

DevOps Capstone Project

DEPLOYING A MOVIE LISTING WEBSITE USING AWS CLOUD

Team Members:

- 1. T Uday Bhaskar - 20A91A1252**
- 2. B Abhishek - 20A91A0571**
- 3. T Rama Reddy - 20A91A05C0**
- 4. V Nithin Reddy - 20A91A1256**
- 5. V Navya - 20A91A1255**
- 6. K Charan - 20A91A1264**
- 7. P Prasad Reddy - 20A91A0547**
- 8. P Devendra - 20A91A0551**

CHAPTER	PAGE NO
ABOUT OF THE PROJECT	1
TOOLS AND TECHNOLOGIES USED IN THIS PROJECT	2-3
IMPLEMENTATAION	4-14
AWS DEPLOYMENT DIAGRAM	15
OBESRVATIONS	16
CONTRIBUTION AND CHALLENGES FACED BY TEAM MEMBERS	17-18
CONCLUSION	19

ABOUT THE PROJECT:

The Movie Listing website is a web application that allows users to upload and view movie details. The website uses ReactJS as the frontend, NodeJS as the backend, and MongoDB as the database. The website allows users to upload movie details, including images. Initially, the images are stored in local storage, but as part of the deployment process on AWS, the images are moved to an S3 bucket for better scalability.

The deployment process involves several steps, including replacing local storage with S3, migrating the database to MongoDB Atlas, deploying the backend in an EC2 instance using Docker, modifying the frontend code to fetch data from the backend, deploying the frontend using Docker into an EC2 instance, creating a load balancer, and using DNS to point to the IP. The use of Docker ensures that the deployment process is consistent across different environments and reduces the chances of deployment-related issues.

The deployment on AWS ensures that the website is scalable and can handle increasing traffic. The use of an EC2 instance and load balancer ensures that traffic is distributed among multiple instances, and the use of MongoDB Atlas ensures that the database is scalable and highly available. The use of S3 for storing images ensures that the website can handle large amounts of data without any issues. The deployment on AWS also allows for easy monitoring and management of the website.

TOOLS AND TECHNOLOGIES USED IN THIS PROJECT:



ReactJS: ReactJS is a JavaScript library for building user interfaces. It allows developers to create reusable UI components and manage the state of the application. ReactJS is used in the frontend of the Movie Listing website.

NodeJS: NodeJS is a JavaScript runtime for building scalable backend applications. It uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. NodeJS is used in the backend of the Movie Listing website.

VISUAL STUDIO CODE: Visual Studio Code is a source-code editor that can be used with a variety of programming languages. Visual Studio Code combines the best of web, native, and language-specific technologies.

Multer: Multer is a middleware for handling file uploads in NodeJS. It allows developers to easily handle multipart/form-data, which is commonly used for uploading files. Multer is used in the backend of the Movie Listing website to handle image uploads.

AWS S3: Amazon Simple Storage Service (S3) is a cloud-based object storage service for storing and retrieving any amount of data from anywhere on the web. It is used in the Movie Listing website to store the images uploaded by users.

MongoDB Atlas: MongoDB Atlas is a cloud-based database service for MongoDB. It provides automatic scaling, automatic backups, and high availability. MongoDB Atlas is used as the cloud-based database for the Movie Listing website.

AWS EC2: Amazon Elastic Compute Cloud (EC2) is a cloud-based virtual server for deploying applications. It provides scalable computing capacity in the cloud. EC2 is used to deploy the backend and frontend of the Movie Listing website.

Docker: Docker is a containerization technology for building and deploying applications. It allows developers to package an application and its dependencies into a container, which can then be deployed in any environment. Docker is used to package the backend and frontend of the Movie Listing website into containers that can be deployed in EC2 instances.

AWS Elastic Load Balancer: Elastic Load Balancing automatically distributes incoming application traffic across multiple targets, such as EC2 instances. It helps to improve the availability and scalability of the application. AWS Elastic Load Balancer is used to distribute incoming traffic to multiple instances of the Movie Listing website.

GitHub: GitHub is a code hosting platform for version control and collaboration on software projects. It is used in the Movie Listing website project to host the source code for the application and to manage code changes made by multiple developers.

IMPLEMENTATION:

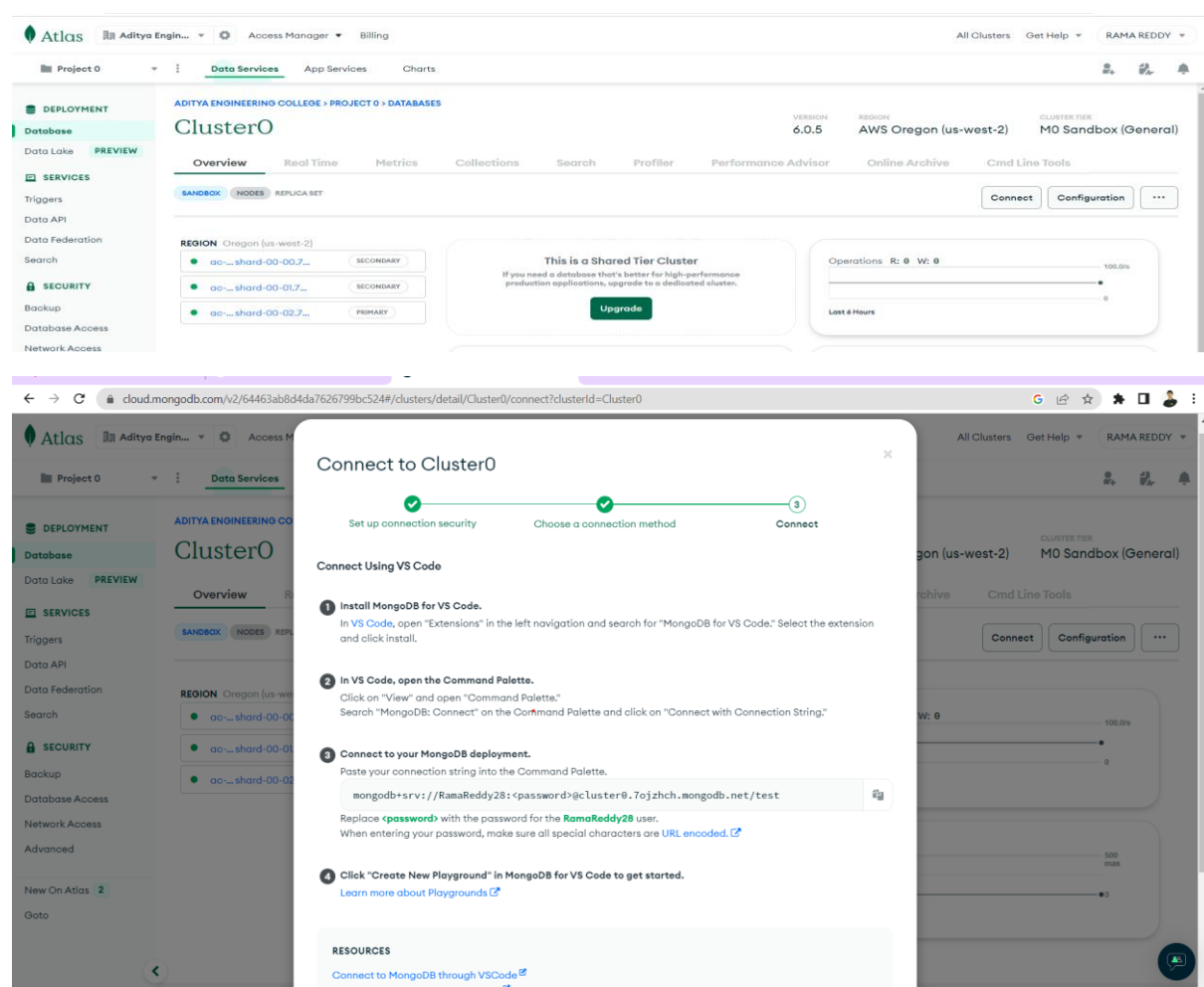
Initially, we cloned the Movie listing website which uses ReactJS as frontend, NodeJS as backend and MongoDB as database from the repo:

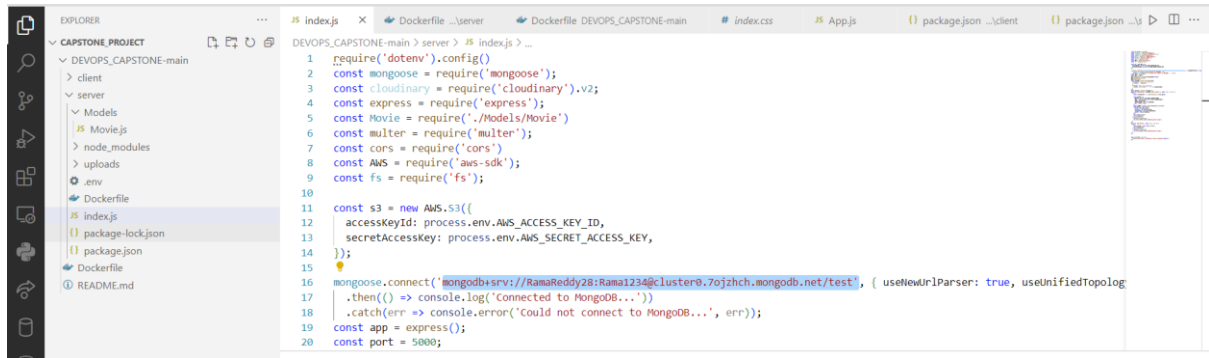
Step 1: MongoDB Atlas Account creation and connection through Server.

Here, we create MongoDB account and create a free tier cluster and given network access as allow access from anywhere and create the cluster.

After creation of the cluster then connect it. Here we able to see the connection link.

Now we must connect the server side with MongoDB, to achieve this we replace the local host link with the connection link where it is found in cluster.

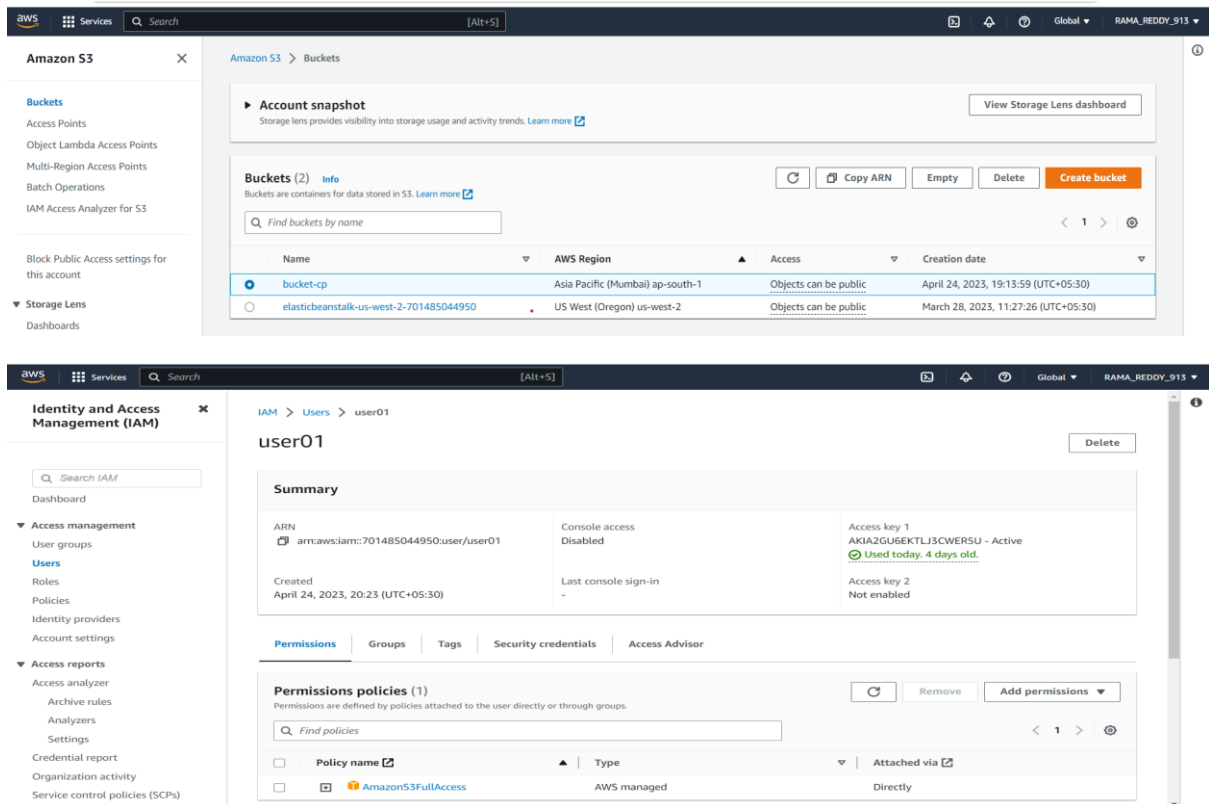




```
1 require('dotenv').config()
2 const mongoose = require('mongoose');
3 const cloudinary = require('cloudinary').v2;
4 const express = require('express');
5 const Movie = require('./Models/Movie');
6 const multer = require('multer');
7 const cors = require('cors');
8 const AWS = require('aws-sdk');
9 const fs = require('fs');
10
11 const s3 = new AWS.S3({
12   accessKeyId: process.env.AWS_ACCESS_KEY_ID,
13   secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
14 });
15
16 mongoose.connect('mongodb+srv://RamaReddy28:Rama1234@cluster0.7ojzhch.mongodb.net/test', { useNewUrlParser: true, useUnifiedTopology: true })
17   .then(() => console.log('Connected to MongoDB...'))
18   .catch(err => console.error('Could not connect to MongoDB...', err));
19 const app = express();
20 const port = 5000;
```

Step2: Creation of s3 bucket for storing images and IAM user for necessary permissions.

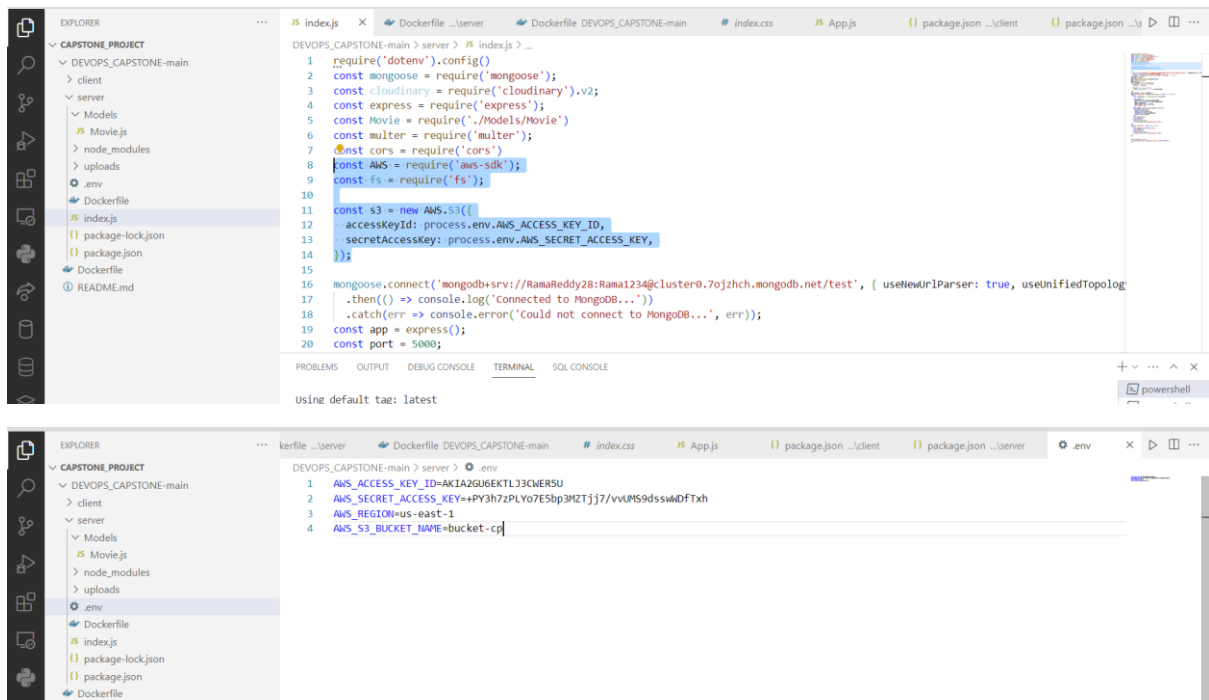
Here, we create s3 bucket and given the bucket policy and after we create IAM user and giving s3fullaccess permission and we generate an access key.



