DevOps CAPSTONE PROJECT

DEPLOYING A MOVIE LISTING WEBSITE INTO AWS CLOUD INFRASTRUCTURE WITH PROPER SCALING

Group-4

Team Members:

1) VENKATA KRISHNA REDDY (TL)	20A91A1239
2) BALAJI REDDY	20A91A0582
3) SUNIL	20A91A05B1
4) VASEEM	20A91A05B8
5) KARTHIK	20MH1A4923
6) YAMUNA	20A91A0512
7) ISWARYA	20A91A0561
8) BLESSY	20A91A0513

PURPOSE OF THE PROJECT

The purpose of the project is to deploy a website on an Amazon Elastic Compute Cloud (EC2) instance using Docker container technology.EC2 is a web service provided by AWS to easily deploy and manage virtual servers. It is highly scalable and flexible to use. Docker is a containerization tool which makes an application run on any platform. Mongo dB to replace the local database so that the data is stored in our cluster and images are stored in S3.

By the concluding phase of this project, we can host a web application on EC2 which is highly reliable, can handle vast web traffic and runs smoothly. This application can be easily maintained and updated because of docker with a load balancer attached to it.

SCOPE OF THE PROJECT

The scope of the project is to deploy a website using Amazon Elastic Compute Cloud (EC2) and Docker container technology. The project includes developing a web application and modify it according to our requirements. Configuring and setting up an EC2 instance also plays a vital role in this project. Mongo dB is used instead of the local database.

The web application is containerized using docker for easier maintenance and deployment. It involves the deployment of web application from our EC2 instance and also attaching a load balancer to reduce the web traffic load on an instance. The project can be concluded by assigning a DNS for accessing the website.

TECHNOLOGIES USED IN THIS PROJECT

AWS: AWS is a cloud computing platform that provides a range of services for individuals and businesses to store, manage, and process data and applications in the cloud. With over 200 services, including computing, storage, databases, analytics, machine learning, IoT, and security, AWS is flexible, cost-effective, and reliable. Some popular AWS services include EC2 for computing, S3 for storage, RDS for databases, Lambda for serverless computing, DynamoDB for NoSQL databases, and ECS for container management.

Docker: Docker is an open-source platform that allows developers to create, deploy, and run applications in containers. Containers are self-contained, portable units of software that include all the components and dependencies an application needs to run. Docker simplifies application development and deployment by enabling developers to package their applications in containers and run them on any system with Docker installed. This provides greater flexibility, portability, and scalability for applications.

GIT: Git is a version control system that allows developers to track changes to their code and collaborate with others on a project. It is a free and open-source tool that provides features such as branching, merging, and version history. Git's distributed architecture enables developers to work independently and collaborate with others, making it a powerful tool for managing software development projects.

Web Development: Web development is the process of creating and maintaining websites and web applications. It involves a combination of programming, design, and content creation to create web pages that are functional and visually appealing. Web development can include both frontend development, which focuses on the user interface, and backend development, which focuses on the server-side programming that powers the website.

TECHNOLOGIES STACK:

Frontend: ReactJS

Backend: NodeJS

Database: MongoDB

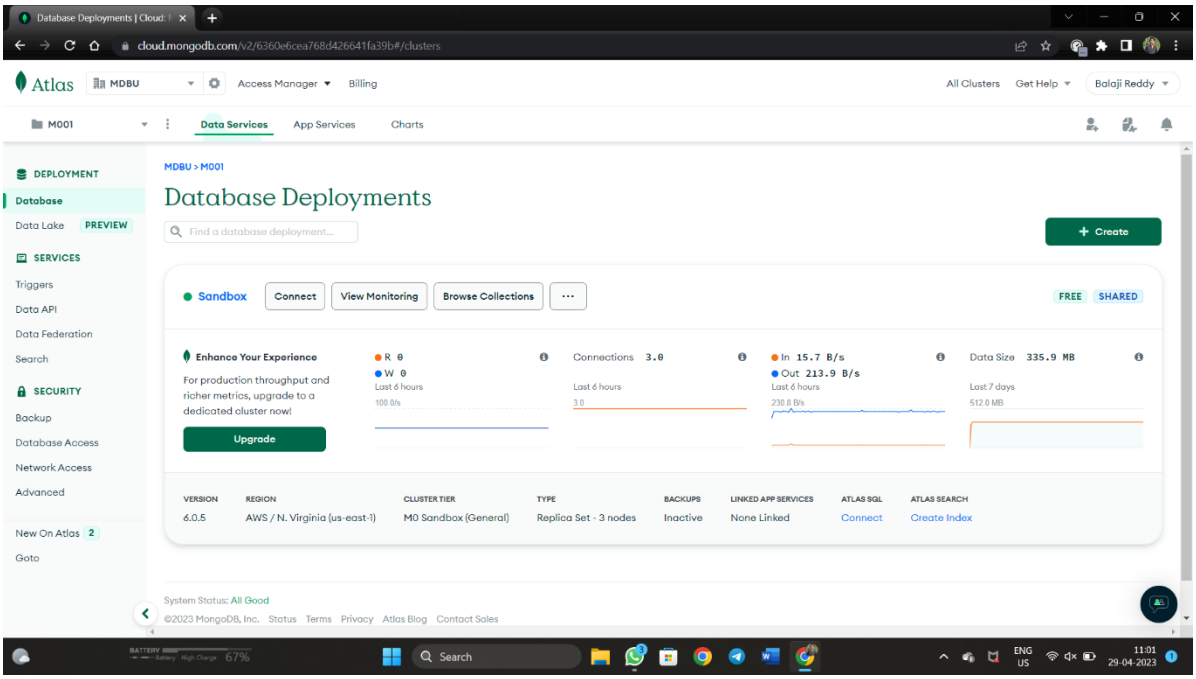
IMPLEMENTATAION and OBSERVATIONS

Step 1: Connecting the Server with Atlas MongoDB

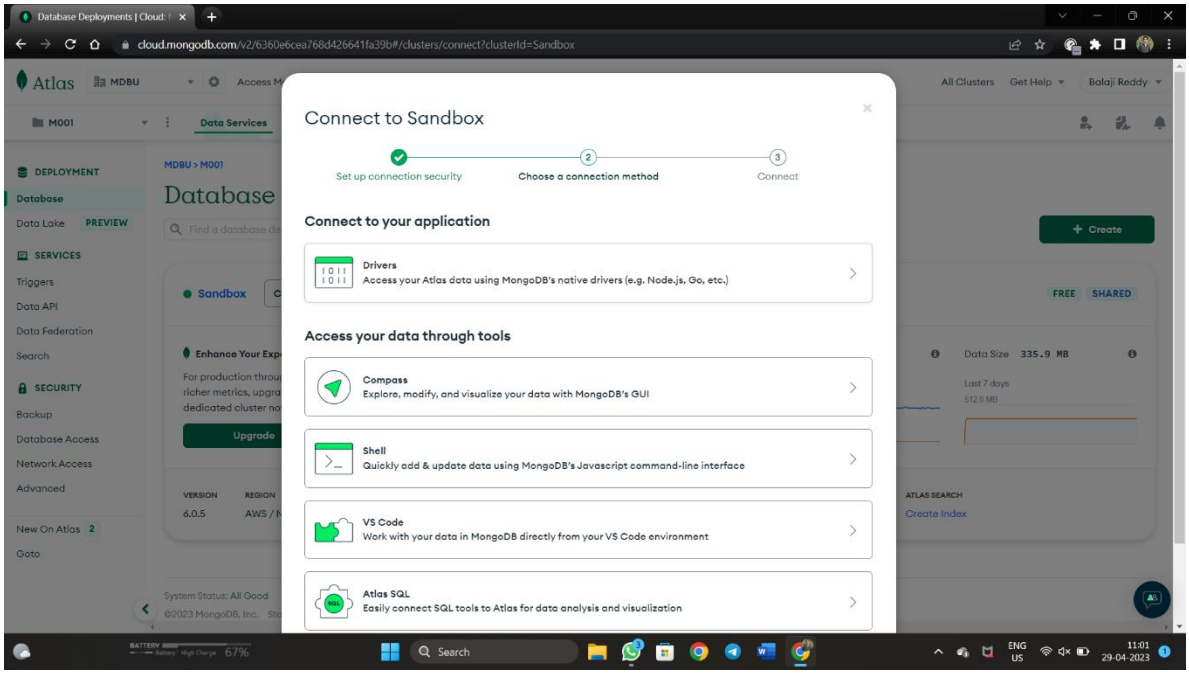
Initially, cloned the “Movie listing” website which uses ReactJS as frontend, NodeJS as backend and MongoDB as database from the repo:

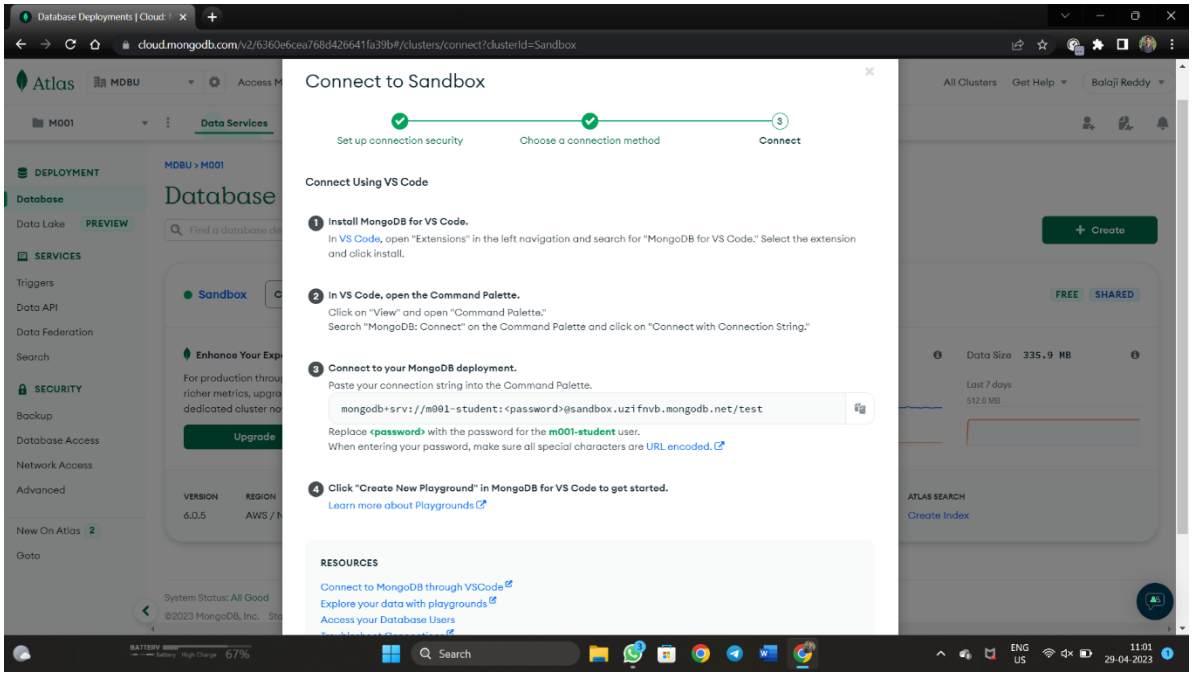
https://github.com/snehal-herovired/DEVOPS_CAPSTONE

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

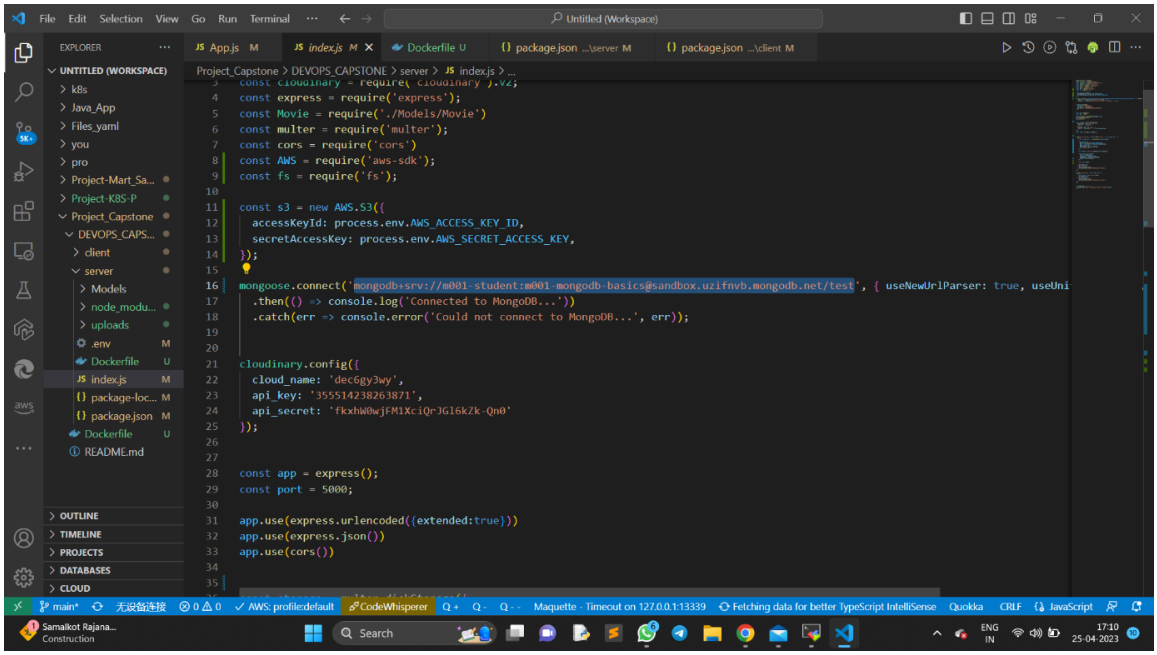


- After creation of the cluster then connect it. Here we able to see the connection link.

