DV1566

INTRODUCTION TO CLOUD COMPUTING

Project Report

Web based Chat application using NodeJs and socket.io

Group Members:

Boya Sada Siva Kumar

Gundala Harshitha

Konka Cherishma

Title

Web based Chat application using NodeJs and socket.io

Goal

The goal is to develop a web-based chat application using HTML, CSS, JavaScript and to also c.io to implement the socket web connections.

Pre requisites

- IDE – VS Code
- Node: latest version
- Node CLI
- Dependencies: Express, nodemon
- Socket.io

Implementation Steps:

- Firstly, the environment setup has been prepared by installing the IDE and node in any favorite operating system you prefer
- Next a project folder had been made
- Initializing the folder as an NPM folder using the “npm init -y” command It creates a json file named package.json which contains the metadata of the project.

```
{
  "name": "wassup",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Fig : Package.json that’s been created

- A folder was created in which the client.js and style.js has been placed

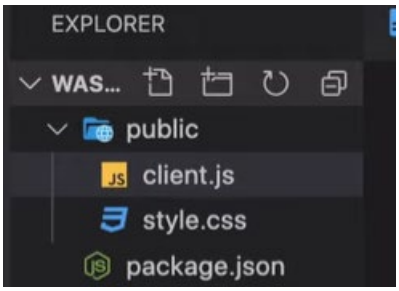


Fig: file structure

- A HTML file has been created in the root project folder.

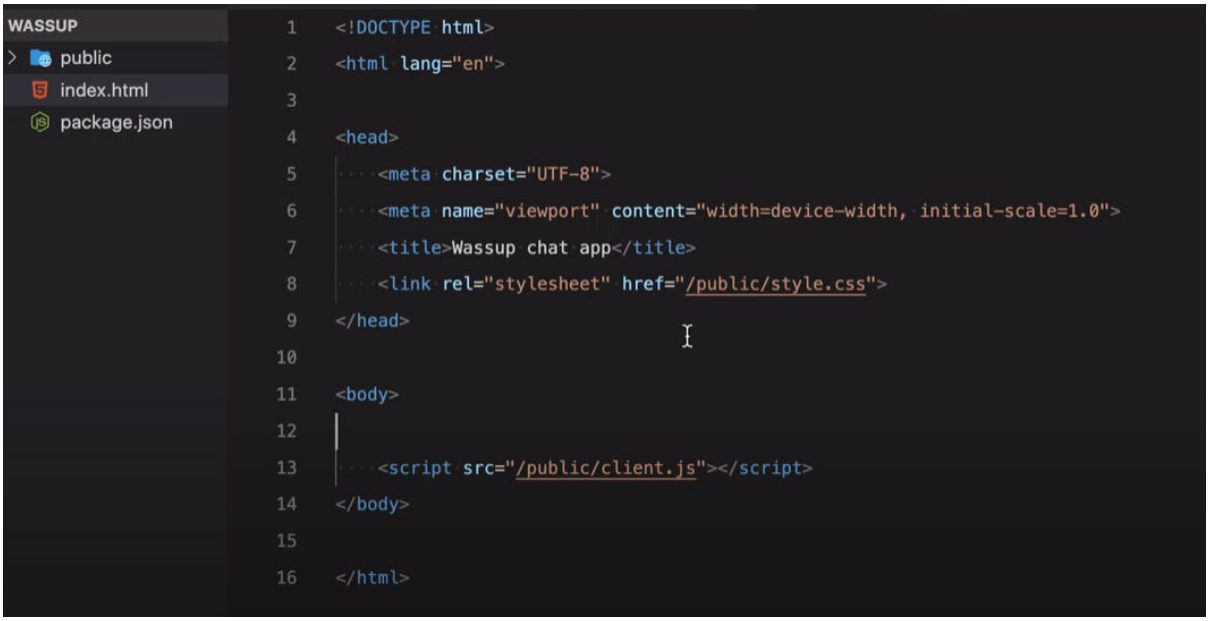


Fig: HTML code with js and css files imported

- Next, the dependencencies were installed
  - npm install express
  - npm install nodemon -D
- By installing nodemon, we would not be required to restart the server every time while making the changes in the code
- Create server.js which is an express server

```
JS server.js > io
1   const express = require('express')
2   const app = express()
3   const http = require('http').createServer(app)
4
5   const PORT = process.env.PORT || 80
6
7   http.listen(PORT, () => {
8       console.log(`Listening on port ${PORT}`)
9   })
10
11  app.use(express.static(__dirname + '/public'))
12
13  app.get('/', (req, res) => {
14      res.sendFile(__dirname + '/index.html')
15  })
16
17  // Socket
18  const io = require('socket.io')(http)
```

Fig : server.js

- The server runs on the port 80
  Port 80 is a default port for localhost , it will help us to run it easily while deployment phase
- The designing has been done using CSS as required
- The installation of socket.io is now required
- Import socket.io in both server.js and client.js and configure it as required
- Completed the code as required

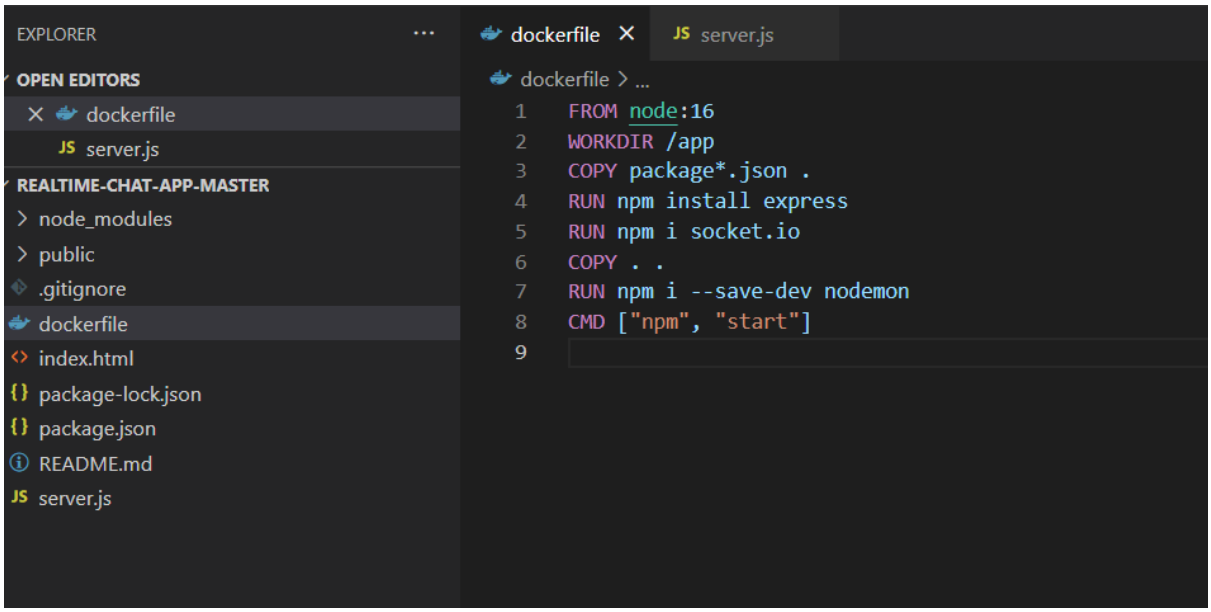
Deployment phase:

Requirements For Deployment

- Docker
- ASW CLI

Deployment Steps:

- Installed the requirements as required
- Start a new terminal/command prompt in your operating system and navigate to the project root folder
- Create a new Docker file



  - We will be using the node image version 16
  - The Working directory is set as /app
  - Firstly, we will be copying the package.json
  - After that, we need to install the required dependencies
  - After installing the dependencies, we need to copy all the remaining files
  - In the final step, mention the CMD command. This command first command that gets executed when we run the container. Usually, we write the command which starts our application.

- Now, it’s the time to build the docker file
  Command “docker build -t app .”
  It builds the dockerfile in layer wise

```
[+] Building 8.9s (12/12) FINISHED
 => [internal] load build definition from Dockerfile                                                                    0.0s
 => => transferring dockerfile: 197B                                                                                    0.0s
 => [internal] load .dockerignore                                                                                       0.4s
 => => transferring context: 2B                                                                                         0.0s
 => [internal] load metadata for docker.io/library/node:16                                                              3.0s
 => [1/7] FROM docker.io/library/node:16@sha256:fd86131ddf8e0faa8ba7a3e49b6bf571745946e663e4065f3bff0a07204c1dde          0.0s
 => [internal] load build context                                                                                       0.2s
 => => transferring context: 106.52kB                                                                                   0.2s
 => CACHED [2/7] WORKDIR /app                                                                                           0.0s
 => CACHED [3/7] COPY package*.json .                                                                                   0.0s
 => CACHED [4/7] RUN npm install express                                                                                0.0s
 => CACHED [5/7] RUN npm i socket.io                                                                                    0.0s
 => [6/7] COPY . .                                                                                                      1.1s
 => [7/7] RUN npm i --save-dev nodemon                                                                                  2.8s
 => exporting to image                                                                                                  1.0s
 => => exporting layers                                                                                                 0.9s
 => => writing image sha256:26041f7f70bff9634c6aae9c0b0f0e6ab3c0ee1d92610ae78492480af12918ad                            0.0s
```

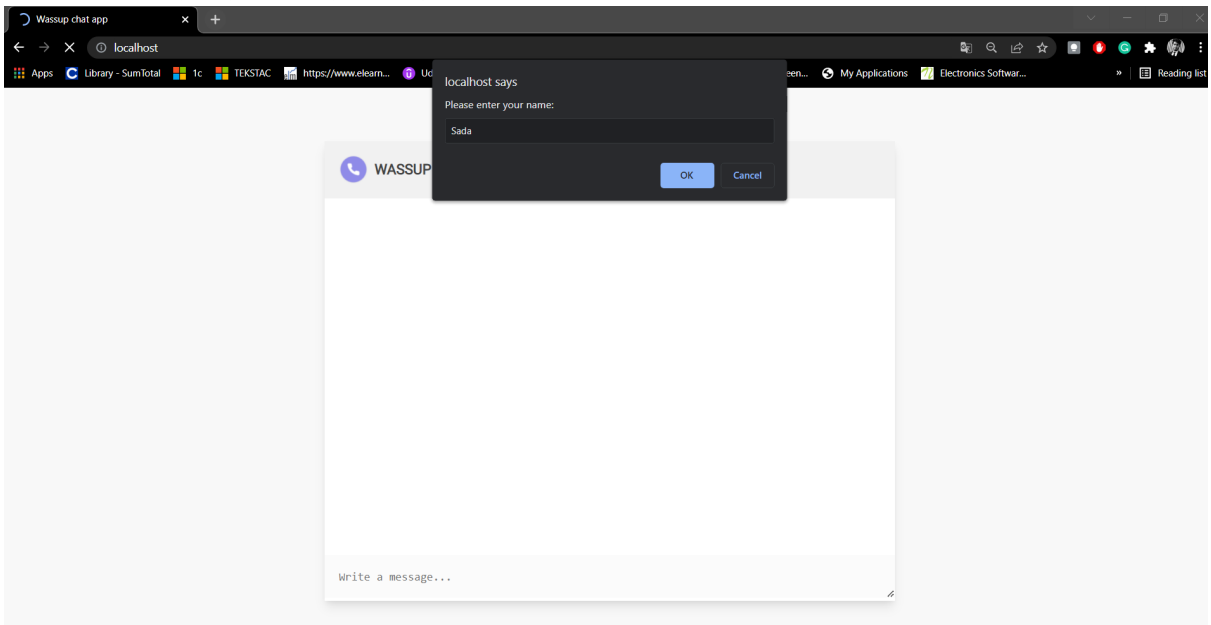
  Here, the build was successful

- Now try to run this container and using port mapping
  “docker run --publish 80:80 app”