Step 3: Now time to change the backend code. For further implementation.

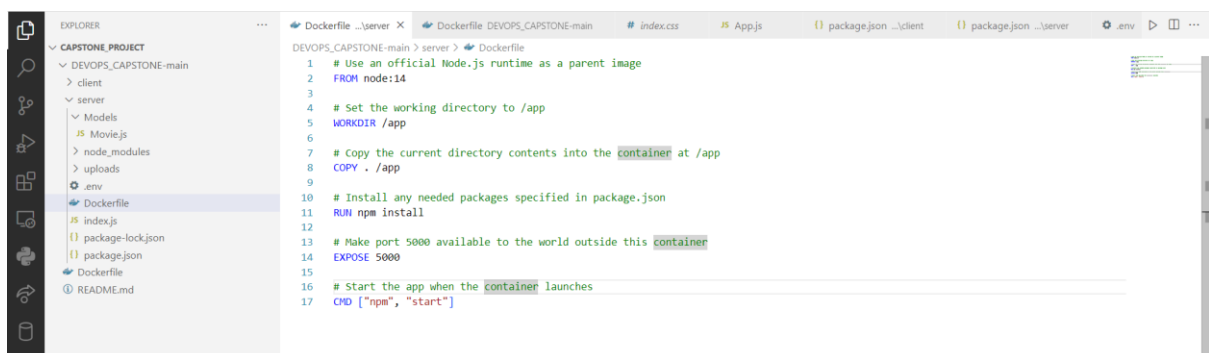
After creating the bucket, update the code to use the multer-s3 library to upload the movie images to the S3 bucket. Multer-s3 is a middleware that extends the multer package, which is used for handling file uploads in Node.js. It allows you to upload files directly to AWS S3 instead of using local storage.

In the backend code, you need to replace the existing multer code with multer-s3 configuration. Multer-s3 requires AWS access keys and the S3 bucket name to configure. You can obtain the access keys from the AWS IAM console.



Step 4: Creation of docker file for image creation in the server.

To create and push the images we must include the docker file. To build an image the docker desktop is running on background and build the image and push it into the docker hub.



Here we use the docker build -t <image name> . to build the image in docker.

We use the docker tag <image name> <username of docker hub>/<image name>

To push the image to the docker hub we use the command docker push <image name>


```

1 # Use an official Node.js runtime as a parent image
2 FROM node:14
3
4 # Set the working directory to /app
5 WORKDIR /app
6
7 # Copy the current directory contents into the container at /app
8 COPY . /app
9
10 # Install any needed packages specified in package.json
11 RUN npm install

```

```

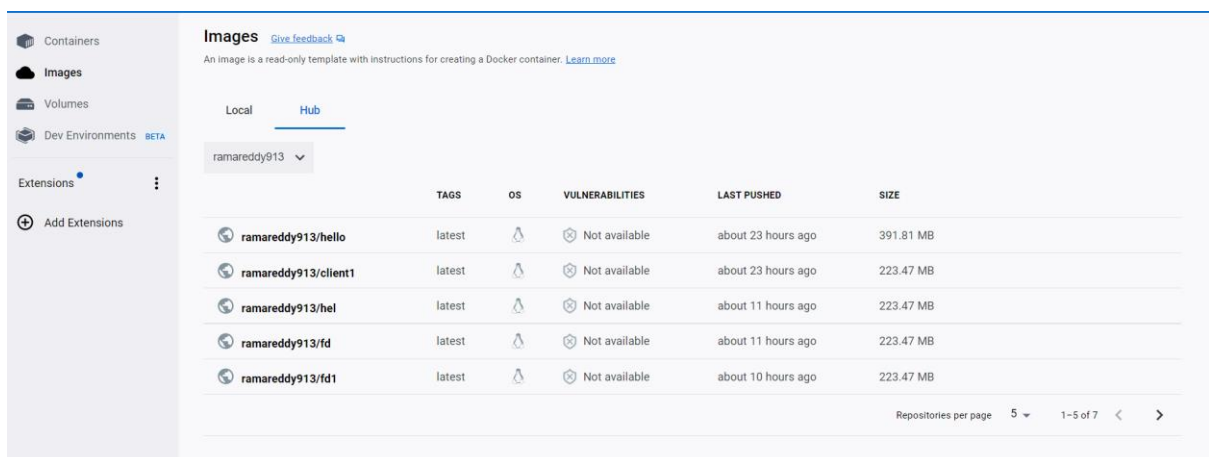
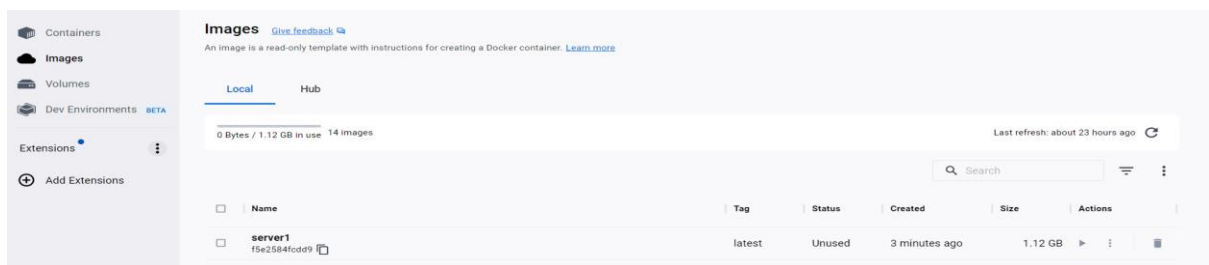
=> => writing image sha256:f5e2584fcd9a19959c7bb4f03cef2df4423d10daa03dbd1c8d46204b1bab74
=> => naming to docker.io/library/server

```

```

PS D:\Capstone_Project\DEVOPS_CAPSTONE-main\server> docker tag server ramareddy913/server
PS D:\Capstone_Project\DEVOPS_CAPSTONE-main\server> docker push ramareddy913/server
docker: "push" is not a docker command.
See 'docker --help'.
PS D:\Capstone_Project\DEVOPS_CAPSTONE-main\server> docker push ramareddy913/server
Using default tag: latest
The push refers to repository [docker.io/ramareddy913/server]
19011574c474: Mounted from ramareddy913/server1
0c6878ac35b6: Mounted from ramareddy913/server1
dbab4df45dd5: Mounted from ramareddy913/server1
005f5a015e5d: Mounted from ramareddy913/server1
3c777b951d62: Mounted from ramareddy913/server1
f8a91d5fc84: Mounted from ramareddy913/server1
cb81227abde5: Mounted from ramareddy913/server1
e01a454893a9: Mounted from ramareddy913/server1
c4566add537: Mounted from ramareddy913/server1
fe0fb3ab4a0f: Mounted from ramareddy913/server1
f1186e061f2: Mounted from ramareddy913/server1
b20ba7477754: Mounted from ramareddy913/server1
latest: digest: sha256:ead47b9ad2bce2c6d09c258059301bf98a4670ecec1e3b4b0194a52a451 size: 2845
PS D:\Capstone_Project\DEVOPS_CAPSTONE-main\server>

