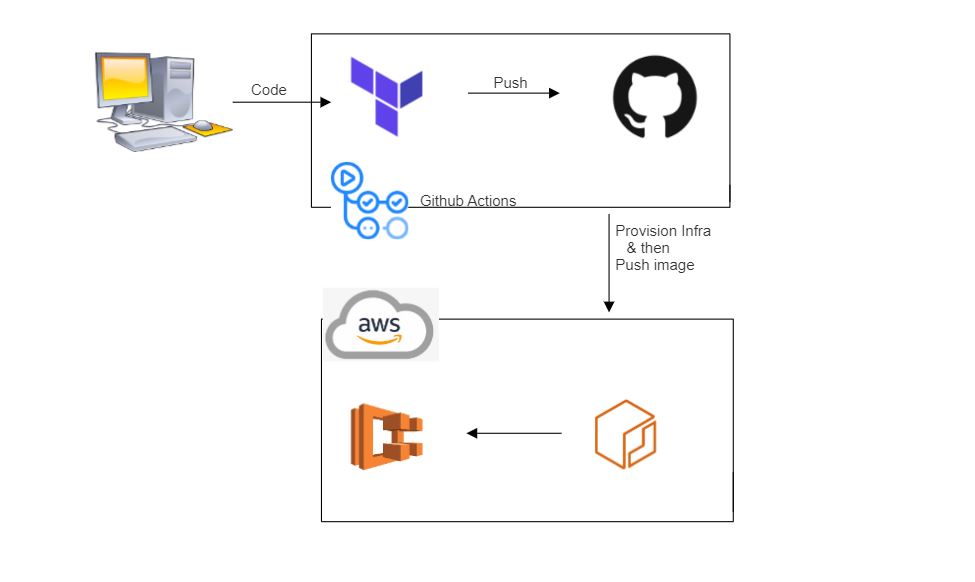
**Description of the services and CI/CD**

****

**Amazon Elastic Container Registry**(Amazon ECR) is an AWS managed container image registry service that is secure, scalable, and reliable. Amazon ECR supports private repositories with resource-based permissions using AWS IAM. This is so that specified users or Amazon EC2 instances can access your container repositories and images. You can use your preferred CLI to push, pull, and manage Docker images, Open Container Initiative (OCI) images, and OCI compatible artifacts.

HashiCorp Terraform is **an infrastructure as code tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share**. You can then use a consistent workflow to provision and manage all of your infrastructure throughout its lifecycle.

**Amazon Elastic Container Service** (Amazon ECS) is a fully managed container orchestration service that helps you easily deploy, manage, and scale containerized applications. As a fully managed service, Amazon ECS comes with AWS configuration and operational best practices built-in. This also means that you don't need to manage control plane, nodes, or add-ons. It's integrated with both AWS and third-party tools, such as Amazon Elastic Container Registry and Docker. This integration makes it easier for teams to focus on building the applications, not the environment. You can run and scale your container workloads across AWS Regions in the cloud, and on-premises, without the complexity of managing a control plane or nodes.

A serverless option with AWS Fargate. With AWS Fargate, you don't need to manage servers, handle capacity planning, or isolate container workloads for security. Fargate handles the infrastructure management aspects of your workload for you. You can schedule the placement of your containers across your cluster based on your resource needs, isolation policies, and availability requirements.

**GitHub Actions** is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. You can create workflows that build and test every pull request to your repository, or deploy merged pull requests to production.GitHub Actions goes beyond just DevOps and lets you run workflows when other events happen in your repository. For example, you can run a workflow to automatically add the appropriate labels whenever someone creates a new issue in your repository.

**The components of GitHub Actions:**

You can configure a GitHub Actions workflow to be triggered when an event occurs in your repository, such as a pull request being opened or an issue being created. Your workflow contains one or more jobs which can run in sequential order or in parallel. Each job will run inside its own virtual machine runner, or inside a container, and has one or more steps that either run a script that you define or run an action, which is a reusable extension that can simplify your workflow.

