# Get started with AWS ECS using Terraform.

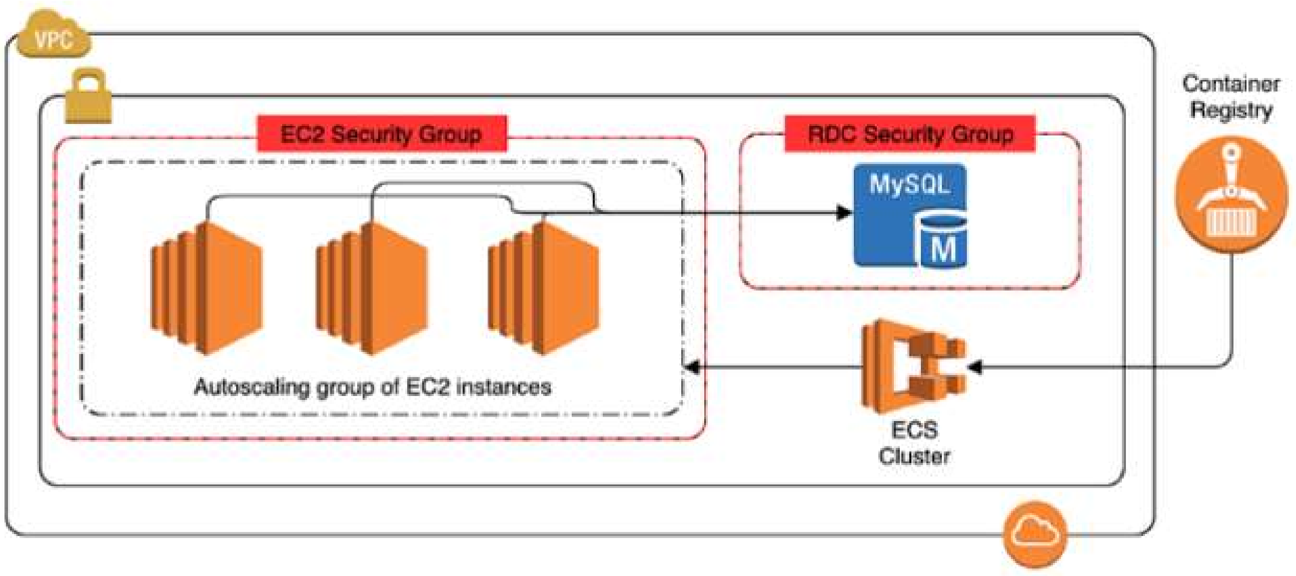
Tim Okito
Oct 13, 2020 · 7 min read

Amazon ECS

Hello world!

Today I will introduce you to Amazon ECS. You may ask "what is ECS?" Amazon ECS is a service for running and maintaining a specified number of task. It is scalable, high-performing container management service that supports Docker containers.

Below is the architecture that we will design.



We will create a VPC (Virtual Private Cloud) which will contain an Autoscaling group with EC2 instances. ECS(Amazon Elastic Container Service) will manage the task that will take place on the EC2 instance based on Docker images stored in ECR (Elastic Container Registry).

Each EC2 instance will serve as a host for a worker that writes something to RDS MySQL. EC2 and MySQL instances will be store in different security group.

Here is a list of all the AWS services that will be part of the building block:

- VPC with a public subnet as an isolated pool for my resources
- Internet Gateway to contact the outer world
- Security groups for RDS MySQL and for EC2s
- Auto-scaling group for ECS cluster with launch configuration
- RDS MySQL instance
- ECR container registry
- ECS cluster with task and service definition

## Terraform State

In order to successfully complete this lab, we must first have a good understanding of the Terraform state and its purpose.

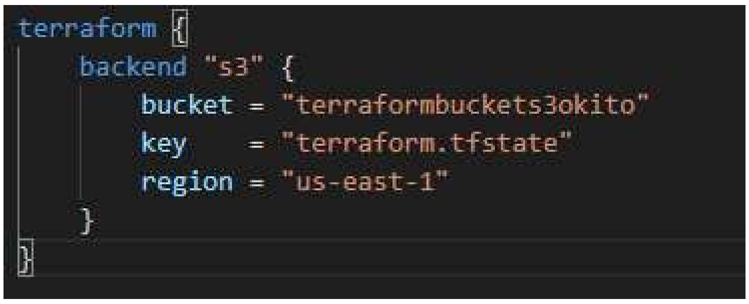
In my opinion the state file is most important file. The state file contains everything in your configuration, including any secrets you might have defined in them. It is the source of truth for the infrastructure being managed.

Although there are some scenarios where the Terraform may be able to function without a state file, it’s not recommended at all.

The state is used by Terraform to map real world resources to your configuration, In a nutshell Terraform looks at what was already provisioned and track the changes in the state file.

It is best practice to store the State file remotely, it helps tremendously when working in a team setting. The ideal location for the state file is an S3 bucket when working with AWS.

Lets get started with the Terraform code now!!!!



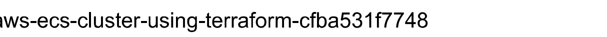
```
terraform {
    backend "s3" {
        bucket = "terraformbuckets3okito"
        key    = "terraform.tfstate"
        region = "us-east-1"
    }
}
```

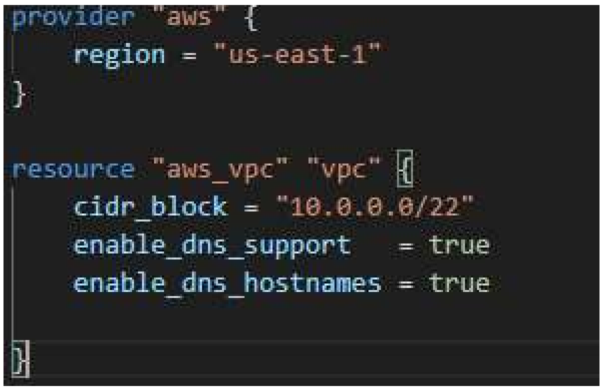
This code will allow Terraform to store the state file in a S3 bucket called "terraformbucketsokito"

In order to keep my login information safe, I will enter AWS Configure to upload my AWS keys without exposing them.