```

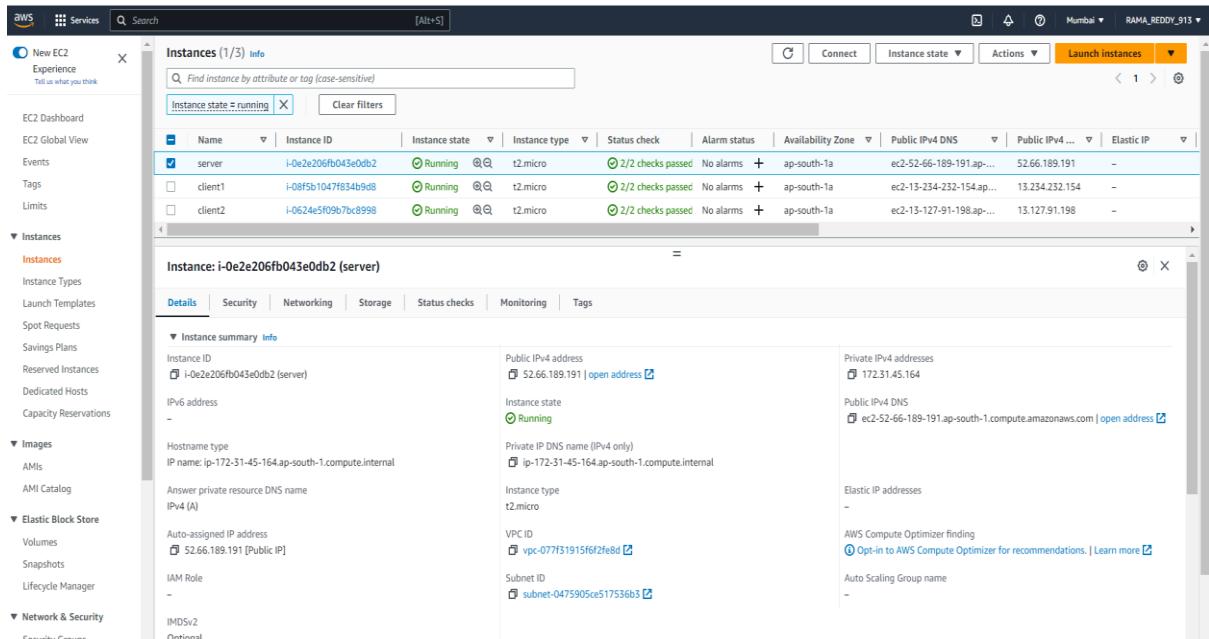


Step 5: Deploy Backend in EC2 instance:

Launch an EC2 instance and configure it with the required dependencies and packages.

Build a Docker image for the NodeJS backend application and push it to AWS ECR or Docker Hub.

Deploy the Docker image on the EC2 instance using Docker.



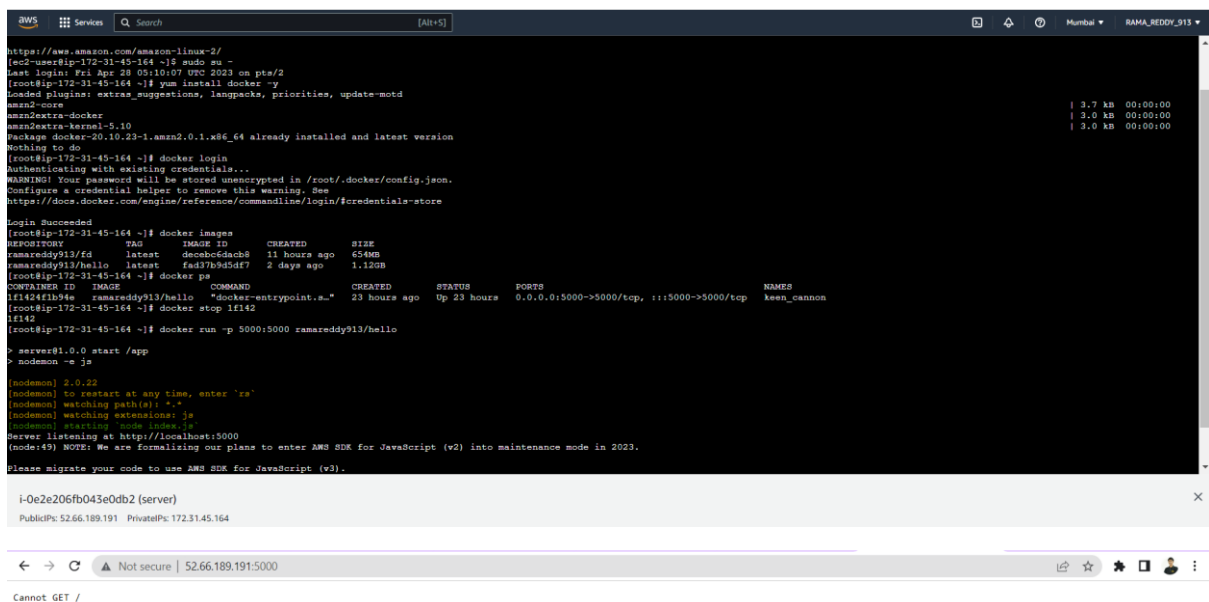
Connect EC2 instance and move to root user and install all necessary plugins and install docker in the

Instance and use docker login command to login into our docker hub account and use the command

`docker pull <user name>/<image name>` command to pull the image.

Now run the image using the command `docker run -p 5000:5000 <username>/<image name>`

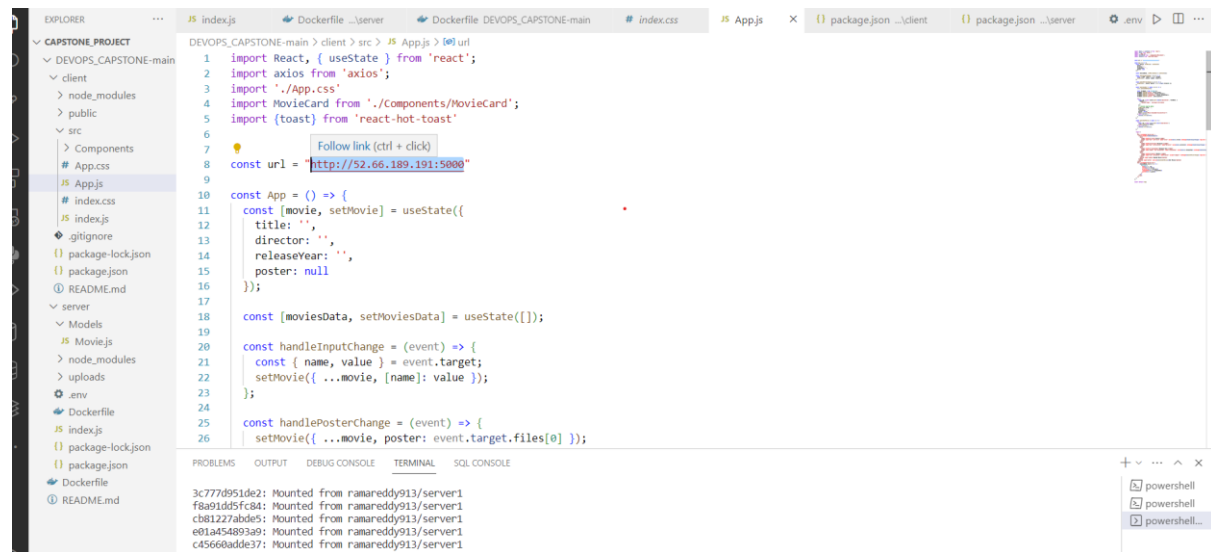
After executing the command, it shows it is connected to MongoDB.



Step 6: Modify Frontend to fetch data from Backend

To modify our frontend to fetch data from your backend, you need to update the frontend code to make API calls to the backend running on the EC2 instance. You can use the **Axios** library to make HTTP requests from your React.js application to your Node.js backend.