```
> wassup@1.0.0 start
> node server.js

Listening on port 80
Connected...
```

  Hence, the project is getting served at “localhost” in our local machine

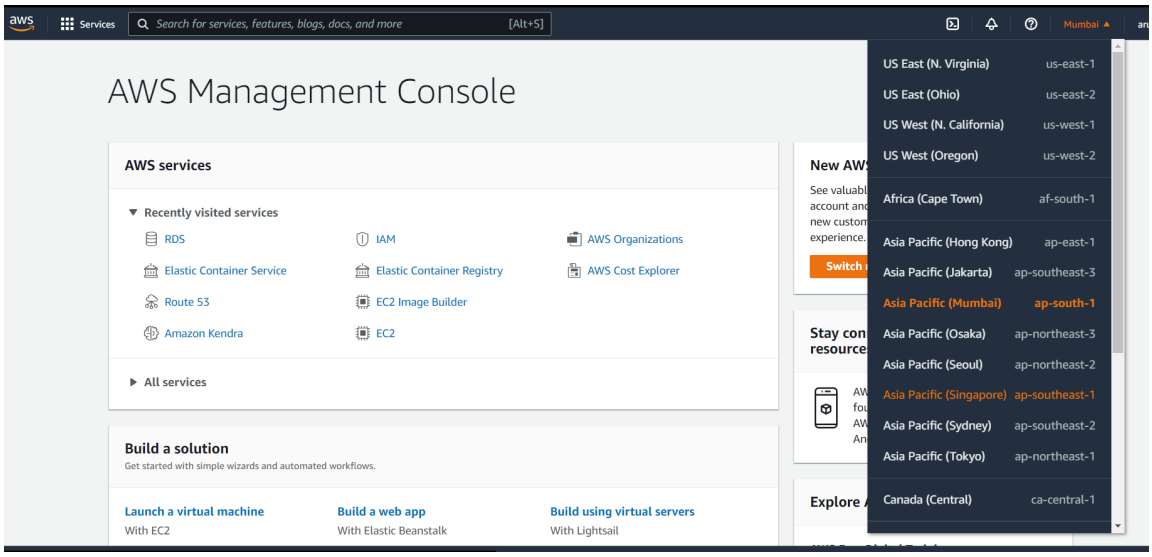


  So the project is working perfectly in a docker container

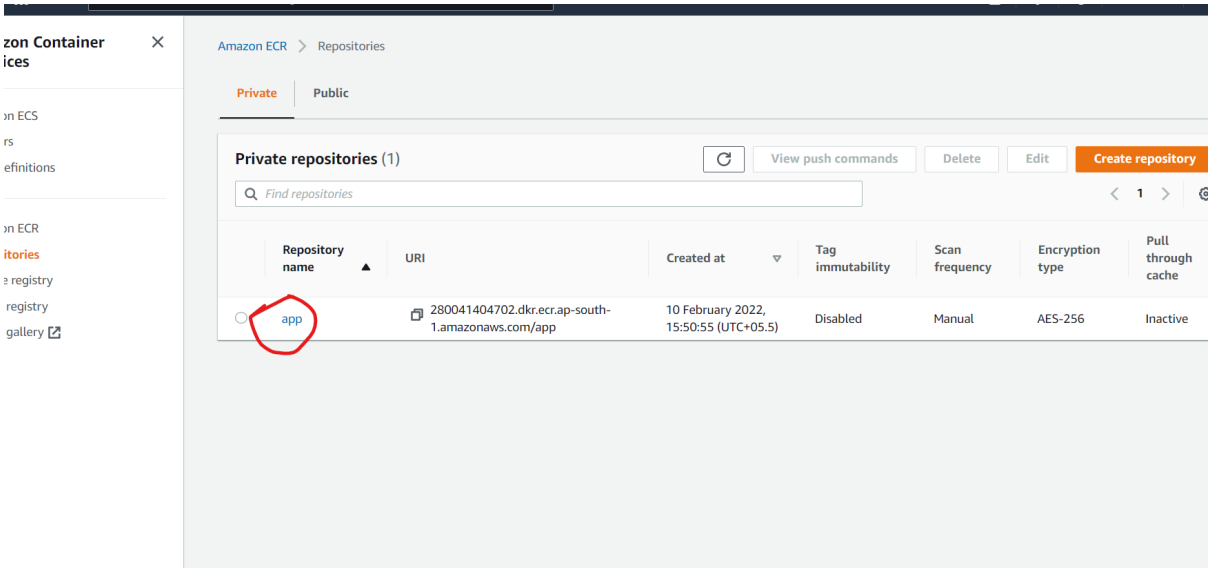
- Now, its time to move to AWS
- I have chosen AWS ECS to deploy the application using Fargate configuration.

ECS provides a cluster .
AWS Fargate is a serverless, pay-as-you-go compute engine that lets you focus on building applications ***without managing servers***.
So, by applying load balancer, auto scaling through Fargate, the application will be scaled up as per the requirement and traffic.
Fargate is a service which is managed by AWS itself. If the any node in the cluster fails, the AWS is responsible for the correction. So, it is automated.

