

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.

```

1  {
2    "name": "wassup",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js"
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC"
12 }
13

```

Fig : Package.json that's been created

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

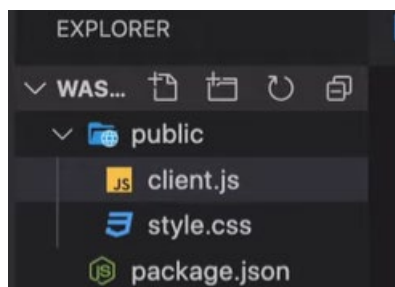


Fig: file structure

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

```

WASSUP
> public
  index.html
  package.json
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <meta charset="UTF-8">
6    <meta name="viewport" content="width=device-width, initial-scale=1.0">
7    <title>Wassup chat app</title>
8    <link rel="stylesheet" href="/public/style.css">
9  </head>
10
11  <body>
12
13    <script src="/public/client.js"></script>
14  </body>
15
16 </html>

```

Fig: HTML code with js and css files imported

- Next, the dependencies 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

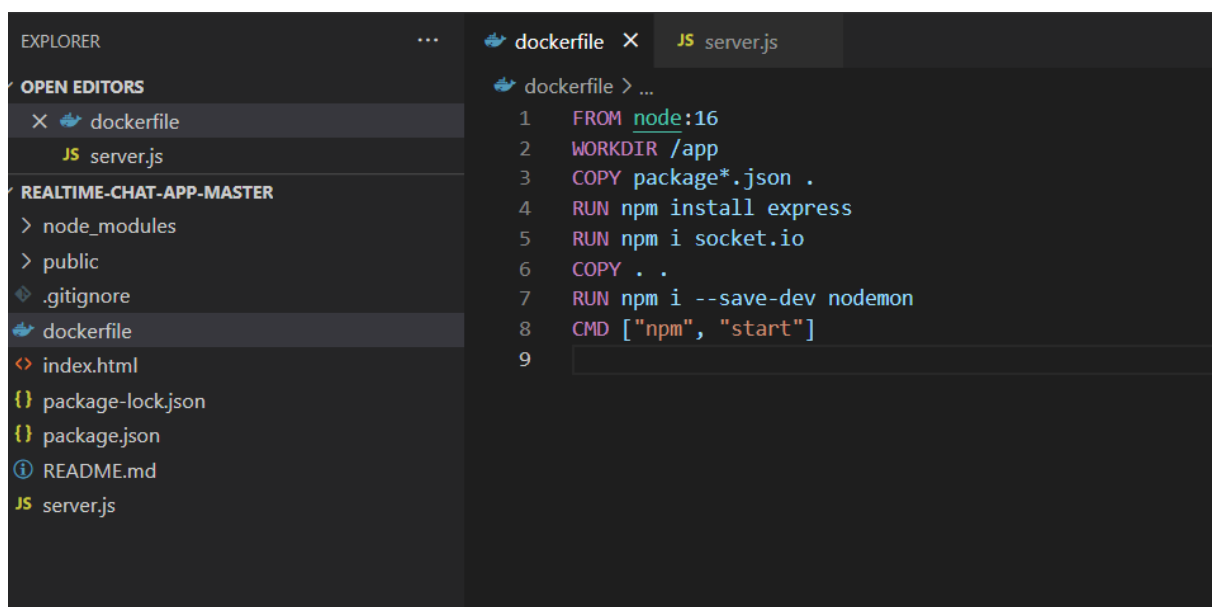
Deployment phase:

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

A screenshot of the Visual Studio Code editor interface. The Explorer sidebar on the left shows the file structure of a project named 'REALTIME-CHAT-APP-MASTER'. It includes folders like 'node_modules' and 'public', and files like '.gitignore', 'dockerfile', 'index.html', 'package-lock.json', 'package.json', 'README.md', and 'server.js'. The 'dockerfile' file is selected. The main editor area shows the content of 'dockerfile' with the following code:

```
1 FROM node:16
2 WORKDIR /app
3 COPY package*.json .
4 RUN npm install express
5 RUN npm i socket.io
6 COPY . .
7 RUN npm i --save-dev nodemon
8 CMD ["npm", "start"]
9
```