## Virtual Private Cloud

First service we will establish is the Virtual Private Cloud.





```
provider "aws" {
    region = "us-east-1"
}

resource "aws_vpc" "vpc" {
    cidr_block = "10.0.0.0/22"
    enable_dns_support   = true
    enable_dns_hostnames = true

}
```

vpc.tf

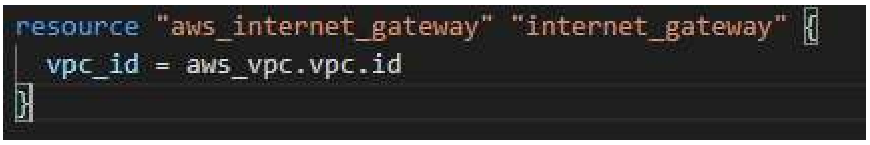
We are going to use AWS as a provider for this lab. The resource that we are creating here is a Virtual Private Cloud.

The Virtual private cloud is the networking layer of the EC2, it allows you to build your own virtual network within AWS. cidr_block here specifies that IPv4 address range of the VPC.

## Internet Gateway

Next we are creating an internet gateway to allow communication between the instances in the VPC and the internet.

We are using aws_vpc.vpc.id in order to get the resource details.



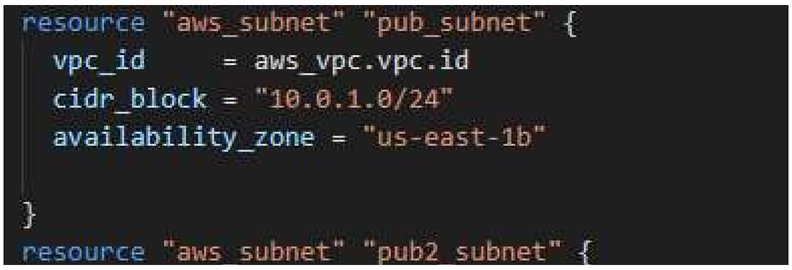
```
resource "aws_internet_gateway" "internet_gateway" {
  vpc_id = aws_vpc.vpc.id
}
```

ig.tf

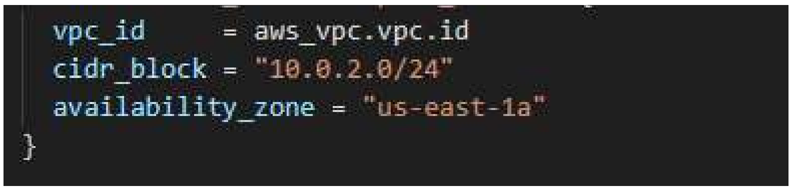
## Subnet

A subnet is a segment of the VPC's IP address range where we are launching the instances. I'm creating 2 subnets here, but they are both in a different Availability zone

Please note that both of the cidr_block are also different, you cannot have the same cidr_block for both of those subnet.



```
resource "aws_subnet" "pub_subnet" {
  vpc_id     = aws_vpc.vpc.id
  cidr_block = "10.0.1.0/24"
  availability_zone = "us-east-1b"

}
resource "aws_subnet" "pub2_subnet" {
```



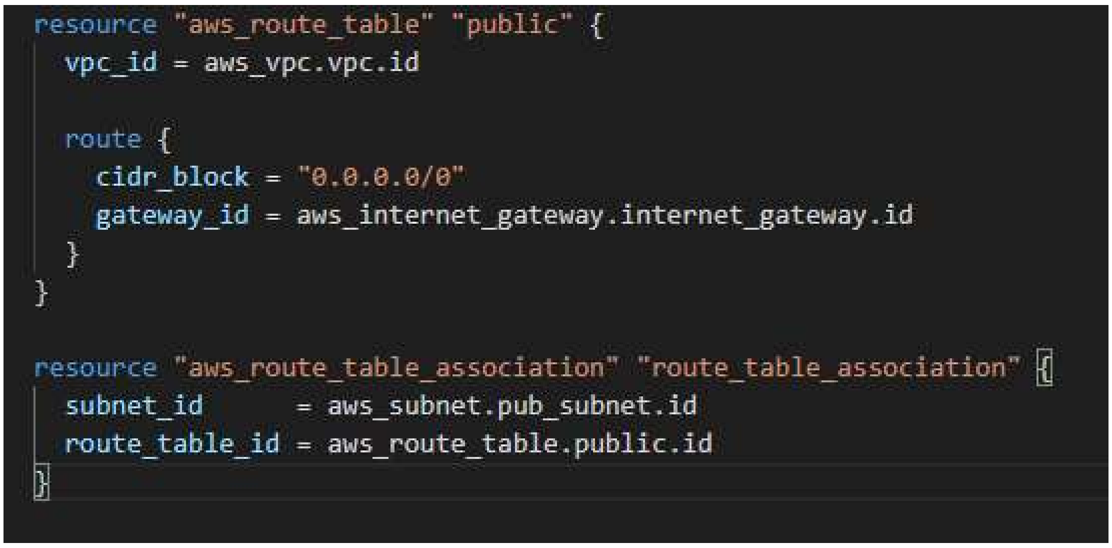
```
  vpc_id     = aws_vpc.vpc.id
  cidr_block = "10.0.2.0/24"
  availability_zone = "us-east-1a"
}
```

subnet.tf

## Route Table

A route table is a logical construct within a VPC that contains a set of rules (called routes) that applied to the subnet and used to determine where network traffic is directed.

By entering (0.0.0.0/0) we are creating a route table that will direct all traffic to the internet gateway and associate this route table with the subnets that we created earlier.



