1. Optimizing Docker Image

Multi-stage Dockerfile to reduce the image size. ensuring only production dependencies are included.

```
FROM node:18 AS build

WORKDIR /app

COPY package.json package-lock.json* ./

RUN npm install

COPY . .

# Stage 2: Final Image (Alpine)

FROM node:18-alpine

RUN addgroup -S appgroup && adduser -S appuser -G appgroup

WORKDIR /app

COPY --from=build /app /app

RUN chown -R appuser:appgroup /app

USER appuser

EXPOSE 5000

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

Single-stage Dockerfile

```
FROM node:18

WORKDIR /app

COPY package.json package-lock.json* ./

RUN npm install

COPY . .

EXPOSE 5000

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

difference in size and security between the original and optimized images

Size: The single-stage build produces a larger image as it includes unnecessary development dependencies, increasing the attack surface. The multi-stage build optimizes this by producing a smaller final image containing only runtime essentials**.**

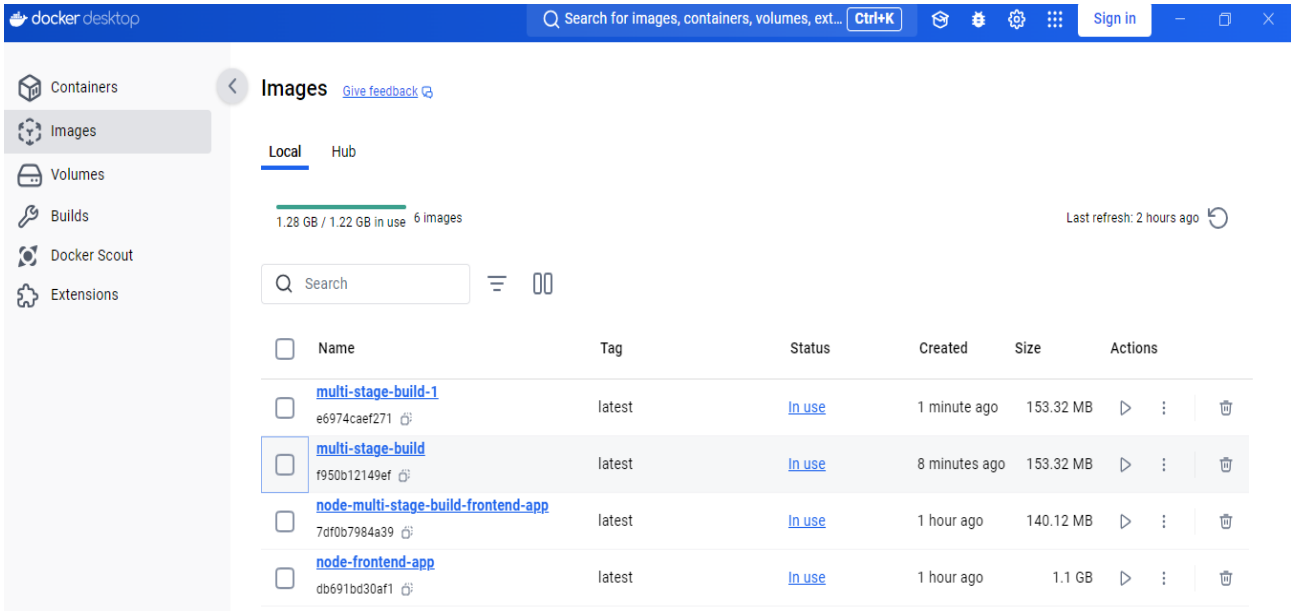
Attack Surface: The single-stage image, with its full Node.js environment, is more vulnerable to exploits targeting unused tools. In contrast, the multi-stage build minimizes this risk by using a lightweight Alpine base image.

Dependency Risks: The multi-stage build reduces the chances of including vulnerable development dependencies, which could be exploited if present in the runtime environment.

Surface Hardening: Alpine's minimal design further enhances security by including fewer packages, making it harder for attackers to find exploitable vulnerabilities.

Best Practices: Multi-stage builds align with security best practices by ensuring only production-ready code and dependencies are deployed, reducing the risk of unintended exposures.

Single stage build Image size is **1.1 GB**, while multi stage build image size is **153.32 MB**.