### **Workflows:** A workflow is a configurable automated process that will run one or more jobs. Workflows are defined by a YAML file checked in to your repository and will run when triggered by an event in your repository, or they can be triggered manually, or at a defined schedule.

### **Events:** An event is a specific activity in a repository that triggers a workflow run. For example, activity can originate from GitHub when someone creates a pull request, opens an issue, or pushes a commit to a repository. You can also trigger a workflow to run on a schedule, by [posting to a REST API](https://docs.github.com/en/rest/repos#create-a-repository-dispatch-event), or manually.

### **Jobs:** A job is a set of steps in a workflow that is executed on the same runner. Each step is either a shell script that will be executed, or an action that will be run. Steps are executed in order and are dependent on each other. Since each step is executed on the same runner, you can share data from one step to another. For example, you can have a step that builds your application followed by a step that tests the application that was built.

### **Actions:** An action is a custom application for the GitHub Actions platform that performs a complex but frequently repeated task. Use an action to help reduce the amount of repetitive code that you write in your workflow files. An action can pull your git repository from GitHub, set up the correct toolchain for your build environment, or set up the authentication to your cloud provider.

### **Runners:** A runner is a server that runs your workflows when they're triggered. Each runner can run a single job at a time. GitHub provides Ubuntu Linux, Microsoft Windows, and macOS runners to run your workflows; each workflow run executes in a fresh, newly-provisioned virtual machine. GitHub also offers larger runners, which are available in larger configurations.

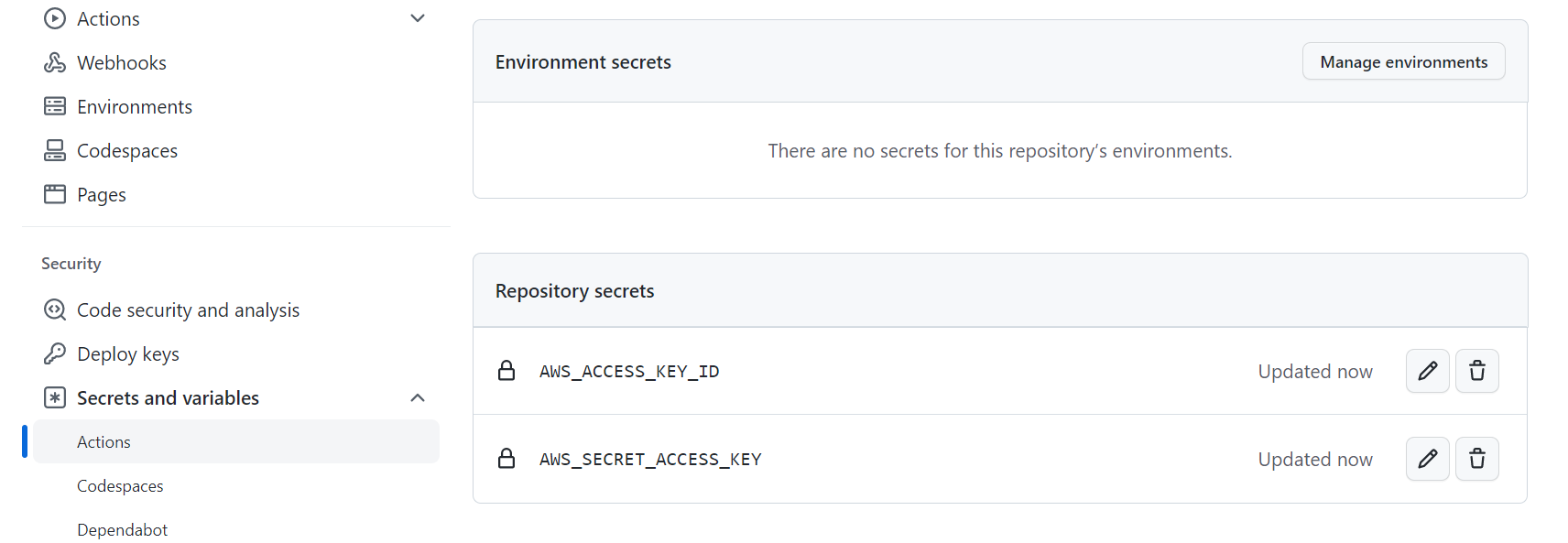
In recent times, most of the organizations have been migrating their monolithic applications to microservices which enables organizations to optimize resources, enhance collaboration, and streamline business processes. As part of the microservices architecture, each service owns its docker repository. Also, as the number of services increases, there will be a requirement to create a new repository for each service. In this example, we use Terraform to provision the infrastructure (ECS +ECR ) . We also create the docker container and push to ecr and then manage the container on eks to deploy to the end user.