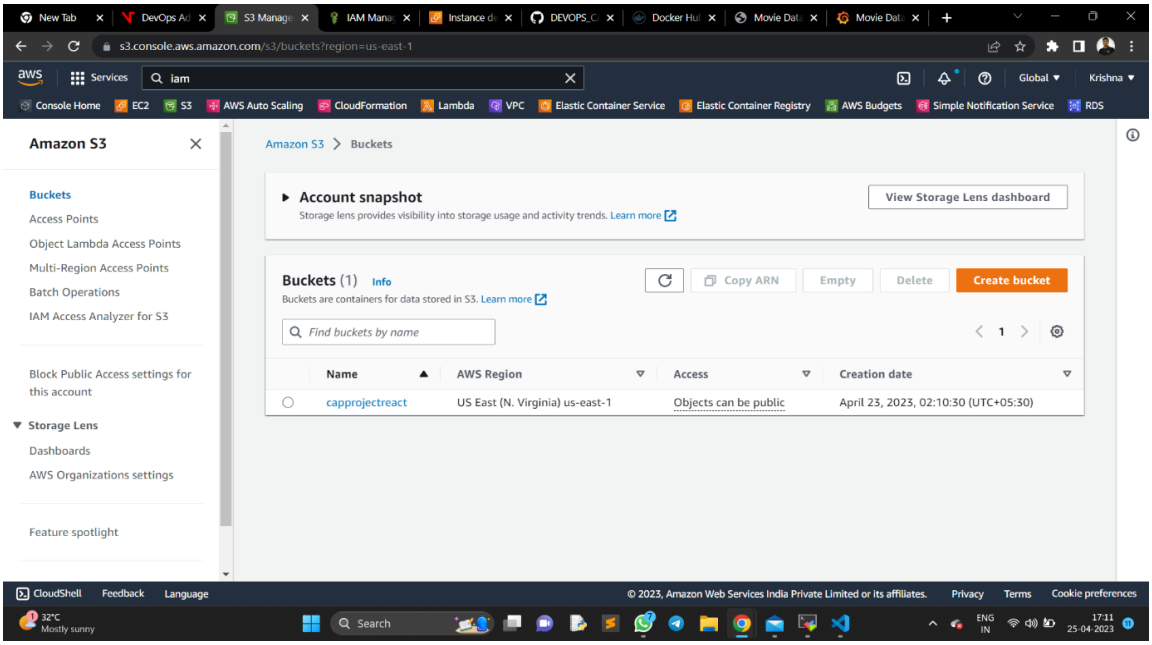




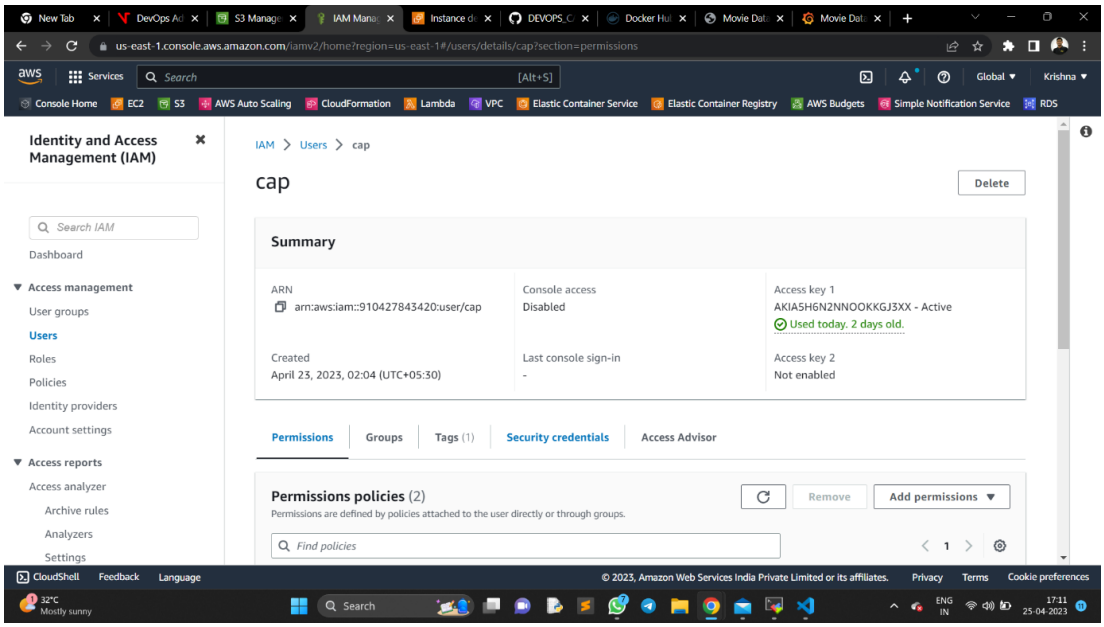
- Now we must connect the server side with MongoDB, to achieve this we replace the local host link with the connection link where it is found in cluster.



- Create a S3 bucket to store the images which will be uploaded in the website.

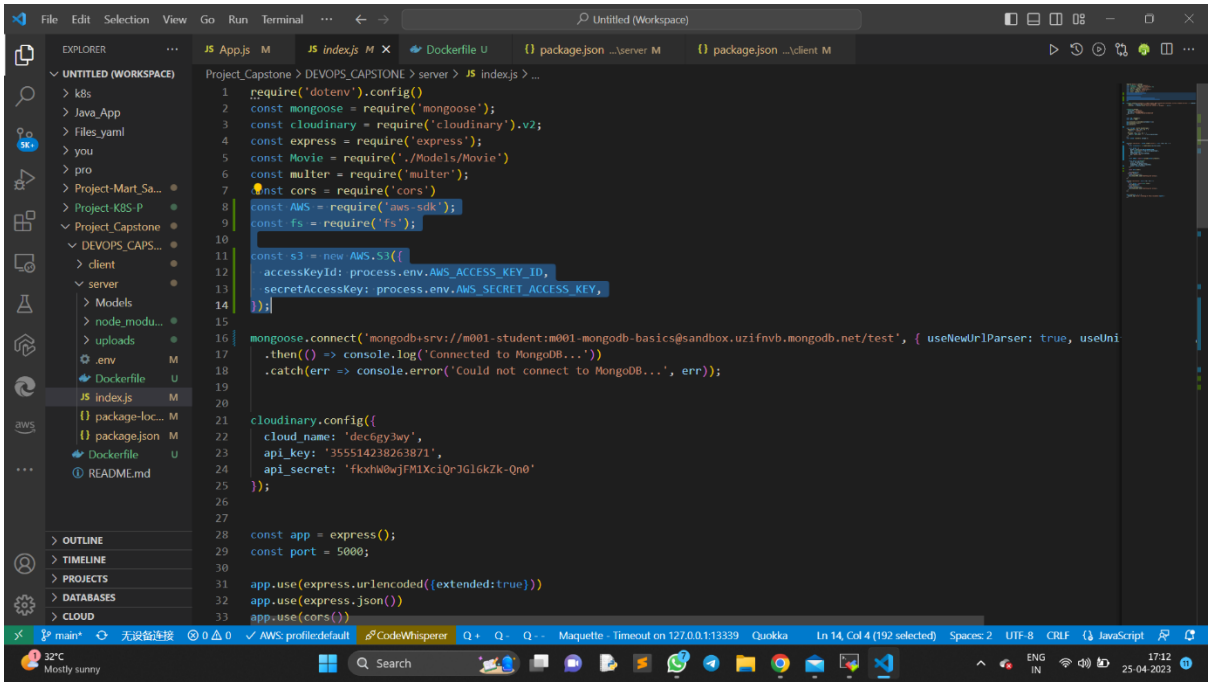


- Create an IAM user and give necessary permissions to upload the images and access it.



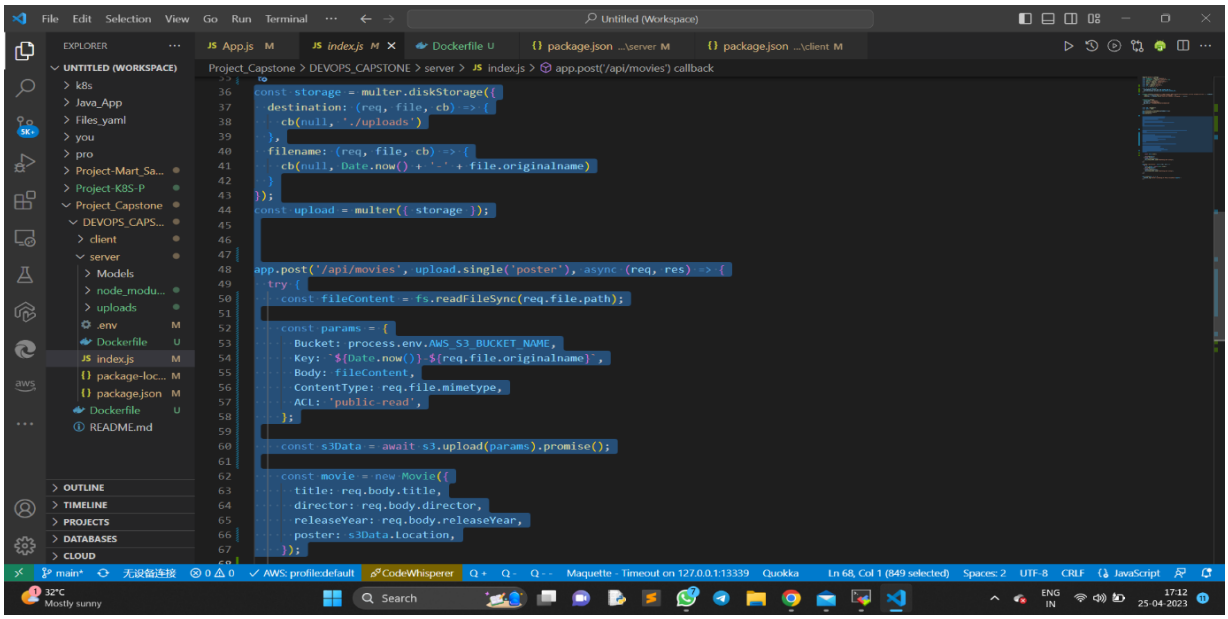
Step 2: Modifying the Frontend and Backend Code

- The AWS SDK for JavaScript supports three runtimes: **JavaScript for browser, Node.js for server, React Native for mobile development**. It also supports cross-runtime: a service client package can be run on browsers, Node.js, and React-Native without code change. Now, we add:



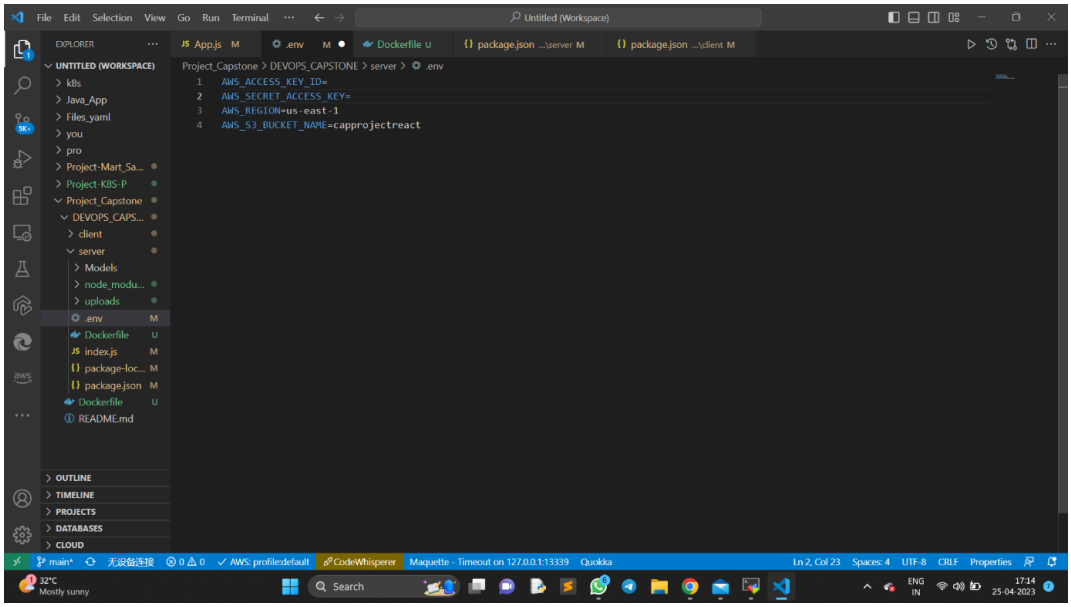
```
require('dotenv').config()
const mongoose = require('mongoose');
const cloudinary = require('cloudinary').v2;
const express = require('express');
const Movie = require('./Models/Movie')
const multer = require('multer');
const cors = require('cors')
const AWS = require('aws-sdk');
const fs = require('fs');

const s3 = new AWS.S3({
  accessKeyId: process.env.AWS_ACCESS_KEY_ID,
  secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
});

mongoose.connect('mongodb+srv://m001-student:m001-mongodb-basics@sandbox.uzifnvb.mongodb.net/test', { useNewUrlParser: true, useUni
  .then(() => console.log('Connected to MongoDB...'))
  .catch(err => console.error('Could not connect to MongoDB...', err));


cloudinary.config({
  cloud_name: 'dec6gy3wy',
  api_key: '355514238263871',
  api_secret: 'fkxhW0wjFM1XciQrJGl6kZk-Qn0'
});


const app = express();
const port = 5000;

app.use(express.urlencoded({extended:true}))
app.use(express.json())
app.use(cors())
```

- Multer is a Node.js middleware used for handling multipart/form-data, which is primarily used for uploading files. it's a function that receives the request and response object when a user from the client-side makes any request. Now, we are adding multer to our code:



```
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, './uploads')
  },
  filename: (req, file, cb) => {
    cb(null, Date.now() + '-' + file.originalname)
  }
});
const upload = multer({ storage });


app.post('/api/movies', upload.single('poster'), async (req, res) => {
  try {
    const fileContent = fs.readFileSync(req.file.path);

    const params = {
      Bucket: process.env.AWS_S3_BUCKET_NAME,
      Key: `${Date.now()}_${req.file.originalname}`,
      Body: fileContent,
      ContentType: req.file.mimetype,
      ACL: 'public-read',
    };

    const s3Data = await s3.upload(params).promise();

    const movie = new Movie({
      title: req.body.title,
      director: req.body.director,
      releaseYear: req.body.releaseYear,
      poster: s3Data.Location,
    });
```

- Now, we have replaced the access keys and ID with our credentials.