2. Deployment of node app on Minikube Cluster via Helm

Prerequisites:- Before you begin, make sure you have the following tools installed and running on your local machine:

- kubectl (Kubernetes command-line tool)
- Docker
- Minikube

Clone the project repository to your local machine:

Github repo url:- https://github.com/shivam086r/Cloudrove-EKS-Helm-CICD.git

Commands:-

git clone https://github.com/shivam086r/Cloudrove-EKS-Helm-CICD.git

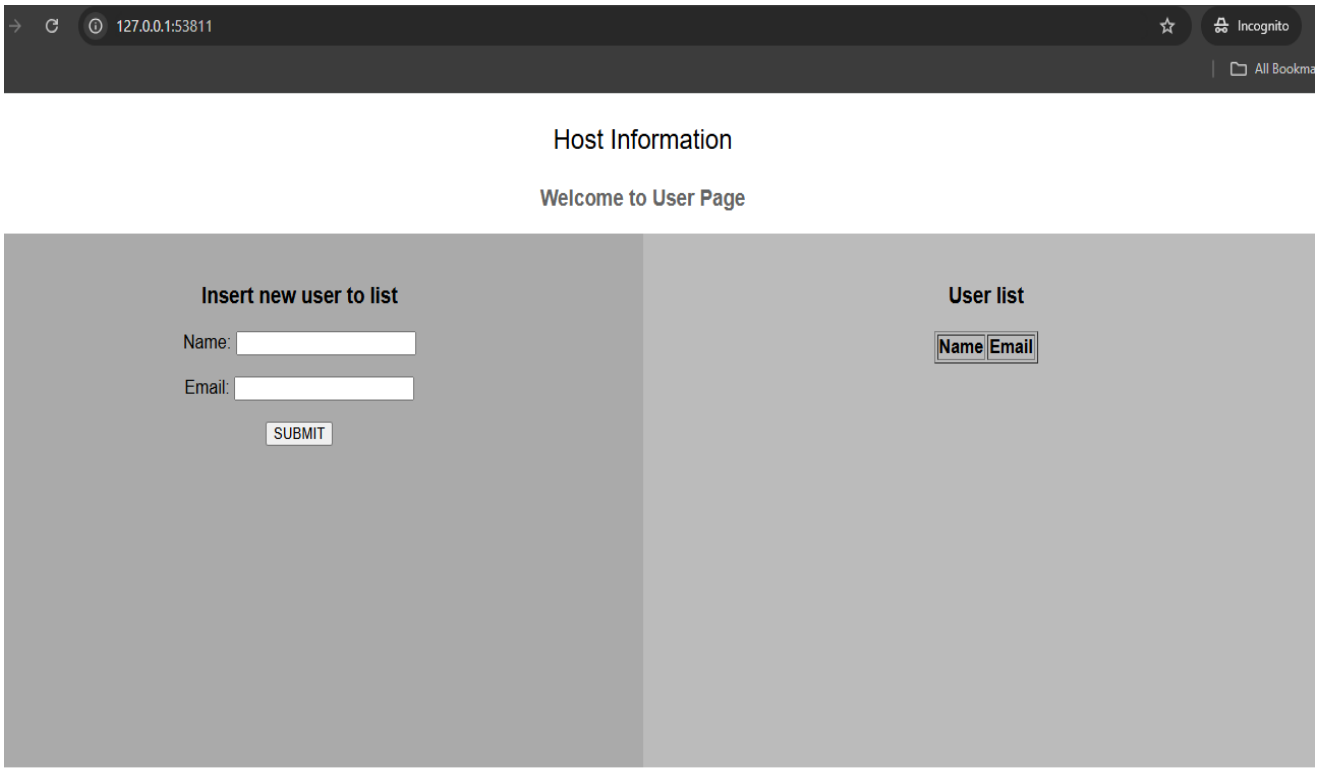
cd Cloudrove-EKS-Helm-CICD

helm install myapp ./myapp-helm-chart

kubectl get svc

minikube service nodejs-app --url

(Note:- You can access the service via this URL)