- Create an AWS account
- In realtime live projects, creating an IAM role will be a good practice such that, root account will be secured
- I have made this deployment in root account itself



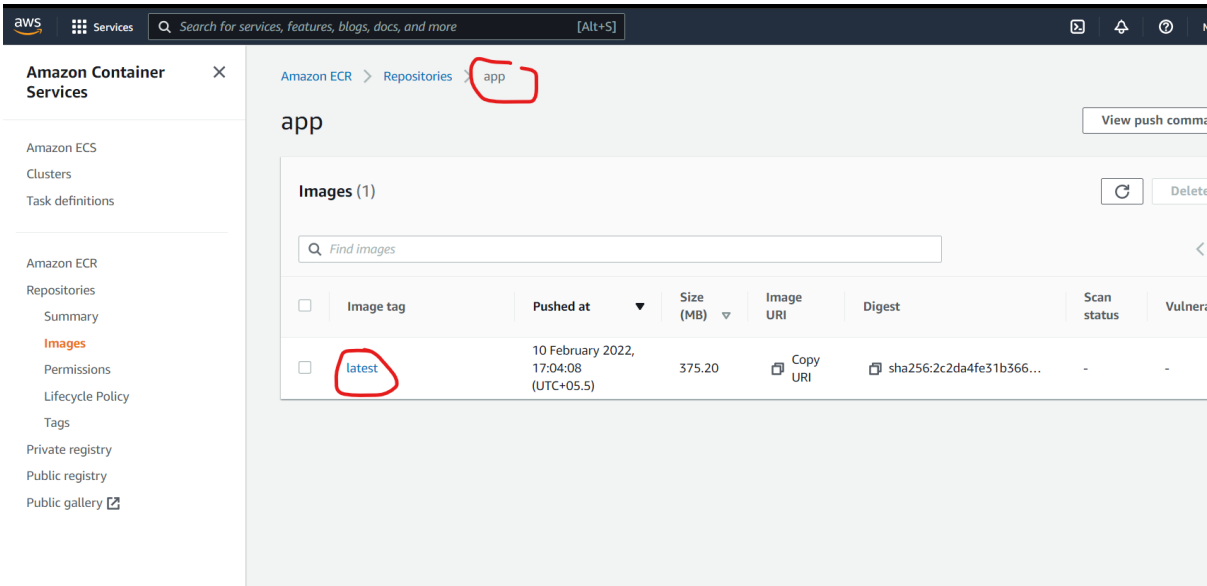
Firstly, select the region



- Goto ECS and open repositories
- I have created a repository named “app”
- We now need to push out docker image to AWS repository that was been created
- Following commads have been used

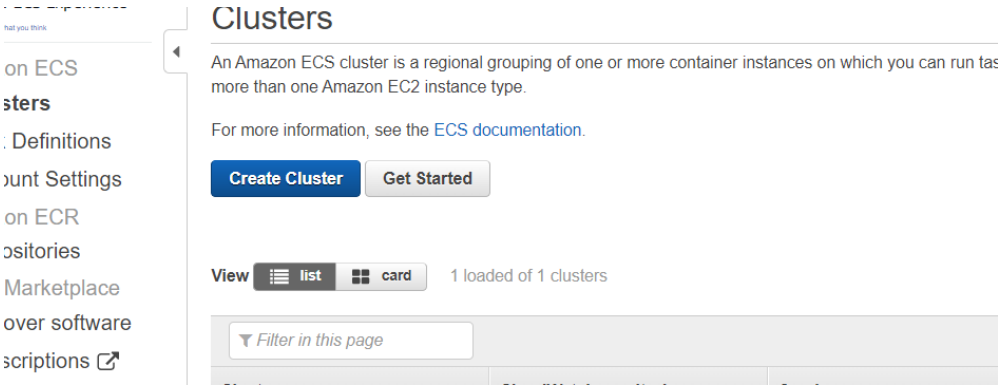
  “docker tag app:latest 280041404702.dkr.ecr.ap-south-1.amazonaws.com/app:latest”

  “docker push 280041404702.dkr.ecr.ap-south-1.amazonaws.com/app:latest”

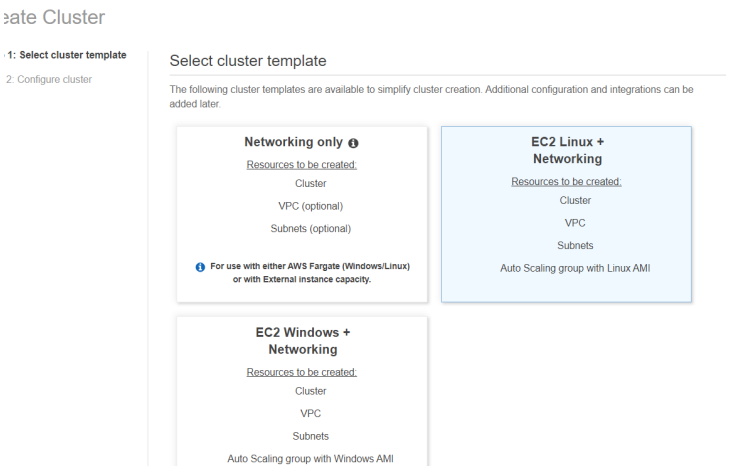


This is the image that’s been pushed into the “app” repository that was been created previously

