Evvo Assignment

Name: Antony Jebinraj Y
Role: DevOps Engineer
Email: jebin1848@gmail.com

Note: (Im not used https because Im not able to buy free domain and I cant use load balancers its so expensive.Next if you want Github action I mentioned the githubaction yaml file also, Next I deleted this infra after execution because its take bill so in interview time i will ready implement this once again)

Steps:

1. Write tf code (terraform) for EC2 creation.
2. Write Shell Script for Jenkins and Kubectl installation.
3. Write tf code for EKS cluster and node creation.
4. Create the Application using Mern Stack.
5. Design Kubernetes deployment and service.
6. Configure application Horizontal auto-scaling.
7. Push the code on GitHub
8. Create a Jenkins pipeline auto-trigger EKS cluster tf code and auto-deployment.

Step: 1.Write tf code (terraform) for EC2 creation.

The code block sets the AWS provider for Terraform to use the "us-west-2" region.

```
#provider.tf
provider "aws" {
  region = "us-west-2"
}
```

This Terraform configuration sets up remote state storage in an S3 bucket named "terraform-jenkins-eks-01" within the "us-west-2" region, using the key "jenkins-server/terraform.tfstate".

```
backend.tf
terraform {
  backend "s3" {
    bucket = "terraform-jenkins-eks-01"
    key    = "jenkins-server/terraform.tfstate"
    region = "us-west-2"
  }
}
```

Defines variables for a Terraform configuration including VPC and subnet CIDR blocks, availability zone, environment prefix, and EC2 instance type.

```
#terraform.tfvars

vpc_cidr_block      = "10.0.0.0/16"
subnet_cidr_block   = "10.0.10.0/24"
availability_zone   = "us-west-2a"
env_prefix          = "dev"
instance_type       = "t2.small"
```

Your provided files define variables for a Terraform configuration:

```
#variable.tf

variable "vpc_cidr_block" {
 type = string
 description = "To set cidr for vpc"
}
variable "subnet_cidr_block" {
 type = string
 description = "To set cidr for subnet"
}
variable "availability_zone" {
 type = string
 description = "To set AWS availability region"
}
variable "env_prefix" {
 type = string
 description = "Set as dev or prod or qa etc. based on desired
environment"
}
variable "instance_type" {
 type = string
 description = "To desired instance type for AWS EC2 instance"
}
```

This Terraform script sets up a VPC with a subnet, an internet gateway, a default route table, and a default security group, all tagged and configured for a Jenkins server environment.

```
#vpc.tf
resource "aws_vpc" "myjenkins-server-vpc" {
  cidr_block = var.vpc_cidr_block
  tags = {
    Name = "${var.env_prefix}-vpc"
  }
}

resource "aws_subnet" "myjenkins-server-subnet-1" {
  vpc_id            = aws_vpc.myjenkins-server-vpc.id
  cidr_block        = var.subnet_cidr_block
  availability_zone = var.availability_zone
  tags = {
    Name = "${var.env_prefix}-subnet-1"
  }
}

resource "aws_internet_gateway" "myjenkins-server-igw" {
  vpc_id = aws_vpc.myjenkins-server-vpc.id
  tags = {
    Name = "${var.env_prefix}-igw"
  }
}

resource "aws_default_route_table" "main-rtbl" {
  default_route_table_id =
aws_vpc.myjenkins-server-vpc.default_route_table_id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.myjenkins-server-igw.id
  }
  tags = {
    Name = "${var.env_prefix}-main-rtbl"
  }
}

resource "aws_default_security_group" "default-sg" {
  vpc_id = aws_vpc.myjenkins-server-vpc.id
  ingress {
```

```
    from_port   = 0
    to_port     = 0
    protocol    = "-1"   # -1 means all protocols
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
  tags = {
    Name = "${var.env_prefix}-default-sg"
  }
}
```