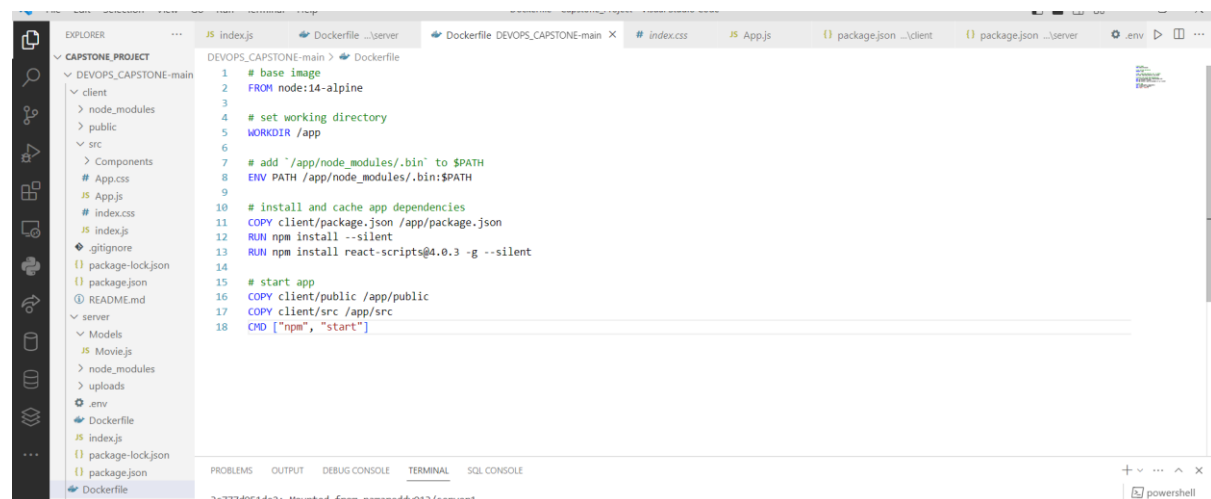
After updating the code, you need to build a Docker image for your React.js frontend application and push it to AWS ECR or Docker Hub. You can then deploy the Docker image on the EC2 instance using Docker.



```
1 import React, { useState } from 'react';
2 import axios from 'axios';
3 import './App.css';
4 import MovieCard from './Components/MovieCard';
5 import { toast } from 'react-hot-toast';
6
7
8 const url = 'http://52.66.189.191:5000';
9
10 const App = () => {
11   const [movie, setMovie] = useState({
12     title: '',
13     director: '',
14     releaseYear: '',
15     poster: null
16   });
17
18   const [moviesData, setMoviesData] = useState([]);
19
20   const handleInputChange = (event) => {
21     const { name, value } = event.target;
22     setMovie({ ...movie, [name]: value });
23   };
24
25   const handlePosterChange = (event) => {
26     setMovie({ ...movie, poster: event.target.files[0] });
27   };
28
29   return (
30     <div>
31       <h1>Movie App</h1>
32       <input type="text" value={movie.title} onChange={handleInputChange} />
33       <input type="text" value={movie.director} onChange={handleInputChange} />
34       <input type="text" value={movie.releaseYear} onChange={handleInputChange} />
35       <input type="text" value={movie.poster} onChange={handlePosterChange} />
36       <button onClick={handleInputChange}>Get Movie</button>
37       <div>
38         {moviesData.map((movie) => <MovieCard key={movie.id} movie={movie} />)}
39       </div>
40     </div>
41   );
42 };
43
44 export default App;
```

Now we must create the image as we done in the server.

So, we must give the docker file in the client and run the commands to build and push the commands.

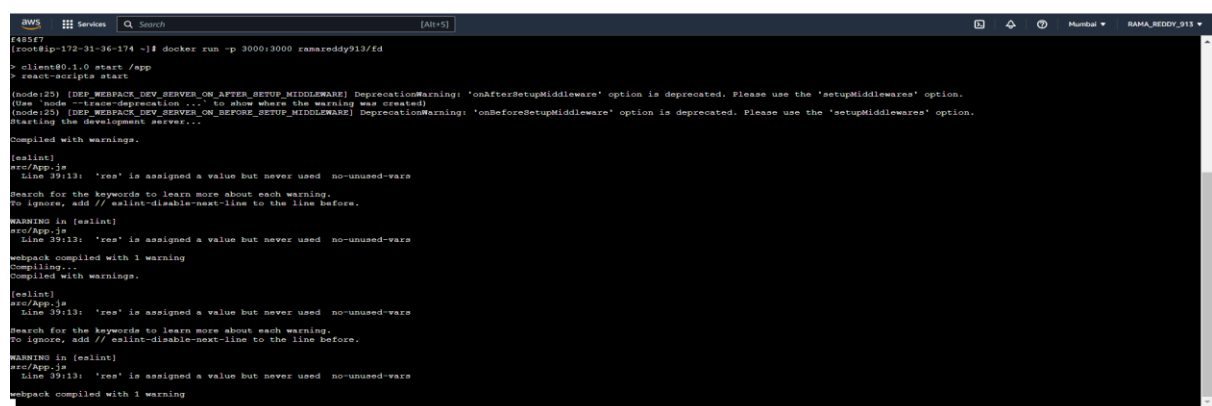


```
1 # base image
2 FROM node:14-alpine
3
4 # set working directory
5 WORKDIR /app
6
7 # add /app/node_modules/.bin to $PATH
8 ENV PATH /app/node_modules/.bin:$PATH
9
10 # install and cache app dependencies
11 COPY client/package.json /app/package.json
12 RUN npm install --silent
13 RUN npm install react-scripts@4.0.3 -g --silent
14
15 # start app
16 COPY client/public /app/public
17 COPY client/src /app/src
18 CMD ["npm", "start"]
```

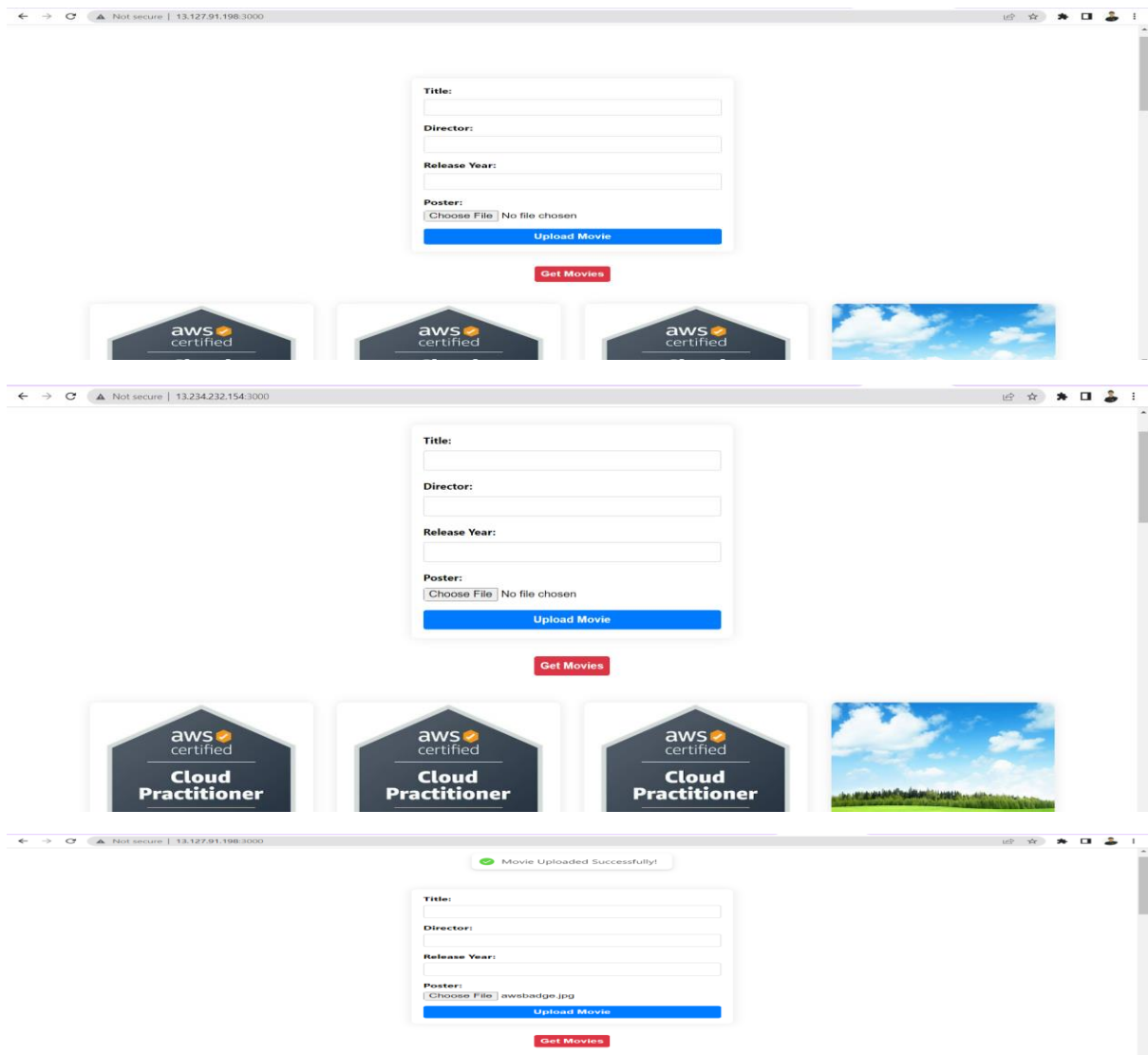
Launch an EC2 instance and configure it with the required dependencies and packages.