- 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
=> => transferring dockerfile: 197B 0.0s
=> [internal] load .dockerignore
=> => 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:fd86131ddf8e0faa8ba7a3e49b6bf571745946e663e4065f3bffa0a7204c1dde 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 sha256:26041f7f70bffa9634c6aae9c0b0f0e6ab3c0ee1d92610ae78492480af12918ad 0.0s
```

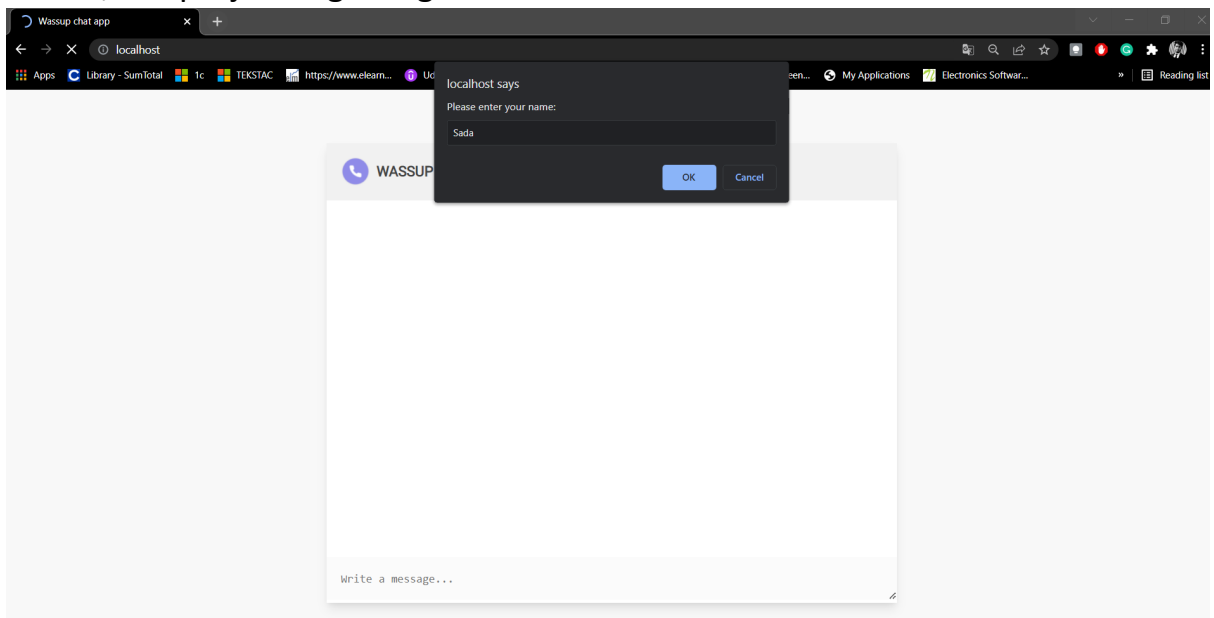
Here, the build was successful

- 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.

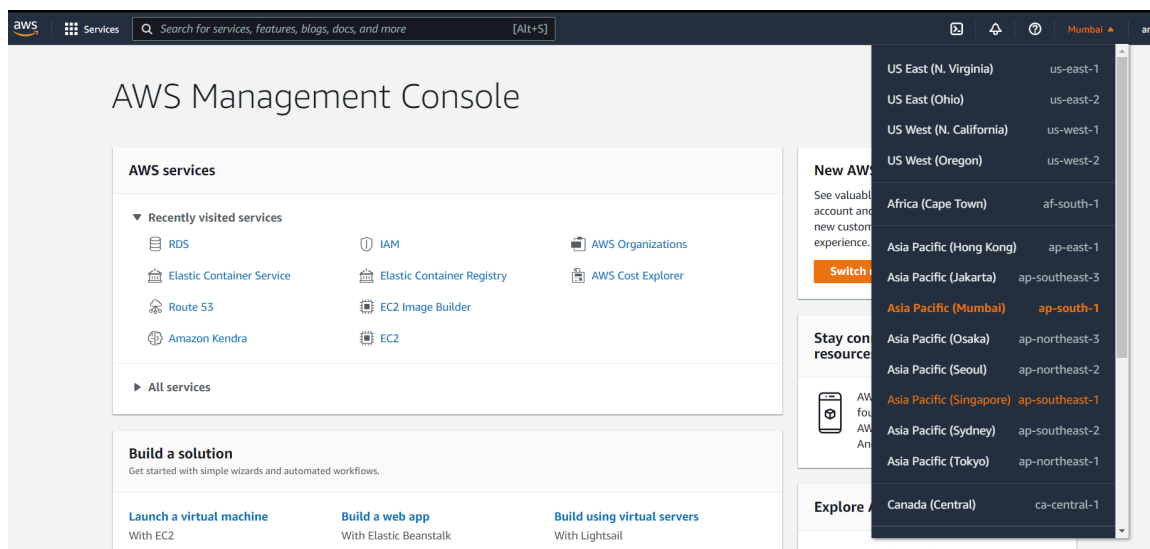
ECS provides a cluster .

AWS Fargate is a serverless, pay-as-you-go compute engine that lets you focus on building applications ***without managing servers***.

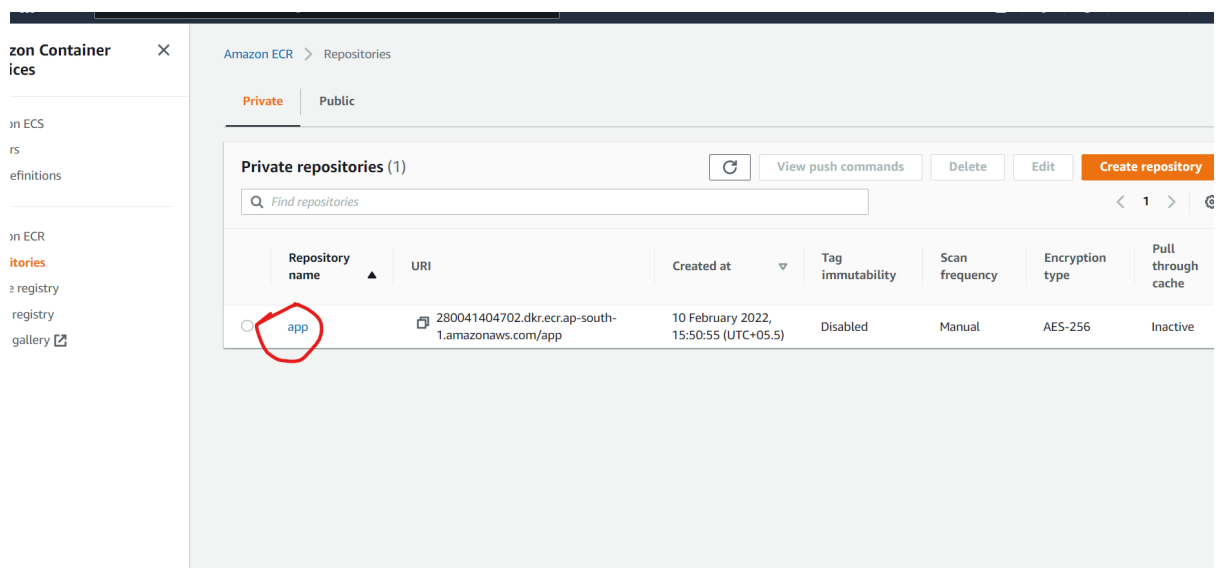
So, by applying load balancer, auto scaling through Fargate, the application will be scaled up as per the requirement and traffic.

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



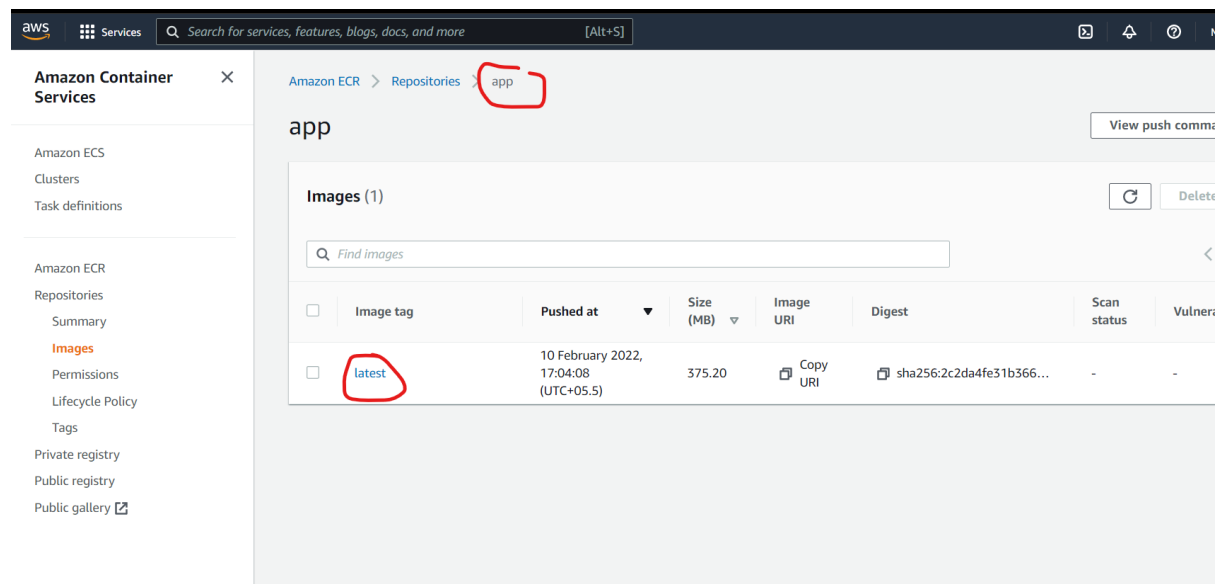
Firstly, select the region



- 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

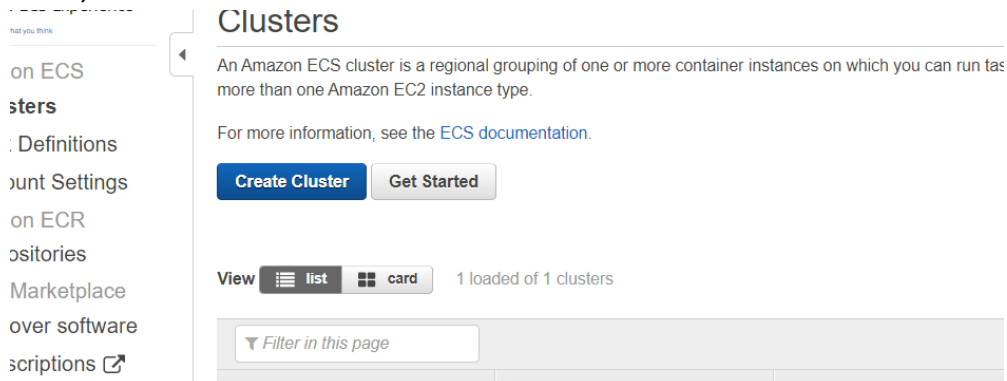
“docker tag app:latest 280041404702.dkr.ecr.ap-south-1.amazonaws.com/app:latest”

“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

- 1: Select cluster template
- 2: Configure cluster

The following cluster templates are available to simplify cluster creation. Additional configuration and integrations can be added later.

- 1: Select cluster template
- 2: Configure cluster

Cluster name*

ChatApp