This Terraform script retrieves the latest Amazon Linux 2 AMI and creates an EC2 instance with specified parameters, setting up Jenkins, and outputs the instance's public IP.

```
#server.tf
data "aws_ami" "latest-amazon-linux-image" {
  most_recent = true
  owners      = ["amazon"]
  filter {
    name   = "name"
    values = ["amzn2-ami-kernel-*-hvm-*-x86_64-gp2"]
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}

resource "aws_instance" "myjenkins-server" {
  ami                         = data.aws_ami.latest-amazon-linux-image.id
  instance_type               = var.instance_type
  key_name                    = "ak"
  subnet_id                   = aws_subnet.myjenkins-server-subnet-1.id
  vpc_security_group_ids      = [aws_default_security_group.default-sg.id]
```

```
  availability_zone           = var.availability_zone
  associate_public_ip_address = true
  user_data                   = file("jenkins-server-setup.sh")
  tags = {
    Name = "${var.env_prefix}-server"
  }
}

output "ec2_public_ip" {
  value = aws_instance. myjenkins-server.public_ip
}
```

Step 2: Write Shell Script for Jenkins and Kubectl installation.

This script updates the system, installs Jenkins, Git, Terraform, and kubectl, and sets up Jenkins to run on system startup on an Amazon Linux system.

```
#jenkins-server-setup.sh
#!/bin/bash
sudo yum update
sudo wget -O /etc/yum.repos.d/jenkins.repo \
    https://pkg.jenkins.io/redhat-stable/jenkins.repo
sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io-2023.key
sudo yum upgrade -y
sudo amazon-linux-extras install java-openjdk11 -y
sudo yum install jenkins -y
sudo systemctl enable jenkins
sudo systemctl start jenkins

# then install git
sudo yum install git -y

#then install terraform
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo
https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
sudo yum -y install terraform
```

```
#finally install kubectl
sudo curl -LO
https://storage.googleapis.com/kubernetes-release/release/v1.23.6/bin/linu
x/amd64/kubectl
sudo chmod +x ./kubectl
sudo mkdir -p $HOME/bin && sudo cp ./kubectl $HOME/bin/kubectl && export
PATH=$PATH:$HOME/bin
```

Executing these terraform tf files for EC2 creation and Jenkins and Kubectl installation.

(terraform init) this command is for initializing our project.

```
C:\ANTONY_JEBINRAJ\Info\eks-terraform-jenkins\terraform-for-jenkins>terraform init

Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v5.53.0...
```

(terraform apply) this command is for applying our infra for provisioning.

```
C:\ANTONY_JEBINRAJ\Info\eks-terraform-jenkins\terraform-for-jenkins>terraform apply
data.aws_ami.latest-amazon-linux-image: Reading...
aws_vpc.myjenkins-server-vpc: Refreshing state... [id=vpc-0ebb7cd98d6f117fc]
data.aws_ami.latest-amazon-linux-image: Read complete after 2s [id=ami-02e8e2a390064c712]
aws_internet_gateway.myjenkins-server-igw: Refreshing state... [id=igw-042afaab0b85c8597]
aws_subnet.myjenkins-server-subnet-1: Refreshing state... [id=subnet-05a2baef070dc077e]
aws_default_security_group.default-sg: Refreshing state... [id=sg-09e56a884254dbe65]
aws_default_route_table.main-rtbl: Refreshing state... [id=rtb-0b15500f367ef43bd]
aws_instance.myjenkins-server: Refreshing state... [id=i-01dc8cf873283ef8d]

Terraform used the selected providers to generate the following execution plan. Resource actions
are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # aws_default_security_group.default-sg will be updated in-place
  ~ resource "aws_default_security_group" "default-sg" {
      ~ egress                 = [
          - {
              - cidr_blocks      = [
                  - "0.0.0.0/0",
                ]
              - from_port        = 0
              - ipv6_cidr_blocks = []
              - prefix_list_ids  = []
              - protocol         = "-1"
              - security_groups  = []
              - self             = false
              - to_port          = 0
                # (1 unchanged attribute hidden)
            },
```

After, Successfully created an EC2 instance and installed Jenkins and Kubectl.