Here install docker in root user. First move to root user using the command `sudo su -`.

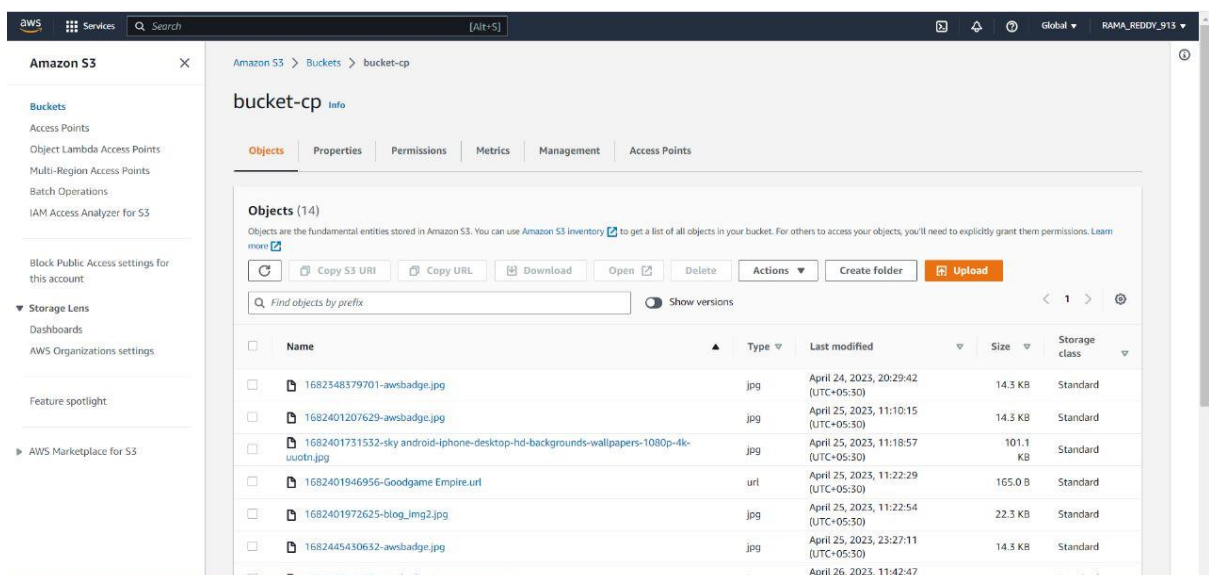
Now pull the image using push command and run the image using the command `docker run -p 3000:3000 <user name>/<image name>`



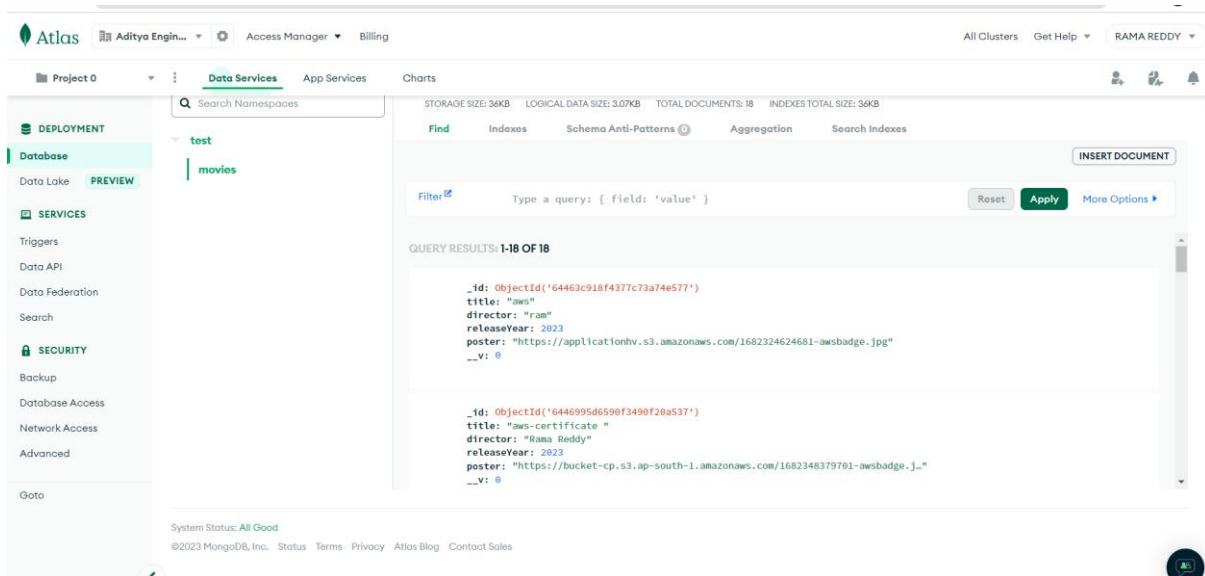
- The deployed web application. Enter the details in the displayed web page and submit the details. The details are uploaded into the mongoDB server and photos are uploaded to the attached S3 bucket.



- The uploaded photos are directly uploaded in the s3 bucket.



- The Details are uploaded in the MongoDB Document.

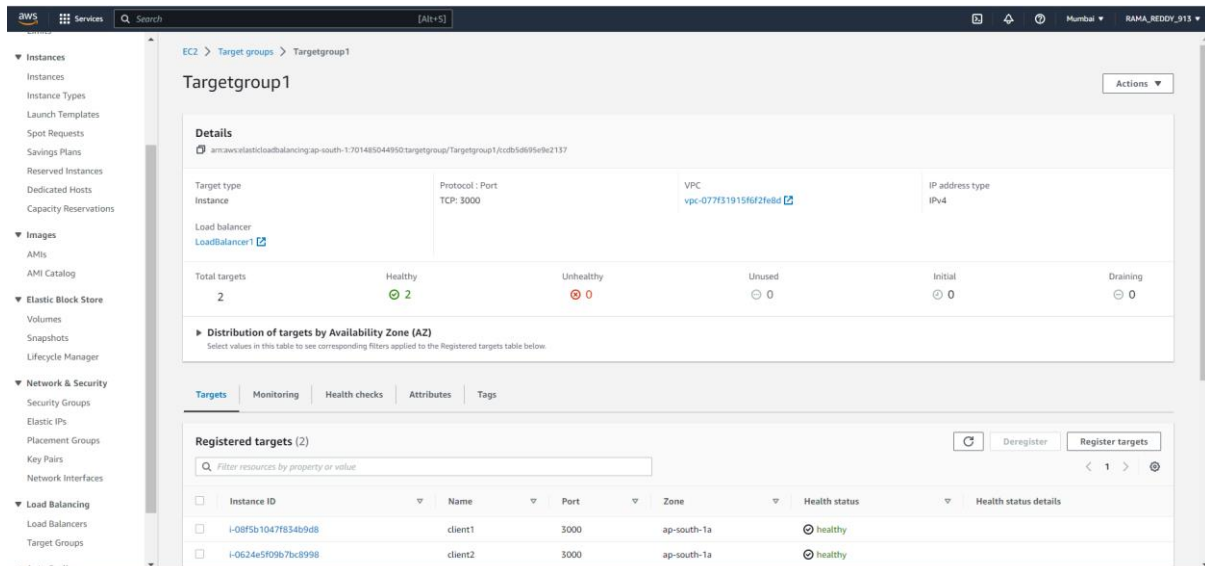


Step 6: Create Target group and load balancer.

Here we have created the target groups.