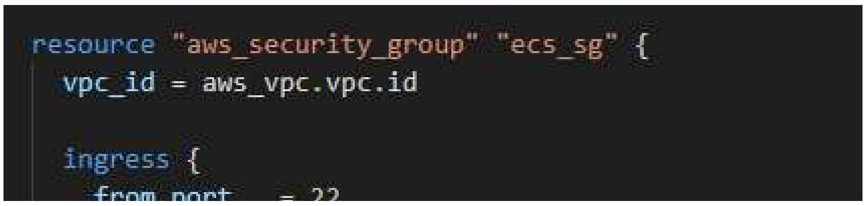
```
resource "aws_route_table" "public" {
  vpc_id = aws_vpc.vpc.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.internet_gateway.id
  }
}

resource "aws_route_table_association" "route_table_association" {
  subnet_id      = aws_subnet.pub_subnet.id
  route_table_id = aws_route_table.public.id
}
```

routetable.tf

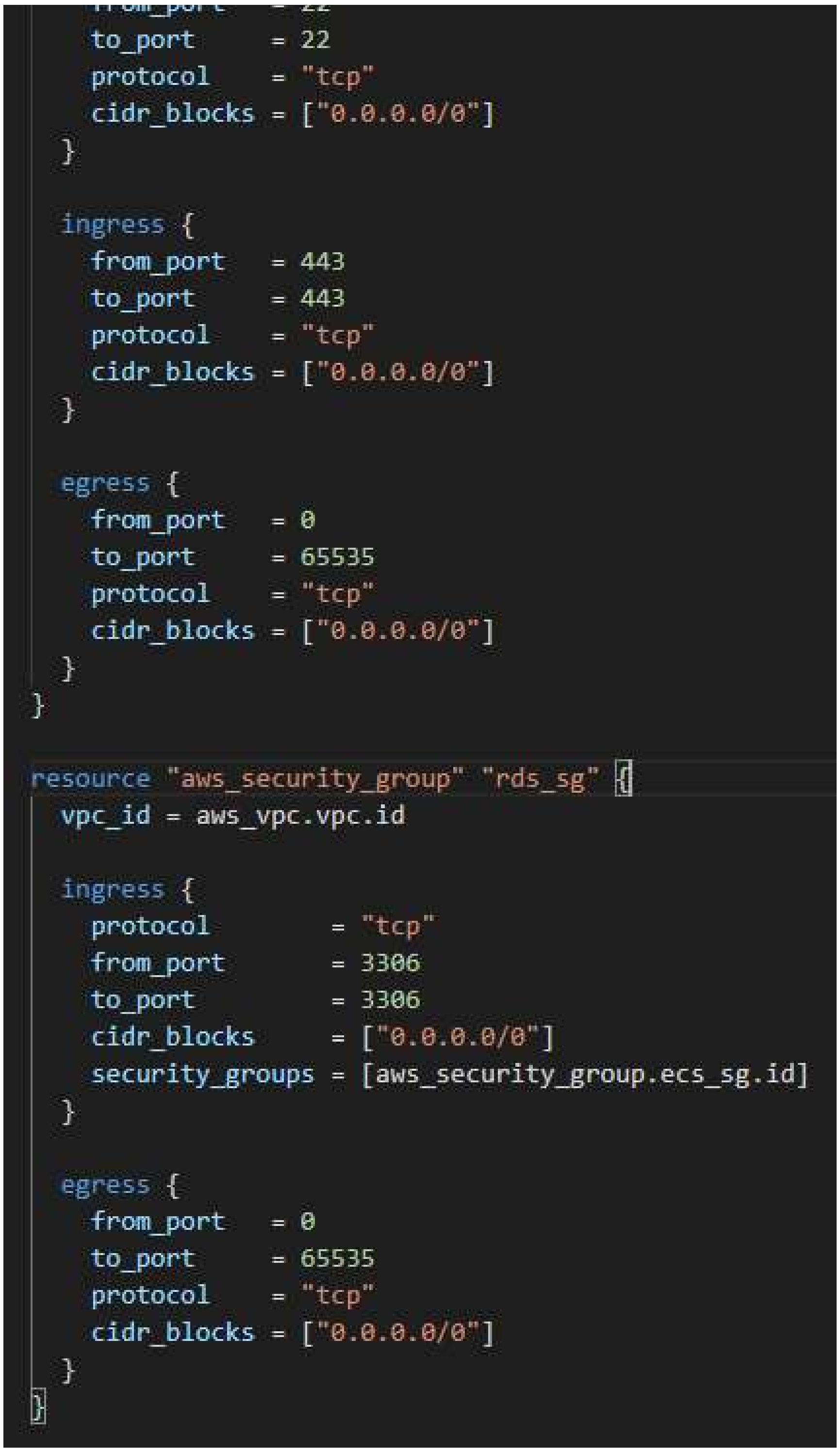
## Security Group

A security group serves as a virtual stateful firewall that controls inbound and outbound network traffic to AWS resources and Amazon EC2 instances. Please note that we are already allowing traffic from the internet to and from the VPC. We have to set some rules in order to secure the instances.