**Procedure for the task:**

**Step 1: Create a github repository (say ecr-ecs-gha-ter)**

After creating the github repository , Set the $SECRET used in github action workflow file .

Create the secret name as AWS\_ACCESS\_KEY\_ID , AWS\_SECRET\_ACCESS\_KEY to configure with aws cloud.



**Step 2: Create the terraform code to provision the ECR & ECS :**

provider "aws" {

  version = "~> 4.53.0"

  region  = "us-east-1"

}

resource "aws\_ecr\_repository" "my\_first\_ecr\_repo" {

 name = "my-first-ecr-repo"

}

resource "aws\_ecr\_repository\_policy" "demo-repo-policy" {

  repository = aws\_ecr\_repository.my\_first\_ecr\_repo.name

  policy     = <<EOF

  {

    "Version": "2008-10-17",

    "Statement": [

      {

        "Sid": "Set the permission for ECR",

        "Effect": "Allow",

        "Principal": "\*",

        "Action": [

          "ecr:BatchCheckLayerAvailability",

          "ecr:BatchGetImage",

          "ecr:CompleteLayerUpload",

          "ecr:GetDownloadUrlForLayer",

          "ecr:GetLifecyclePolicy",

          "ecr:InitiateLayerUpload",

          "ecr:PutImage",

          "ecr:UploadLayerPart"

        ]

      }

    ]

  }

  EOF

}

resource "aws\_ecs\_cluster" "my\_cluster" {

  name = "my-cluster"

}

resource "aws\_ecs\_task\_definition" "my\_first\_task" {

  family                   = "my-first-task" # Naming our first task

  container\_definitions    = <<DEFINITION

  [

    {

      "name": "my-first-task",

      "image": "${aws\_ecr\_repository.my\_first\_ecr\_repo.repository\_url}",

      "essential": true,

      "portMappings": [

        {

          "containerPort": 3100,

          "hostPort": 3100

        }

      ],

      "memory": 512,

      "cpu": 256

    }

  ]

  DEFINITION

  requires\_compatibilities = ["FARGATE"]

  network\_mode             = "awsvpc"

  memory                   = 512

  cpu                      = 256

  execution\_role\_arn       = "${aws\_iam\_role.ecsTaskExecutionRole.arn}"

}

resource "aws\_iam\_role" "ecsTaskExecutionRole" {

  name               = "ecsTaskExecutionRole"

  assume\_role\_policy = "${data.aws\_iam\_policy\_document.assume\_role\_policy.json}"

}

data "aws\_iam\_policy\_document" "assume\_role\_policy" {

  statement {

    actions = ["sts:AssumeRole"]

    principals {

      type        = "Service"

      identifiers = ["ecs-tasks.amazonaws.com"]

    }

  }

}

resource "aws\_iam\_role\_policy\_attachment" "ecsTaskExecutionRole\_policy" {

  role       = "${aws\_iam\_role.ecsTaskExecutionRole.name}"

  policy\_arn = "arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy"

}

# Providing a reference to our default VPC

resource "aws\_default\_vpc" "default\_vpc" {

}

# Providing a reference to our default subnets

resource "aws\_default\_subnet" "default\_subnet\_a" {

  availability\_zone = "us-east-1a"

}

resource "aws\_default\_subnet" "default\_subnet\_b" {

  availability\_zone = "us-east-1b"