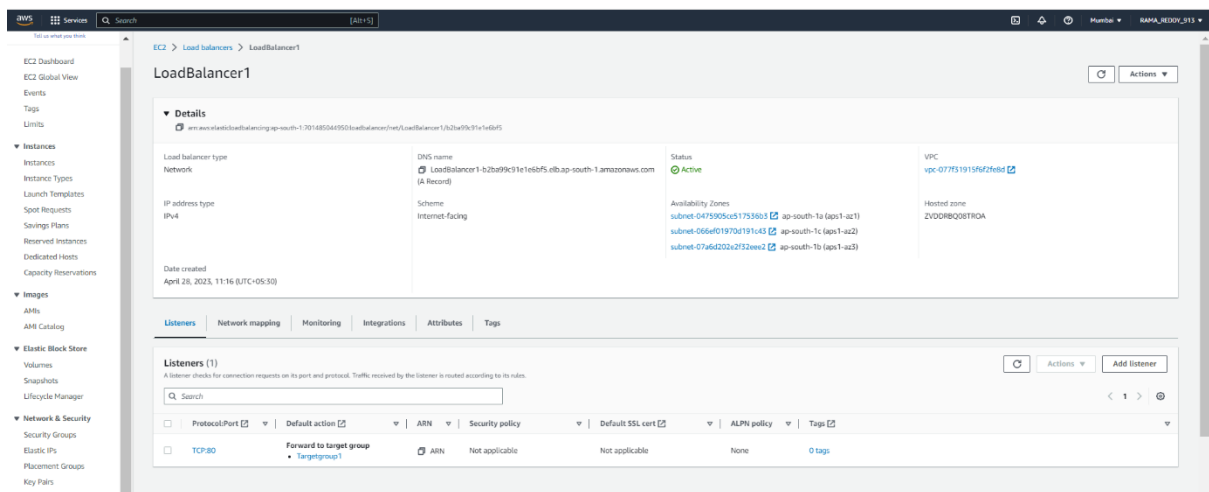
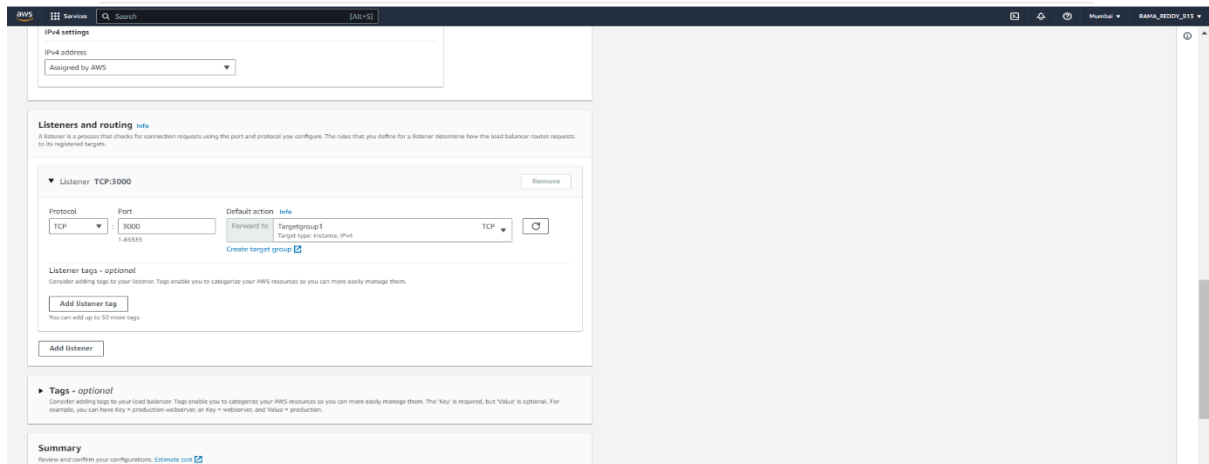
Now specify the group details as select the instances because we are doing the whole process using the ec2 instances and name the target group and select the protocol as TCP and port number as 3000, because we are run the images on the port 3000 and register the targets (client) and create the target group.

The target group is successful when it is healthy.

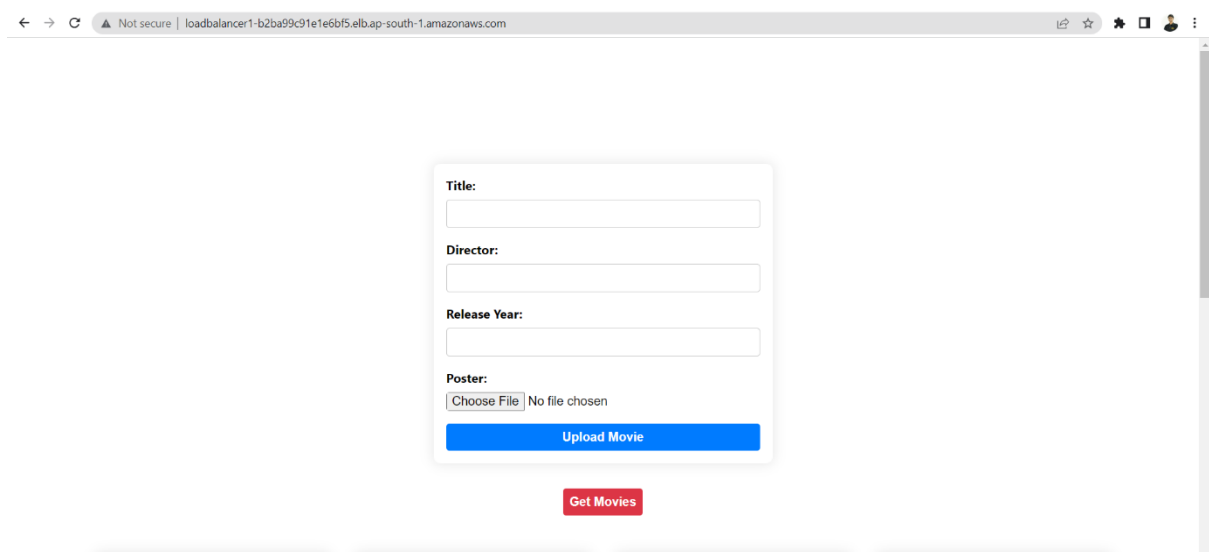


- Create the load balancer which is used to increase capacity (concurrent users) and reliability of applications. Here we create load balancer for server and client.

Here we create the load balancer by giving the basic configuration and network mapping as default and select the mapping and listeners and routing where we must give our target group.



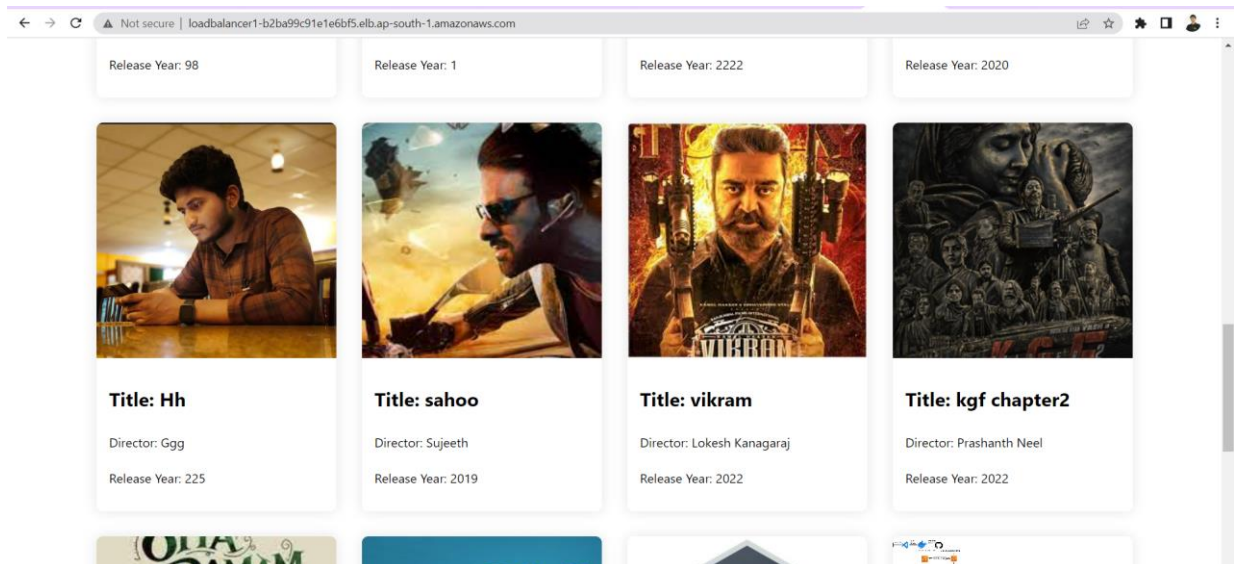
Here in this process, we get a DNS (Domain Name System).



Step 7: Use DNS to point to the load balancer:

Here we update the existing one to point to the load balancer's DNS name.