The two instances that we will create today are an EC2 and RDS MySQL. In this case we are going to need two security groups.



```
resource "aws_security_group" "ecs_sg" {
  vpc_id = aws_vpc.vpc.id

  ingress {
    from_port    = 22
```



```
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 65535
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_security_group" "rds_sg" {
  vpc_id = aws_vpc.vpc.id

  ingress {
    protocol        = "tcp"
    from_port       = 3306
    to_port         = 3306
    cidr_blocks     = ["0.0.0.0/0"]
    security_groups = [aws_security_group.ecs_sg.id]
  }

  egress {
    from_port   = 0
    to_port     = 65535
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

sg.tf

The first security group focuses on the EC2 will be stored in the ECS cluster. Inbound traffic is being narrowed to two port : 22 for SSH and 443 for HTTPS in order to download the docker image from ECR.

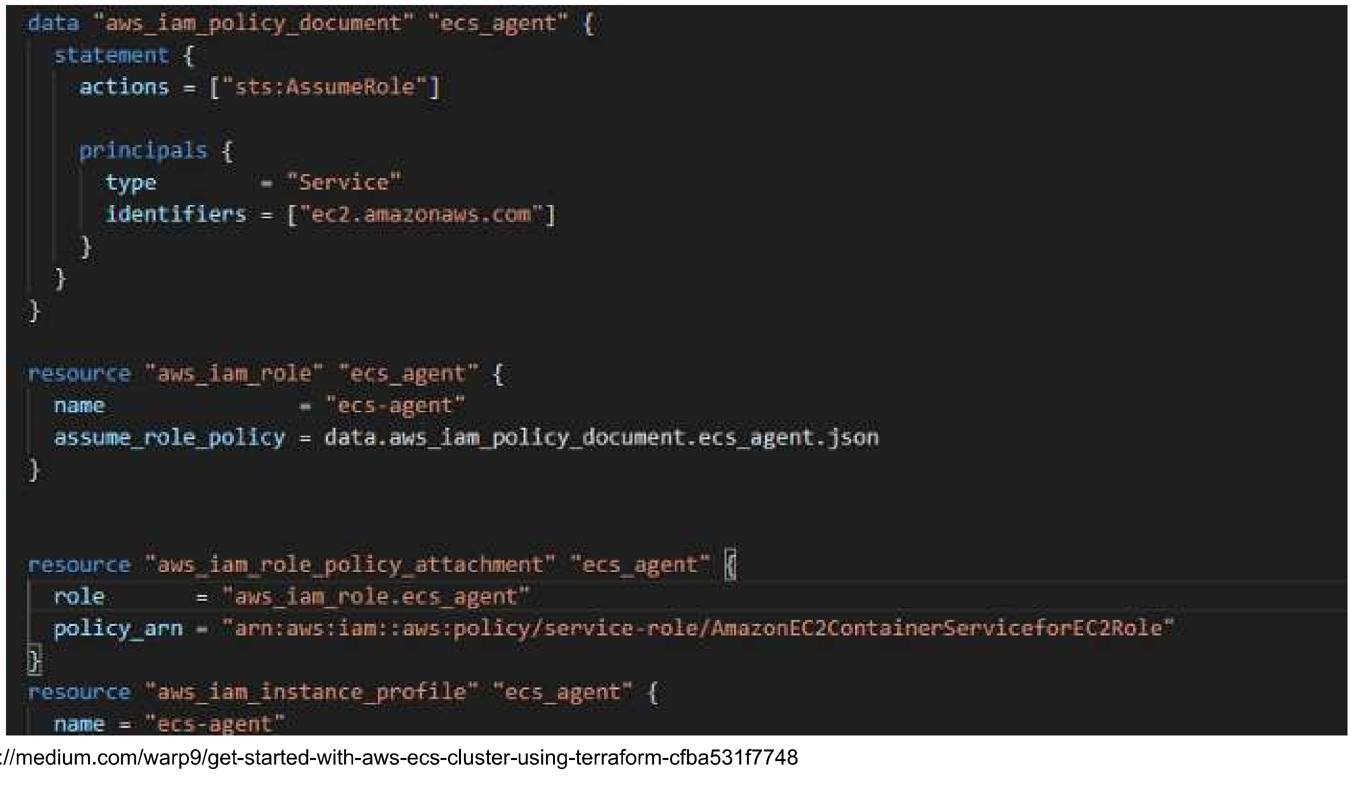
The second security group focuses on RDS, we have only one port here for MySQL which is 3306.

Inbound traffic coming from the internet is open, that's why we have the cidr_block of (0.0.0.0/0). In production environments there should be some limitations within a IP range.

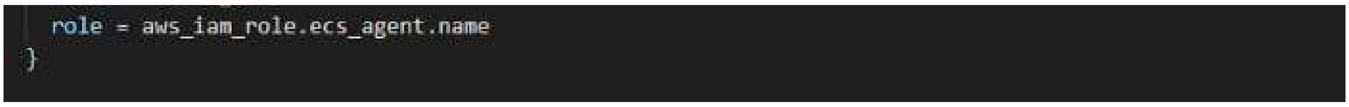
Now with a Security group, Route Table, Subnet and Internet Gateway we are now done with the networking part of the architecture.

## Autoscaling group

An Auto Scaling group is a collection of EC2 instances managed by the Auto Scaling Service. Before we launch our container instances and register them we have to create an IAM role for those instances.



```
data "aws_iam_policy_document" "ecs_agent" {
  statement {
    actions = ["sts:AssumeRole"]

    principals {
      type        = "Service"
      identifiers = ["ec2.amazonaws.com"]
    }
  }
}

resource "aws_iam_role" "ecs_agent" {
  name               = "ecs-agent"
  assume_role_policy = data.aws_iam_policy_document.ecs_agent.json
}