Step 3: Write tf code(terraform) for the EKS cluster and node creation. The code block sets the AWS provider for Terraform to use the "us-west-2" region.

```
#provider.tf
provider "aws" {
  region = "us-west-2"
}
```

This Terraform configuration sets up remote state storage in an S3 bucket named "terraform-jenkins-eks-01" within the "us-west-2" region, using the key "jenkins-server/terraform.tfstate".

```
backend.tf
terraform {
  backend "s3" {
    bucket = "terraform-jenkins-eks-01"
    key    = "jenkins-server/terraform.tfstate"
    region = "us-west-2"
  }
}
```

This Terraform variable definition file specifies the CIDR blocks for a VPC, along with its associated private and public subnets.

```
#terraform.tfvars
vpc_cidr_block = "10.0.0.0/16"
private_subnet_cidr_blocks=["10.0.1.0/24","10.0.2.0/24","10.0.3.0/24"]
public_subnet_cidr_blocks=["10.0.4.0/24","10.0.5.0/24","10.0.6.0/24"]
```

This Terraform configuration declares variables for VPC CIDR block, lists of private subnet CIDR blocks, and lists of public subnet CIDR blocks.

```
#variable.tf
variable "vpc_cidr_block" {
    type = string
}

variable "private_subnet_cidr_blocks" {
```

```
    type = list(string)
}
variable "public_subnet_cidr_blocks" {
    type = list(string)
}
```

This Terraform module creates a VPC with public and private subnets, enables NAT gateway, DNS hostnames, and assigns Kubernetes tags for Jenkins server deployment on AWS.

```
#vpc.tf
data "aws_availability_zones" "azs" {}
module "myjenkins-server-vpc" {
  source          = "terraform-aws-modules/vpc/aws"
  name            = "myjenkins-server-vpc"
  cidr            = var.vpc_cidr_block
  private_subnets = var.private_subnet_cidr_blocks
  public_subnets  = var.public_subnet_cidr_blocks
  azs             = data.aws_availability_zones.azs.names

  enable_nat_gateway   =  true
  single_nat_gateway   =  true
  enable_dns_hostnames = true

  tags = {
    "kubernetes.io/cluster/myjenkins-server-eks-cluster" = "shared"
  }

  public_subnet_tags = {
    "kubernetes.io/cluster/myjenkins-server-eks-cluster" = "shared"
    "kubernetes.io/role/elb"                  = 1
  }

  private_subnet_tags = {
    "kubernetes.io/cluster/myjenkins-server-eks-cluster" = "shared"
    "kubernetes.io/role/internal-elb"         = 1
  }
}
```

This Terraform module configures an Amazon EKS cluster named "myjenkins-server-eks-cluster" with version "1.24", accessible publicly, linked to a VPC and subnets, and managed node group with specified instance types for a development environment.

```
#eks-cluster.tf
module "eks" {
    source  = "terraform-aws-modules/eks/aws"
    version = "~> 19.0"
    cluster_name = "myjenkins-server-eks-cluster"
    cluster_version = "1.24"

    cluster_endpoint_public_access   = true

    vpc_id = module.myjenkins-server-vpc.vpc_id
    subnet_ids = module.myjenkins-server-vpc.private_subnets

    tags = {
        environment = "development"
        application = "myjenkins-server"
    }

    eks_managed_node_groups = {
        dev = {
            min_size = 1
            max_size = 3
            desired_size = 2

            instance_types = ["t2.small"]
        }
    }
}
```

Step 4: Create the Application using Mern Stack.