Now, we need to create an ECS cluster

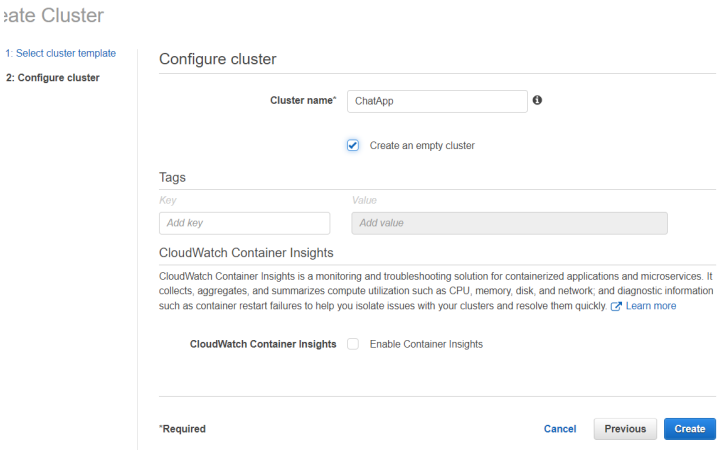


Click on create cluster button



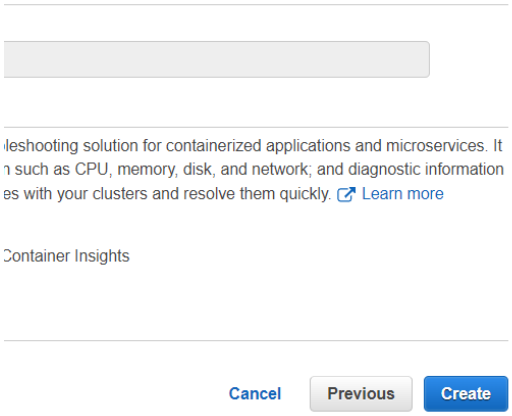
Now, we need to select the cluster template. Here, I have used EC2 Linux+ Networking

---

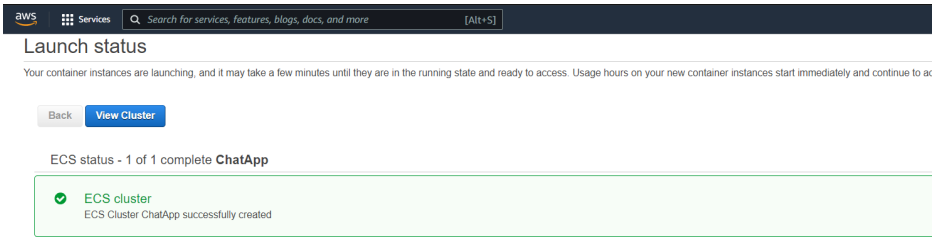


Give a name to the cluster

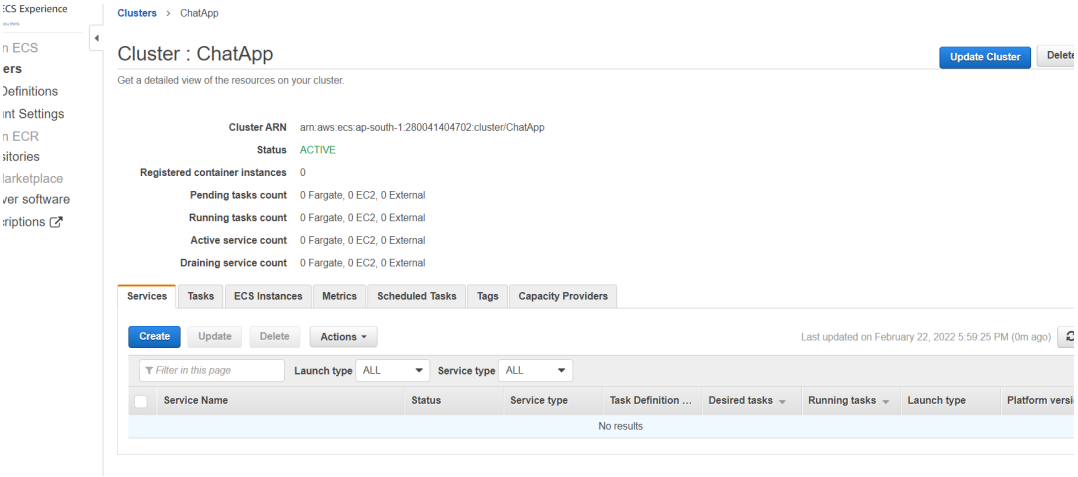
And tick the “create an empty cluster” check box



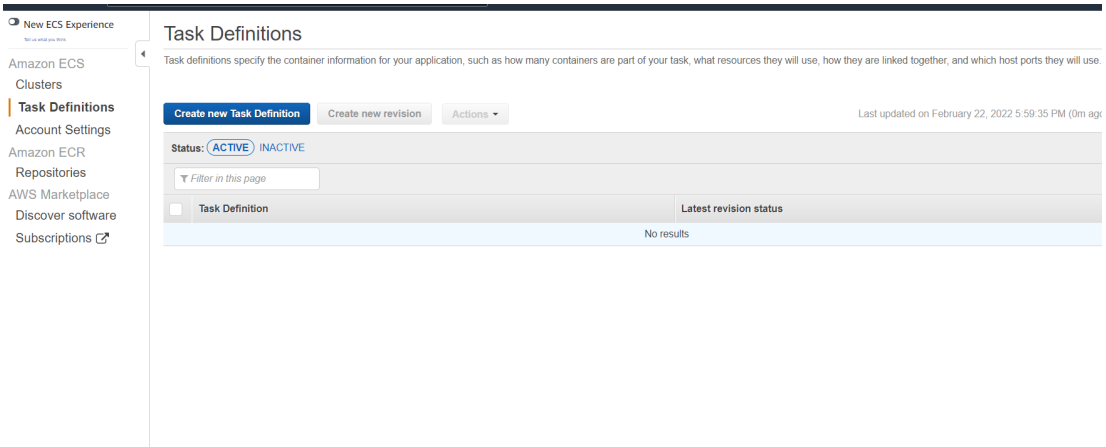
Click on create button



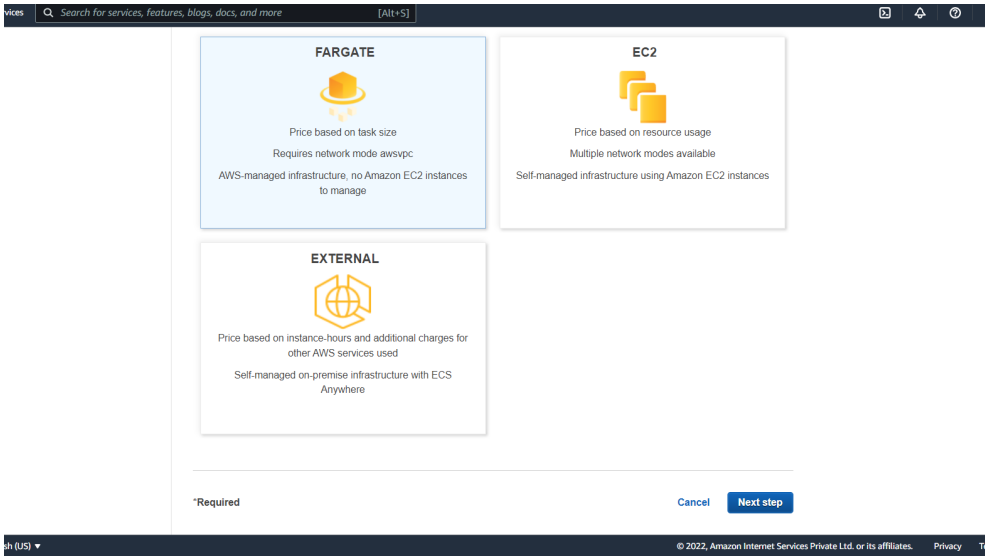
This appears after the cluster gets created successfully



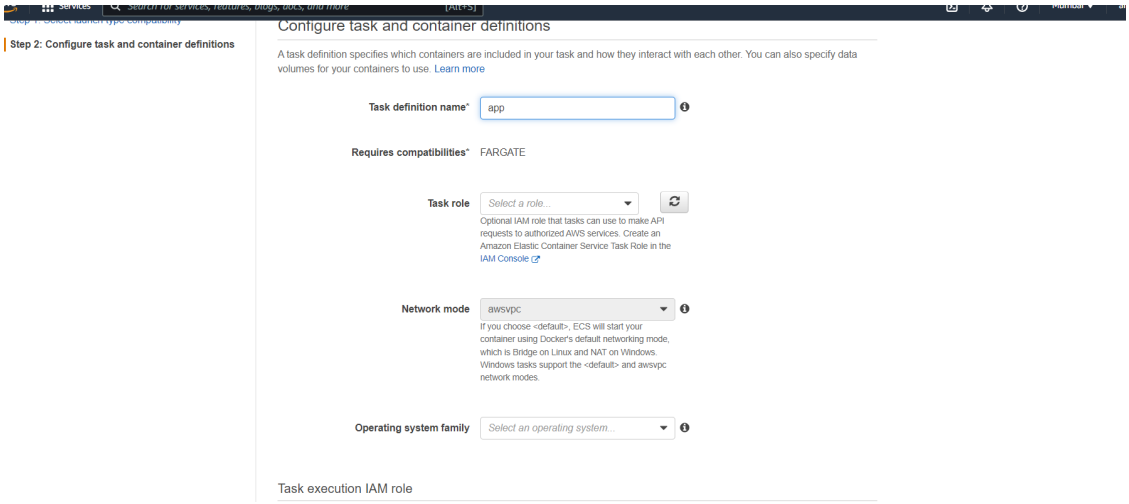
Check your cluster is active or not



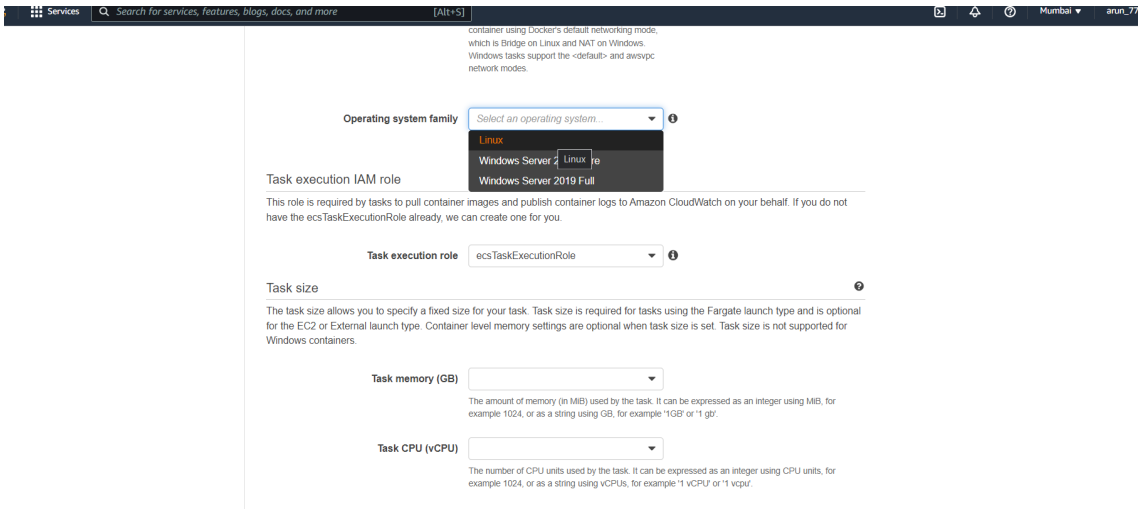
Now we need to create a task definition



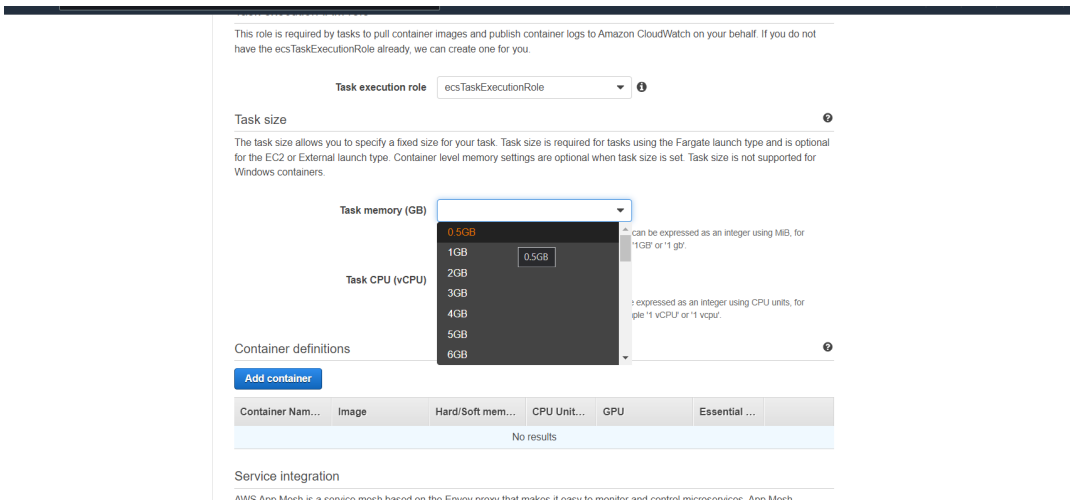