resource "aws_iam_role_policy_attachment" "ecs_agent" {
  role       = "aws_iam_role.ecs_agent"
  policy_arn = "arn:aws:iam::aws:policy/service-role/AmazonEC2ContainerServiceforEC2Role"
}
resource "aws_iam_instance_profile" "ecs_agent" {
  name = "ecs-agent"
```

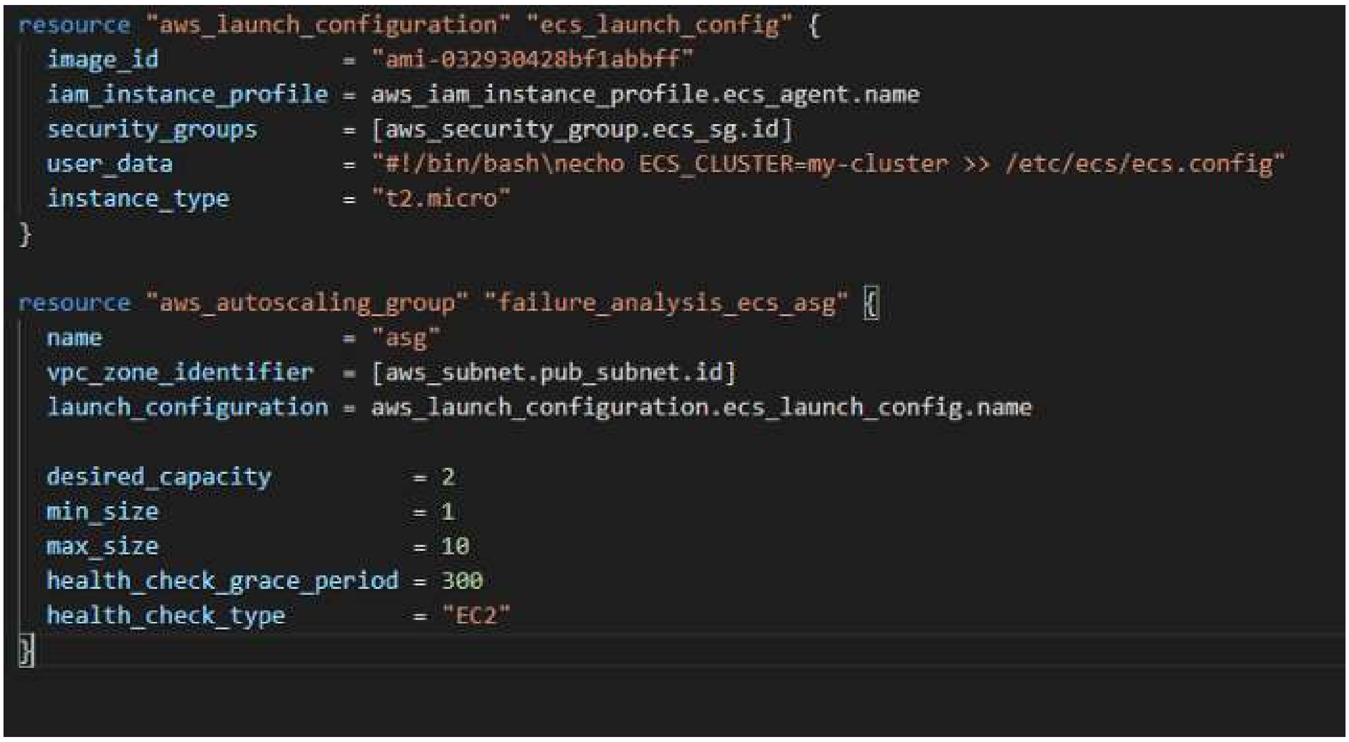


```
  role = aws_iam_role.ecs_agent.name
}
```

iam.tf

Now that we have an IAM role, we can now create an Autoscaling group.

Please note that the AMI being used here is a special one because it comes with ECS-optimized image with preinstalled docker and it also falls under the free-tier.

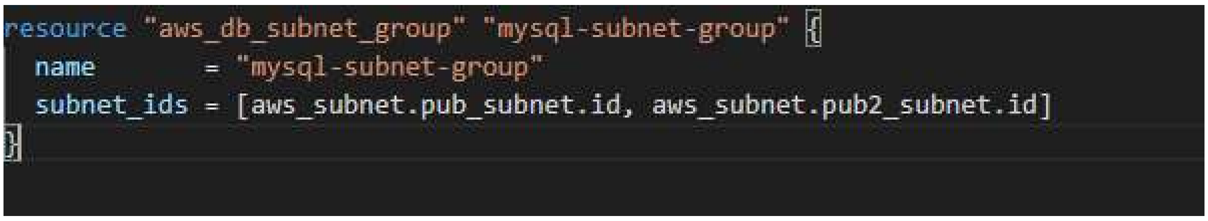


```
resource "aws_launch_configuration" "ecs_launch_config" {
  image_id             = "ami-032930428bf1abbff"
  iam_instance_profile = aws_iam_instance_profile.ecs_agent.name
  security_groups      = [aws_security_group.ecs_sg.id]
  user_data            = "#!/bin/bash\necho ECS_CLUSTER=my-cluster >> /etc/ecs/ecs.config"
  instance_type        = "t2.micro"
}

resource "aws_autoscaling_group" "failure_analysis_ecs_asg" {
  name                 = "asg"
  vpc_zone_identifier  = [aws_subnet.pub_subnet.id]
  launch_configuration = aws_launch_configuration.ecs_launch_config.name

