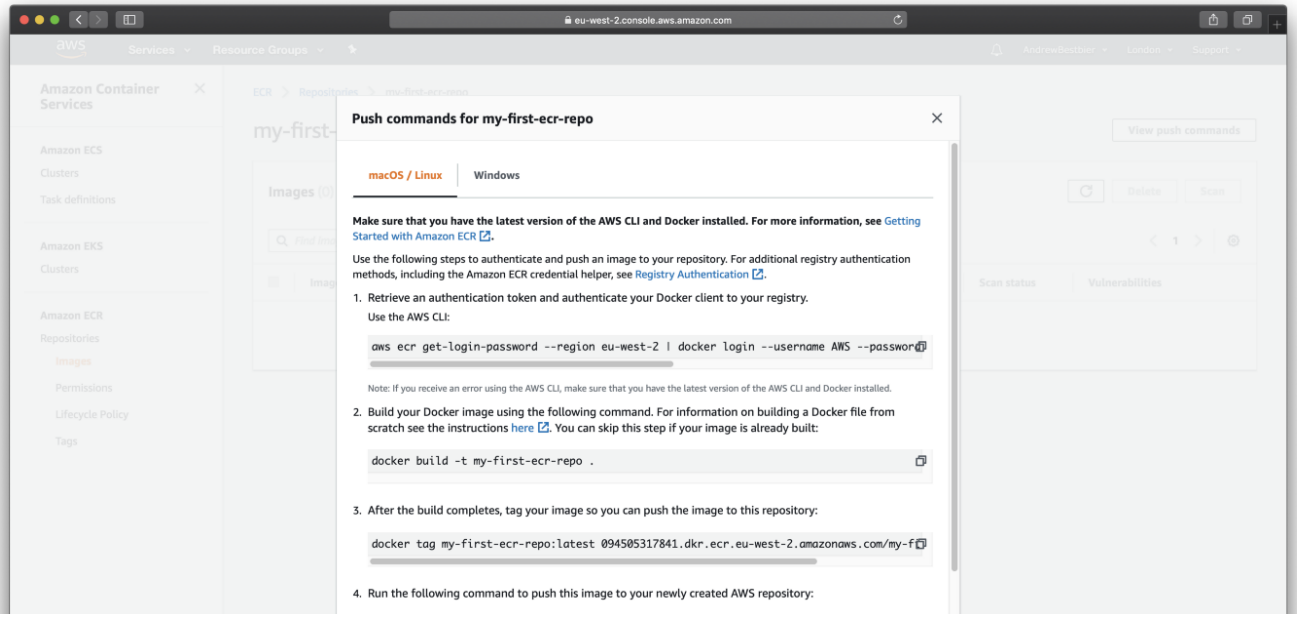
You will then be shown an execution plan with the changes terraform will make on AWS. Type *yes*:



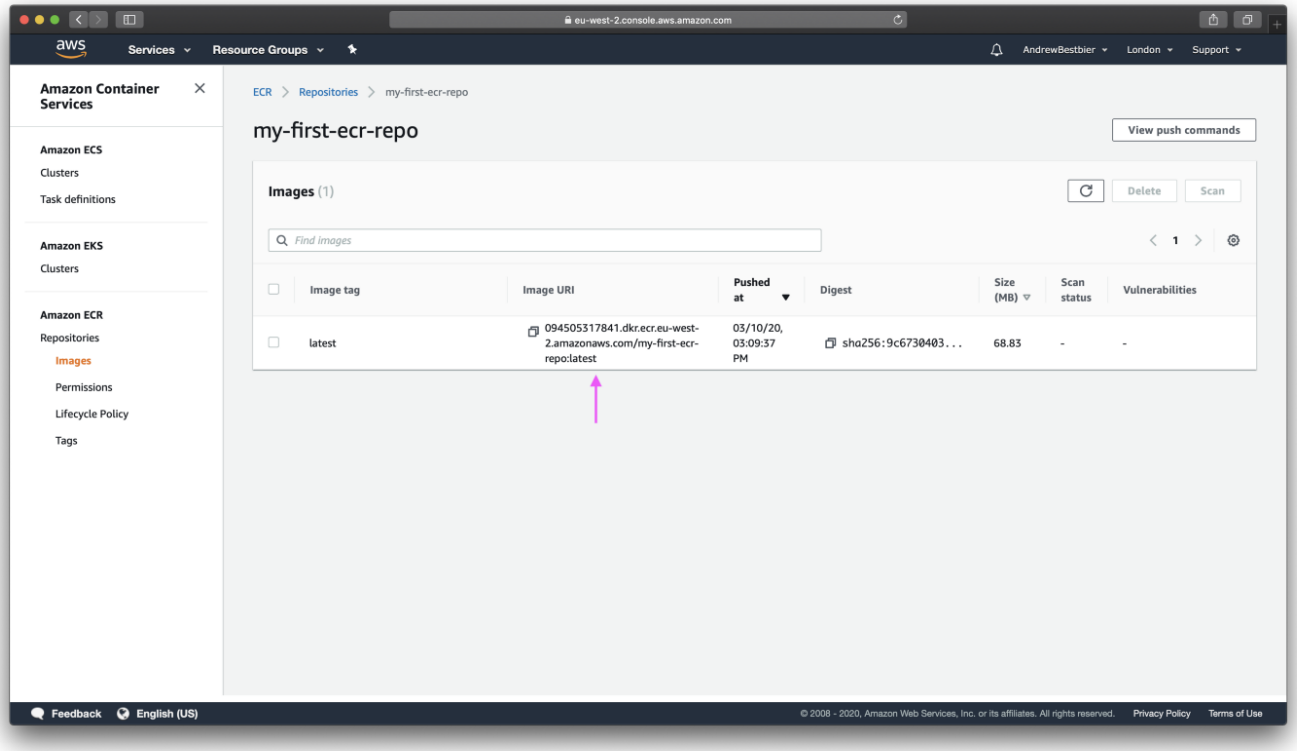
We will be using the `terraform apply` command repeatedly throughout this tutorial to deploy our changes. If you then navigate to the AWS ECR service, you should see your newly created repository:




Now we can push our Node application image up to this repository. Click on the repository and click View push commands. A modal will appear with four commands you need to run locally in order to have your image pushed up to your repository:




Once you have run these commands, you should see your pushed image in your repository:




## AWS ECS

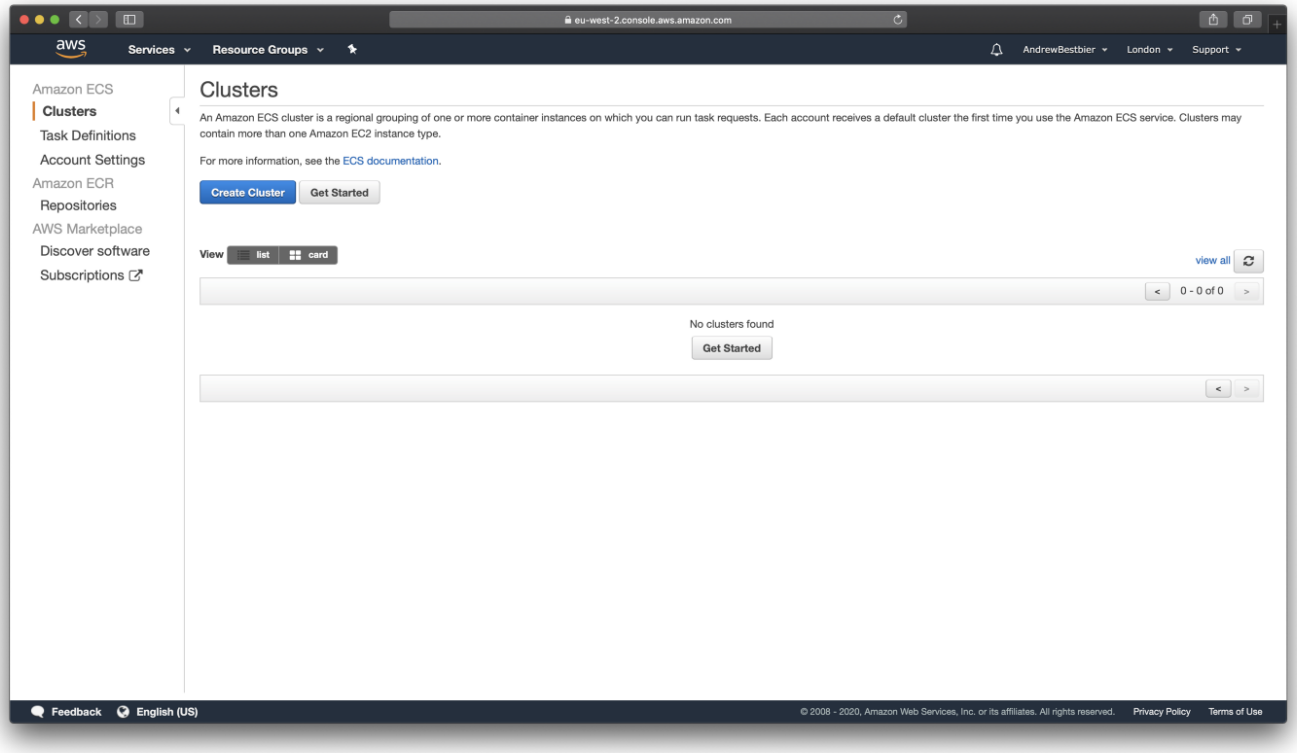
Amazon Elastic Container Service (Amazon ECS) is a fully managed container orchestration service. AWS ECS is a fantastic service for running your containers. In this guide we will be using ECS Fargate, as this is a serverless compute service that allows you to run containers without provisioning servers.