Create the project and the working directory called node-react-k8s You can choose any name you want for your project. In the node-react-k8s folder, create a new folder named, and change the directory into the folder using the following command

mkdir backend and cd backend

The backend folder will contain all the code needed to create the Express server. Within the newly created folder, run the following command to initialize the Node.js project.

npm init --y

This command will initialize the Node.js project and create the ‘package.json’ to install the project dependencies. Next, we install the dependencies necessary to build and run the application. We will install the dependencies Express, and nodemon to help us create the backend server, monitor the application script, and detect changes, and errors.

npm i express and npm i -g nodemon

Thirdly, open the ‘package.json’ file, and add the following script under the “script” section.

"dev": "nodemon -L app.js"

4. Within the backend folder, create a file and name it `app.js` . Add the following code.

```
const express = require('express')

const cors = require('cors')
```

```
const app = express()




app.use(cors())




app.get('/', (req, res) => {

  res.json([

    {

      "id":"1",

      "title":"Movie Review: The Perk of Being a Wallflower"

    },

    {

      "id":"2",

      "title":"Game Review: Need for Speed"

    },
```

```
    {

      "id":"3",

      "title":"Show Review: Looking for Alaska"

    }

  ])

})



app.listen(4000, () => {

  console.log('listening for requests on port 4000')

})
```

With the code above, you will create an Express application using the ‘get’ route that will set APIs to the React front end. This will be listening for all the requests coming from port 4000 on the server. To run the application, use the following command on your server.

npm run dev

This will create the development server to run your application. Creating React.js Application. After building your backend server, change the directory out of the backend folder, and create a React.js project using the following command

npx create-react-app frontend

We'd use this command to create a new folder named "frontend" in the working directory. Within the 'frontend' folder, navigate to the "src" folder, and open "App.js".

```
import { useEffect, useState } from 'react'

import './App.css';



function App() {

  const [blogs, setBlogs] = useState([])

  useEffect(() => {

    fetch('http://localhost:4000/')

      .then(res => res.json())
```

```
      .then(data => setBlogs(data))

  }, [])



  return (

    <div className="App">

      <header className="App-header">

        <h1>all blogs</h1>

        {blogs && blogs.map(blog => (

          <div key={blog.id}>{blog.title}</div>

        ))}

      </header>

    </div>

  );

}
```

```
export default App;
```

Taking a look at the ‘fetch’ section, you can see that it is fetching the API from the backend server to display on the front end. Don’t worry if you do not understand. run your application using the following code.

```
npm start
```

Step 5: Design Kubernetes deployment and service.

Dockerizing the Node.js application and Navigate to the ‘backend’ directory and create a Dockerfile. This Dockerfile will have all the commands that you will need to build the backend of the application. The code will look like the one below.

```
#Dockerfile

FROM node:18-alpine

RUN npm install -g nodemon

WORKDIR /app
```

```
COPY package.json .

RUN npm install

COPY . .

EXPOSE 4000

CMD ["npm", "run", "dev"]
```

This file defines a Kubernetes Deployment and Service configuration for a Node.js application, specifying resource limits, ports, and replicas.

```
#deployment-service.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

  name: node-js-deployment

spec:

  replicas: 2
```

```
selector:

  matchLabels:

    app: node-js

template:

  metadata:

    labels:

      app: node-js

  spec:

    containers:

    - name: node-js

      image: obusorezekiel/node-js

      resources:

        limits:

          memory: "356Mi"
```

```
            cpu: "500m"

        ports:

        - containerPort: 4000




---




apiVersion: v1

kind: Service

metadata:

  name: node-js-service

spec:

  type: NodePort

  selector:

    app: node-js
```

```
  ports:

  - port: 4000

    targetPort: 4000

    protocol: TCP
```

Just as we have dockerized the Node.js application, we are required to Dockerize the React.js application too.Navigate to the ‘front end’ and create a Dockerfile in the folder. Open the “Dockerfile” and add the following code.

```
FROM node:18-alpine

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 3000
```

```
CMD ["npm", "start"]
```