  desired_capacity          = 2
  min_size                  = 1
  max_size                  = 10
  health_check_grace_period = 300
  health_check_type         = "EC2"
}
```

autoscaling.tf

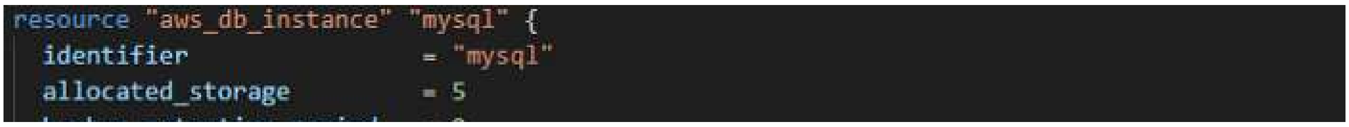
## Database Instance

Now that we have a subnet and a security group for RDS we need to provision database and add both subnets were previously created and then create the actual database instance.

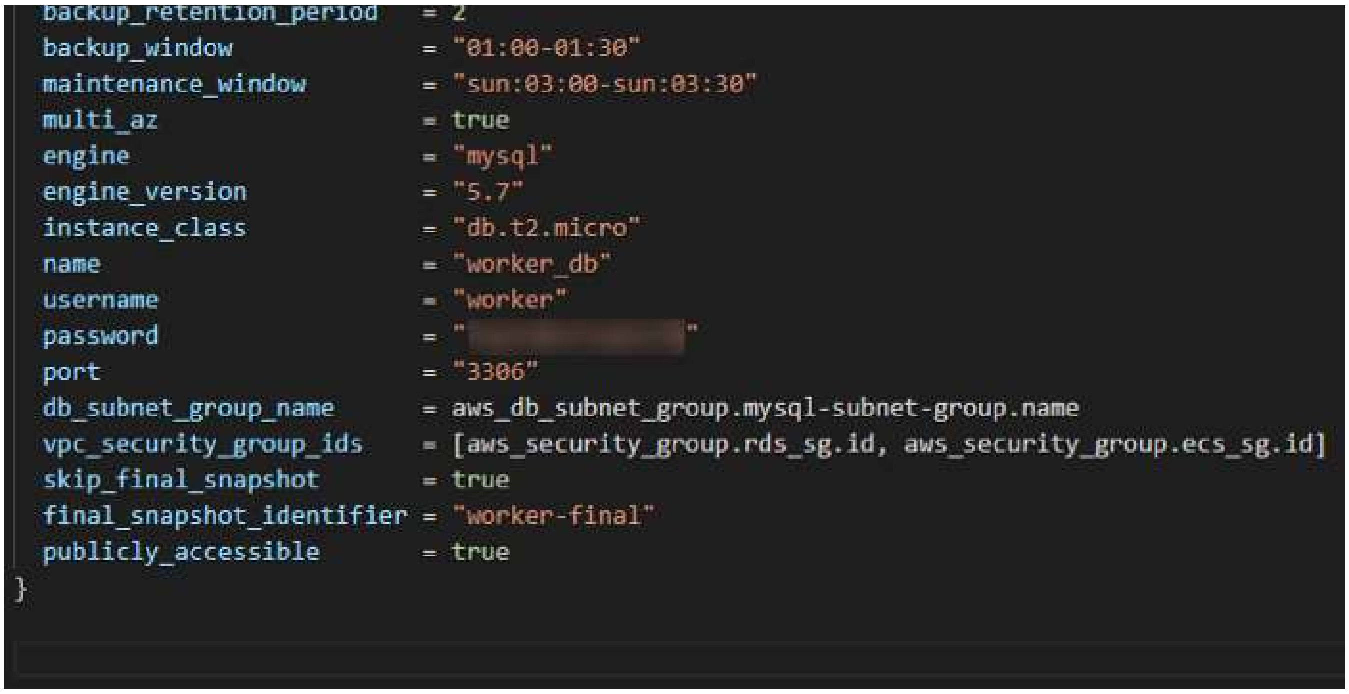


```
resource "aws_db_subnet_group" "mysql-subnet-group" {
  name       = "mysql-subnet-group"
  subnet_ids = [aws_subnet.pub_subnet.id, aws_subnet.pub2_subnet.id]
}
```

db_subnet.tf



```
resource "aws_db_instance" "mysql" {
  identifier                = "mysql"
  allocated_storage         = 5
```



```
  backup_retention_period   = 2
  backup_window             = "01:00-01:30"
  maintenance_window        = "sun:03:00-sun:03:30"
  multi_az                  = true
  engine                    = "mysql"
  engine_version            = "5.7"
  instance_class            = "db.t2.micro"
  name                      = "worker_db"
  username                  = "worker"
  password                  = "[redacted]"
  port                      = "3306"
  db_subnet_group_name      = aws_db_subnet_group.mysql-subnet-group.name
  vpc_security_group_ids    = [aws_security_group.rds_sg.id, aws_security_group.ecs_sg.id]
  skip_final_snapshot       = true
  final_snapshot_identifier = "worker-final"
  publicly_accessible       = true
}
```

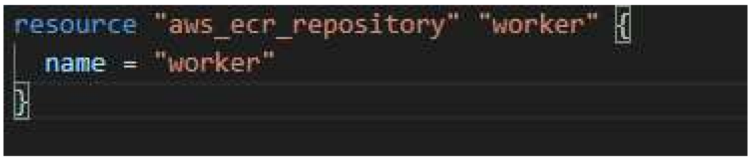
Now its time for the heavy lifting!!!!!!!!!

## Elastic Container Service

Amazon ECS provides a complete container management system supporting Docker containers and windows server containers which allows us to use third-party plug-ins and customizations from Kubernetes community.

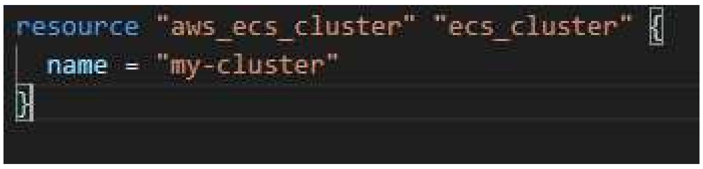
ECS allows you to setup a cluster of EC2 instances running docker in a selected VPC.

We will use ECR to push the images and use them while launching the EC2 instances within our cluster.



```
resource "aws_ecr_repository" "worker" {
  name = "worker"
}
```

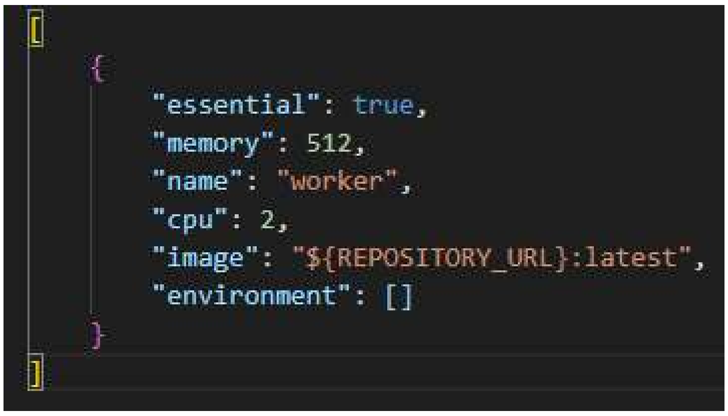
ecr.tf