I have used Fargate and go to the next step



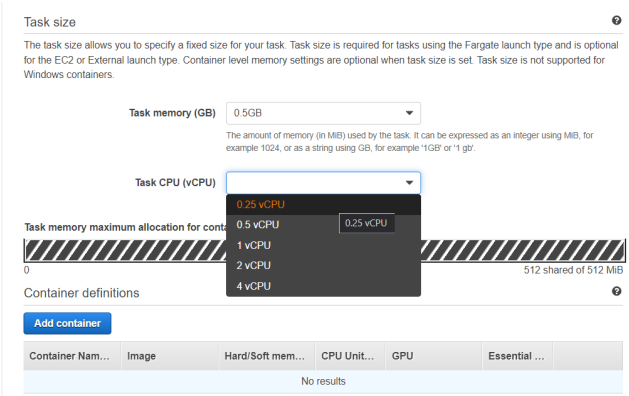
Give a name to your task definition



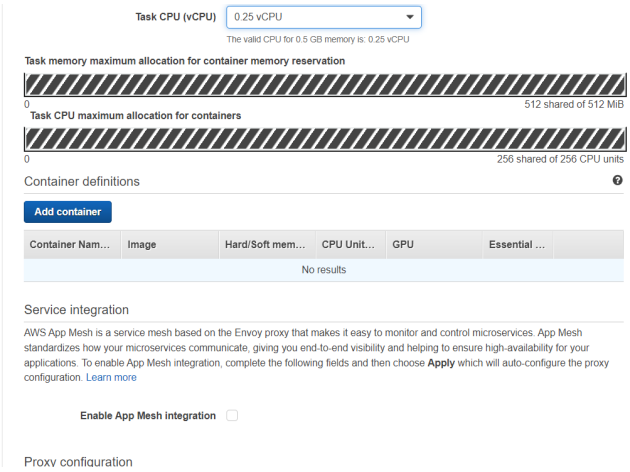
OS family linux, we can use windows as well



Task memory, I have opted for the lowest which is sufficient

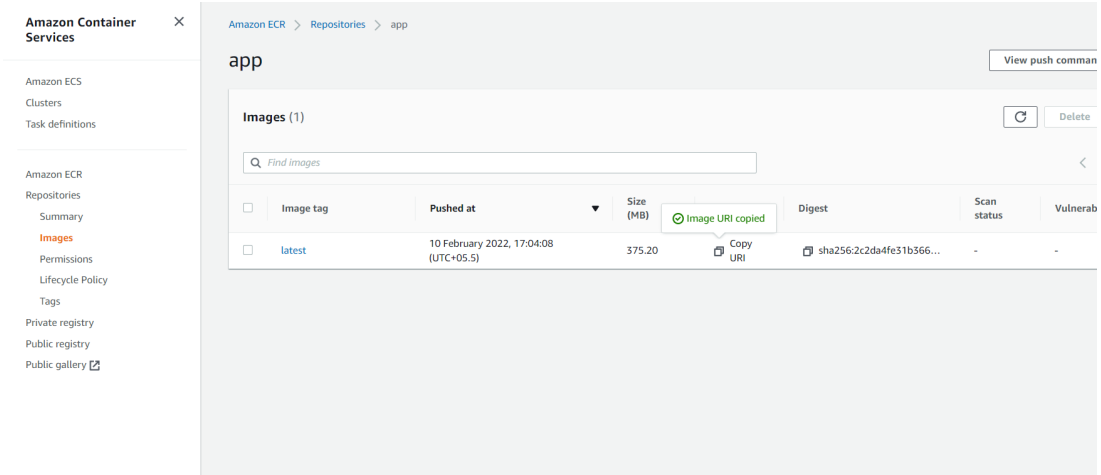


CPU ,I have opted for the lowest which is sufficient

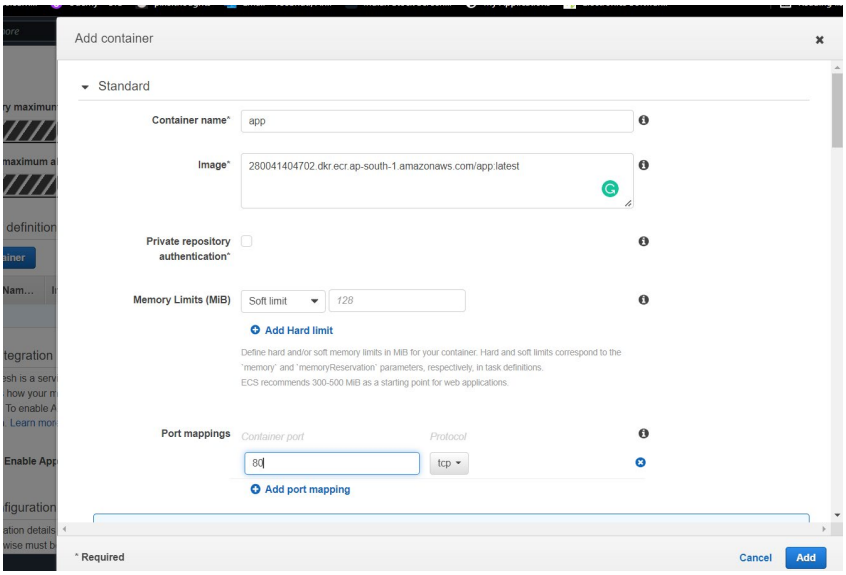


Click on the add container button

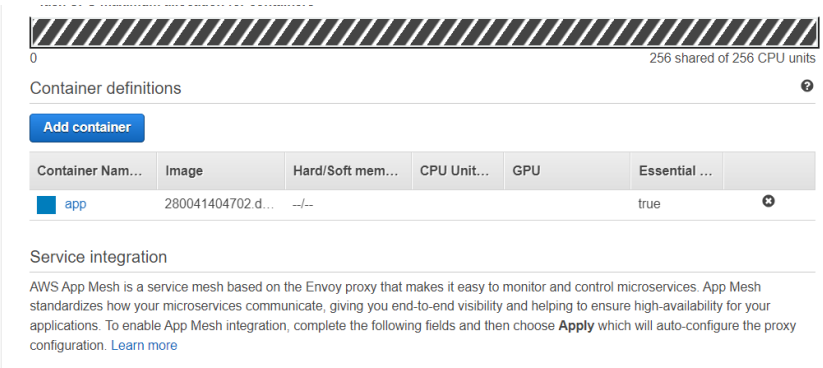
We need to add the details of the container that we have made in the docker and pushed into the Container repository



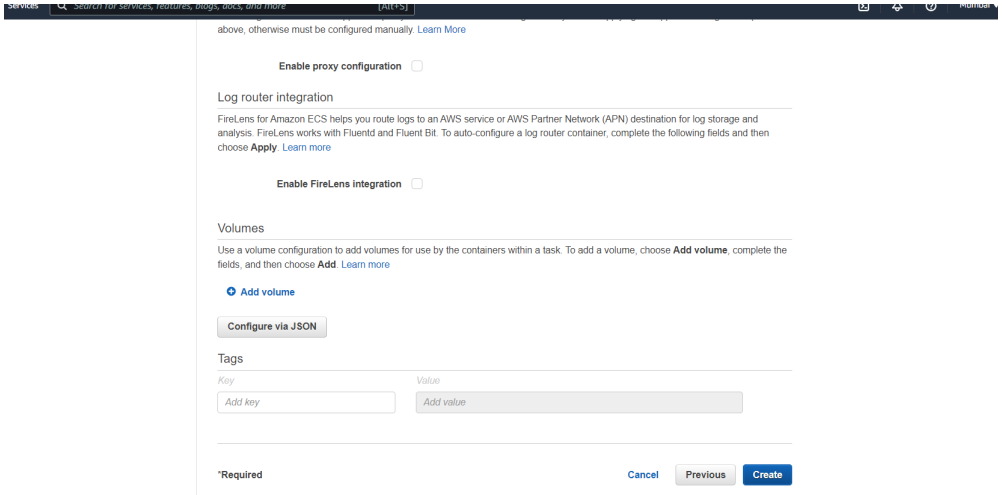
We now go to the ECR(repository) , copy the URI



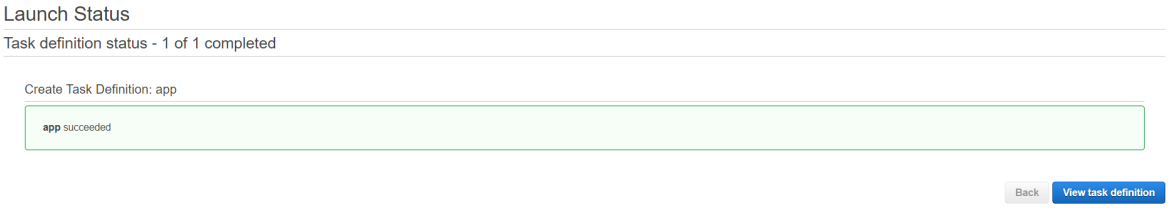
Fill the details of the container



After adding the container, it should look like this