☒ Create an empty cluster

Tags

Key

Add key

Value

Add value

CloudWatch Container Insights

CloudWatch Container Insights is a monitoring and troubleshooting solution for containerized applications and microservices. It collects, aggregates, and summarizes compute utilization such as CPU, memory, disk, and network, and diagnostic information such as container restart failures to help you isolate issues with your clusters and resolve them quickly. [Learn more](#)

CloudWatch Container Insights

☐ Enable Container Insights

*Required

Cancel

Previous

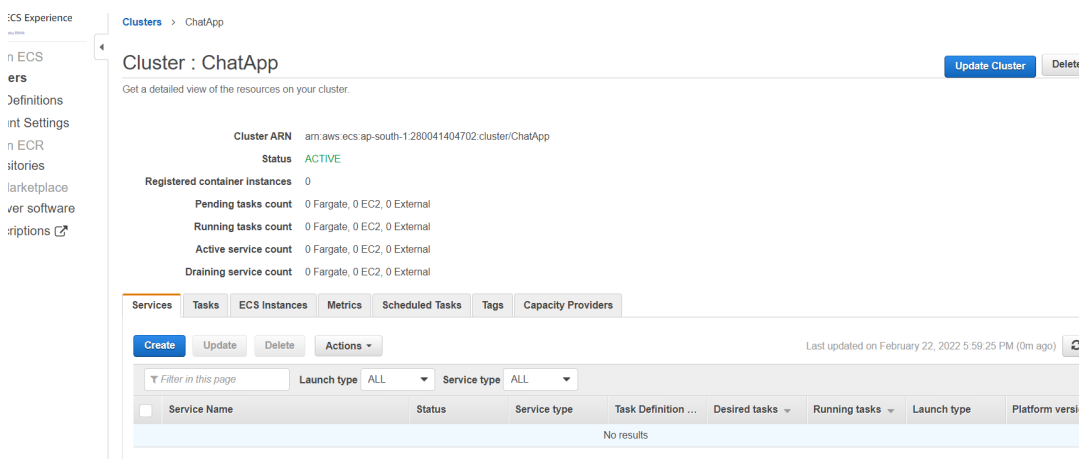
Create

reshooting solution for containerized applications and microservices. It can show you such as CPU, memory, disk, and network; and diagnostic information. You can work with your clusters and resolve them quickly. [🔗 Learn more](#)

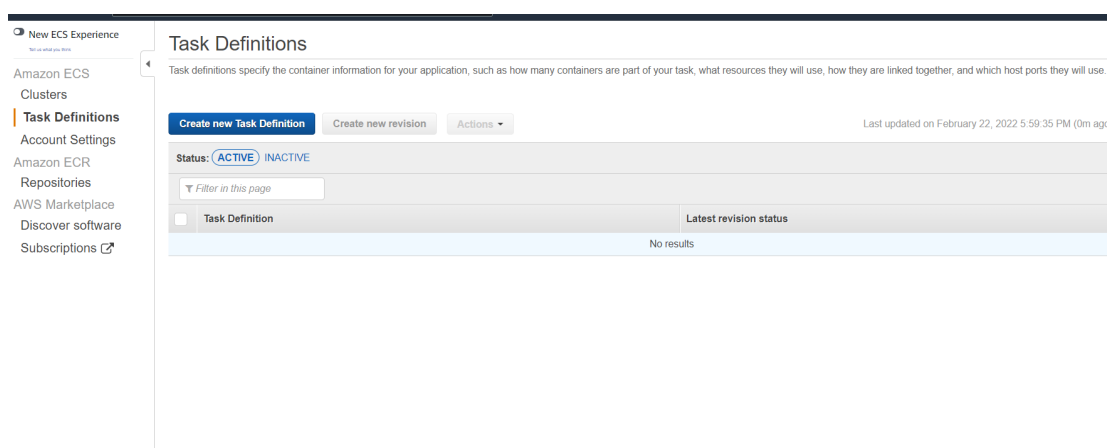
Click on create button



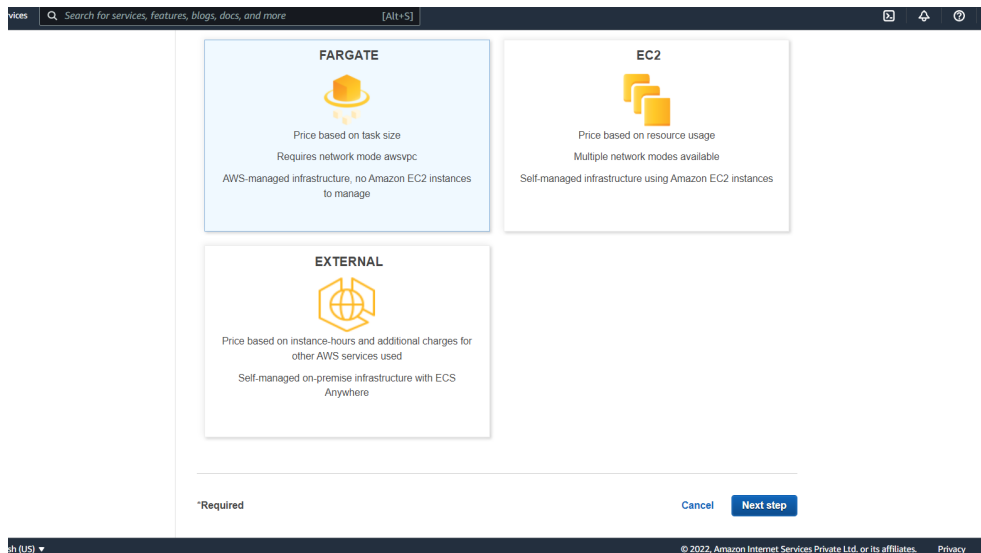
This appears after the cluster gets created successfully



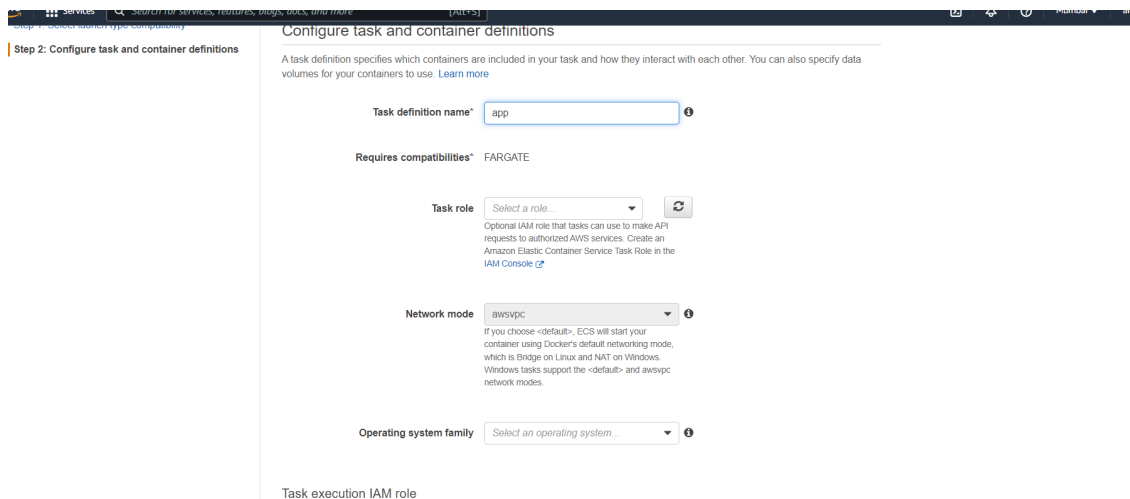
Check your cluster is active or not



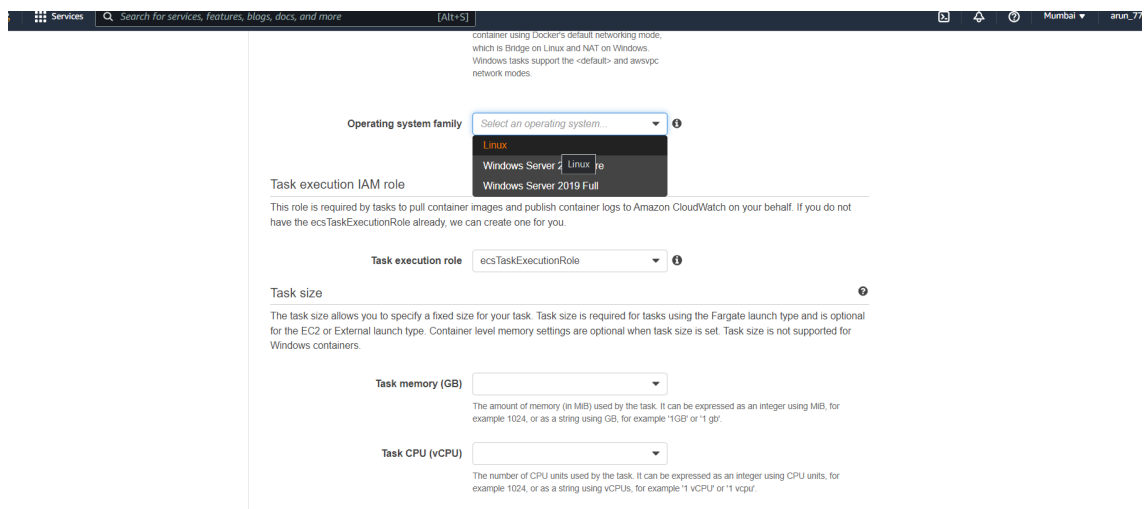
Now we need to create a task definition



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

This role is required by tasks to pull container images and publish container logs to Amazon CloudWatch on your behalf. If you do not have the `ecsTaskExecutionRole` already, we can create one for you.

Task execution role: `ecsTaskExecutionRole`

Task size