```
resource "aws_ecs_cluster" "ecs_cluster" {
  name = "my-cluster"
}
```

ecs.tf

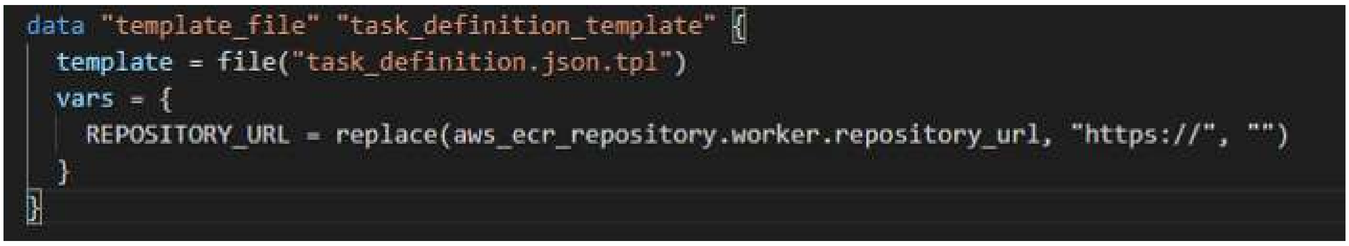
Containers are launched using a task definition. which is a set of simple instructions understood by the ECS cluster. Its a JSON file that is kept separately.



```
[
    {
        "essential": true,
        "memory": 512,
        "name": "worker",
        "cpu": 2,
        "image": "${REPOSITORY_URL}:latest",
        "environment": []
    }
]
```

task_definition.json.tpl

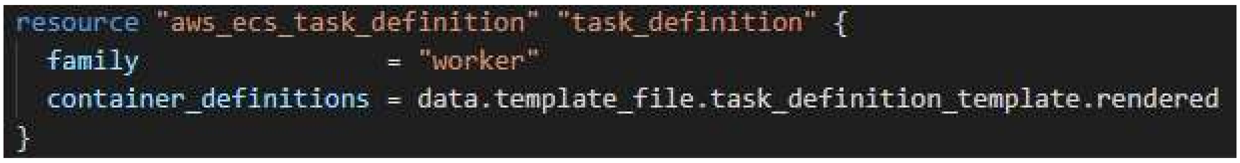
It comes with a Terraform template_file definition.



```
data "template_file" "task_definition_template" {
  template = file("task_definition.json.tpl")
  vars = {
    REPOSITORY_URL = replace(aws_ecr_repository.worker.repository_url, "https://", "")
  }
}
```

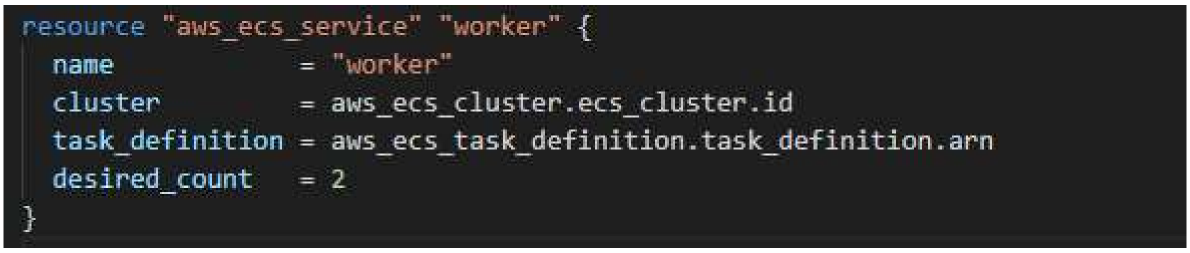
template_file.tf

We are defining what image will be used using a template variable in the template_file data resource as repository_url.



```
resource "aws_ecs_task_definition" "task_definition" {
  family                = "worker"
  container_definitions = data.template_file.task_definition_template.rendered
}
```

task_definition.tf

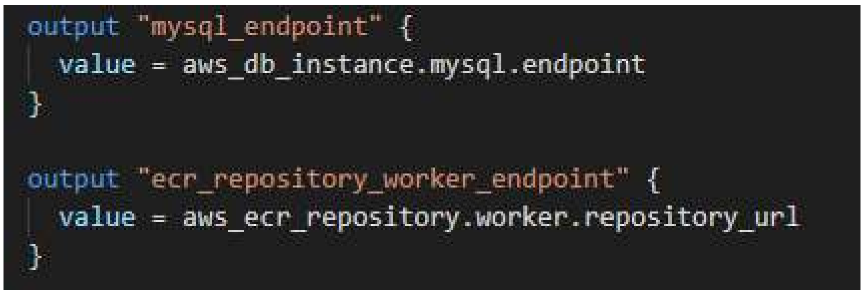


```
resource "aws_ecs_service" "worker" {
  name            = "worker"
  cluster         = aws_ecs_cluster.ecs_cluster.id
  task_definition = aws_ecs_task_definition.task_definition.arn
  desired_count   = 2
}
```

ecs_service.tf

We are all set with the last part of the architecture.

One last thing we need to do is to set an output for the provisioned components.



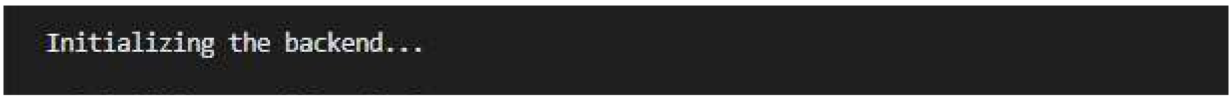
```
output "mysql_endpoint" {
  value = aws_db_instance.mysql.endpoint
}