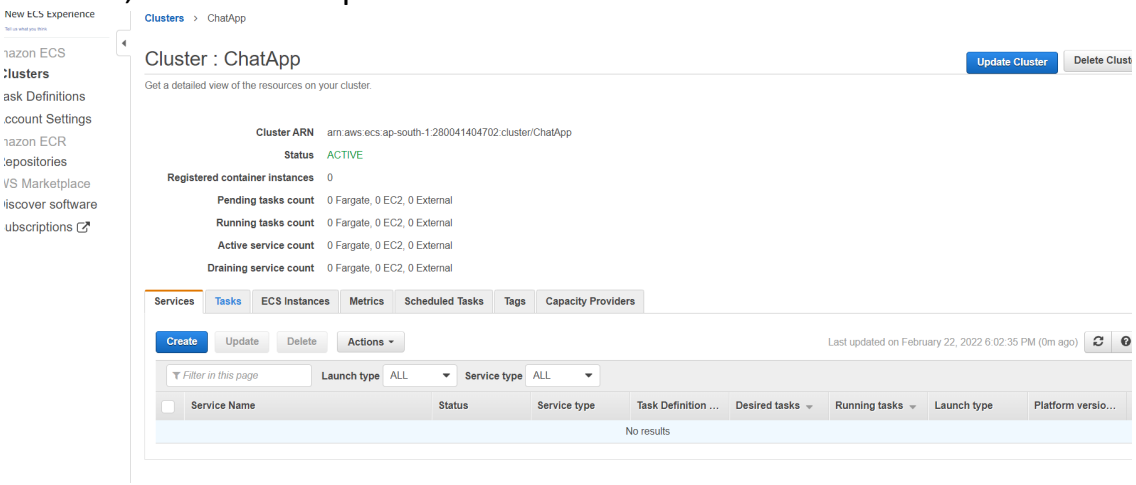


Click on create button

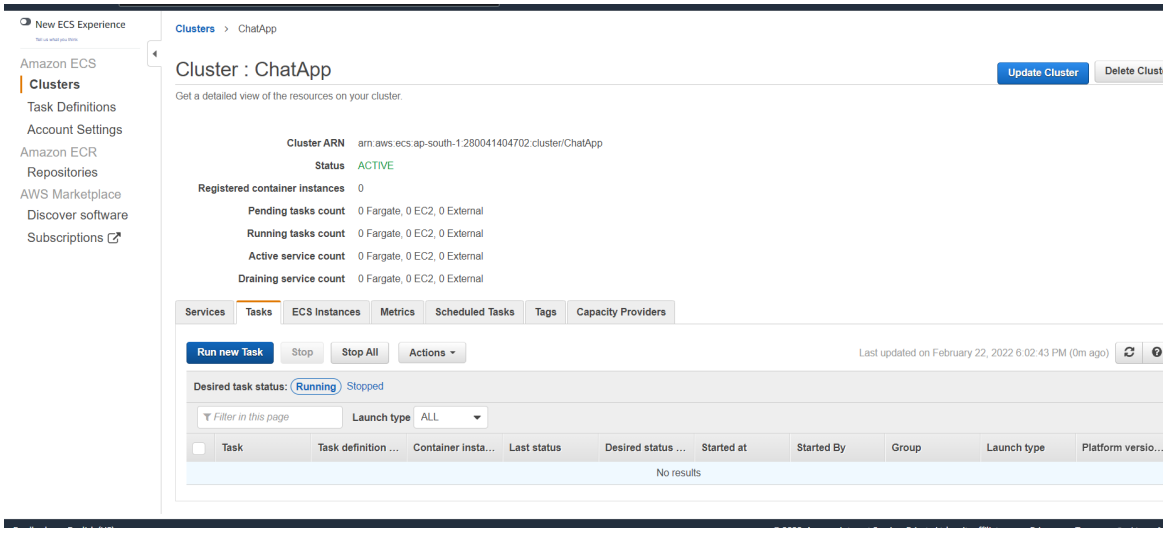


This message appears after the successful creation of the task definition

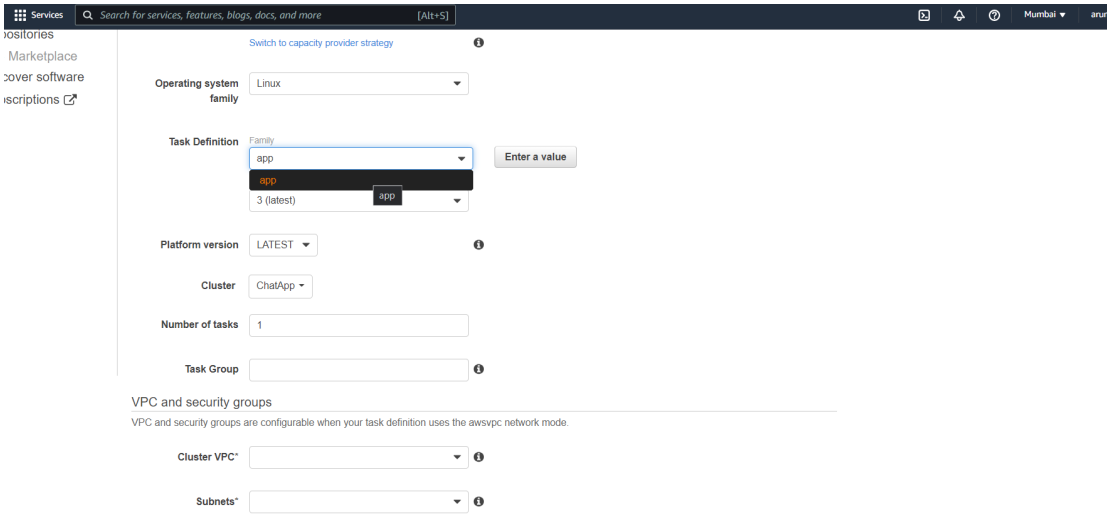
We now need to run the task definition
For that, we need to open the cluster



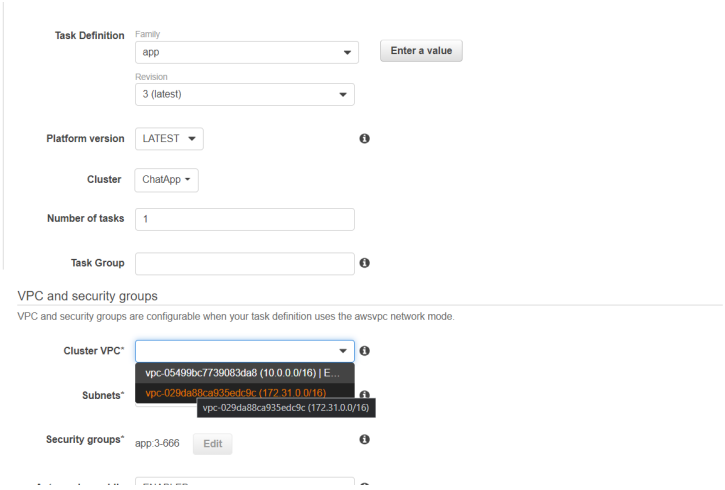
Our cluster



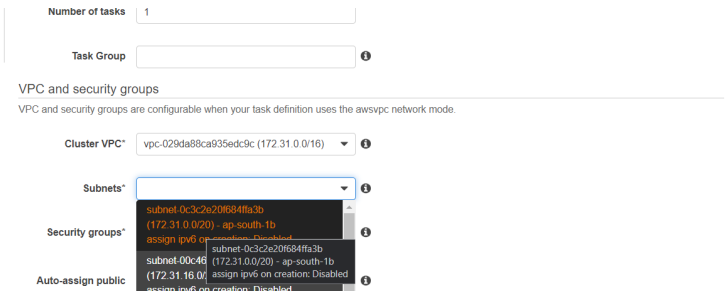
We now, navigate to the tasks where “Run new task button appears”



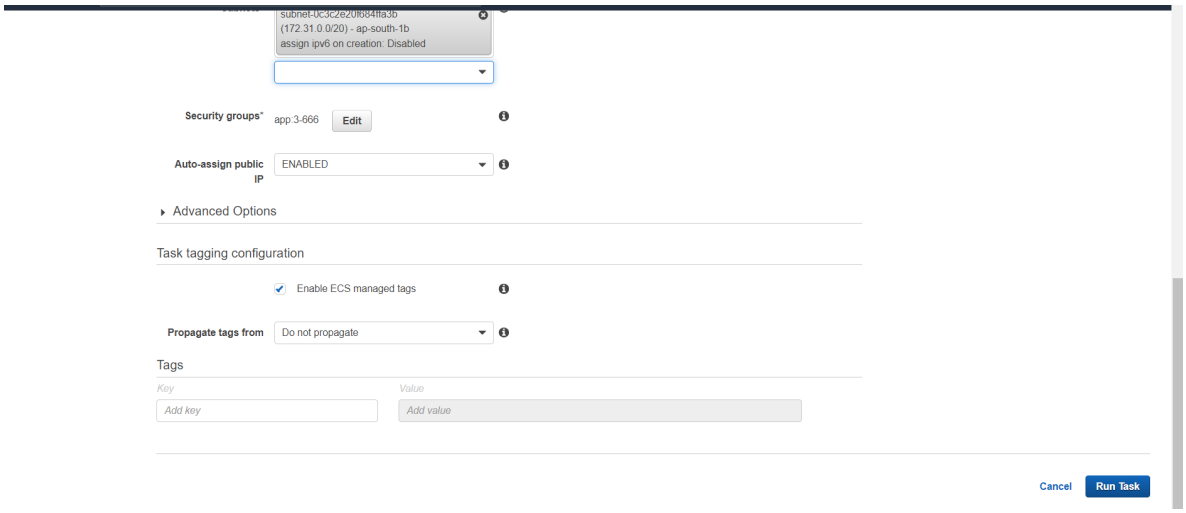
Select the task (app is my task name)



Select VPC

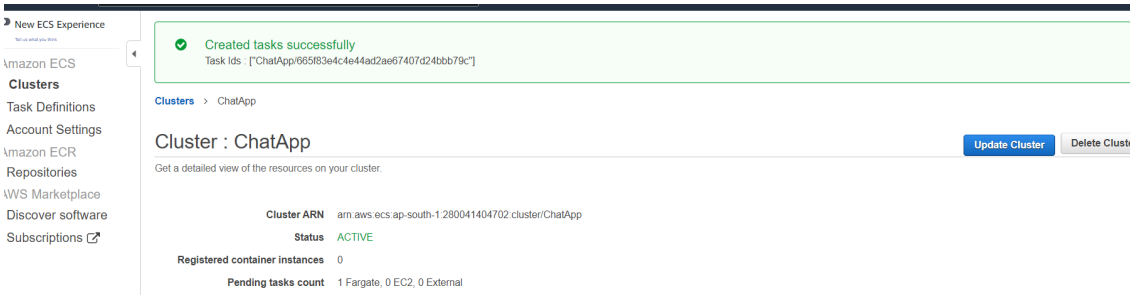


Select a subnet

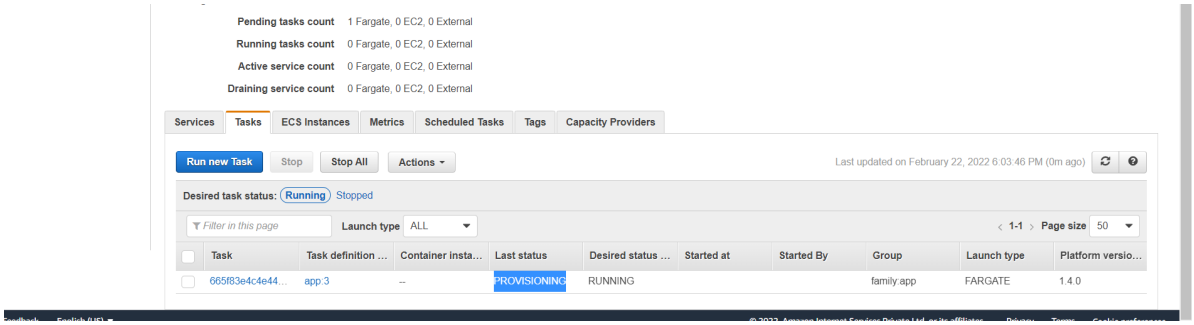


We can add security groups

After that, we need to click the “Run Task” button

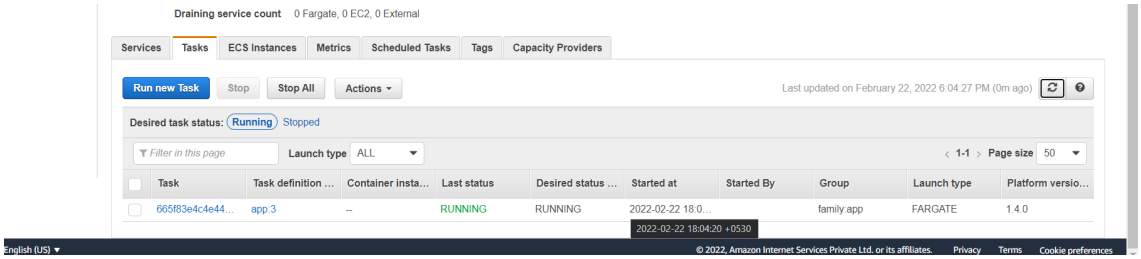


Task is now running perfectly