The task size allows you to specify a fixed size for your task. Task size is required for tasks using the Fargate launch type and is optional for the EC2 or External launch type. Container level memory settings are optional when task size is set. Task size is not supported for Windows containers.

Task memory (GB): **0.5GB**

Task CPU (vCPU): **0.5GB**

Container definitions

[Add container](#)

Container Name...	Image	Hard/Soft mem...	CPU Unit...	GPU	Essential ...
No results					

Service integration

AWS App Mesh is a service mesh based on the Envoy proxy that makes it easy to monitor and control microservices. App Mesh standardizes how your microservices communicate, giving you end-to-end visibility and helping to ensure high-availability for your applications. To enable App Mesh integration, complete the following fields and then choose **Apply** which will auto-configure the proxy configuration. [Learn more](#)

☐ Enable App Mesh integration

Proxy configuration

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

Task size

The task size allows you to specify a fixed size for your task. Task size is required for tasks using the Fargate launch type and is optional for the EC2 or External launch type. Container level memory settings are optional when task size is set. Task size is not supported for Windows containers.

Task memory (GB): **0.5GB**

The amount of memory (in MiB) used by the task. It can be expressed as an integer using MiB, for example 1024, or as a string using GB, for example 1GB or 1 gi.

Task CPU (vCPU): **0.25 vCPU**