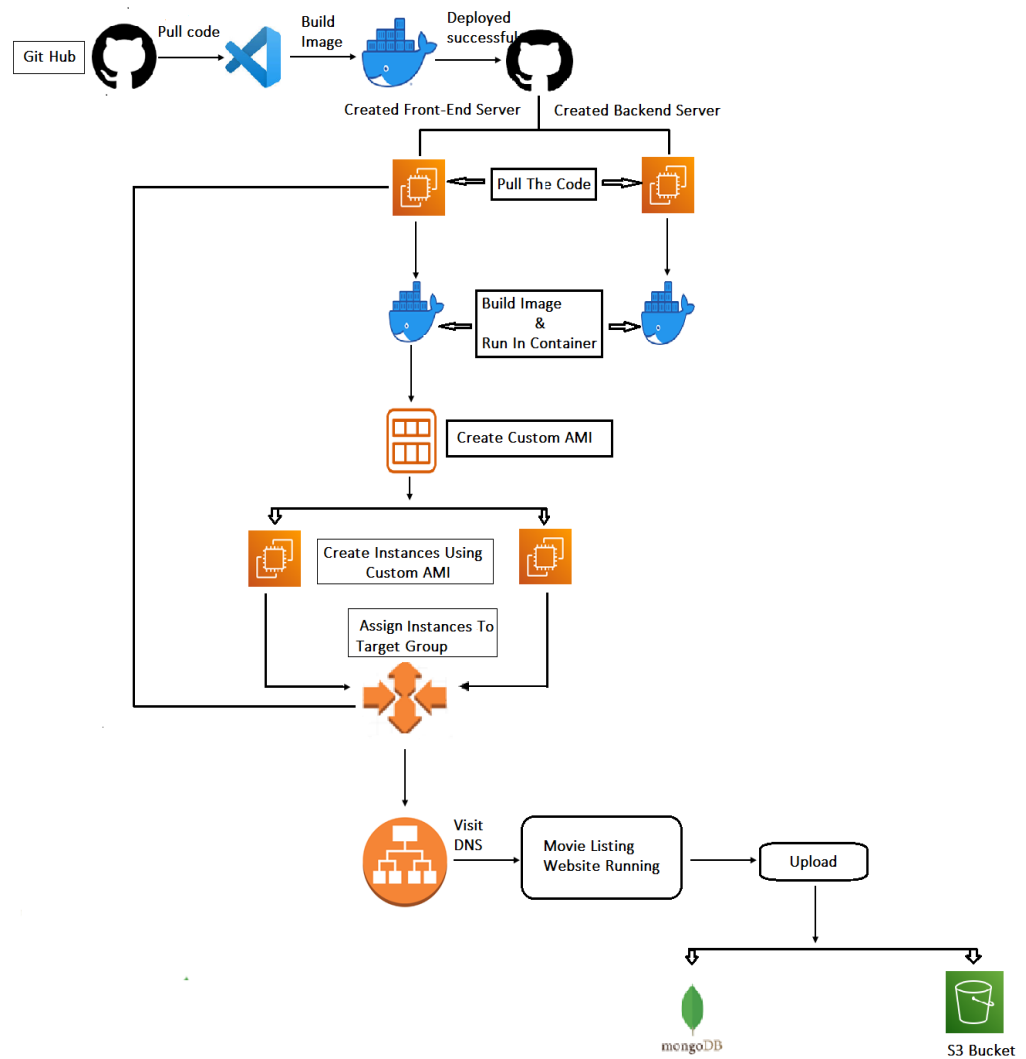
DNS Name: <http://loadbalancer1-b2ba99c91e1e6bf5.elb.ap-south-1.amazonaws.com/>



DNS LINK: <http://loadbalancer1-b2ba99c91e1e6bf5.elb.ap-south-1.amazonaws.com/>

GITHUB LINK: https://github.com/UdaybhaskarTalari/CAPSTONE_PROJECT

AWS DEPLOYMENT DIAGRAM:



OBSERVATIONS:

Some observations on the Movie Listing Website deployment on AWS:

Technology Stack: The technology stack used in the Movie Listing website is a common choice for building web applications. ReactJS and NodeJS are popular choices for building scalable and efficient web applications, and MongoDB is a commonly used NoSQL database. Using Multer for file uploads is also a common approach.

Cloud Infrastructure: The deployment of the application on AWS infrastructure is a good choice for scalability and availability. Using AWS S3 for image storage and MongoDB Atlas for database hosting provides automatic scaling and backup features. Using EC2 instances and Docker for deploying the backend and frontend is also a good choice for scalability and portability.

Load Balancing: The use of AWS Elastic Load Balancer is a good choice for distributing incoming traffic across multiple targets. This helps to improve the availability and scalability of the application.

DNS: Using AWS Route53 for DNS routing is a good choice for directing traffic to the load balancer for the Movie Listing website.

Containerization: Using Docker to package the backend and frontend into containers is a good choice for portability and consistency. Hosting the Docker images in AWS ECR is also a good choice for ease of management.

CONTRIBUTION AND CHALLENGES FACED BY TEAM MEMBERS:

Task 1: Connecting the Server with Atlas MongoDB

The task1 is done by Devendra, Prasad, and Navya. Here they are done the process of connecting the server with Mongo db. They Faced some challenges like the connection is only with the only their IP address, then they resolve it by making connection allow to access anywhere in network access option.

Task 2: Creating IAM user and s3.

The task2 is done by Devendra, Charan, and Prasad. They create s3 bucket with necessary bucket policies and created the IAM user.

Task 3: Configuring the Application Code with S3-multer.

The task3 is done by Uday, Abhishek and Rama reddy. They configure the Application code with s3-multer. They faced the challenges that the images are uploaded into s3 bucket and configuration also having some mistakes, they rectify my modifying the code in the index.js file and .env file.

Task 4: Containerization of the code using Docker file.

The task 4 is done by Nithin, Abhishek, and Rama reddy. They done the containerization of the code using docker file. Here created the docker file in server and client for building and pushing the image to docker.

Task 5: Deploying server on EC2 using Docker.

The task 5 is done by Navya, Devendra and Uday. They deploying server on EC2 using the docker. They faced the problems of the images are not pulling into Docker hub. They rectify it by correct configuration ec2 instances and docker login in ec2 instances.

Task 6: Deploying client on EC2 using Docker.

The task 6 is done by Uday, Charan and Prasad. They deployed Client on EC2 using the docker. They challenges are the client is not responding and the not uploading the images. They resolved it by changing the port number in app.js file in client and making proper configurations of Docker file in client.

Task 7: Creation a target group and Load Balancer.

The task 7 is done by Uday, Rama reddy, Navya. They created the target grouo and load balancer for our website. They faced problems like load balncer is not working and we rectify it by running the images with same port number and giving Tcp as protocol and using the same port number in the client instances.

Task 8: Creating AWS Deployment Diagram.

The task 8 is done by Nithin, Abhishek and Uday. They tried very hard for creating AWS Deployment Diagram.

Task 9: Preparing Project Documentation.

The task 9 is done by Rama reddy, Nithin and Abhishek. They kept their full efforts in preparing the documentation and gathered information and put it in the documentation

CONCLUSION:

At the end of this project, we successfully deployed frontend using docker into EC2 instance. The deployment of the Movie Listing website on AWS involved several technologies, including ReactJS, NodeJS, MongoDB, AWS S3, AWS EC2, AWS Elastic Load Balancer, and AWS Route53. The project involved replacing the local storage with AWS S3 for storing images, replacing the local database with MongoDB Atlas cloud infrastructure, and deploying the backend and frontend code on AWS EC2 instances using Docker containers. A load balancer was also created to scale the website traffic and a DNS was used to point to the website's IP.

The project highlighted the complexities involved in deploying a web application on AWS, including setting up the necessary infrastructure components, configuring Docker containers, integrating AWS services, troubleshooting issues, and optimizing costs. To overcome these challenges, it was necessary to have a solid understanding of the technologies involved, as well as experience in deploying applications on cloud infrastructure.

Overall, the successful deployment of the Movie Listing website on AWS demonstrates the potential benefits of cloud infrastructure, such as scalability, flexibility, and cost-effectiveness. However, it also highlights the need for careful planning, best practices, and expertise to ensure a successful deployment.