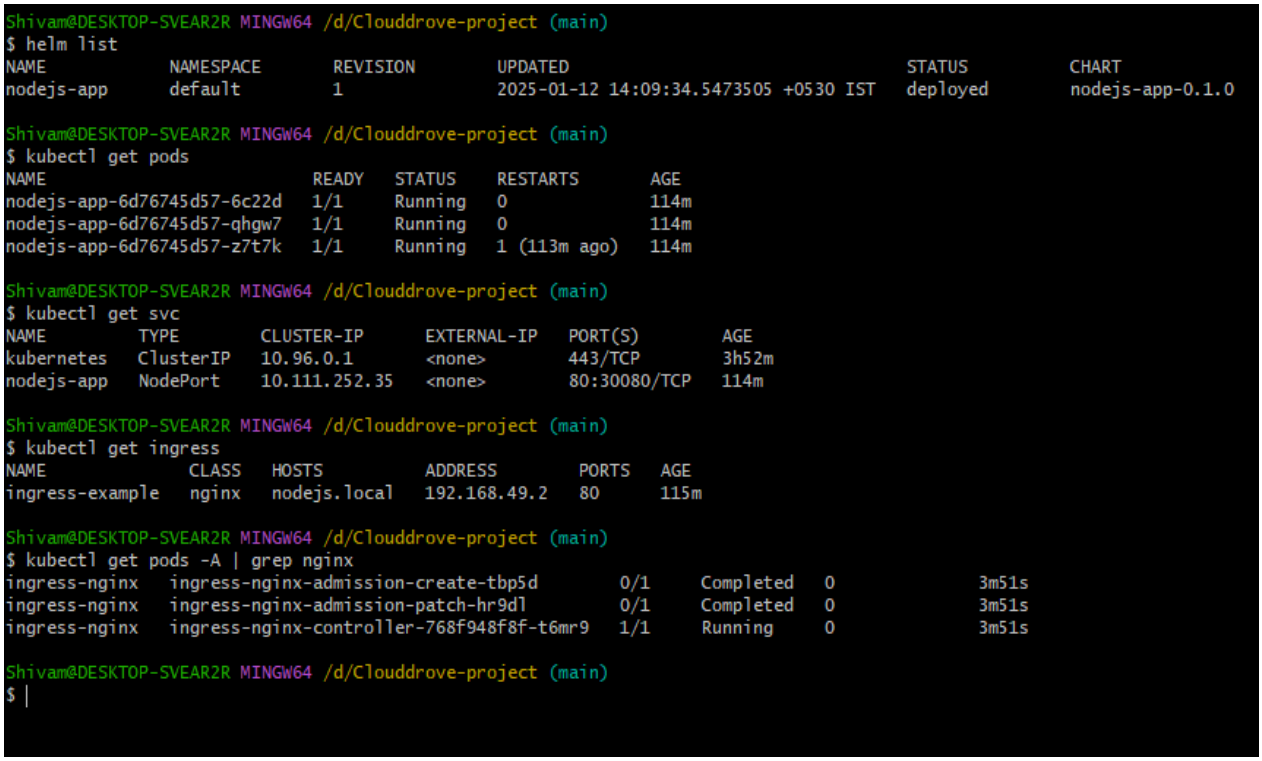


Adding ingress controller

Commands:-

minikube addons enable ingress

kubectl get pods -A | grep nginx



##Adding Ingress Controller IP with hostname inside the local hosts file

##for linux open host file and add IP and domain

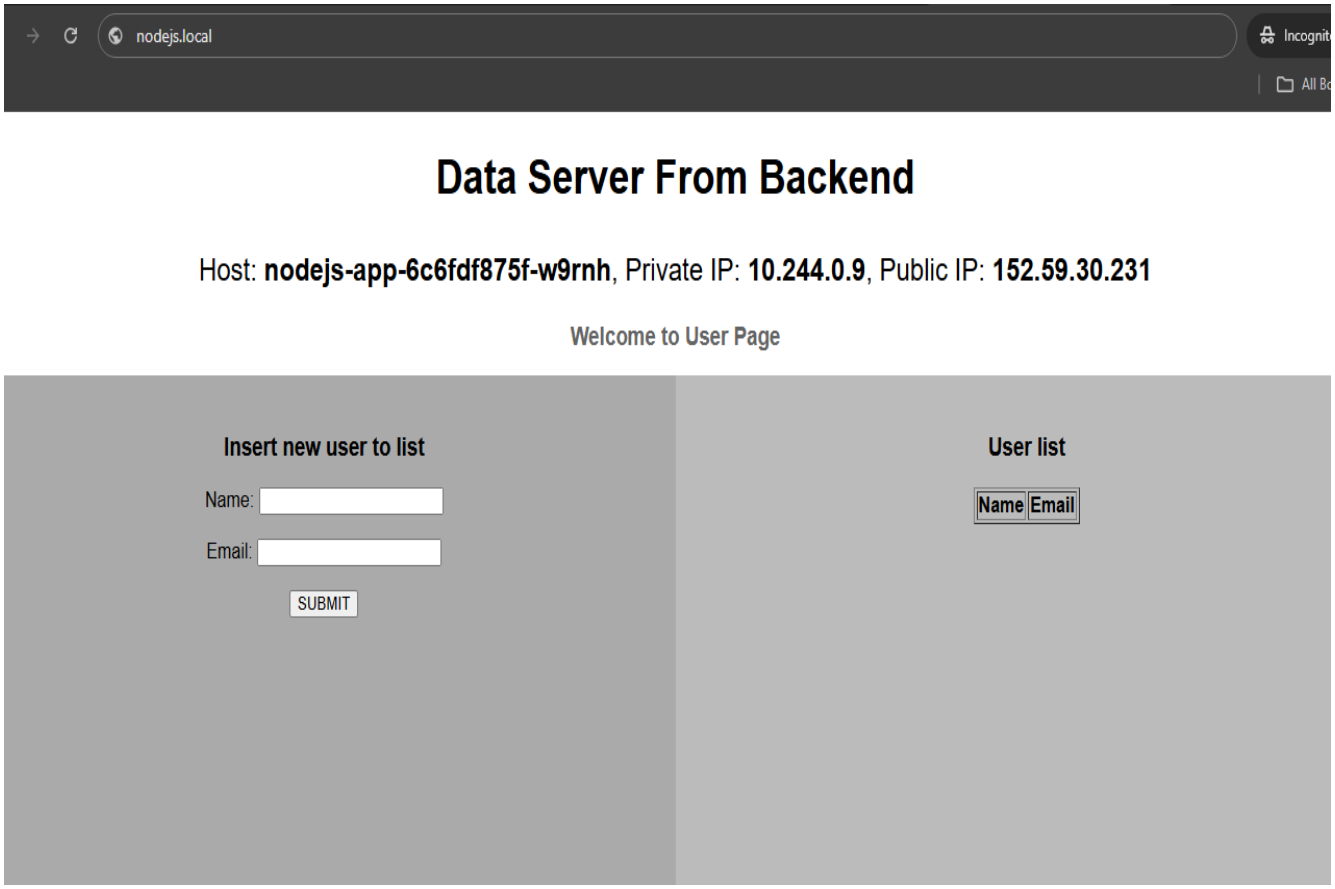
/etc/hosts

192.168.49.2 nodejs.local

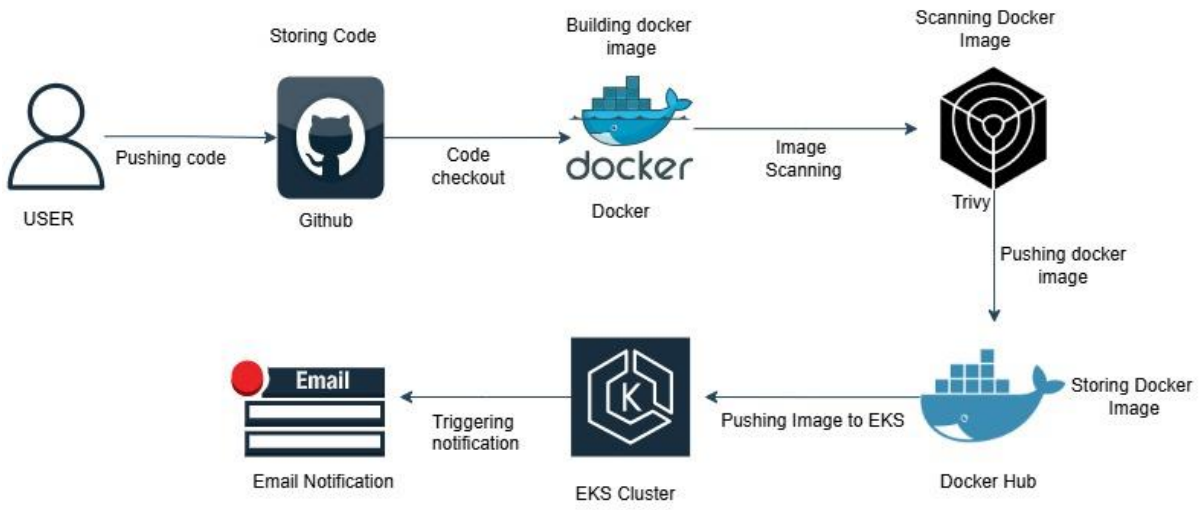
##for windows open host file and add IP and domain

C:\Windows\System32\drivers\etc/hosts

192.168.49.2 nodejs.local


nodejs.local
Incognit
All Bo
Data Server From Backend
Host: nodejs-app-6c6fdf875f-w9rnh, Private IP: 10.244.0.9, Public IP: 152.59.30.231
Welcome to User Page
Insert new user to list
Name:
Email:
SUBMIT
User list
Name
Email

3.EKS Cluster Creation with EKSCTL, Deploying app on EKS via Helm and CICD configuration



CICD WORKFLOW

Prerequisite:

- ➔ AWS CLI and kubectl should be installed on your machine
- ➔ Create IAM user with admin access and create access key id and secret access key

Commands:-

aws --version

aws configure

(provide your access-key-id, secret-access-key and region)