Task memory maximum allocation for container memory reservation: 0 to 512 shared of 512 MiB

Task CPU maximum allocation for containers: 0 to 256 shared of 256 CPU units

Container definitions

[Add container](#)

Container Name...	Image	Hard/Soft mem...	CPU Unit...	GPU	Essential ...
No results					

CPU ,I have opted for the lowest which is sufficient

Task CPU (vCPU): **0.25 vCPU**

The valid CPU for 0.5 GB memory is: 0.25 vCPU

Task memory maximum allocation for container memory reservation: 0 to 512 shared of 512 MiB

Task CPU maximum allocation for containers: 0 to 256 shared of 256 CPU units

Container definitions

[Add container](#)

Container Name...	Image	Hard/Soft mem...	CPU Unit...	GPU	Essential ...
No results					

Service integration

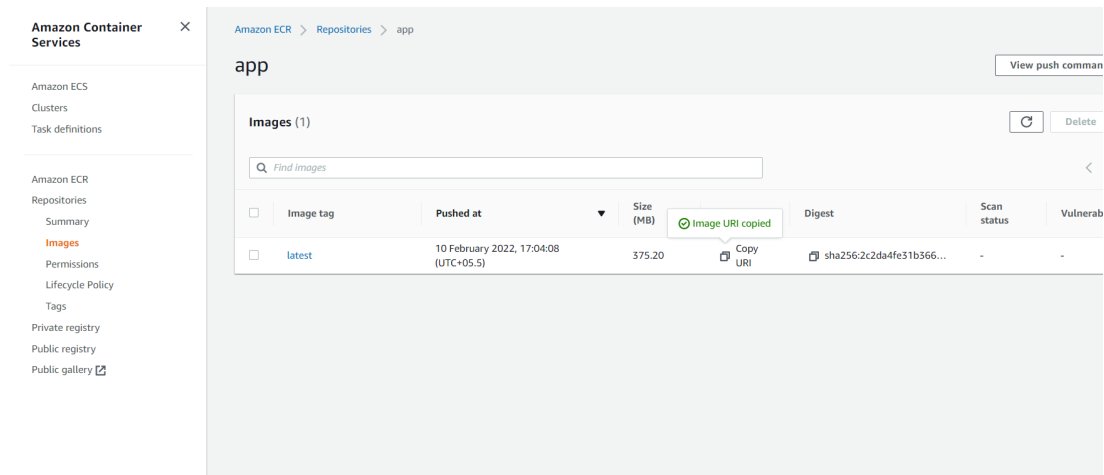
AWS App Mesh is a service mesh based on the Envoy proxy that makes it easy to monitor and control microservices. App Mesh standardizes how your microservices communicate, giving you end-to-end visibility and helping to ensure high-availability for your applications. To enable App Mesh integration, complete the following fields and then choose **Apply** which will auto-configure the proxy configuration. [Learn more](#)

☐ Enable App Mesh integration

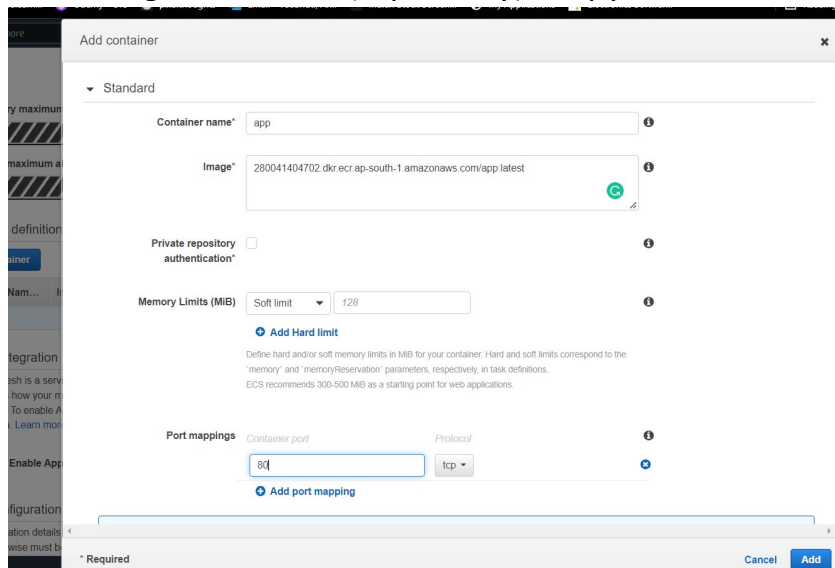
Proxy configuration

Click on the add container button

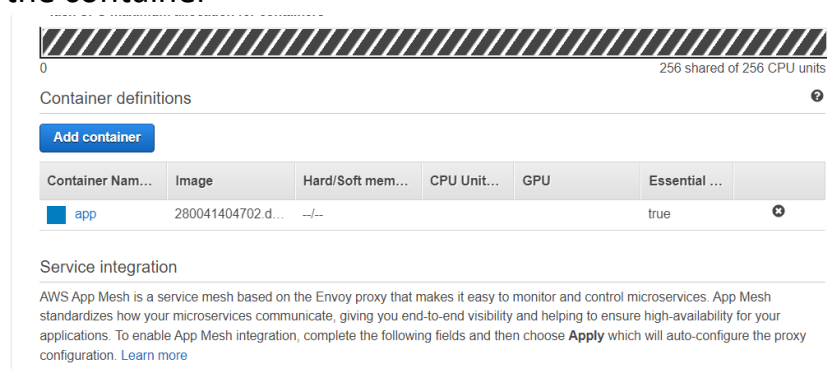
We need to add the details of the container that we have made in the docker and pushed into the Container repository