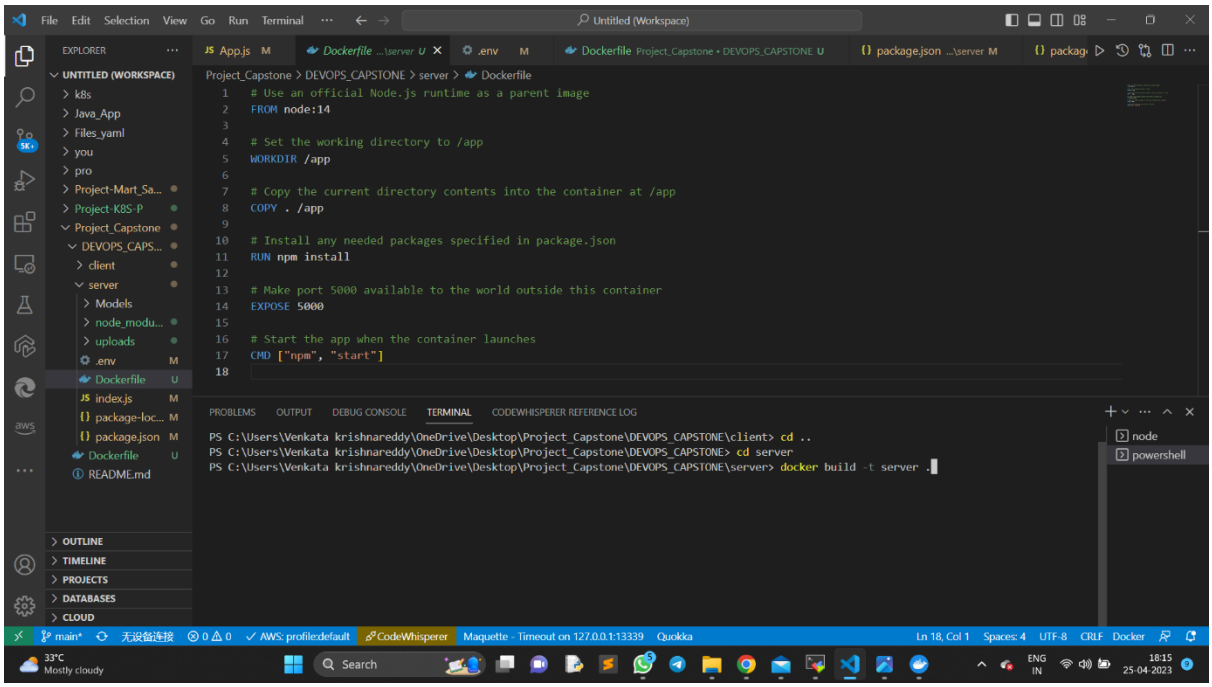
Step-3: Containerization of the code

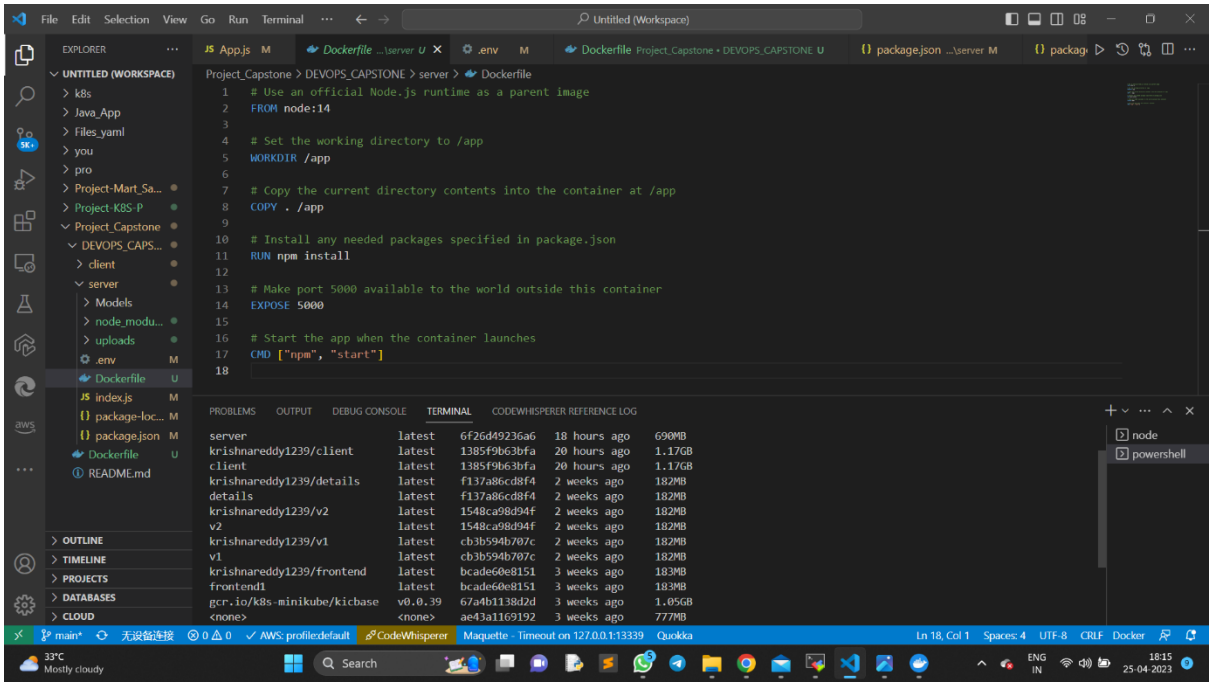
- Here, we are converting our frontend and backend code into docker images to run. Docker can build images automatically by reading the instructions from a Docker file. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Hence, the dockerfiles are written for both server and client.

Dockerfile for server:

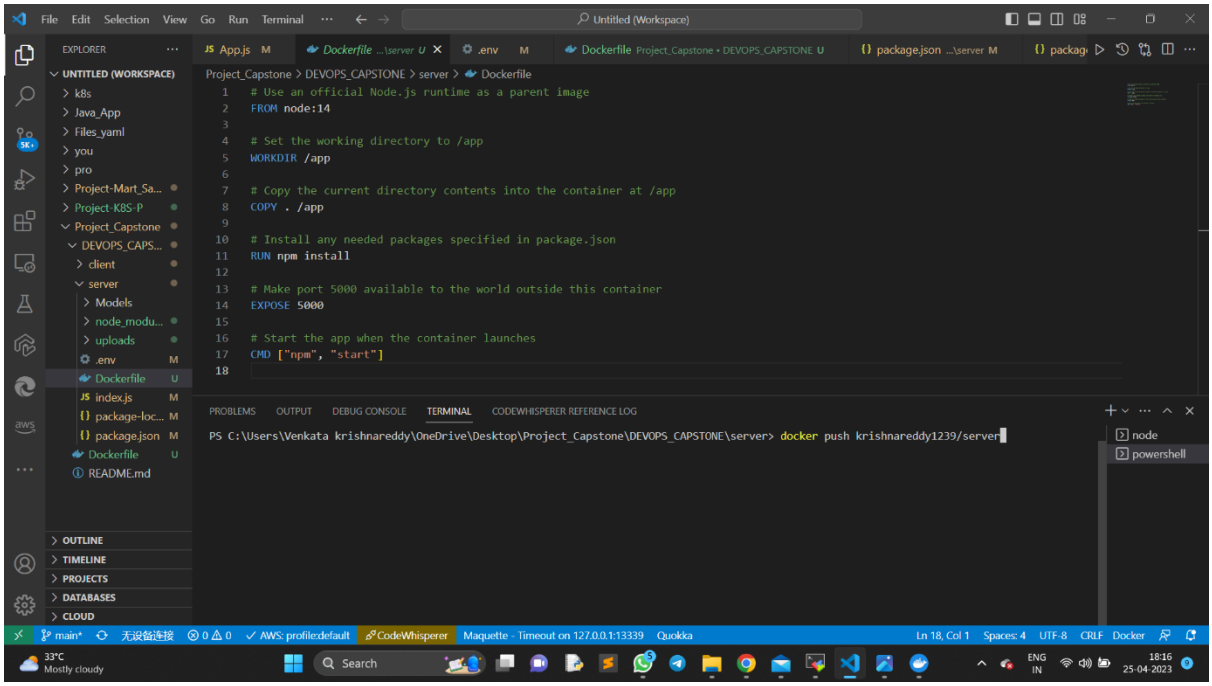
```
# Use an official Node.js runtime as a parent image
FROM node:14

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY. /app

# Install any needed packages specified in package.json
RUN npm install

# Make port 5000 available to the world outside this container
EXPOSE 5000

# Start the app when the container launches
CMD ["npm", "start"]
```

- The docker build command builds Docker images from a Dockerfile and a “context”. A build's context is the set of files located in the specified PATH or URL.

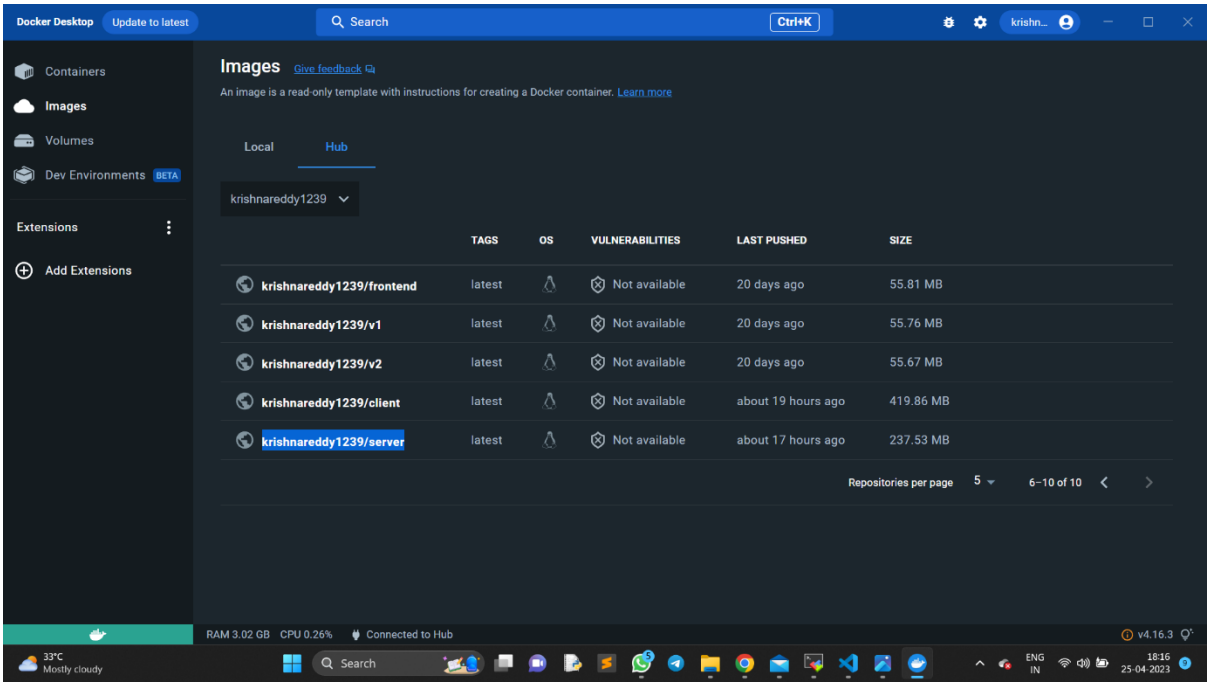




- Docker Push is a command that is used to push or share a local Docker image or a repository to a central repository; it might be a public registry like https://hub.docker.com or a private registry or a self-hosted registry.