This Kubernetes manifest defines a Deployment and a Service for a React.js application, with two replicas, resource limits, exposing port 3000.

```
#deployment-service.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

  name: react-js-deployment

spec:

  replicas: 2

  selector:

    matchLabels:

      app: react-js

  template:
```

```
  metadata:

    labels:

      app: react-js

  spec:

    containers:

    - name: react-js

      image: obusorezekiel/react-js

      resources:

        limits:

          memory: "356Mi"

          cpu: "500m"

      ports:

      - containerPort: 3000
```

```yaml
---




apiVersion: v1

kind: Service

metadata:

  name: react-js-service

spec:

  type: NodePort

  selector:

    app: react-js

  ports:

  - protocol: TCP

    port: 3000

    targetPort: 3000
```

Step 6: Configure application Horizontal auto-scaling.

Install metric server

```
[root@ip-10-0-10-168 ~]# kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.3.6/components.yaml

clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
serviceaccount/metrics-server created
deployment.apps/metrics-server created
service/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
error: unable to recognize "https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.3.6/components.yaml": no matches fo
r kind "APIService" in version "apiregistration.k8s.io/v1beta1"
```

Create an Horizontal Pod Auto scaler yaml file

```yaml
#hpa.yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: myapp-backend-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: node-js-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Step 7: Push the code on GitHub

- Create Repo (eks-terraform-jenkins) and using these command for push.

```
git add .
git commit -m ("update")
git push origin main
```

Step 8: Create a Jenkins pipeline auto-trigger EKS cluster tf code and auto-deployment.

This Jenkins pipeline script automates the creation of an EKS cluster, builds Docker images for both backend and frontend applications, pushes them to ECR, and deploys them to Kubernetes.

```
pipeline {
    agent any
    environment {
        AWS_ACCESS_KEY_ID = credentials('AWS_ACCESS_KEY_ID')
        AWS_SECRET_ACCESS_KEY = credentials('AWS_SECRET_ACCESS_KEY')
        AWS_DEFAULT_REGION = "us-west-2"
    }
    stages {
        stage("Create an EKS Cluster") {
            steps {
                script {
                    dir('terraform-for-eks/terraform-for-cluster') { sh
                        "terraform init"
                        sh "terraform apply -auto-approve"
                    }
                }
            }
```

```
    }
    stage("Deploy to Backend") {
      steps {
        script {
          dir('my-app/backend') {
            sh "aws eks update-kubeconfig --name myjenkins-server-eks-cluster --region us-west-2" sh
            "docker build -t myapp-backend ."
            sh "aws ecr get-login-password --region us-west-2 | docker login --username AWS
--password-stdin 038675313786.dkr.ecr.us-west-2.amazonaws.com" sh
            "docker tag myapp-backend:latest
038675313786.dkr.ecr.us-west-2.amazonaws.com/myapp-backend:latest" sh
            "docker push
038675313786.dkr.ecr.us-west-2.amazonaws.com/myapp-backend:latest" sh
            "kubectl apply -f deployment-service.yaml"
          }
        }
      }
    }
    stage("Deploy to Frontend") {
      steps {
        script {
          dir('my-app/frontend') {
            sh "aws eks update-kubeconfig --name myjenkins-server-eks-cluster --region us-west-2" sh
            "docker build -t myapp-frontend ."
            sh "aws ecr get-login-password --region us-west-2 | docker login --username AWS
--password-stdin 038675313786.dkr.ecr.us-west-2.amazonaws.com" sh
            "docker tag myapp-frontend:latest
038675313786.dkr.ecr.us-west-2.amazonaws.com/myapp-frontend:latest" sh
            "docker push
038675313786.dkr.ecr.us-west-2.amazonaws.com/myapp-frontend:latest" sh
            "kubectl apply -f deployment-service.yaml"