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

above, otherwise must be configured manually. [Learn More](#)

Enable proxy configuration ☐

Log router integration

FireLens for Amazon ECS helps you route logs to an AWS service or AWS Partner Network (APN) destination for log storage and analysis. FireLens works with Fluentd and Fluent Bit. To auto-configure a log router container, complete the following fields and then choose **Apply**. [Learn more](#)

Enable FireLens integration ☐

Volumes

Use a volume configuration to add volumes for use by the containers within a task. To add a volume, choose **Add volume**, complete the fields, and then choose **Add**. [Learn more](#)

[Add volume](#)

Configure via JSON

Tags

Key	Value
Add key	Add value

*Required

[Cancel](#) [Previous](#) [Create](#)

Click on create button

Launch Status

Task definition status - 1 of 1 completed

Create Task Definition: app

app succeeded

[Back](#) [View task definition](#)

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

New ECS experience

Amazon ECS

Clusters

Task Definitions

Account Settings

Amazon ECR

Repositories

VS Marketplace

Discover software

Subscriptions

[Clusters](#) > ChatApp

Cluster : ChatApp [Update Cluster](#) [Delete Cluster](#)

Get a detailed view of the resources on your cluster.

Cluster ARN am-aws-ecs:ap-south-1:280041404702:cluster/ChatApp

Status ACTIVE

Registered container instances 0

Pending tasks count 0 Fargate, 0 EC2, 0 External

Running tasks count 0 Fargate, 0 EC2, 0 External