output "ecr_repository_worker_endpoint" {
  value = aws_ecr_repository.worker.repository_url
}
```

output.tf

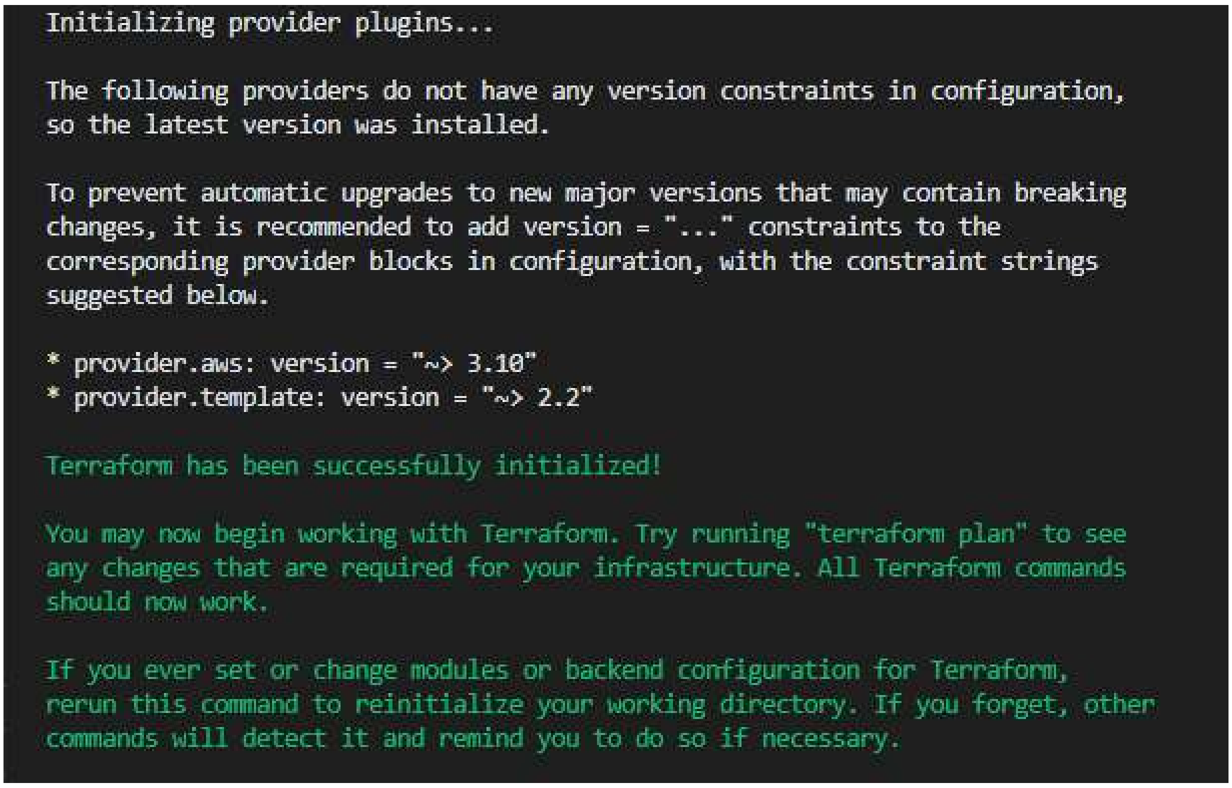
Its now time to initialize our directory by typing Terraform init

This command will initialize the directory containing a Terraform configuration. The initialization verifies the state backend and downloads modules, plugins and providers.

Below is the result that I received after running Terraform init



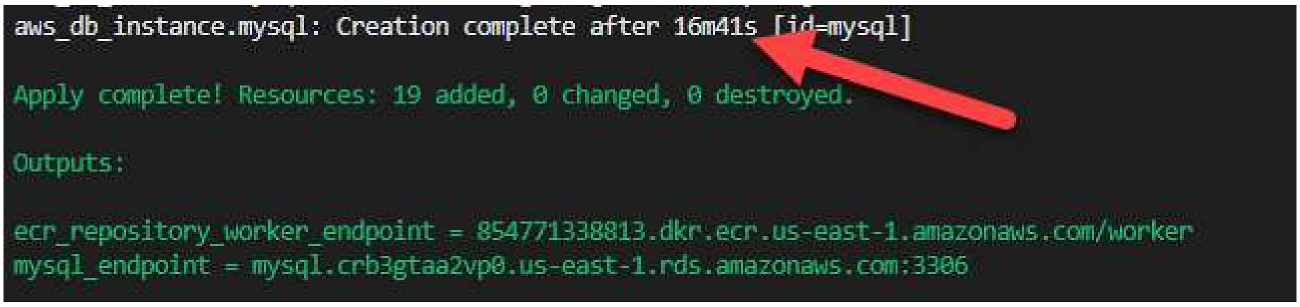
```
Initializing the backend...
```



```
Initializing provider plugins...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 3.10"
* provider.template: version = "~> 2.2"

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
commands will detect it and remind you to do so if necessary.
```

No errors! We are now free to proceed!

We should now be able to run Terraform apply to start executing the changes. Please note that this step will take a lil while. So go grab yourself or some coffee.



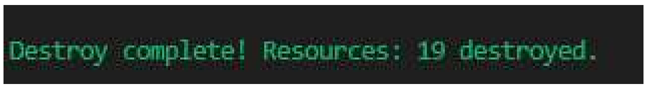
```
aws_db_instance.mysql: Creation complete after 16m41s [id=mysql]

Apply complete! Resources: 19 added, 0 changed, 0 destroyed.

Outputs:

ecr_repository_worker_endpoint = 854771338813.dkr.ecr.us-east-1.amazonaws.com/worker
mysql_endpoint = mysql.crb3gtaa2vp0.us-east-1.rds.amazonaws.com:3306
```

It took 16 minutes to create the MySQL instance, we are now able to see that everything worked. We are also able to see the outputs.

**Cleanup time**

In order to save some money, we are going to destroy this this lab.

Run Terraform destroy, you should get the same result that I received below.



```
Destroy complete! Resources: 19 destroyed.
```

All set!

Terraform is such a powerful tool, It allowed to change my perspective when it comes building an architecture. I'm willing to learn and grow as an engineer so any feedback will be appreciated.

Here is a link to my GitHub Repo for this lab!

https://github.com/ptokito/ECSClusterWithTerraform

AWS Docker DevOps Containers Terraform