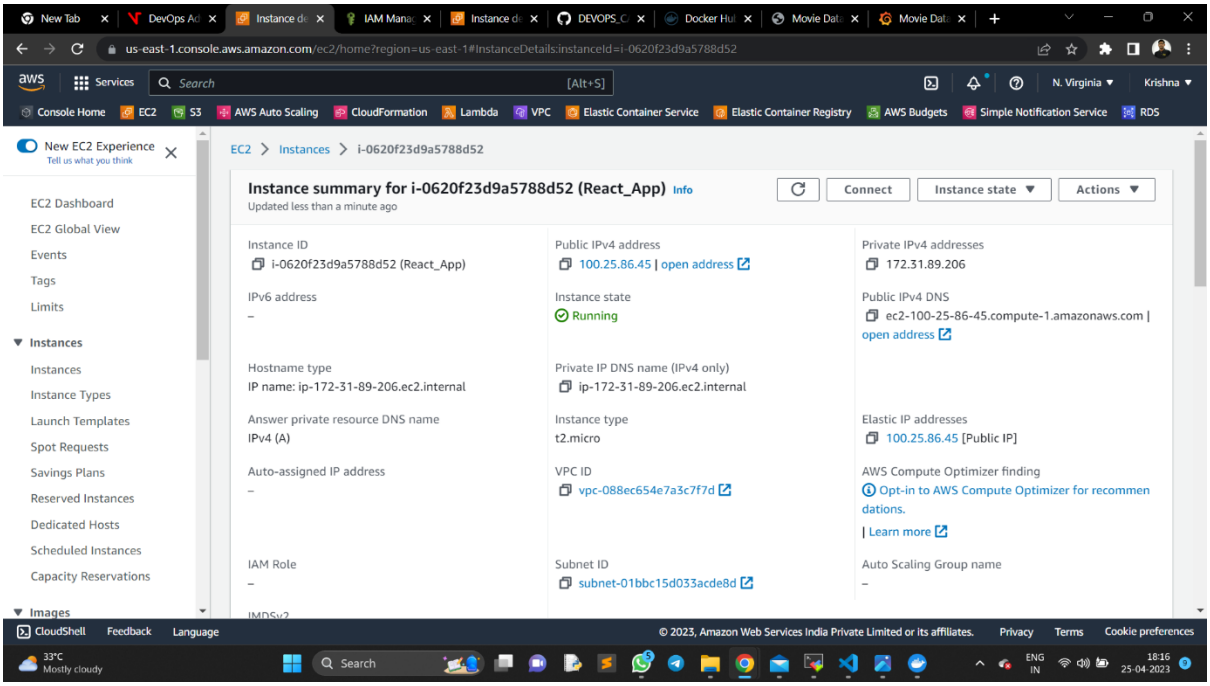


- The images built in previous steps are stored in docker as shown below:

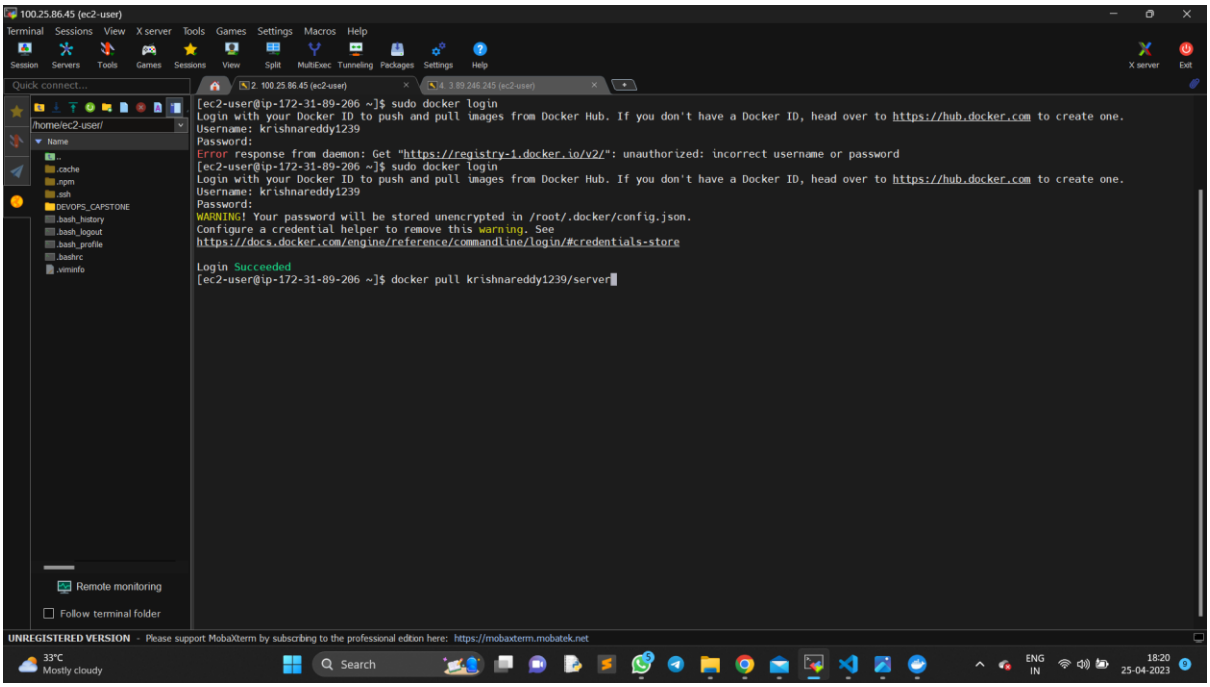


Step 4: Deploying server on EC2 using Docker

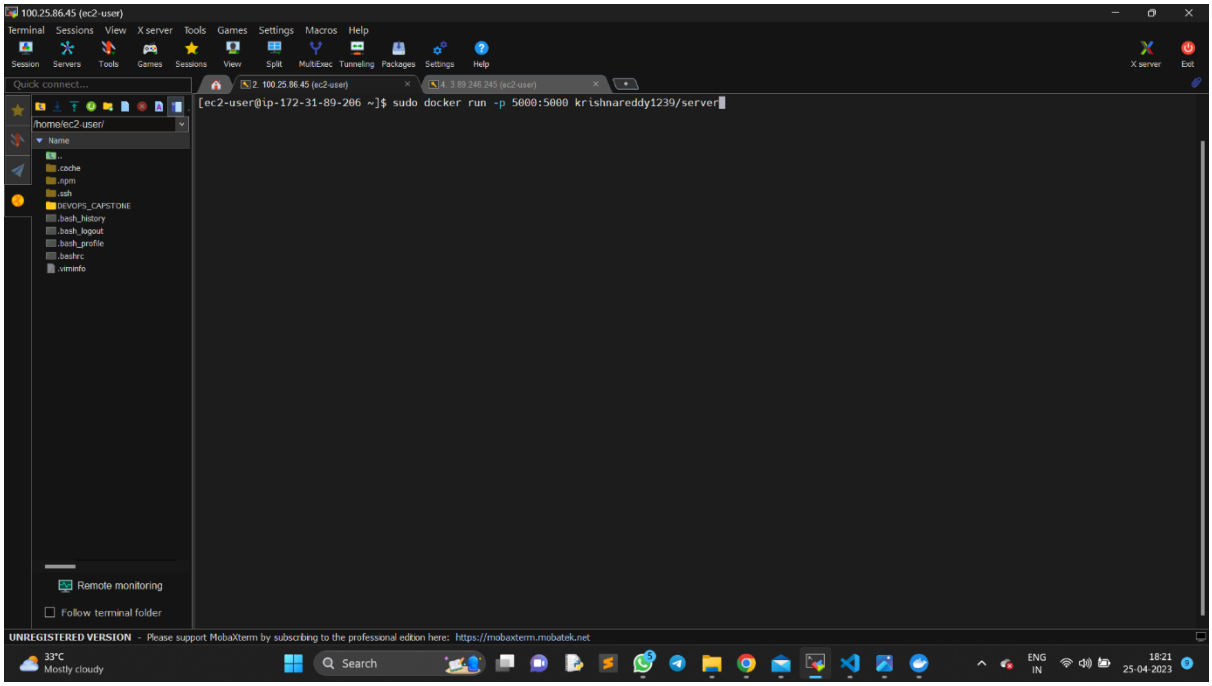
- Now create an EC2 instance to deploy the backend and it is deployed by using docker.

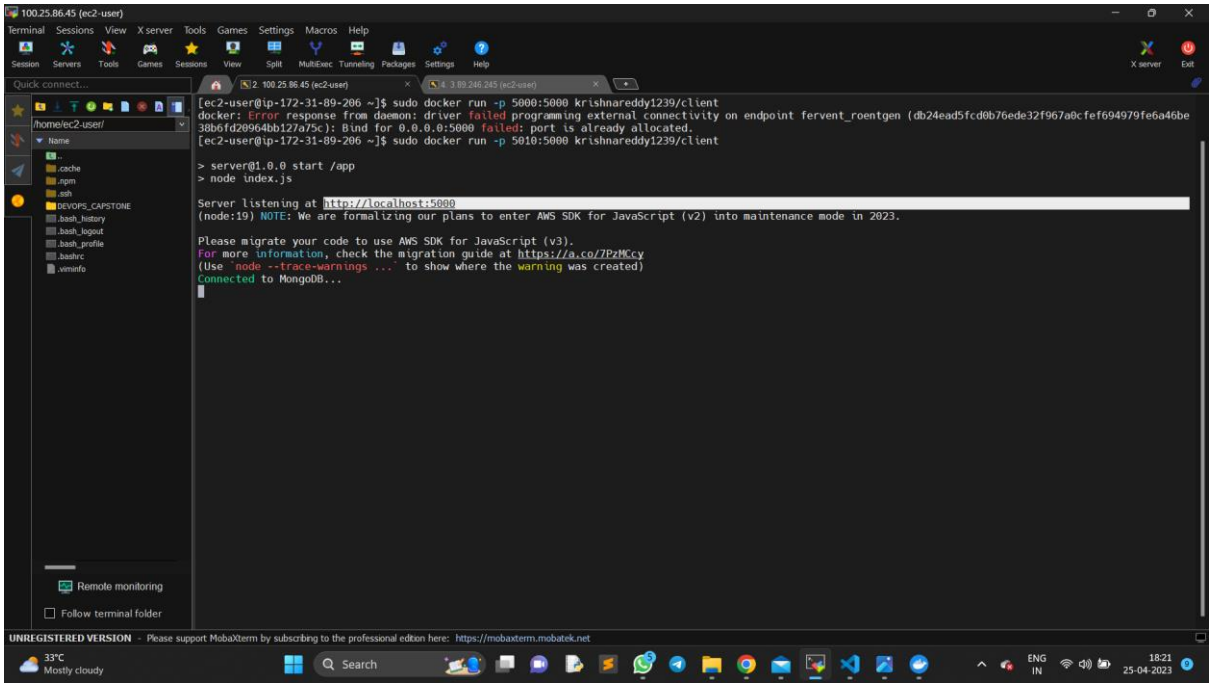


- Connect EC2 instance through local machine using MobaXterm and login into docker through EC2 instance by using the command **sudo docker login** and pull the docker image using **docker pull** command.



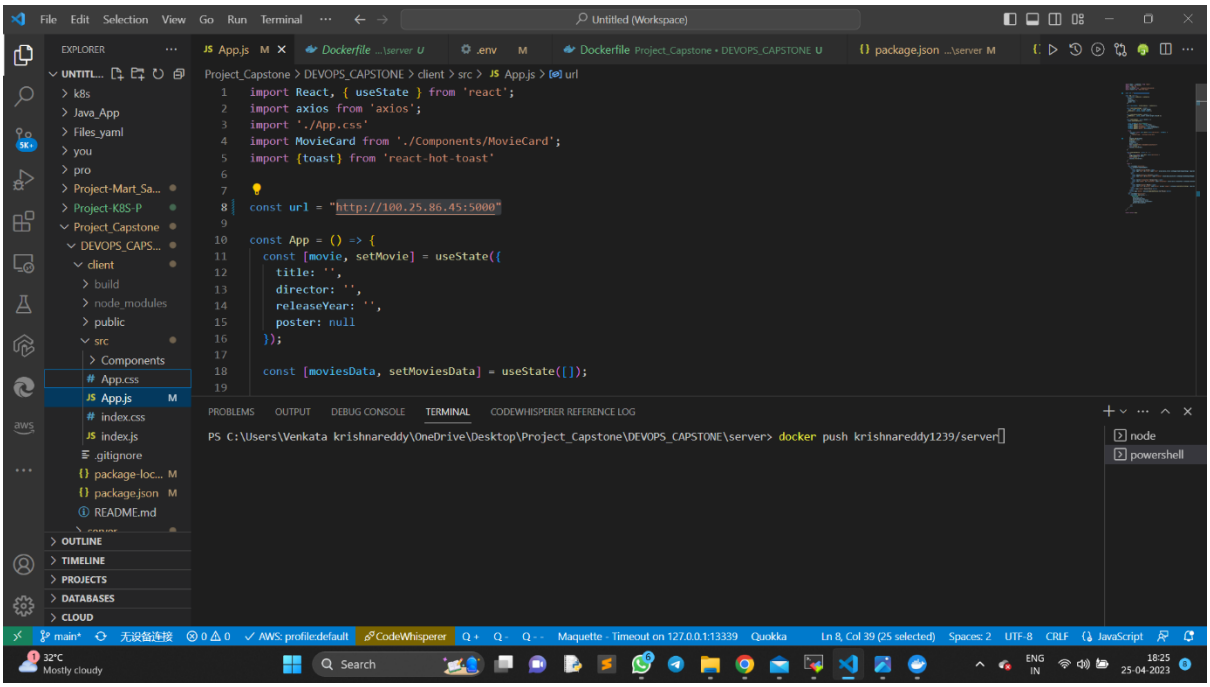
- Now run the docker image by listening the port number using the command **sudo docker run -p 5000:5000 image_name.** It shows that file is connecting to the MongoDB.