}

resource "aws\_default\_subnet" "default\_subnet\_c" {

  availability\_zone = "us-east-1c"

}

resource "aws\_alb" "application\_load\_balancer" {

  name               = "test-lb-tf" # Naming our load balancer

  load\_balancer\_type = "application"

  subnets = [ # Referencing the default subnets

    "${aws\_default\_subnet.default\_subnet\_a.id}",

    "${aws\_default\_subnet.default\_subnet\_b.id}",

    "${aws\_default\_subnet.default\_subnet\_c.id}"

  ]

  # Referencing the security group

  security\_groups = ["${aws\_security\_group.load\_balancer\_security\_group.id}"]

}

# Creating a security group for the load balancer:

resource "aws\_security\_group" "load\_balancer\_security\_group" {

  ingress {

    from\_port   = 80 # Allowing traffic in from port 80

    to\_port     = 80

    protocol    = "tcp"

    cidr\_blocks = ["0.0.0.0/0"] # Allowing traffic in from all sources

  }

  egress {

    from\_port   = 0 # Allowing any incoming port

    to\_port     = 0 # Allowing any outgoing port

    protocol    = "-1" # Allowing any outgoing protocol

    cidr\_blocks = ["0.0.0.0/0"] # Allowing traffic out to all IP addresses

  }

}

resource "aws\_lb\_target\_group" "target\_group" {

  name        = "target-group"

  port        = 80

  protocol    = "HTTP"

  target\_type = "ip"

  vpc\_id      = "${aws\_default\_vpc.default\_vpc.id}"

  health\_check {

    matcher = "200,301,302"

    path = "/"

  }

}

resource "aws\_lb\_listener" "listener" {

  load\_balancer\_arn = "${aws\_alb.application\_load\_balancer.arn}"

  port              = "80"

  protocol          = "HTTP"

  default\_action {

    type             = "forward"

    target\_group\_arn = "${aws\_lb\_target\_group.target\_group.arn}"

  }

}

resource "aws\_ecs\_service" "my\_first\_service" {

  name            = "my-first-service"

  cluster         = "${aws\_ecs\_cluster.my\_cluster.id}"

  task\_definition = "${aws\_ecs\_task\_definition.my\_first\_task.arn}"

  launch\_type     = "FARGATE"

  desired\_count   = 3 # Setting the number of containers to 3

  load\_balancer {

    target\_group\_arn = "${aws\_lb\_target\_group.target\_group.arn}"

    container\_name   = "${aws\_ecs\_task\_definition.my\_first\_task.family}"

    container\_port   = 3100 # Specifying the container port

  }

  network\_configuration {

    subnets          = ["${aws\_default\_subnet.default\_subnet\_a.id}", "${aws\_default\_subnet.default\_subnet\_b.id}", "${aws\_default\_subnet.default\_subnet\_c.id}"]

    assign\_public\_ip = true                                                # Providing our containers with public IPs

    security\_groups  = ["${aws\_security\_group.service\_security\_group.id}"] # Setting the security group

  }

}

resource "aws\_security\_group" "service\_security\_group" {

  ingress {

    from\_port = 0

    to\_port   = 0

    protocol  = "-1"

    # Only allowing traffic in from the load balancer security group

    security\_groups = ["${aws\_security\_group.load\_balancer\_security\_group.id}"]

  }

  egress {

    from\_port   = 0 # Allowing any incoming port

    to\_port     = 0 # Allowing any outgoing port

    protocol    = "-1" # Allowing any outgoing protocol

    cidr\_blocks = ["0.0.0.0/0"] # Allowing traffic out to all IP addresses

  }

}

**Step 3: Create a github action workflow file to create the ECR and ECS on aws cloud and build the image using Dockerfile and push to ECR & deploying it to ECS,when pushing the code to main branch.**

name: Node.js CI

on: [push]

jobs:

  build:

    runs-on: ubuntu-latest

    steps:

    - uses: actions/checkout@v3

    - name: Use Node.js

      uses: actions/setup-node@v3

      with:

        node-version: '12.x'

    - name: Install dependencies

      run: npm install

    - run: npm init --y

    - run: npm install express

    - name: Checkout Repo

      uses: actions/checkout@v2

    - name: Terraform Setup

      uses: hashicorp/setup-terraform@v1

    - name: Terraform Init

      run: terraform init

      env:

        GITHUB\_TOKEN: ${{ secrets.GITHUB\_TOKEN }}

        TF\_ACTION\_WORKING\_DIR: '.'