Cloning the Repo

git clone https://github.com/shivam086r/Clouddrove-project.git

cd Cloudrove-EKS-Helm-CICD

Creating EKS Cluster via eksctl :-

Commands:-

eksctl create cluster -f cluster-config.yaml

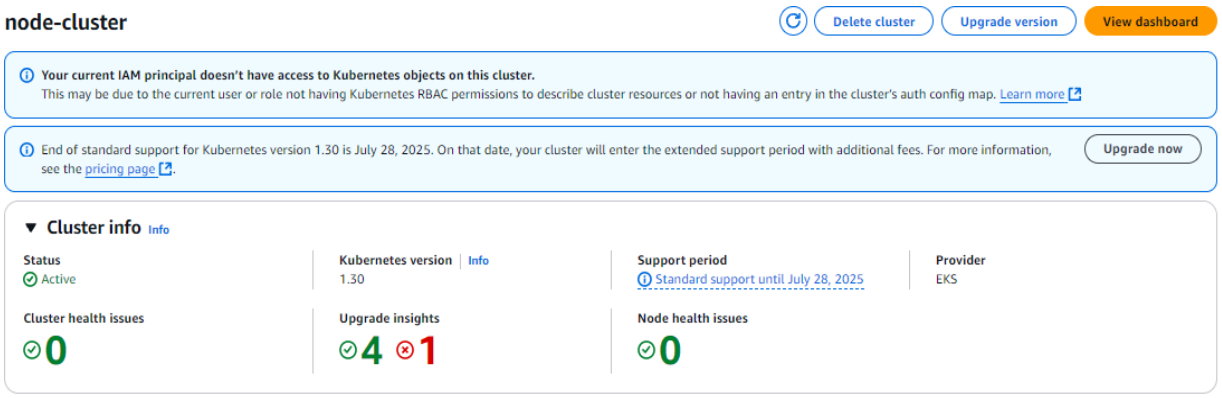
configuring oidc-provider and addon vpc-cni

Commands:-

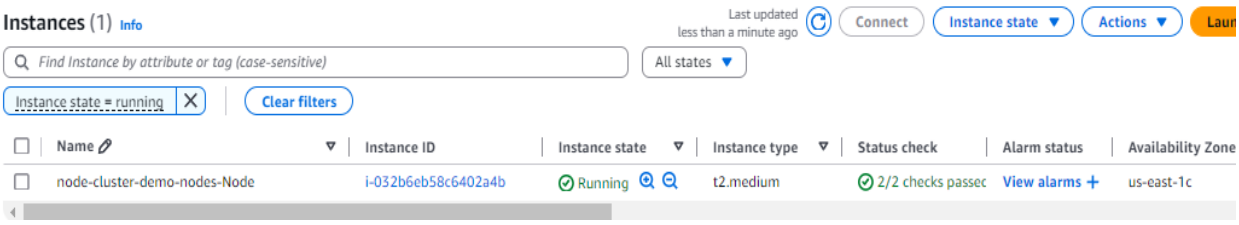
eksctl utils associate-iam-oidc-provider --region us-east-1 --cluster node-cluster --approve