Step 5: Deploying client on EC2 using Docker

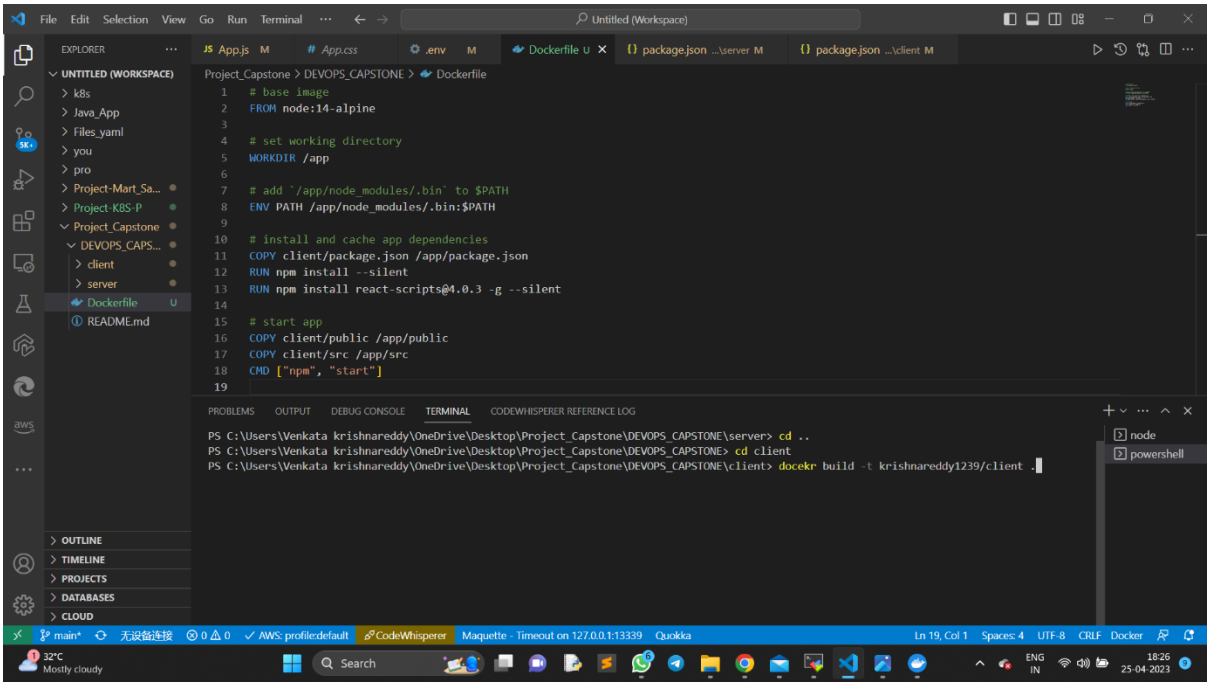
- Modified the port number for listening to the client



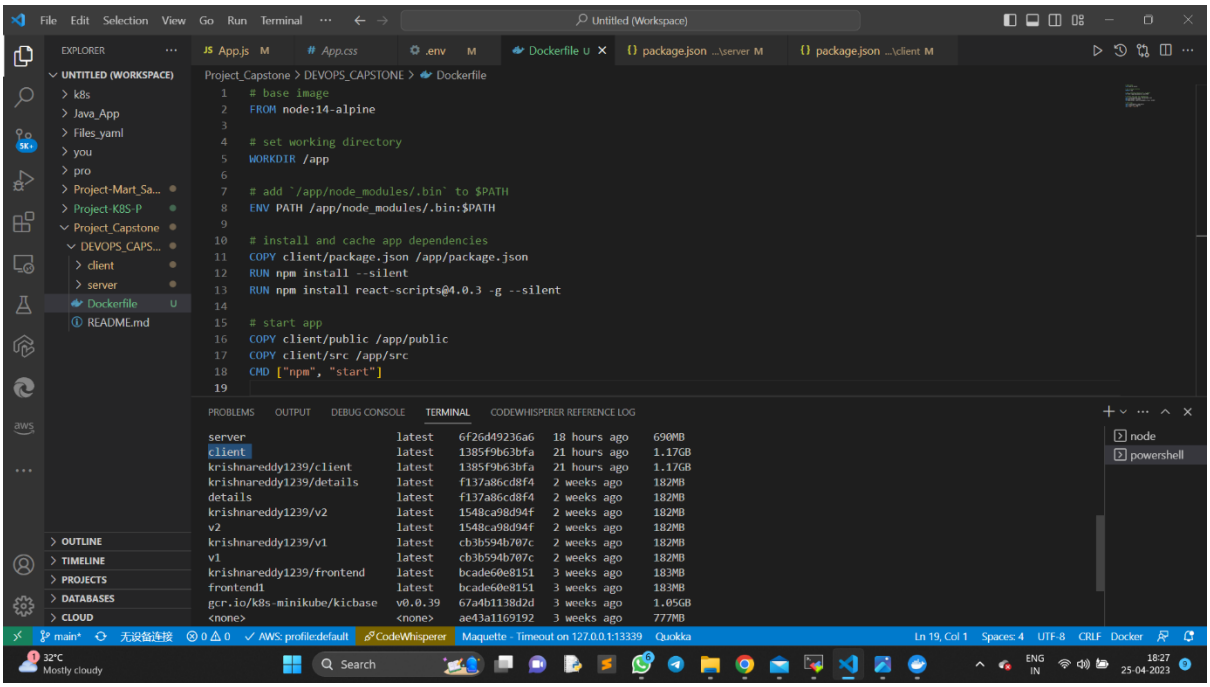
- Now add the Dockerfile which is used to call all the commands on the command line and build the docker image using the command **docker build -t image path/name .**

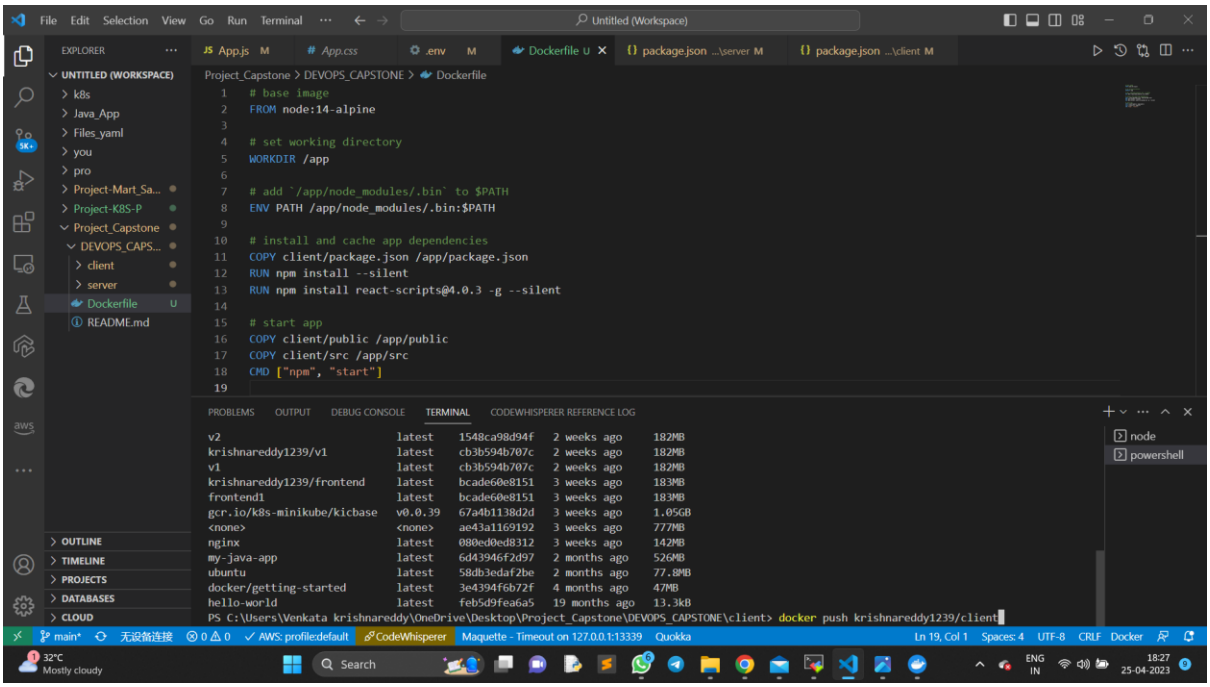
Dockerfile for Client:

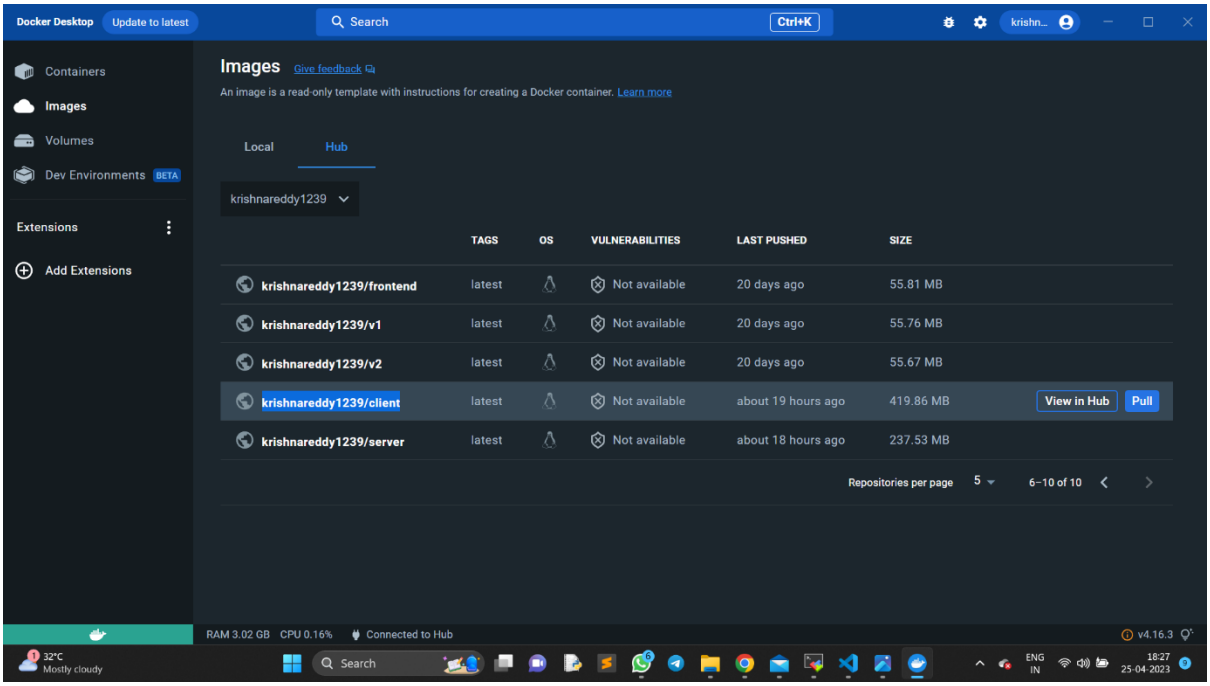
```
# base image
FROM node:14-alpine

# set working directory
WORKDIR /app

# add `/app/node_modules/.bin` to $PATH
ENV PATH /app/node_modules/.bin:$PATH

# install and cache app dependencies
COPY client/package.json /app/package.json
RUN npm install --silent
RUN npm install react-scripts@4.0.3 -g --silent

# start app
COPY client/public /app/public
COPY client/src /app/src
CMD ["npm", "start"]
```



- To check the status of running images in the docker using **docker ps** and check the image is created in the docker desktop.

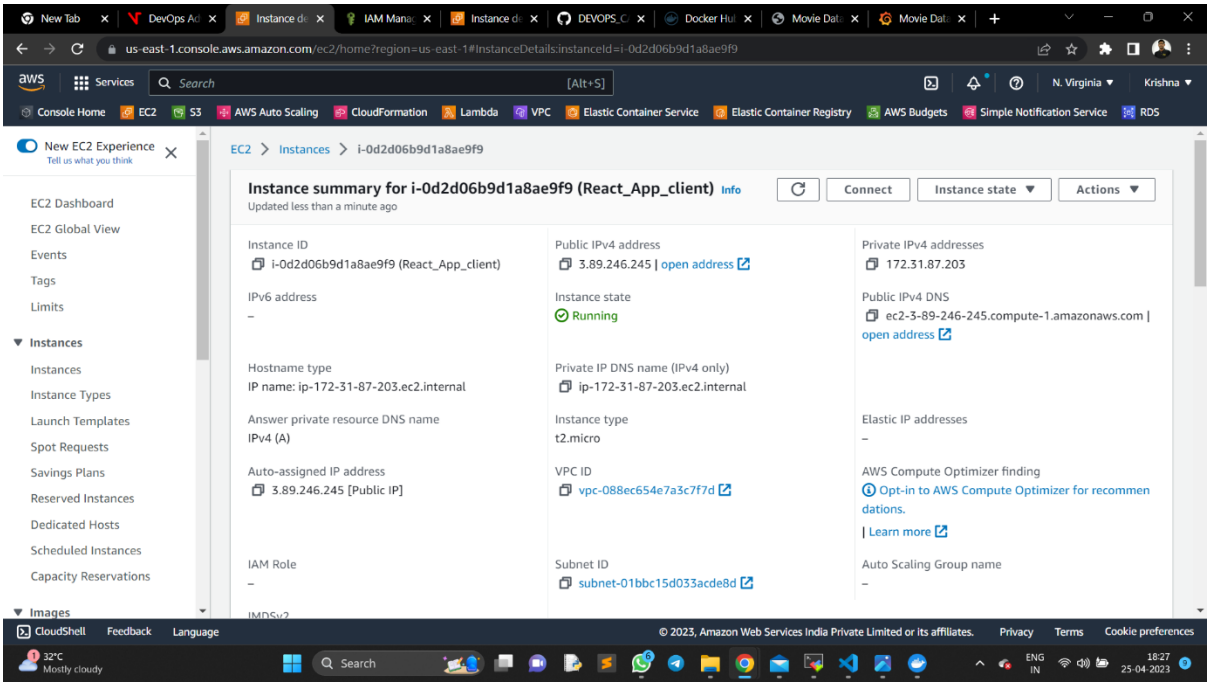



Project_Capstone > DEVOPS_CAPSTONE > Dockerfile
base image
FROM node:14-alpine
set working directory
WORKDIR /app
add `/app/node_modules/.bin` to $PATH
ENV PATH /app/node_modules/.bin:$PATH
install and cache app dependencies
COPY client/package.json /app/package.json
RUN npm install --silent
RUN npm install react-scripts@4.0.3 -g --silent
start app
COPY client/public /app/public
COPY client/src /app/src
CMD ["npm", "start"]
PS C:\Users\Venkata krishnareddy\OneDrive\Desktop\Project_Capstone\DEVOPS_CAPSTONE\client> docker push krishnareddy1239/client