        AWS\_ACCESS\_KEY\_ID:  ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

        AWS\_SECRET\_ACCESS\_KEY:  ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

    - name: Terraform validate

      run: terraform validate

    - name: Terraform Apply

      run: terraform apply -auto-approve

      env:

        GITHUB\_TOKEN: ${{ secrets.GITHUB\_TOKEN }}

        TF\_ACTION\_WORKING\_DIR: '.'

        AWS\_ACCESS\_KEY\_ID:  ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

        AWS\_SECRET\_ACCESS\_KEY:  ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

    - name: Check out code

      uses: actions/checkout@v2

    - name: Configure AWS credentials

      uses: aws-actions/configure-aws-credentials@v1

      with:

        aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

        aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

        aws-region: us-east-1

    - name: Login to Amazon ECR

      id: login-ecr

      uses: aws-actions/amazon-ecr-login@v1

    - name: Build, tag, and push image to Amazon ECR

      env:

        ECR\_REGISTRY: ${{ steps.login-ecr.outputs.registry }}

        ECR\_REPOSITORY: my-first-ecr-repo

        IMAGE\_TAG: docker\_image

      run: |

        docker build -t $ECR\_REGISTRY/$ECR\_REPOSITORY:$IMAGE\_TAG .

        docker push $ECR\_REGISTRY/$ECR\_REPOSITORY:$IMAGE\_TAG

        echo "::set-output name=image::$ECR\_REGISTRY/$ECR\_REPOSITORY:$IMAGE\_TAG"

**Step 4: Creating the nodejs app**

const express = require('express')

const app = express()

const port = 3100

app.get('/', (req, res) => res.send('Simple APP with Terraform!'))

app.listen(port, () => console.log(`Example app listening on port ${port}!`))

**Step 5: Create the DOCKERFILE to create the docker image of the nodejs app.**

# 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 . .

# Make port 3100 available to the world outside this container

EXPOSE 3100

# Run index.js when the container launches

CMD ["node", "index.js"]

**Step 6: Push the code to github repository from local repository.**

$ git add .

$ git commit -m “commit”

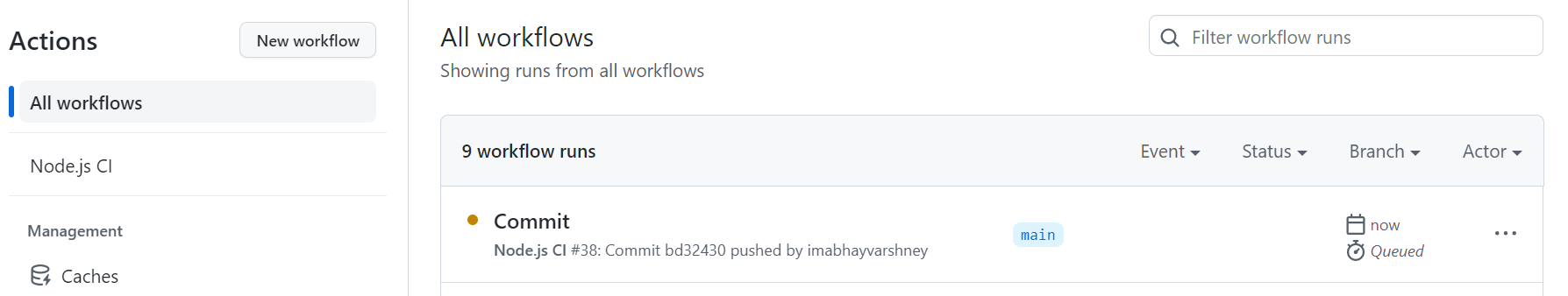
$ git push -u origin main

Graphical user interface, text, application, email

Description automatically generated

**Step 7 : Browse the alb url to see the deployed nodejs application.**

**Step 8: SNAPSHOTS :**

Workflow started and created the infrastructure & build & push the image to ECR and deployed to ECS. 