Active service count 0 Fargate, 0 EC2, 0 External

Draining service count 0 Fargate, 0 EC2, 0 External

Services **Tasks** **EC2 Instances** **Metrics** **Scheduled Tasks** **Tags** **Capacity Providers**

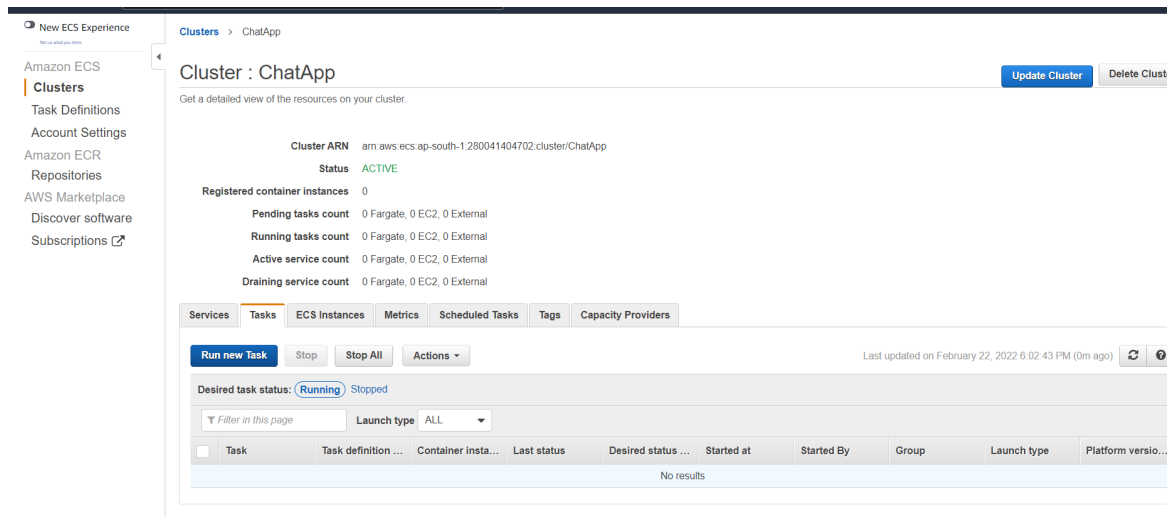
[Create](#) [Update](#) [Delete](#) [Actions](#)

Last updated on February 22, 2022 6:02:35 PM (1m ago)

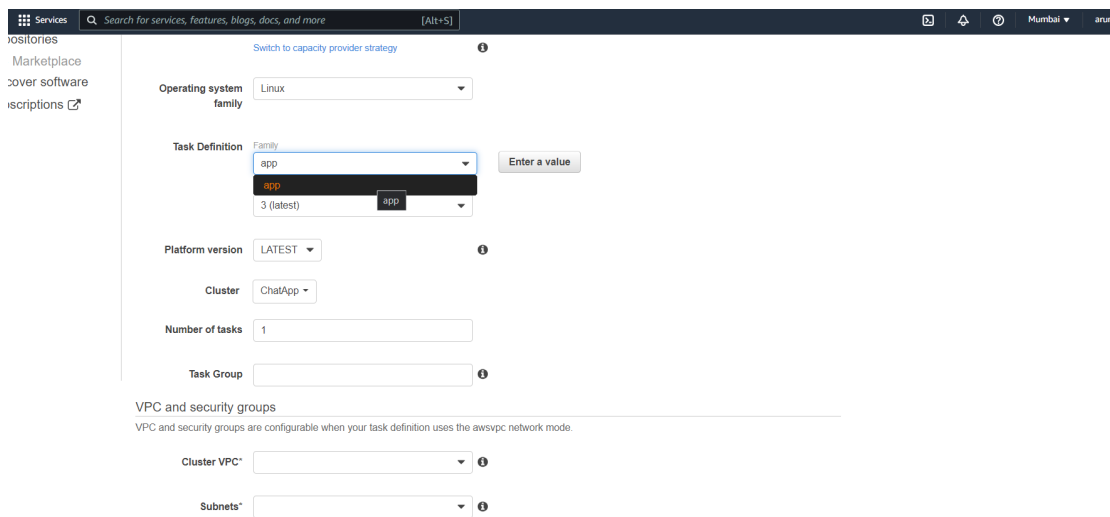
Launch type ALL **Service type** ALL

<input type="checkbox"/>	Service Name	Status	Service type	Task Definition ...	Desired tasks	Running tasks	Launch type	Platform versio...
No results								

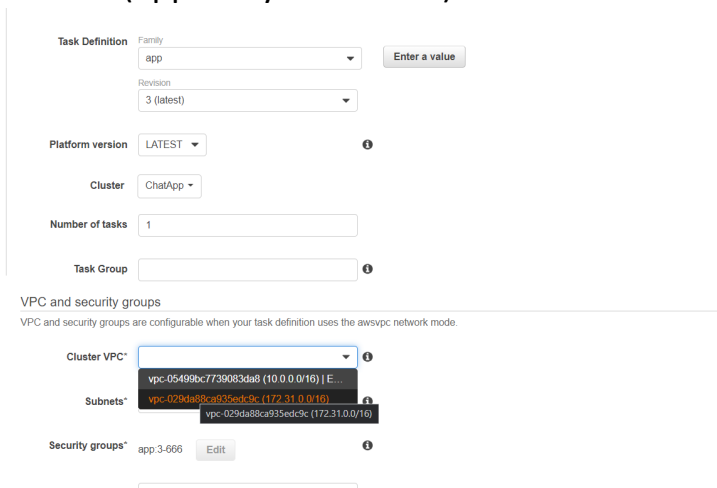
Our cluster



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



Select the task (app is my task name)



Select VPC

Number of tasks

Task Group

VPC and security groups

VPC and security groups are configurable when your task definition uses the awsvpc network mode.

Cluster VPC*

Subnets*

Security groups*

Auto-assign public IP

Select a subnet

Subnets*

Security groups*

Auto-assign public IP

Advanced Options

Task tagging configuration

☒ Enable ECS managed tags

Propagate tags from

Tags

Key

Value

Cancel Run Task

We can add security groups

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

New ECS Experience

Amazon ECS

Clusters

Task Definitions

Account Settings

Amazon ECR

Repositories

AWS Marketplace

Discover software

Subscriptions

Created tasks successfully

Task ids : ["ChatApp/665f83e4c4e44ad2ae67407d24bbb79c"]

Clusters > ChatApp

Cluster : ChatApp