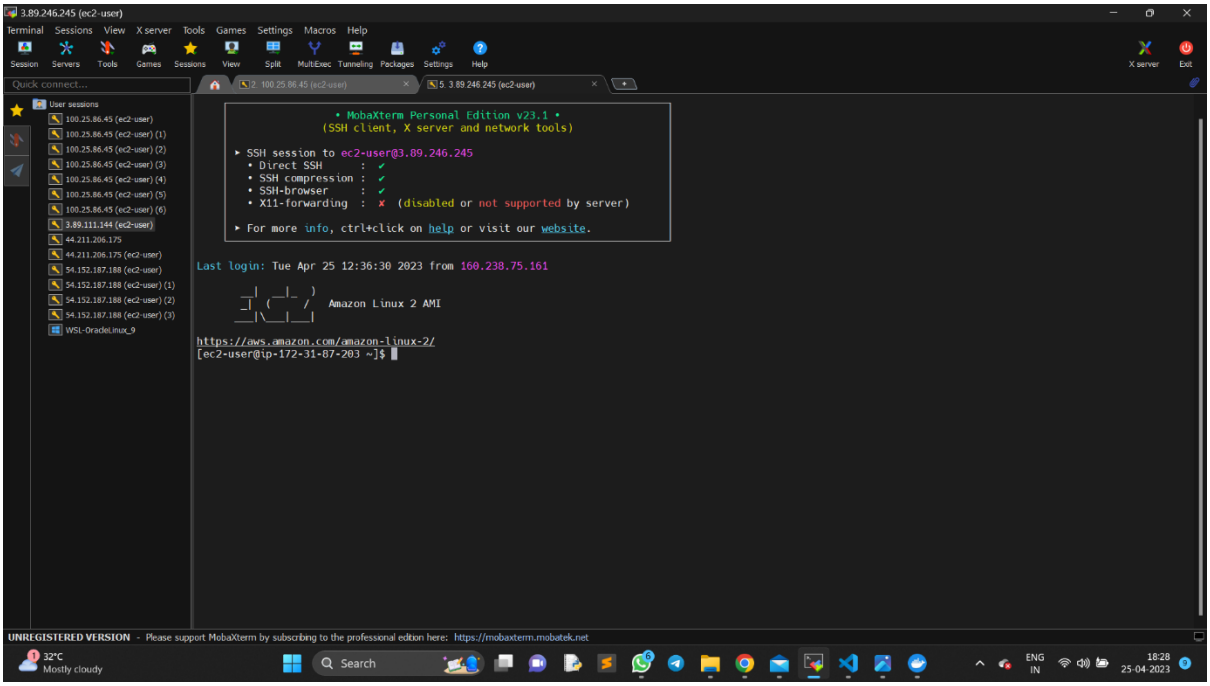

Docker Desktop
Update to latest
Search
Ctrl+K
Containers
Images
Volumes
Dev Environments
BETA
Extensions
Add Extensions
Images
Give feedback
An image is a read-only template with instructions for creating a Docker container. Learn more
Local
Hub
krishnareddy1239
TAGS
OS
VULNERABILITIES
LAST PUSHED
SIZE
krishnareddy1239/frontend latest Not available 20 days ago 55.81 MB
krishnareddy1239/v1 latest Not available 20 days ago 55.76 MB
krishnareddy1239/v2 latest Not available 20 days ago 55.67 MB
krishnareddy1239/client latest Not available about 19 hours ago 419.86 MB View in Hub Pull
krishnareddy1239/server latest Not available about 18 hours ago 237.53 MB
Repositories per page 5
6–10 of 10
RAM 3.02 GB CPU 0.16%
Connected to Hub
v4.16.3

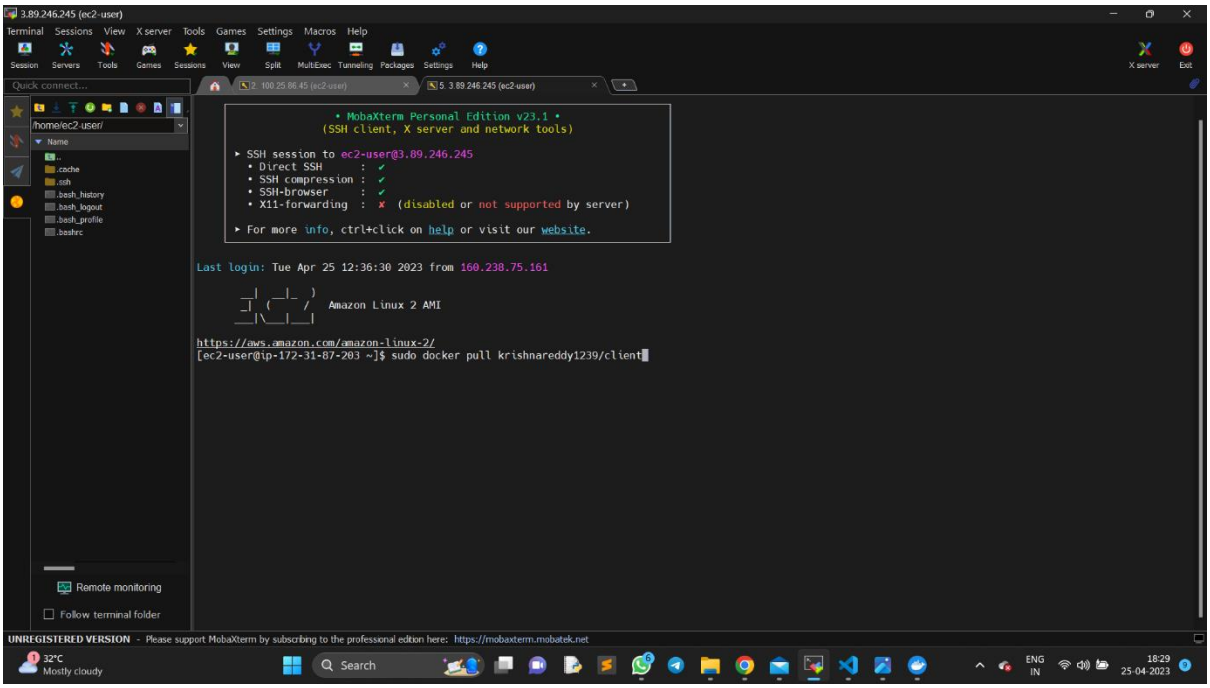
Step 6: Deploying EC2 instances for client

- Create another EC2 instance to deploy the frontend(client)

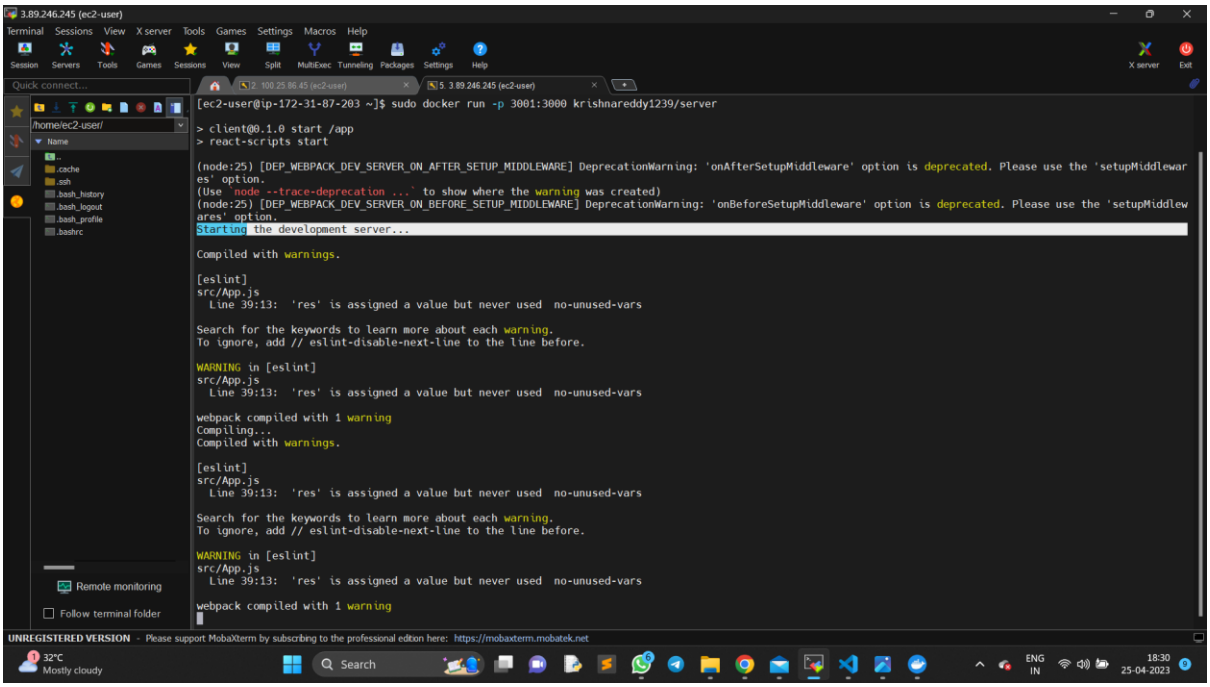


- Connect EC2 instance through local machine using MobaXterm.

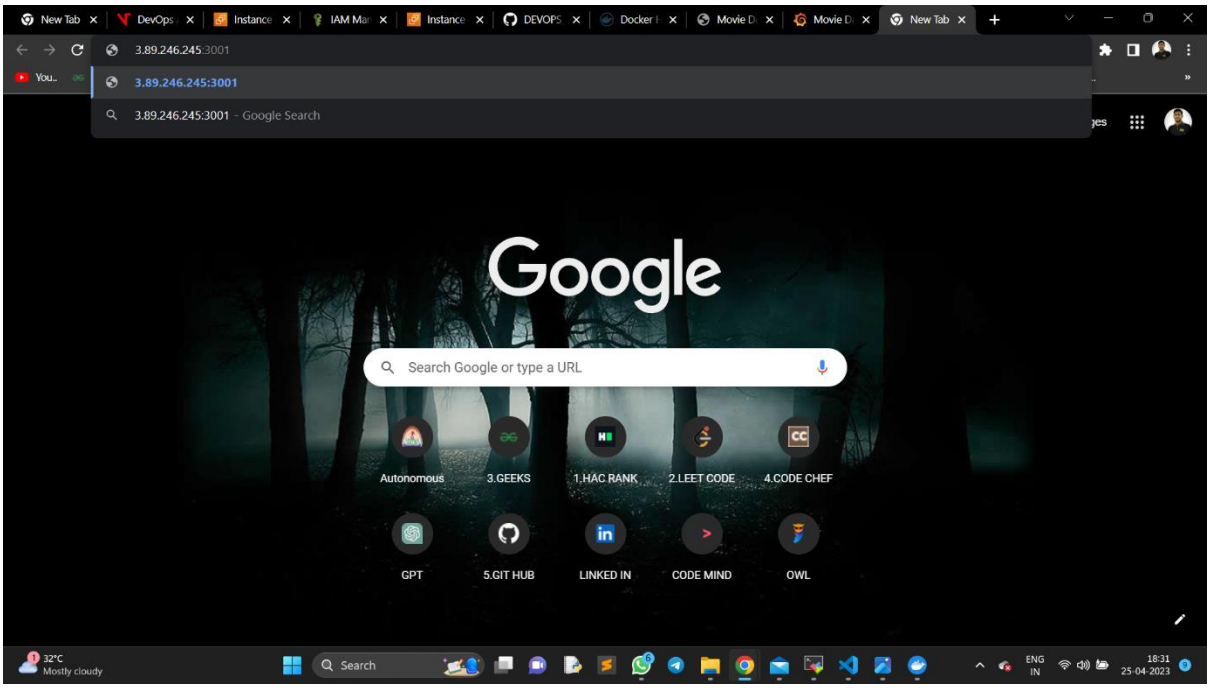




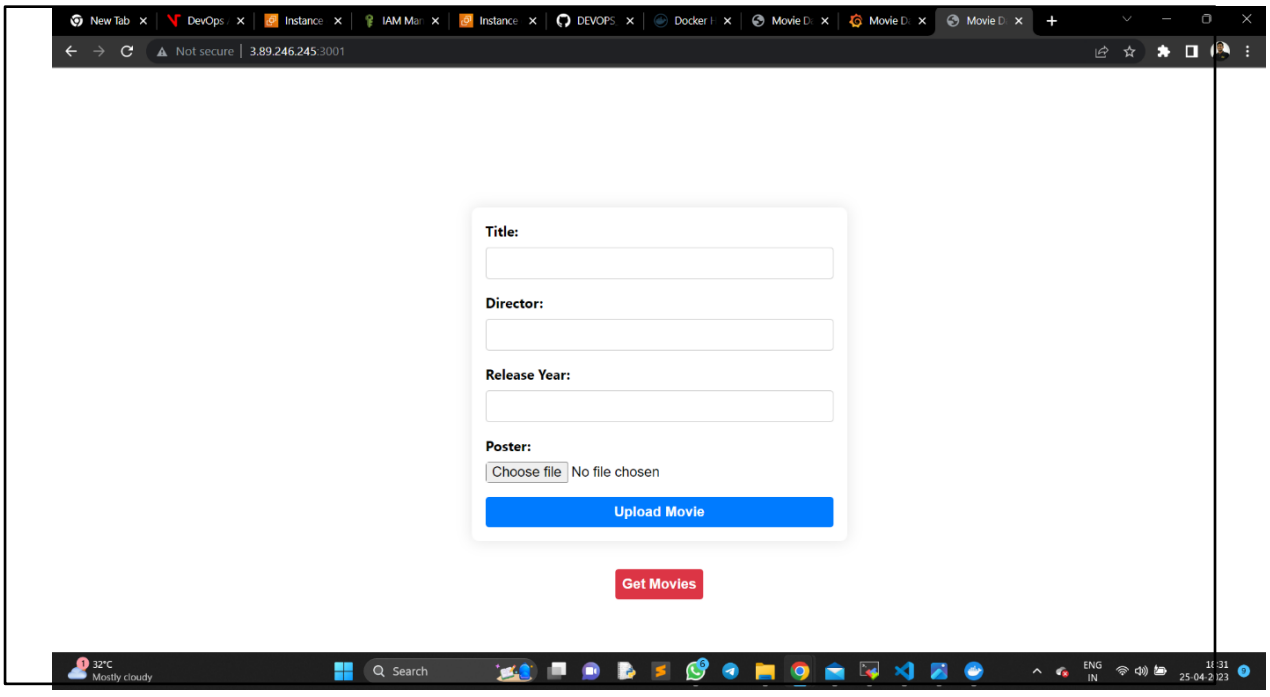
- Now run the docker image by listening the port number using the command **sudo docker run -p 5000:5000 image_name.** It shows starting the development server.