Text

Description automatically generated

Text

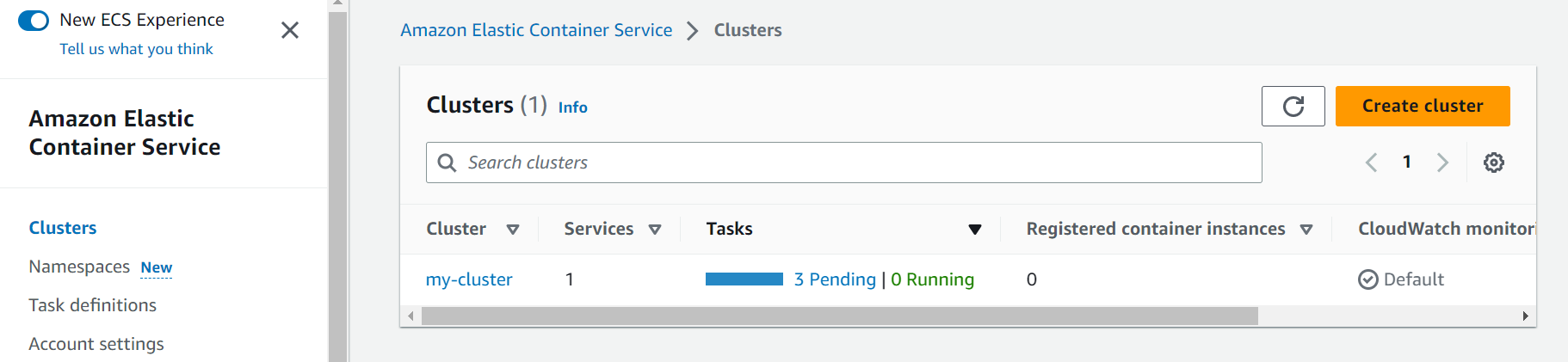
Description automatically generated

Created ECR :

Graphical user interface, text, application, email

Description automatically generated

Created ECS :



Pushed image in ECR:

Graphical user interface, application

Description automatically generated

Created service :

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

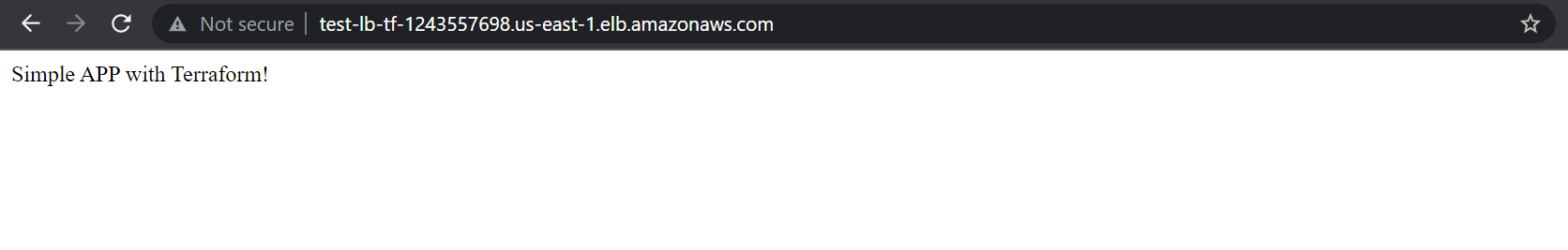
Description automatically generated

Created load balancer:

Graphical user interface, text, application, email

Description automatically generated

Deployed node js app:



You can use my github repository for the the code and workflow:

https://github.com/imabhayvarshney/ecr-ecs-gha-ter.git