eksctl update addon --name vpc-cni --cluster node-cluster --region us-east-1

Cluster



Worker-Node



<u>Deploying app on EKS Cluster</u>

Updating kubeconfig file

Command:-

aws eks --region us-east-1 update-kubeconfig --name <cluster-name>

Deploying app via helm-Chart

Command:-

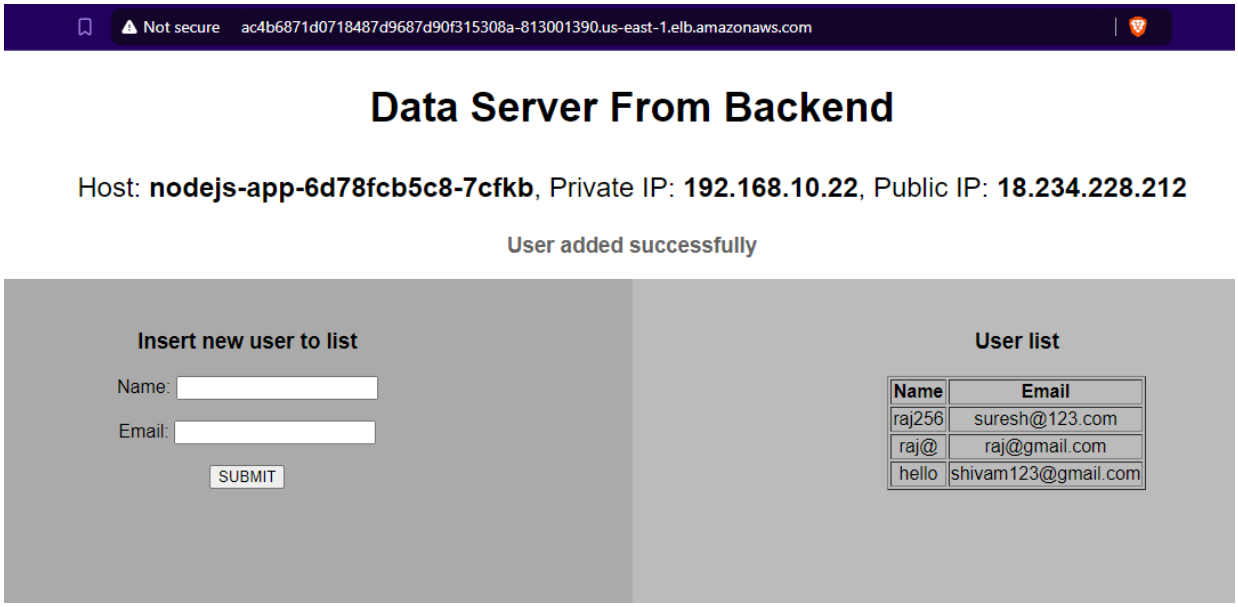
kubectl create namespace app-namespace

helm install myapp ./myapp-helm-chart

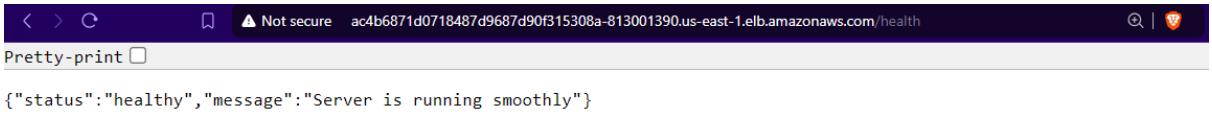
kubectl get all -n app-namespace

```
Shivam@DESKTOP-SVEAR2R MINGW64 /d/CloudDrove-update-project/test-clouddrove (main)
$ kubectl get all -n app-namespace
NAME                          READY   STATUS    RESTARTS   AGE
pod/mongodb-8444b757d6-bc6gx      1/1     Running   0          49m
pod/nodejs-app-6d78fcb5c8-7cfkb   1/1     Running   0          6m30s
pod/nodejs-app-6d78fcb5c8-ggc42   1/1     Running   0          6m32s

NAME                      TYPE           CLUSTER-IP      EXTERNAL-IP                                                               PORT(S)        AGE
service/mongodb-service   ClusterIP      10.100.57.136   <none>                                                                    27017/TCP      49m
service/nodejs-service    LoadBalancer   10.100.24.73    ac4b6871d0718487d9687d90f315308a-813001390.us-east-1.elb.amazonaws.com   80:30675/TCP   49m

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mongodb      1/1     1            1           49m
deployment.apps/nodejs-app   2/2     2            2           49m

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/mongodb-8444b757d6      1         1         1       49m
replicaset.apps/nodejs-app-6d78fcb5c8   2         2         2       6m34s
replicaset.apps/nodejs-app-7bbbdd85d4   0         0         0       49m
```