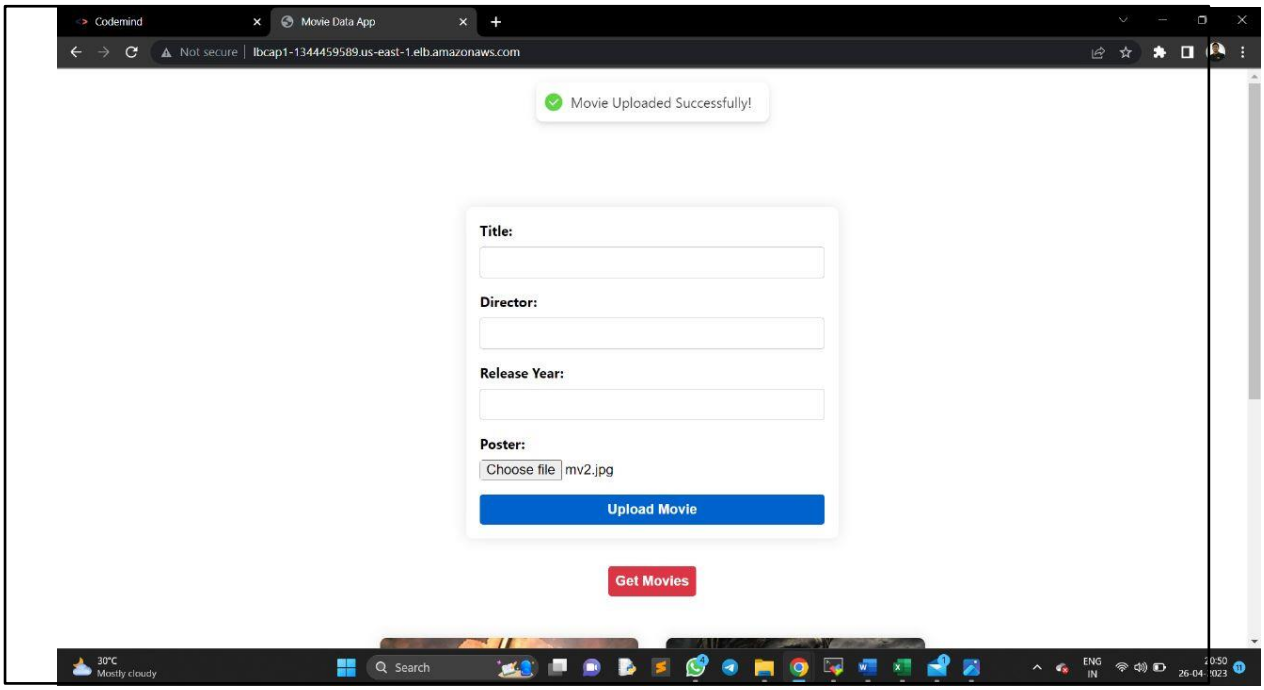


- After deploying to check the website we use the public IP address of the Client Instance along with the listening port number **3.89.246.245:3001**

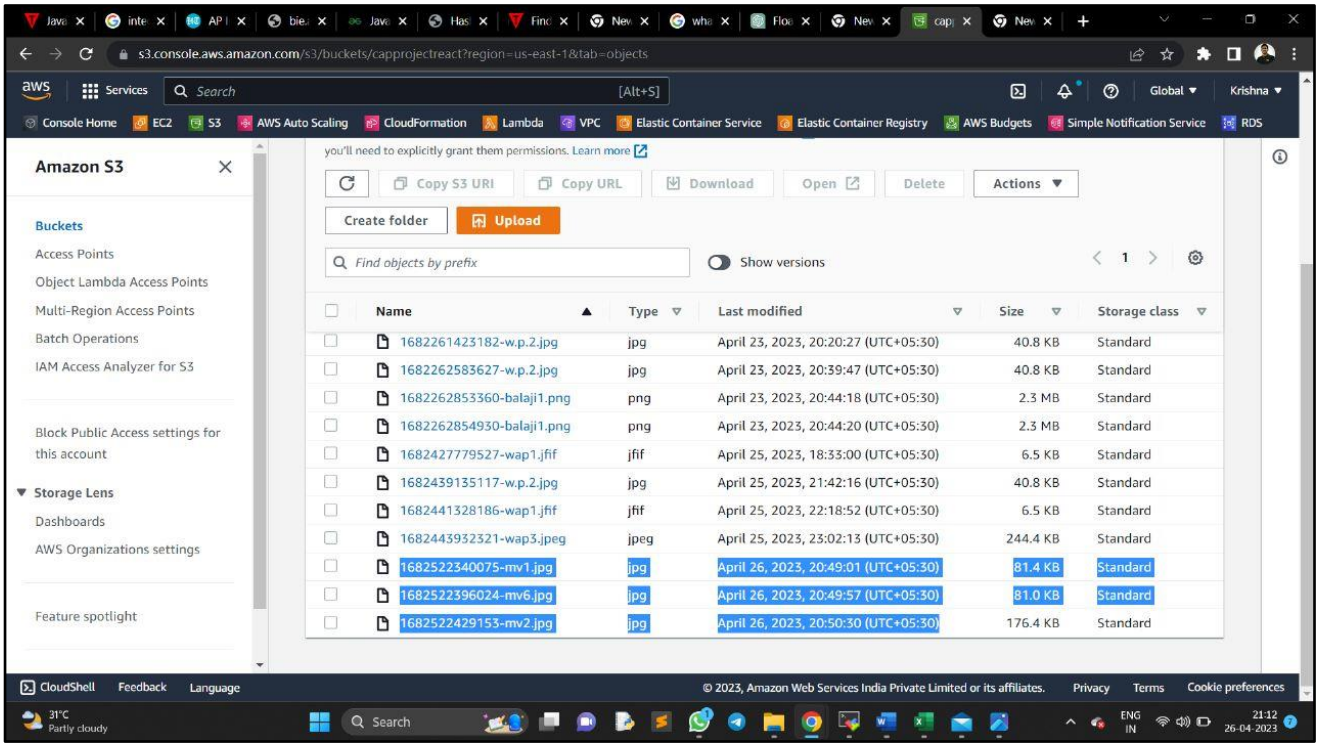


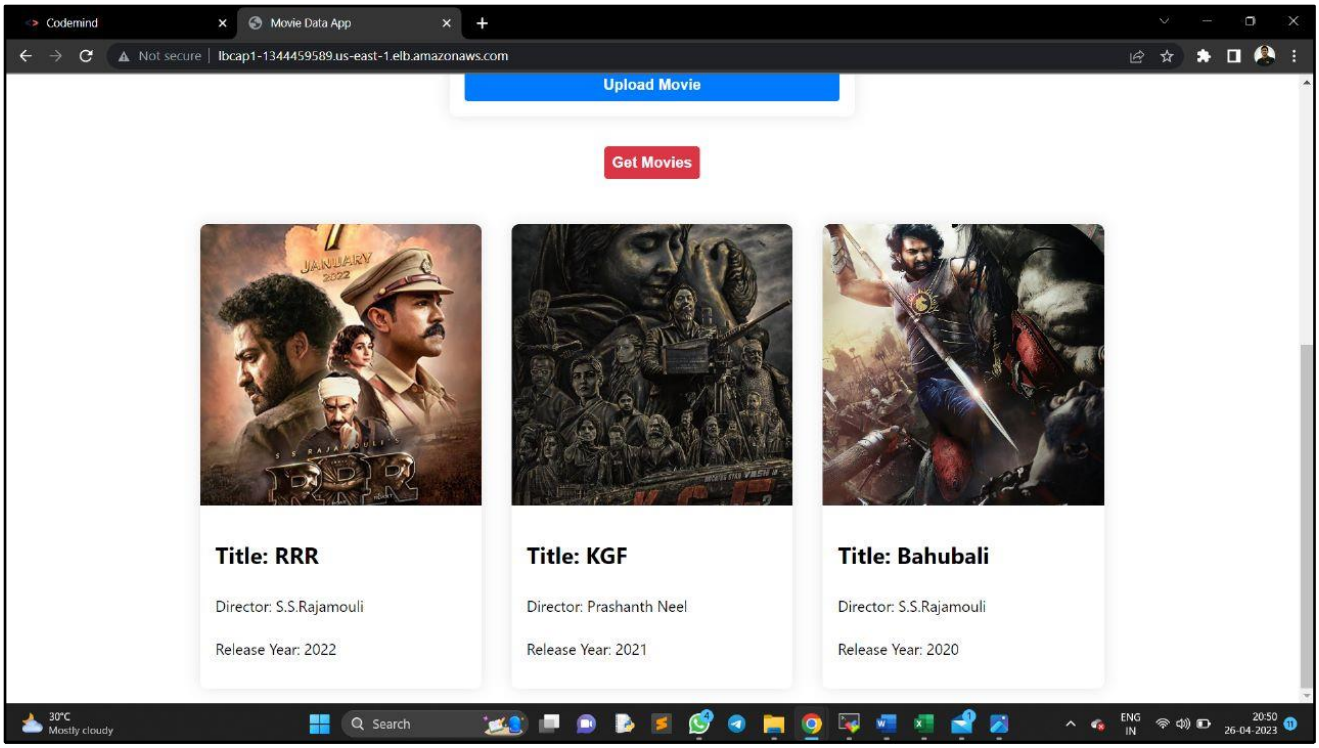
- The deployed web application. Enter the details in the displayed web page and submit the details. The details are uploaded into the mongoDB server and photos are uploaded to the attached S3 bucket.



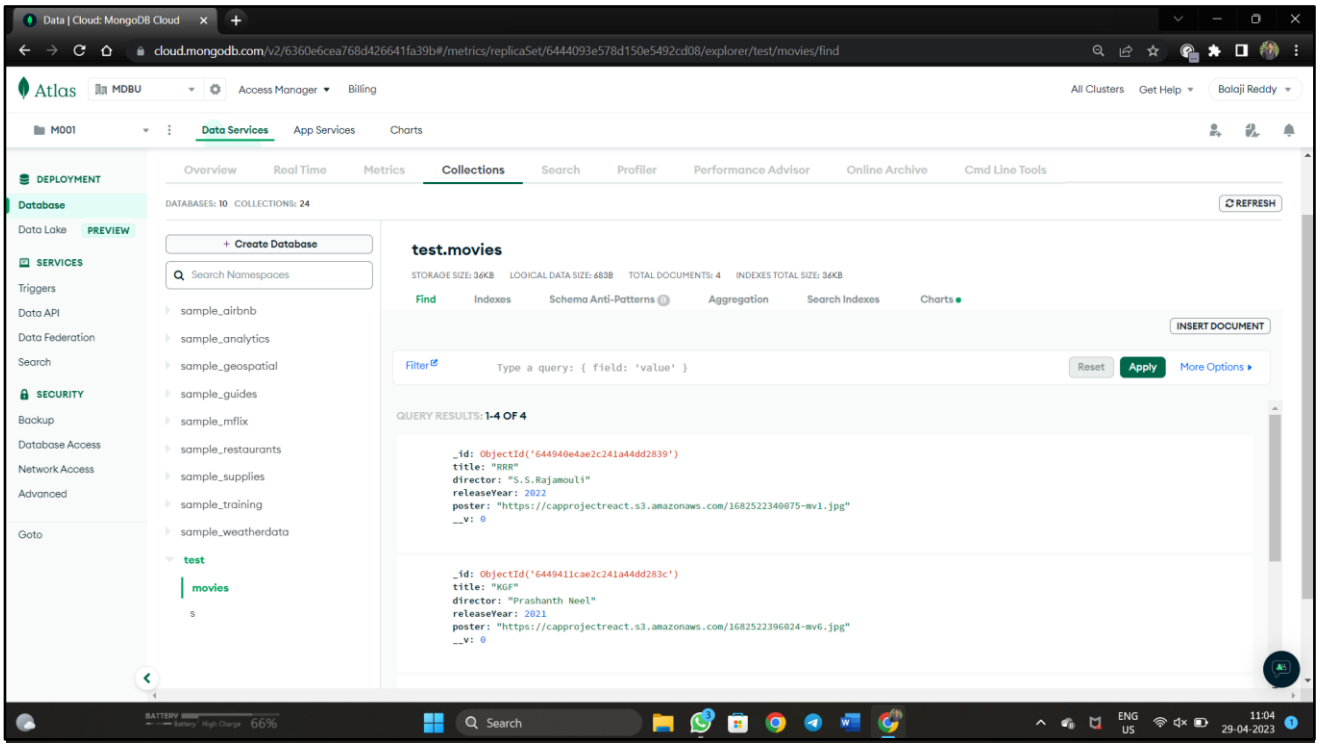


- Photos are uploaded in the S3 bucket.