ECS has three parts: clusters, services, and tasks.

**Tasks** are JSON files that describe how a container should be run. For example, you need to specify the ports and image location for your application. A **service** simply runs a specified number of tasks and restarts/kills them as needed. This has similarities to an auto-scaling group for EC2. A **cluster** is a logical grouping of services and tasks. This will become more clear as we build.

## Step 4. Create the Cluster

Navigate to the AWS ECS service and click **Clusters**. You should see the following empty page:




Next, add this code to your terraform file and redeploy your infrastructure with `terraform apply`:

You should then see your new cluster:

## Step 5. Create the First Task

Creating a task is a bit more involved than creating a cluster. Add the following commented code to your terraform script:

Notice how we specify the image by referencing the repository URL of our other terraform resource. Also notice how we provide the port mapping of 3000. We also create an IAM role so that tasks have the correct permissions to execute. If you click Task Definitions in AWS ECS, you should see your new task:

## Step 6. Create the First Service

Great! Now we have a cluster and a task definition. It is time that we spun up a few containers in our cluster through the creation of a service that will use our newly created task definition as a blueprint. If we examine the documentation in Terraform for an ECS service, we find that we need the following terraform code as a minimum:

However, if you try to deploy this, you will get the following error:

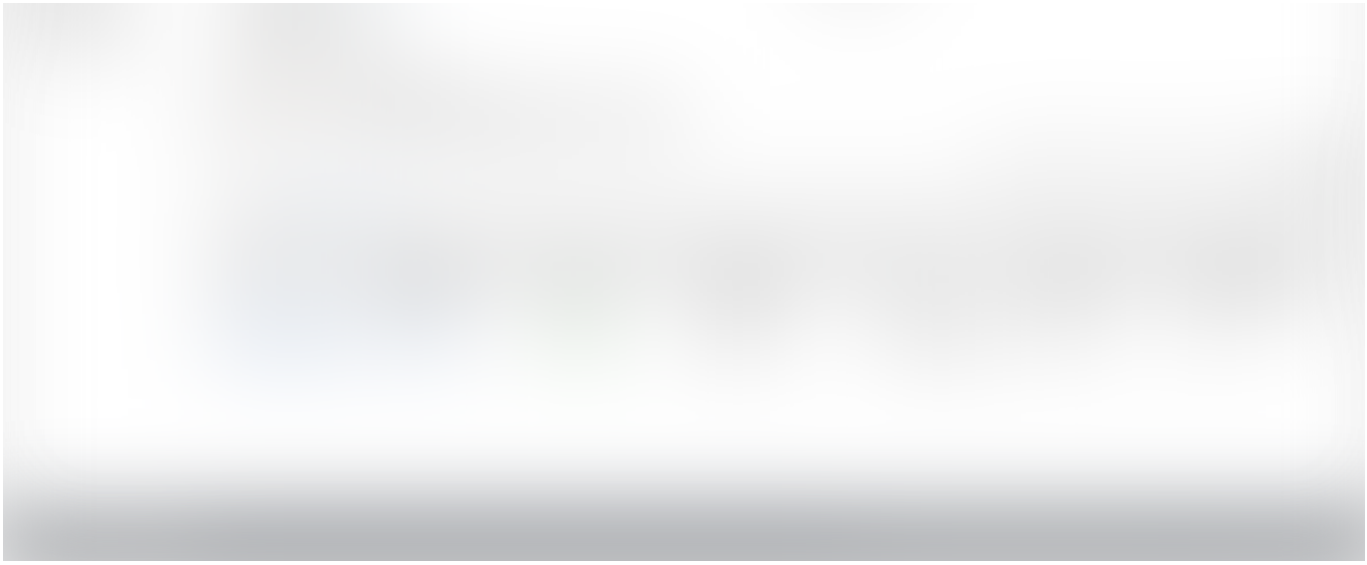
```
Network Configuration must be provided when networkMode 'awsvpc' is
specified
```

As we are using Fargate, our tasks need to specify that the network mode is awsvpc. As a result, we need to extend our service to include a network configuration. You may have not known it yet, but our cluster was automatically deployed into your account's default VPC. However, for a service, this needs to be explicitly stated, even if we wish to continue using the default VPC and subnets. First, we need to create reference resources to the default VPC and subnets so that they can be referenced by our other resources:

Next, adjust your service to reference the default subnets:

Once deployed, click on your cluster, and you should then see your service:

If you click on your service and the Tasks tab, you should also see that three tasks/containers have been spun up:



## Step 7. Create a Load Balancer

The final step in this process is to create a load balancer through which we can access our containers. The idea is to have a single URL provided by our load balancer that, behind the scenes, will redirect our traffic to our underlying containers. Add the following commented terraform code:

Note how we also create a security group for the load balancer. This security group is used to control the traffic allowed to and from the load balancer. If you deploy your code, navigate to EC2, and click Load balancers, you should see the following:

Note that if you click on the URL (pink arrow above), you will see an error, as you have not specified where the traffic should be directed:

To direct traffic we need to create a target group and listener. Each target group is used to route requests to one or more registered targets (in our case, containers). When you create each listener rule, you specify a target group and conditions. Traffic is then forwarded to the corresponding target group. Create these with the following:

If you view the Listeners tab of your load balancer, you should see a listener that forwards traffic to your target group:

If you click on your target group and then click the Targets tag, you will see a message saying, “There are no targets registered to this target group”:

This is because we have not linked our ECS service to our load balancer. We can change this by altering our service code to reference the targets:

Then if you refresh the Targets tab, you should see your three containers:

Note how the status of each container is unhealthy. This is because the ECS service does not allow traffic in by default. We can change this by creating a security group for the ECS service that allows traffic only from the application load balancer security group:

If you check your target containers, they should now be healthy:

You should also be able to access your containers through your load balancer URL:

## Conclusion

Congratulations on making it this far! I hope you learned a lot. See the final code on GitHub. Please let me know your thoughts and feedback, and best of luck with your AWS journey.

👋 **Join us today !!**

Follow us on LinkedIn, Twitter, Facebook, and Instagram


https://avmconsulting.net/

If this post was helpful, please click the clap 👏 button below a few times to show your support! ⬇

---

## Sign up for AVM Consulting

By AVM Consulting Blog

We are developing blogging to help community with Cloud/DevOps services Take a look.

Get this newsletter

Emails will be sent to johnkraus3@gmail.com.
Not you?

AWS JavaScript Docker Containers Programming