          }
        }
      }
    }

  }
```

```
}
```

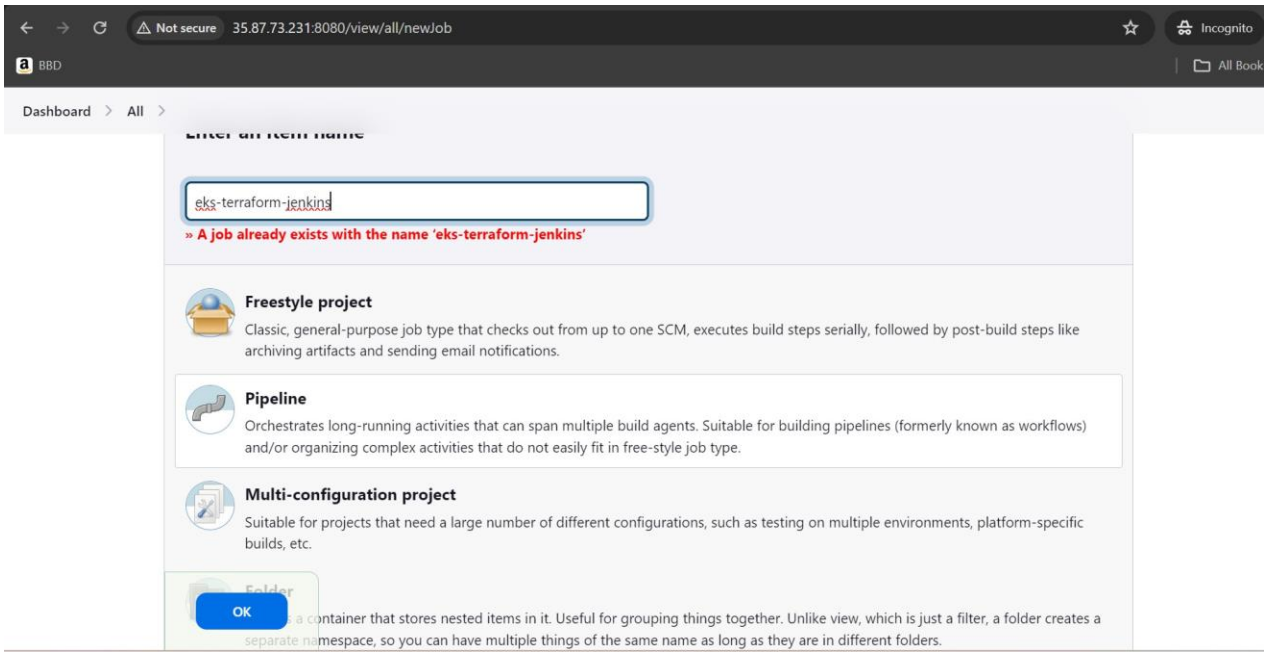
Go to our Jenkins console using this URL: http://35.87.73.231:3000/ (username: admin: password:admin)

If you want github actions, you can use this:

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout 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: Set up Docker Buildx
      uses: docker/setup-buildx-action@v1

    - name: Log in to Docker Hub
      uses: docker/login-action@v1
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}

    - name: Build and push Docker image
      run: |
        docker build -t antonyjebinraj/my-app .
        docker push antonyjebinraj/my-app

    - name: Install dependencies
      run: |
        sudo apt-get update
        sudo apt-get install -y gnupg software-properties-common curl
```