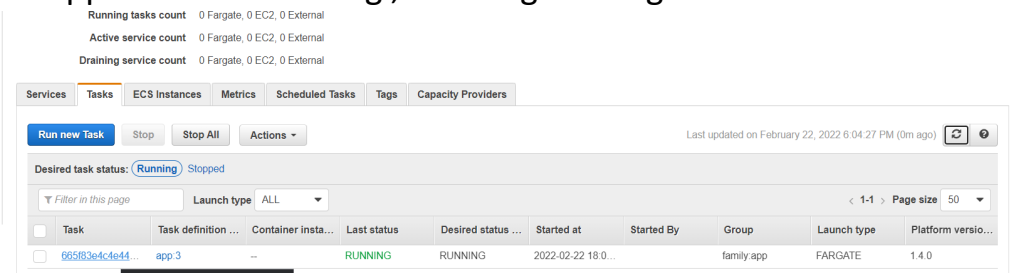


Wait until the status turns to RUNNING

You can refresh from here to check the status

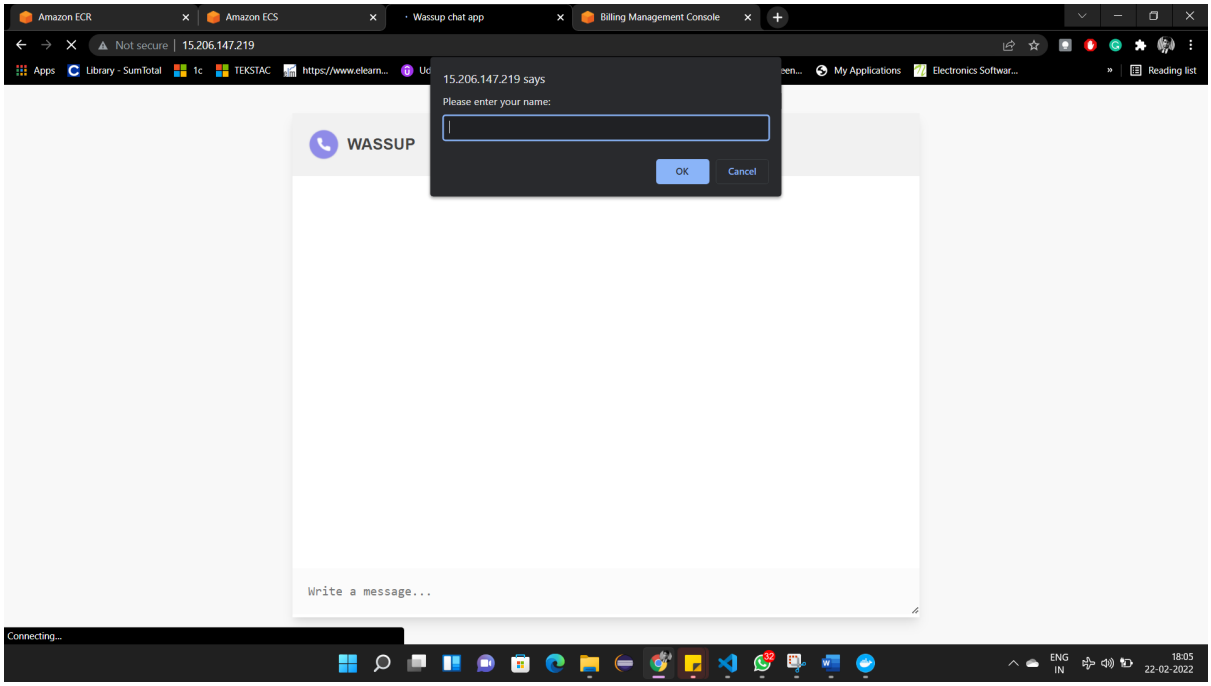


Once it appears to be running , we are good to go



Open the task

Find the public IP and open it in your browser



Hence the application is running perfectly

Testing :

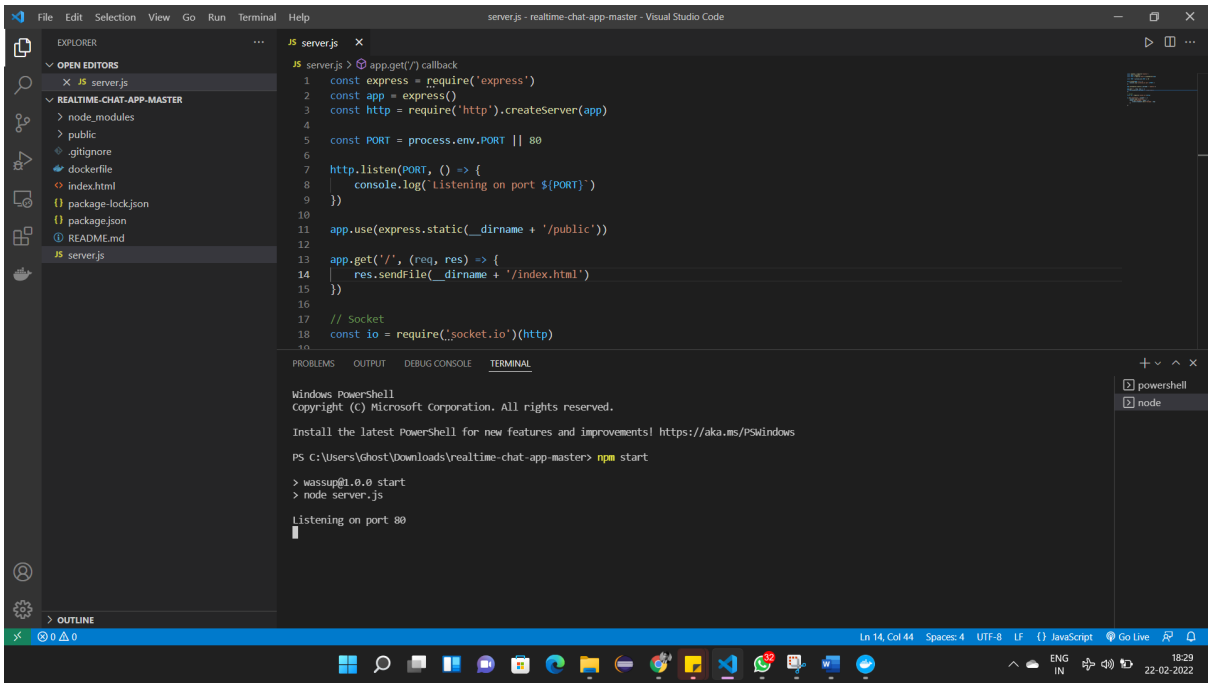
We have done Manual testing of the app

Test design:
The test will be of three phases

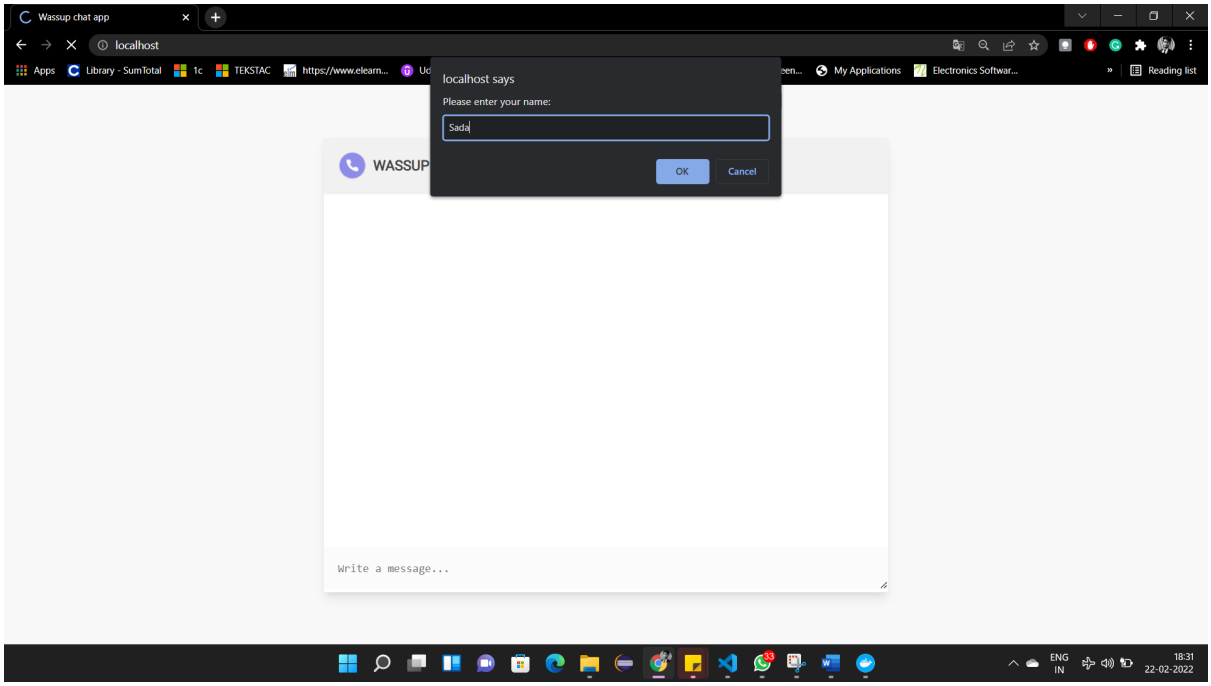
- Test the app on the local machine in different browsers and tabs
- Second phase: The app will be dockerised and it will be running in the local machine
- Third: the app will be run and tested after deploying on AWS

Test:

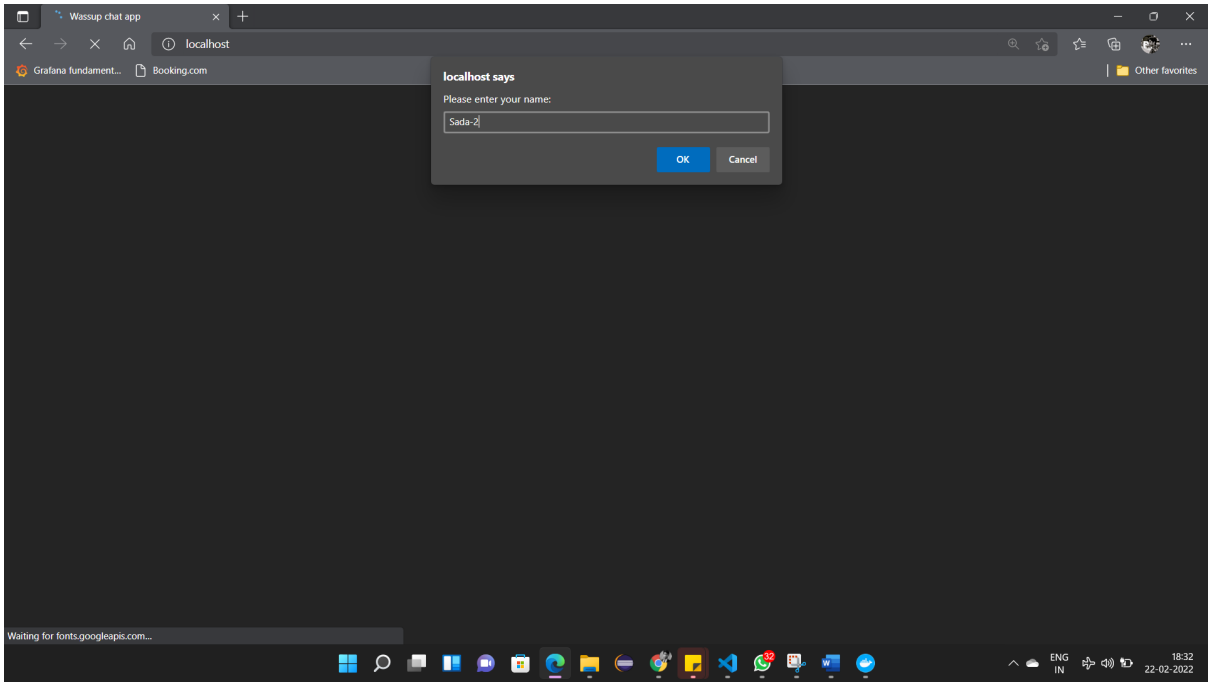
Testing locally:



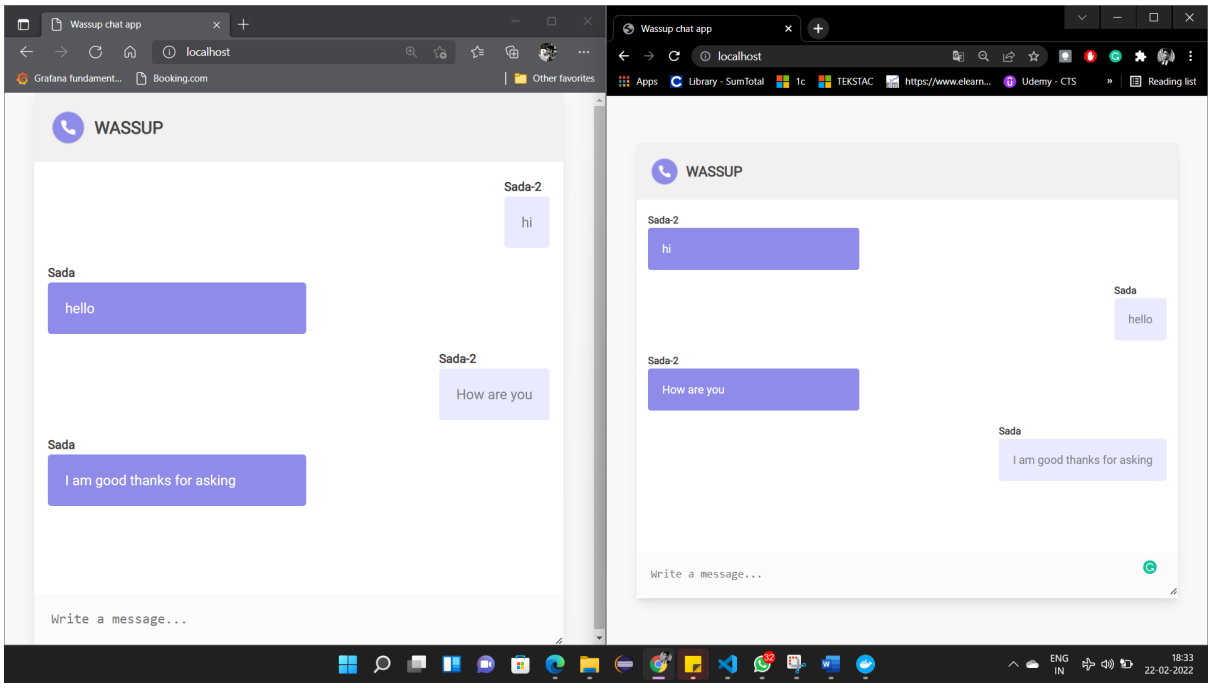
Starting the server



App running locally on the browser

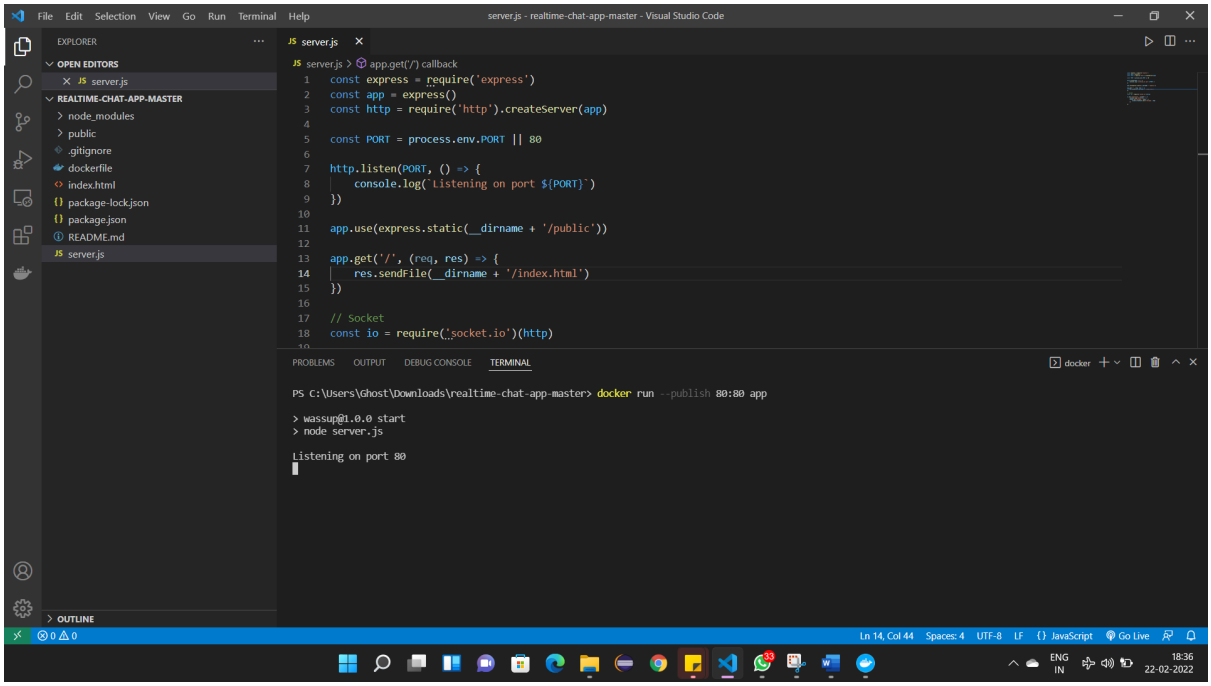


App running from another browser locally with different username

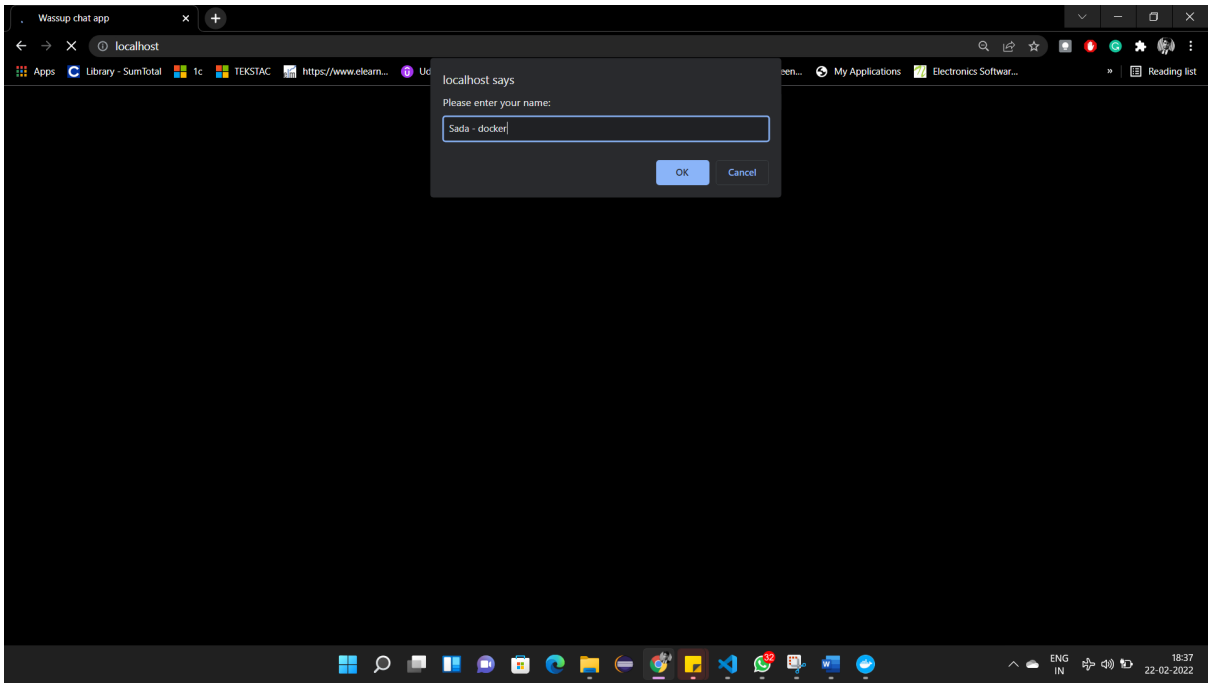


Chat application is running perfectly (local server)

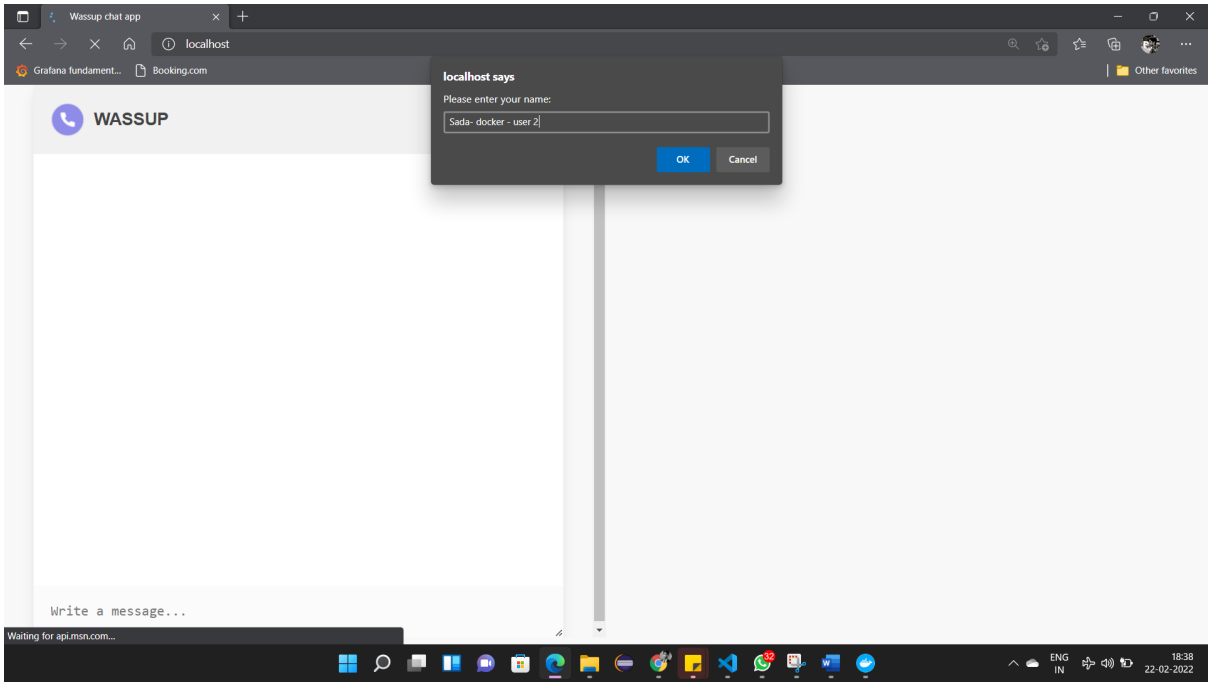
- Phase 2 testing (from docker container)



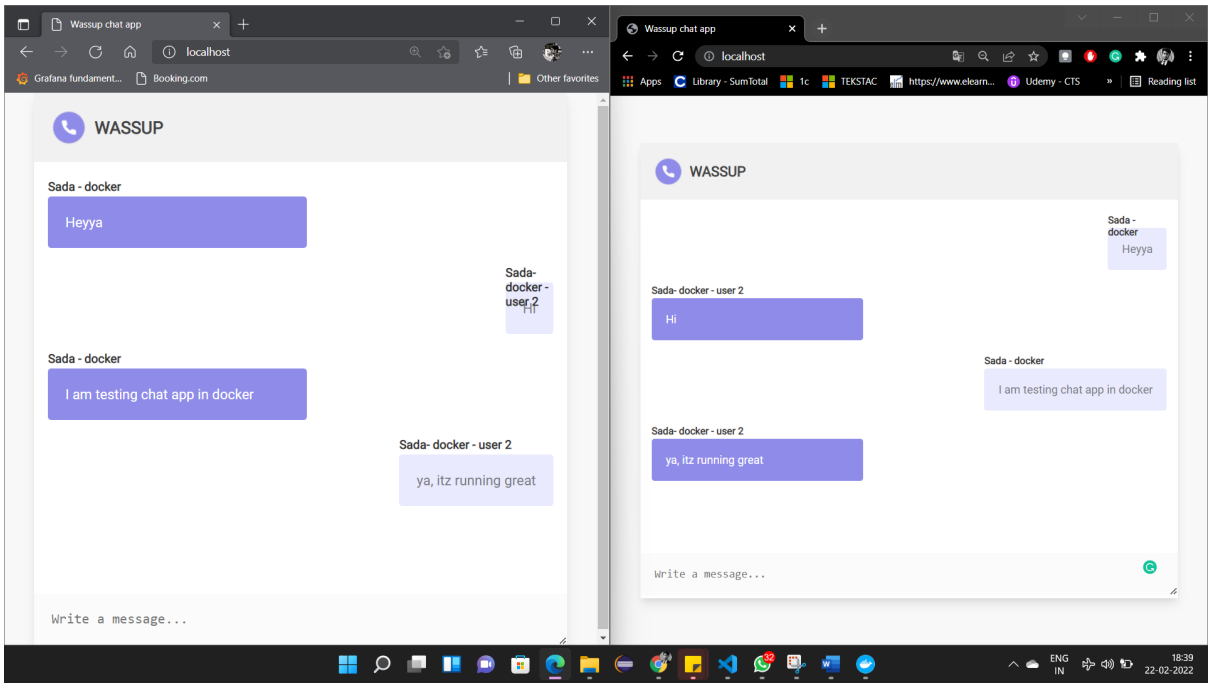
Docker run with port mapping



User 1

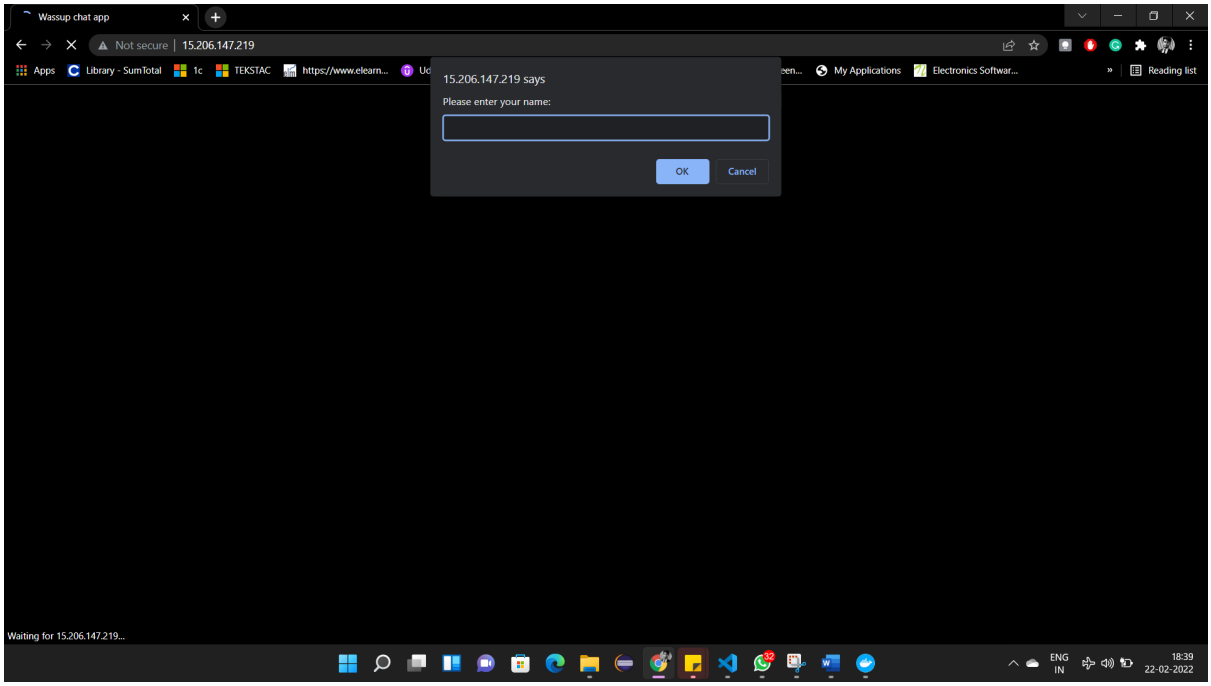


user 2 from another browser

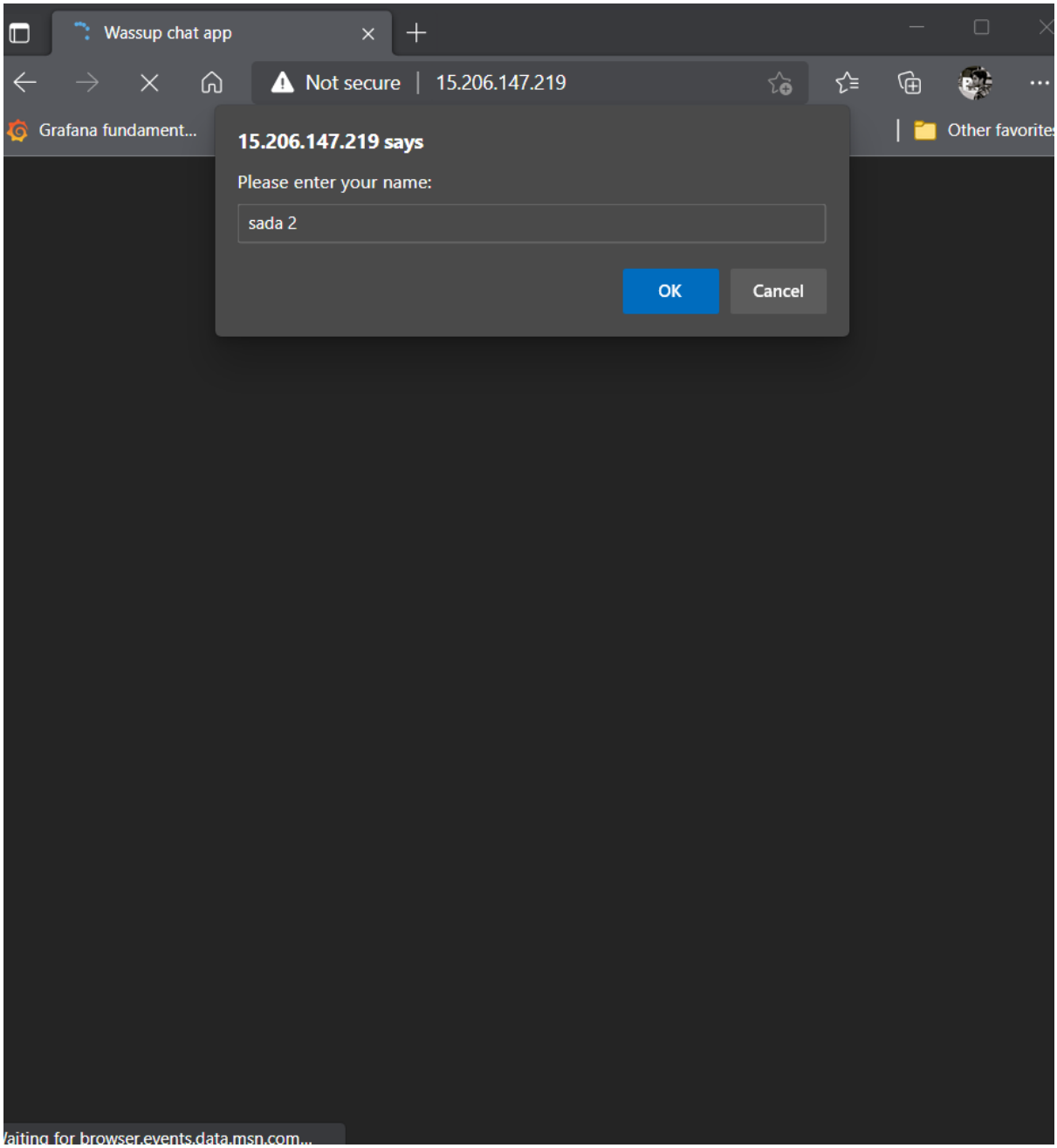


Chat app working from docker container as well

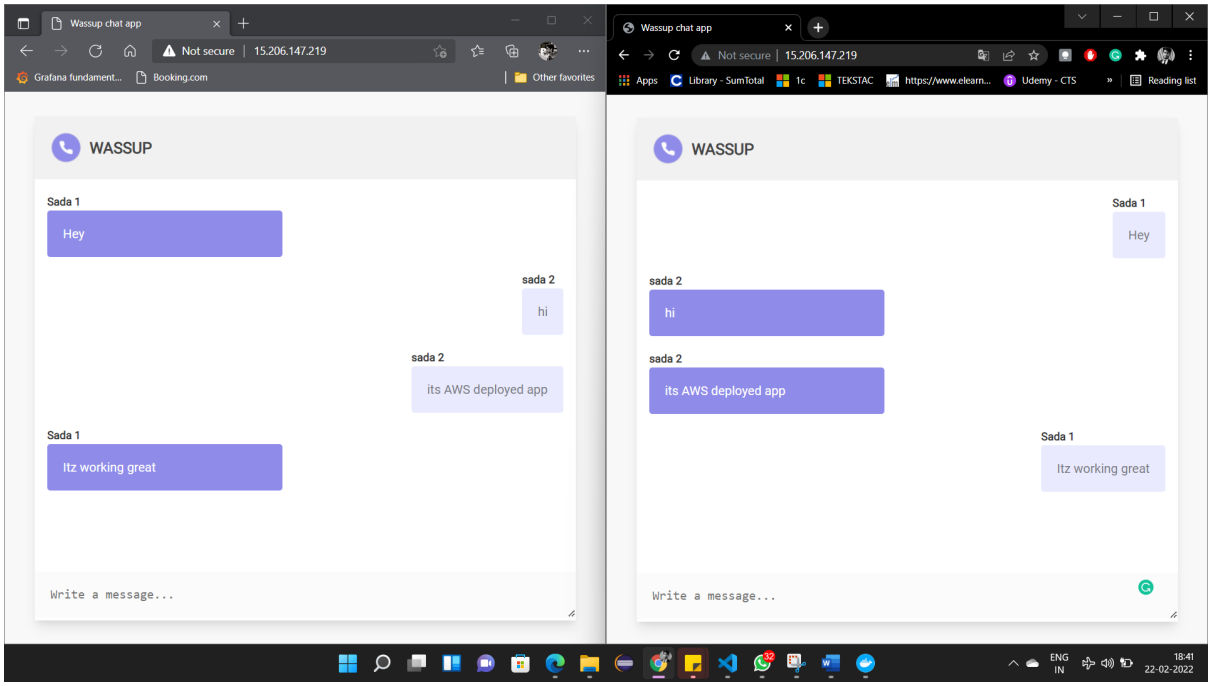
- Test phase 3
  Deployed app



User 1 from one browser



User from browser 2



We have tested the AWS deployed app in multiple devices and it worked well