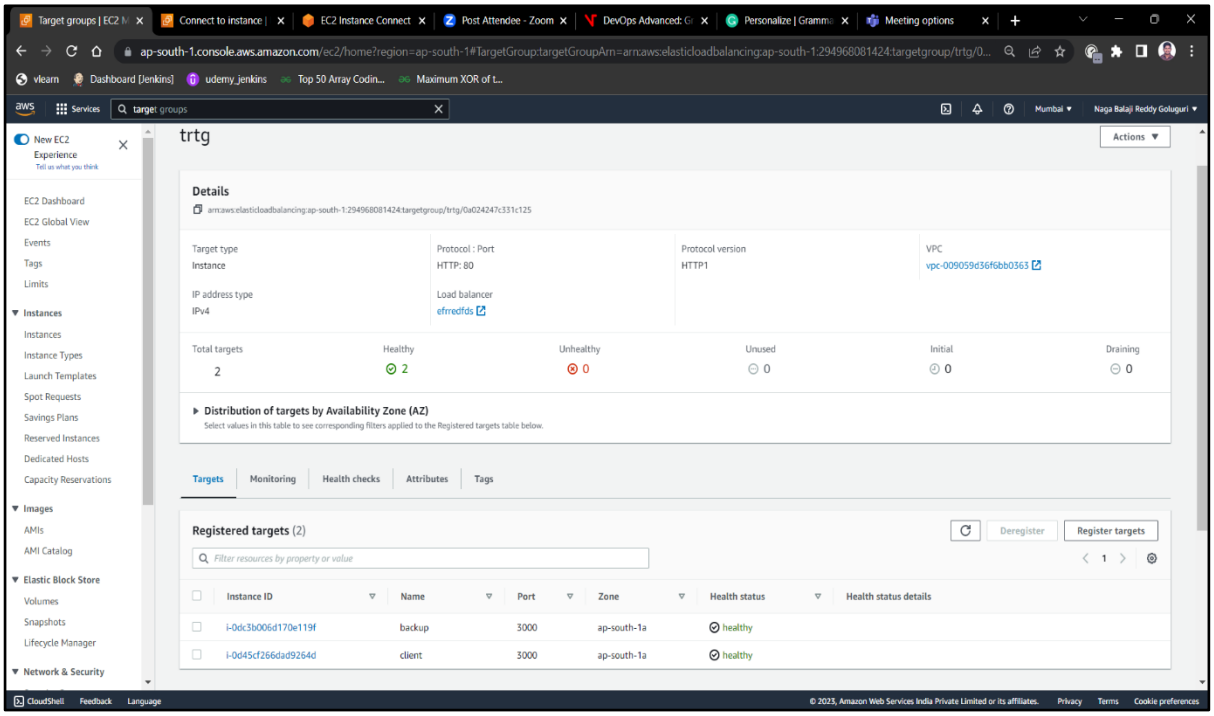


- Details are uploaded in the MongoDB Document.

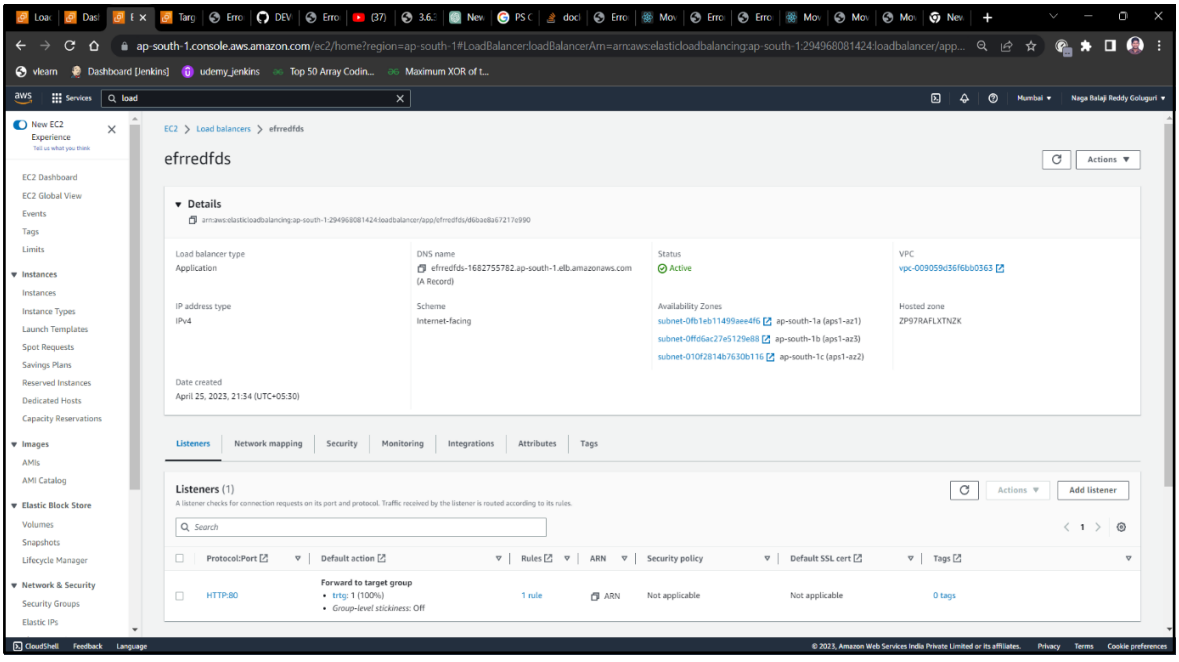


Step 7 : Creation a target group and Load Balancer

- Create a target group which tells a load balancer where to direct traffic to EC2 instances, fixed IP addresses; or AWS Lambda functions, amongst others.
- When creating a load balancer, you create one or more listeners and configure listener rules to direct the traffic to one target group



- 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.

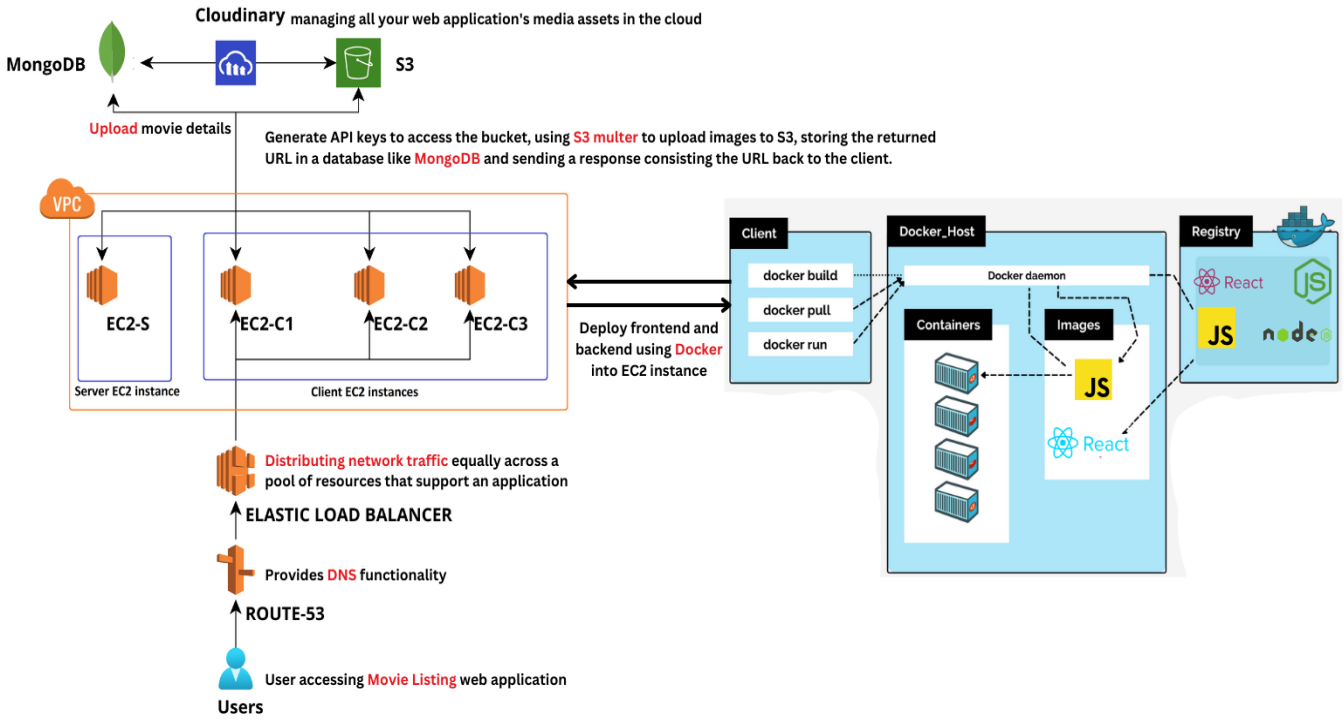


- While creating Load balancer we got one **DNS (Domain Name System)** name which is used to translate a domain name into the appropriate IP address.

DNS NAME: http://lbcap1-1344459589.us-east-1.elb.amazonaws.com/

Project Git-Hub repository Link: DevOps_Capstone

AWS DEPLOYMENT DIAGRAM:



This project includes the utilization of various tools. Multiple observations are noted based on the tools. Some key observations are:

Visual Studio Code: Visual Studio Code is a source-code editor that can be used with a variety of programming languages. The code has been according to the requirements using VS Code.

MongoDB Atlas: MongoDB Atlas is a fully-managed cloud database that handles all the complexity of deploying, managing, and healing your deployments on the cloud service provider of your choice (AWS, Azure, and GCP). The local database has been replaced by this database and the data uploaded is stored here.

EC2: EC2 (Elastic Compute Cloud) is a web service provided by Amazon Web Services (AWS) that allows users to rent virtual computers, known as instances, on which they can run their own applications. The containerized frontend and backend have been deployed using EC2.The client instances are also attached to the load balancer to manage the traffic.

S3: Amazon S3 (Simple Storage Service) is a cloud-based storage service that enables users to store and retrieve data from anywhere on the internet. The images we have uploaded in the frontend have been stored in S3 buckets which was created.

IAM: AWS Identity and Access Management (IAM) is a web service provided by Amazon Web Services (AWS) that enables users to securely manage access to AWS resources. A user have been created for the access controls and full administrator access was given.

Route 53: Amazon Route 53 is a highly available and scalable cloud-based Domain Name System (DNS) service provided by Amazon Web Services (AWS). It helps the developers and business owners by translating the domain name into IP address. DNS has been assigned which can be accessed by anyone to view the website.

GitHub: GitHub is a web-based platform that provides developers with a variety of tools for software development and collaboration. The project has been uploaded in GitHub which would be accessible for future changes.

A project requires clear objectives and ground rules to work on to achieve the goal of any aspect.

- **Jira** has been used to plan the sprints, track the changes and complete the project efficiently.
- **Slack** is a great communication tool where we can interact efficiently and to collaborate.
- **Teams** is a collaboration app that kept us organized, enabled conversations, and let us schedule meetings to get the project done efficiently and effectively**.**

TEAM MEMBERS CONTRIBUTION:

During the course of the project, we distributed the workload among team members based on their individual strengths and interests.

We **communicated regularly** through daily **Team meetings** and daily updates on our project management tool through WhatsApp group. This allowed us to stay on top of any issues that arose and to make adjustments to our plan as needed.

To track the contributions of each team member, we created an Excel spreadsheet with "**Task Contributed**" and "**Members**" as columns. In the "Task Contributed" column, we listed each task that needed to be completed for the project, along with a brief description of the task. In the "Members" column, we listed the name of the team member who was responsible for completing that task.

By using this spreadsheet, we were able to keep track of who was responsible for each task and ensure that **everyone was contributing equally to the project.** We updated the spreadsheet regularly to reflect any changes to the task assignments, and we used it as a reference during our team meetings to discuss progress and identify any issues.

S No.	TASK CONTRIBUTED	MEMBERS
1	Connecting the Server with Atlas MongoDB.	Yamuna, Balaji, Karthik
2	Creating IAM user and Configuring S3-bucket.	Blessy, Iswarya, Yamuna
3	Configuring the Application Code with S3-multer.	Vaseem, Krishna, Sunil
4	Containerization of the code using Dockerfile.	Krishna, Sunil, Iswarya
5	Deploying server on EC2 using Docker.	Karthik, Balaji, Krishna
6	Deploying client on EC2 using Docker.	Balaji, Vaseem, karthik
7	Creation a target group and Load Balancer.	Balaji, Blessy, Sunil
8	Creating AWS Deployment Diagram.	Sunil, Krishna, Vaseem
9	Preparing Project Documentation.	Blessy , Yamuna, Iswarya

Overall, we believe that our team was successful in distributing the workload effectively and working together to achieve our goals. By outlining our roles and responsibilities, maintaining open communication, and evaluating our performance, we were able to complete the project on time and to a high standard.

CHALLENGES FACED

- Due to lack of node of react knowledge, it took lot of time to analyse the code.
- We face most of the issues while modifying the server code for configuring with S3-multer.
- Due to lack of good internet, Docker image creation took a lot of time.
- When something went wrong, we had to reframe the frontend (connection) and dockarize the frontend code again.
- Due to our lack of knowledge on how to create an AWS architecture diagram, we had to learn how to work with different editing tools like Canva and figma...etc.

RESOURCES

MongoDB : https://www.mongodb.com/docs/

AWS : https://docs.aws.amazon.com/

S3-multer : https://www.npmjs.com/package/multer-s3

Docker : https://docs.docker.com/

Git : https://git-scm.com/docs

AWS Architecture Diagram Developing tools: https://aws.amazon.com/architecture/icons/

CONCLUSION

At the end of this project, we successfully deployed frontend using docker into EC2 instance. The given capstone project involves using ReactJS as Frontend, NodeJS as Backend, MongoDB as Database, multer in Backend for file upload and cloud infrastructure (AWS) to deploy. The deployed frontend application gives us the required “Movie Listing” web site in which users can upload movie details.

For achieving this, we used multer-s3 library for storing images, replaced local database with Atlas MongoDB cloud infrastructure to take the database into the cloud, deployed backend and frontend in EC2 instances using Docker and attached Elastic IP to this instance and modified the Frontend Code to be able to fetch data from Backend.

The frontend user interface of the application having 4 fields like **Title, Director, Release year** and **Poster**. And having 2 functions **upload Movie** and **Get Movies**. When the user enters the details of the movie the data will be stored in mongoDB database. The poster field having images these images are stored in S3 bucket using S3-multer and by using cloudinary which provides a URL of that image which is stored in mongoDB database. When the user clicks on Get Movies button the user will get details of all the movies which are stored in mongoDB database and images are fetched from S3 bucket.