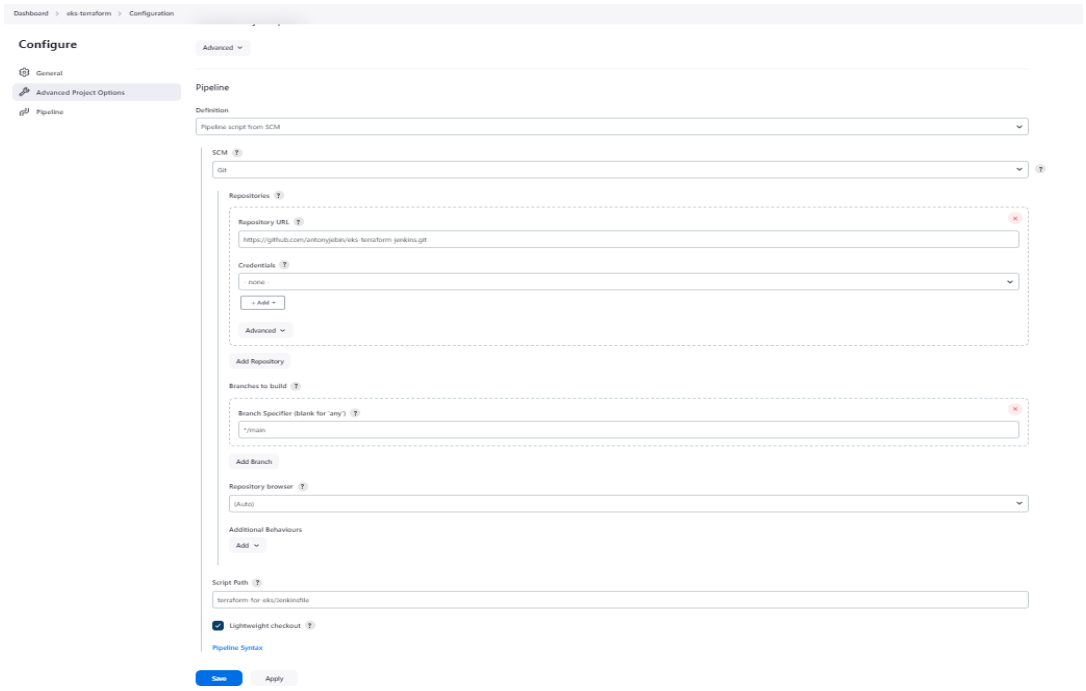
```
    - name: Add HashiCorp GPG key
      run: |
        curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -

    - name: Add HashiCorp repository
      run: |
        sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com
$(lsb_release -cs) main"

    - name: Install Terraform
      run: |
        sudo apt-get update
        sudo apt-get install -y terraform

    - name: Initialize Terraform
      run: terraform init

    - name: Apply Terraform configuration
      run: terraform apply -auto-approve

env:
    AWS_DEFAULT_REGION: us-east-1
    AWS_REGION: us-east-1
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

Next, create a pipeline using a console like:

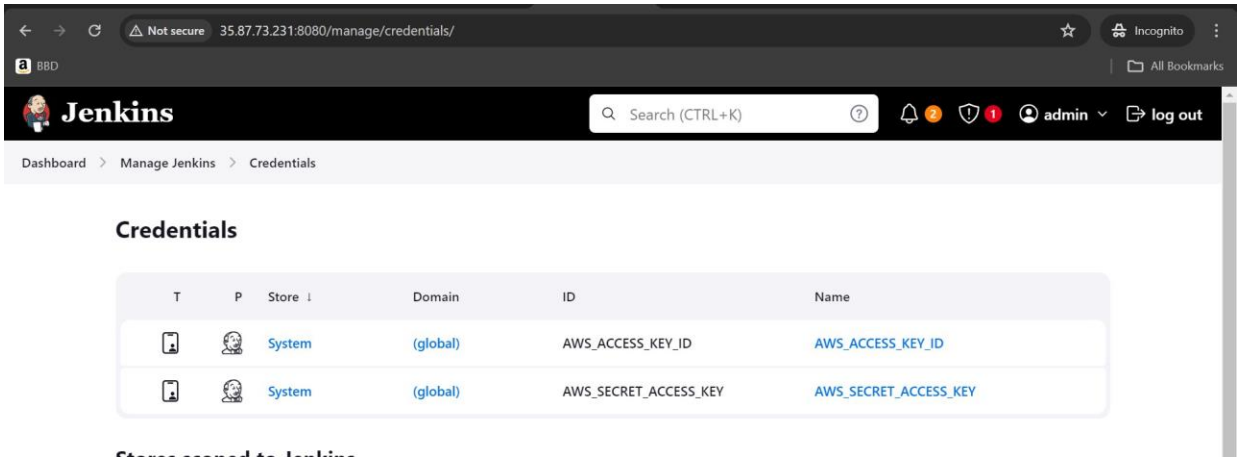


Choose pipeline option.
Select Git.

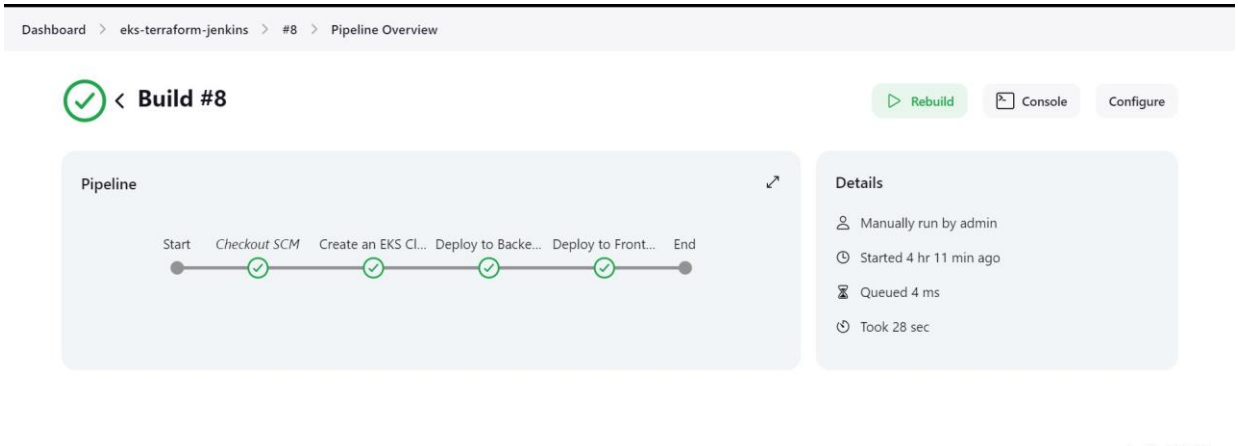
Configure our git URL.
Configure our Jenkinfile with path location.
Save and Apply.



Next, Go Dashboard>Manage Jenkins>Credentials.
Create AWS Access key and secret key credentials to access our AWS account.



Next, Click Build button its automatically trigger our jenkins pipeline and create of an EKS cluster, builds Docker images for both backend and frontend applications, pushes them to ECR, and deploys them to Kubernetes.




```
[root@ip-10-0-10-168 ~]# kubectl get all
NAME                                        READY   STATUS
RESTARTS      AGE

pod/node-js-deployment-5c874fc856-n97d7     1/1     Running              0
29m
pod/node-js-deployment-5c874fc856-zfmh8     1/1     Running              0
29m

pod/react-js-deployment-67764899cd-ph6bx    1/1     Running              0
29m
pod/react-js-deployment-67764899cd-thbkk    1/1     Running             0
29m
```

```
NAME                          TYPE        CLUSTER-IP       EXTERNAL-IP
3000:32000/TCP    106s
service/kubernetes            ClusterIP   172.20.0.1       <none>
443/TCP           6h43m
service/node-js-service       NodePort    172.20.126.209   <none>
4000:31140/TCP    29m


service/react-js-service      NodePort    172.20.44.147    <none>
3000:32752/TCP    29m

NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE

deployment.apps/node-js-deployment      2/2     2            2           29m
```

```
106s
replicaset.apps/node-js-deployment-5c874fc856    2         2         2
29m

NAME                                                        REFERENCE
TARGETS          MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/myapp-backend-hpa
Deployment/node-js-deployment   <unknown>/50%   2         10        2
```

Successfully set up all these things!