Accessing app on browser via Load balancer URL and adding Data



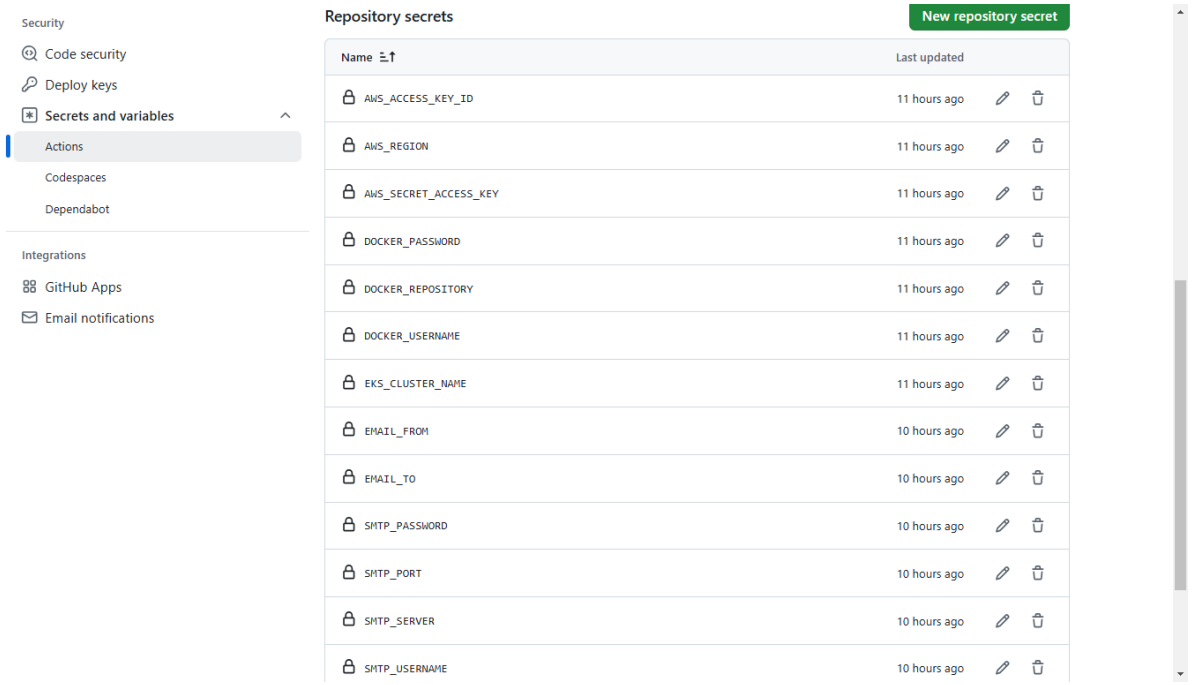
Checking /health endpoint:-



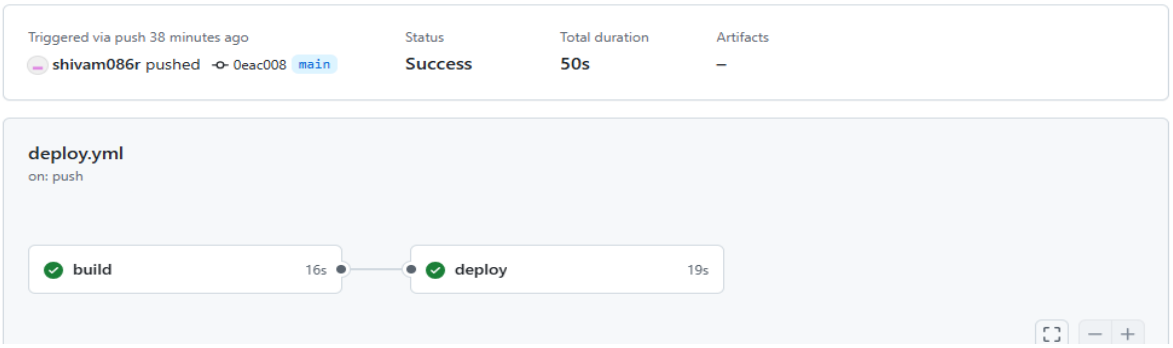
Setting up CICD pipeline

creating **.github/workflow/deploy.yaml** file and adding all the necessary CICD steps there

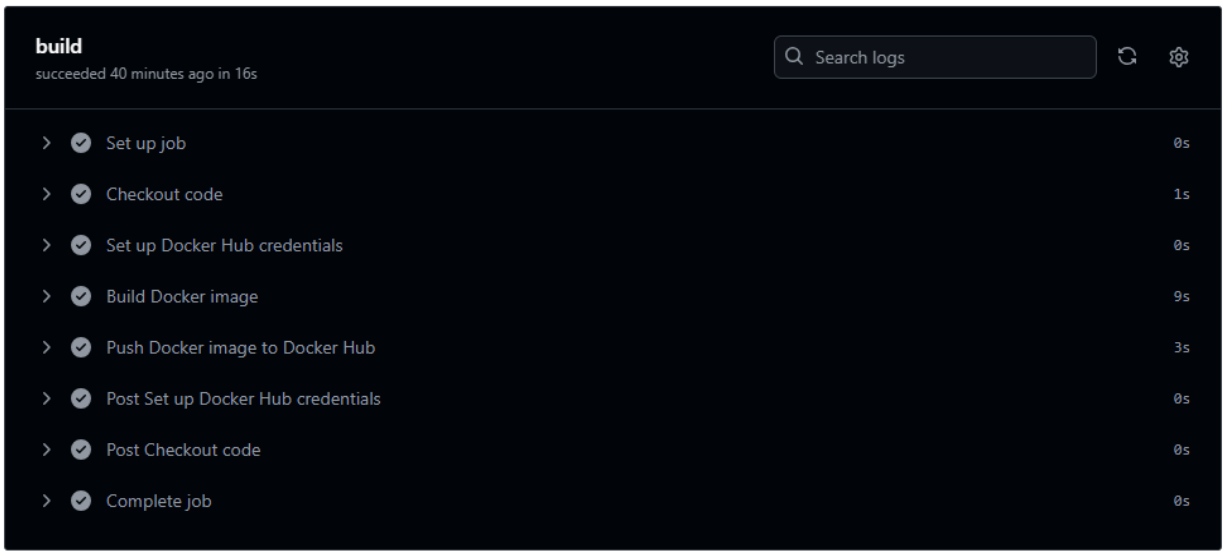
In your repository settings, add the following secrets:



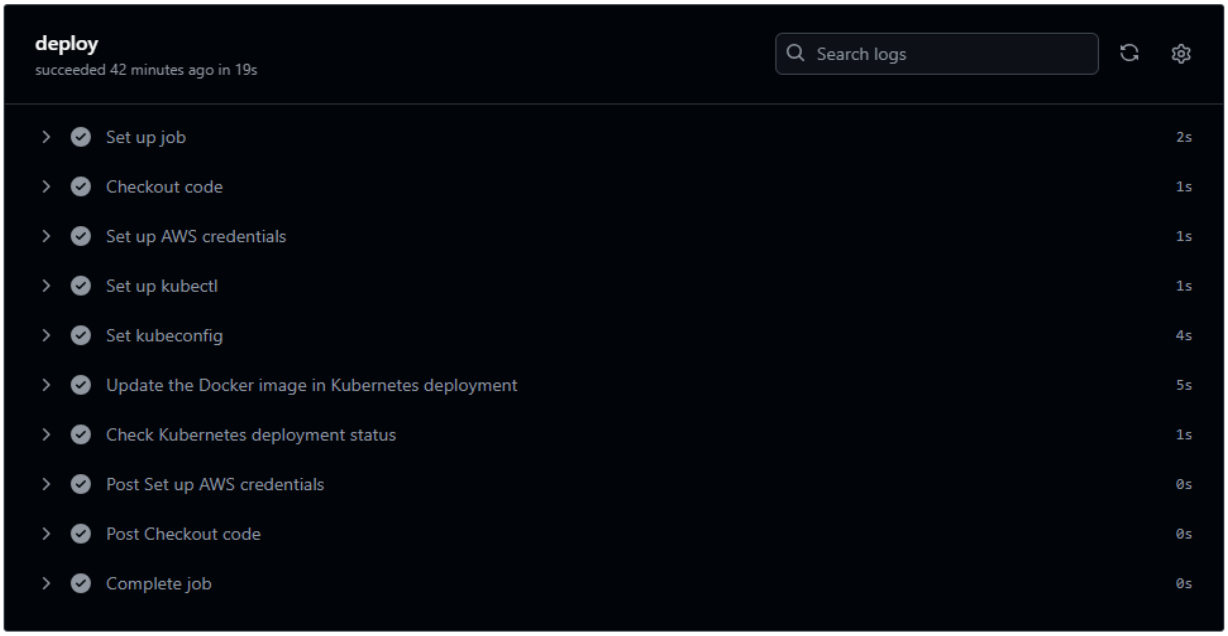
After pushing the changes in repo, Github actions work flow stages :-



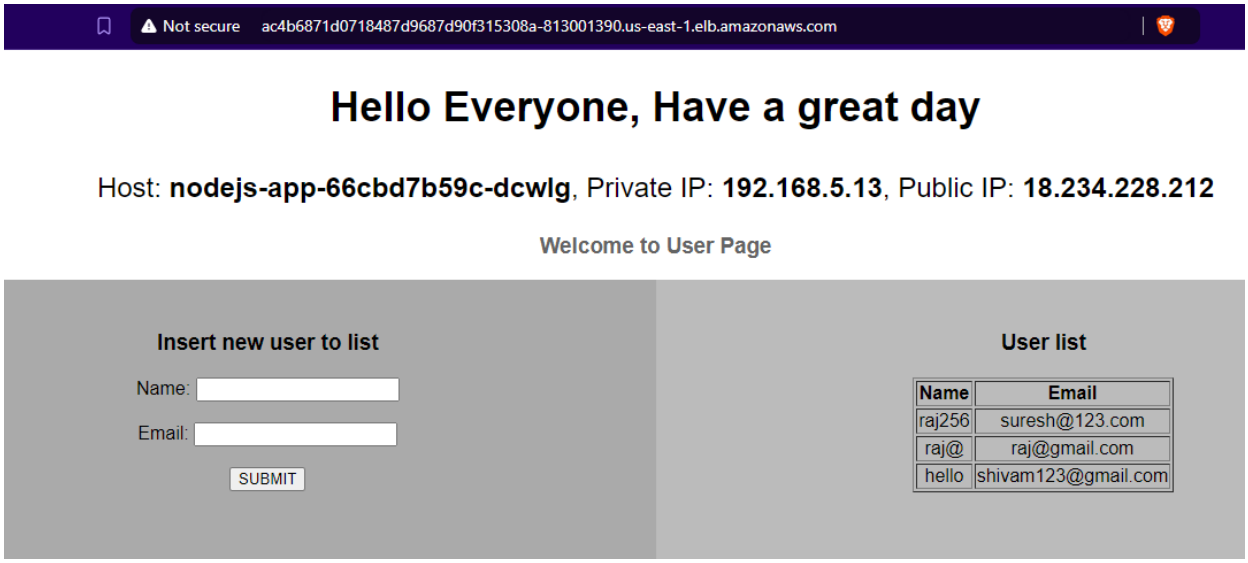
Build stage workflow:-



Deploy-stage-workflow:-



Updated changes on the webpage after pushing new code to GitHub and deploying the new Docker image to the EKS cluster via the CI/CD pipeline



Verifying the health of the deployed application using a /health endpoint.

Adding code for **/health** endpoint code in **index.js** file and commit the code:

```
// Health check route
app.get('/health', (req, res) => {
   res.status(200).json({
      status: 'healthy',
      message: 'Server is running smoothly'
   });
});
```

Checking health Endpoint