Update Cluster Delete Cluster

Get a detailed view of the resources on your cluster.

Cluster ARN

Status

Registered container instances

Pending tasks count

Task is now running perfectly

Pending tasks count

Running tasks count

Active service count

Draining service count

Services Tasks ECS Instances Metrics Scheduled Tasks Tags Capacity Providers

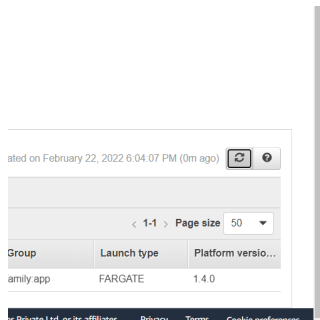
Run new Task Stop Stop All Actions

Desired task status: Stopped

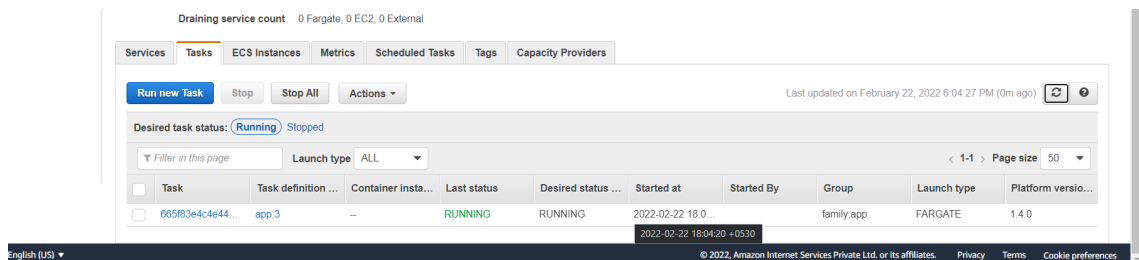
Filter in this page Launch type ALL

Task	Task definition	Container insta...	Last status	Desired status	Started at	Started By	Group	Launch type	Platform versio...
<input type="checkbox"/> 665f83e4c4e44...	app-3	...	PROVISIONING	RUNNING			family app	FARGATE	1.4.0

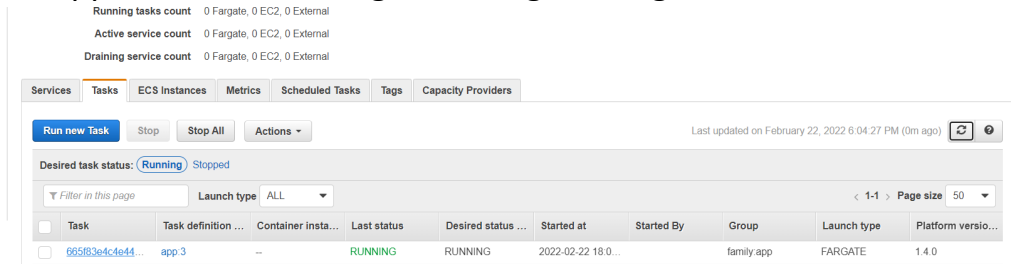
Wait until the status turns to RUNNING



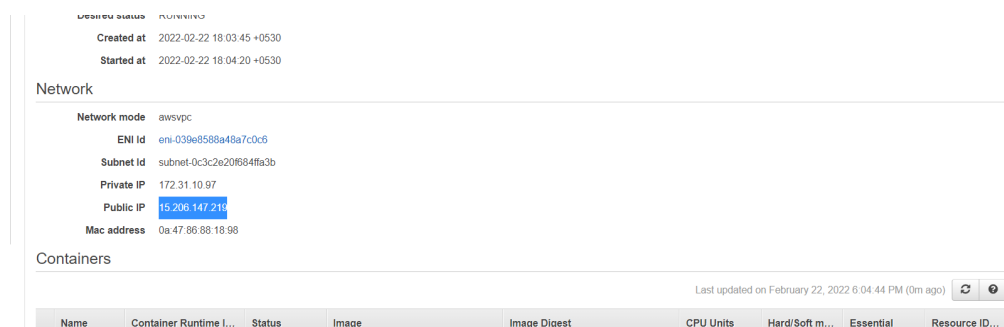
You can refresh from here to check the status



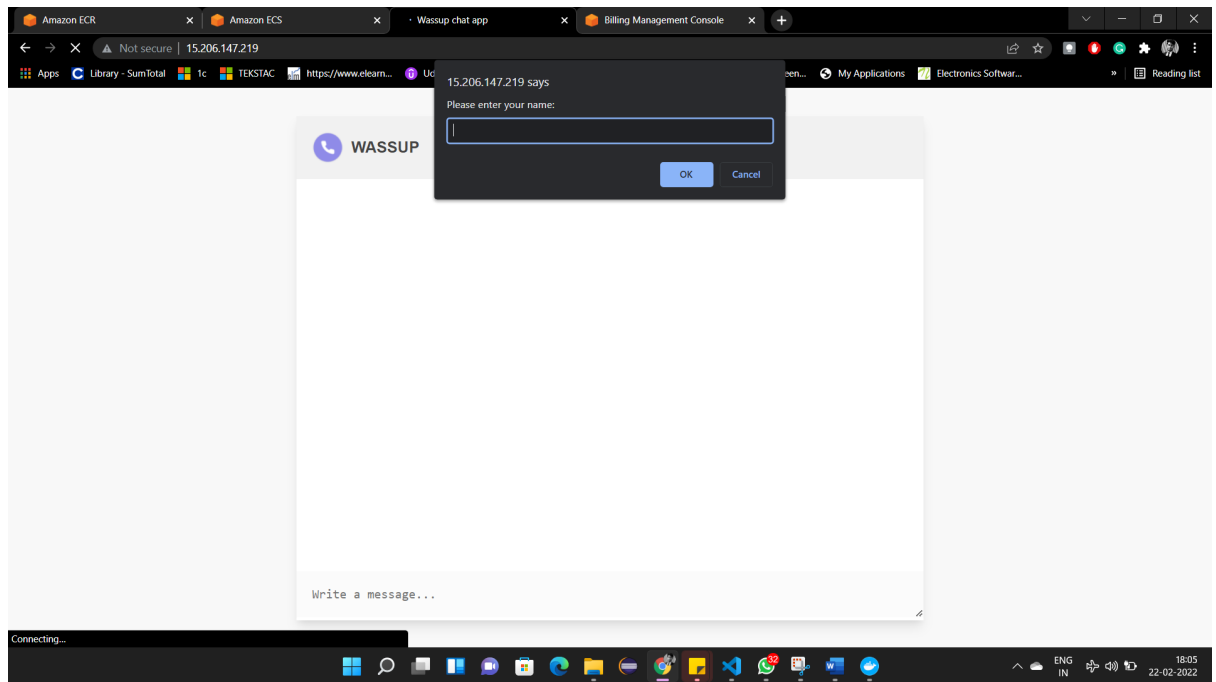
Once it appears to be running , we are good to go



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

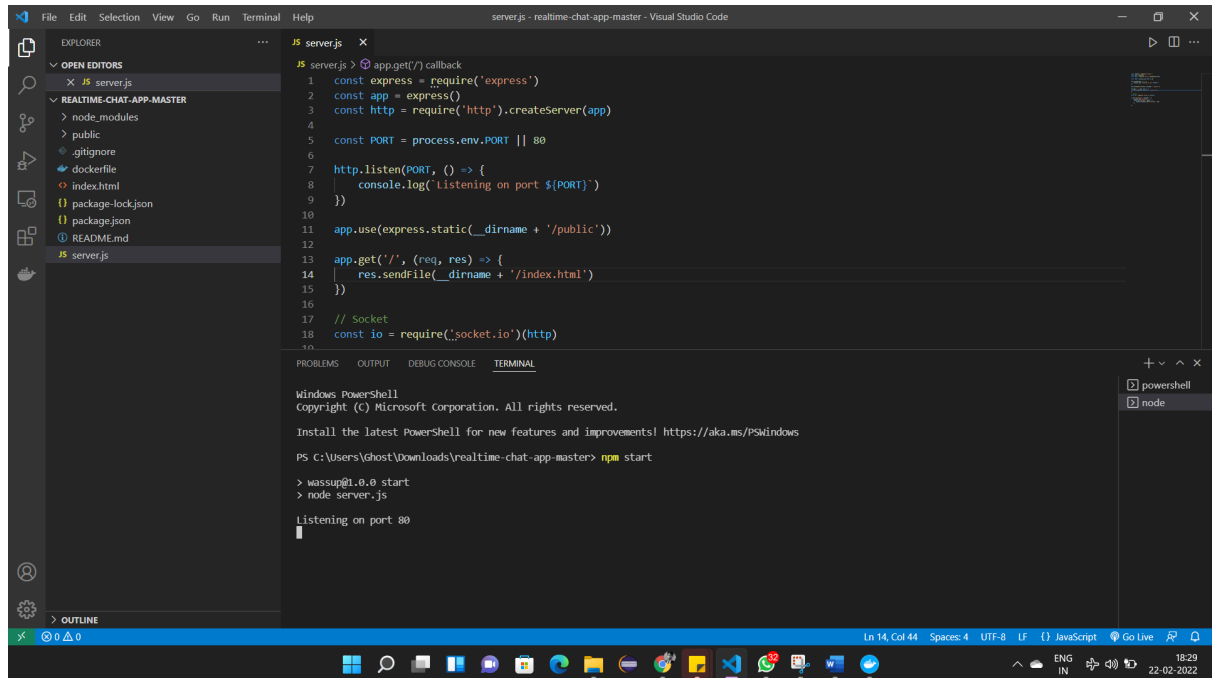
Test design:

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:



The screenshot shows the Visual Studio Code editor with a file named `server.js` open. The file contains the following JavaScript code:

```
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)
```

The terminal window at the bottom shows the command prompt output:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

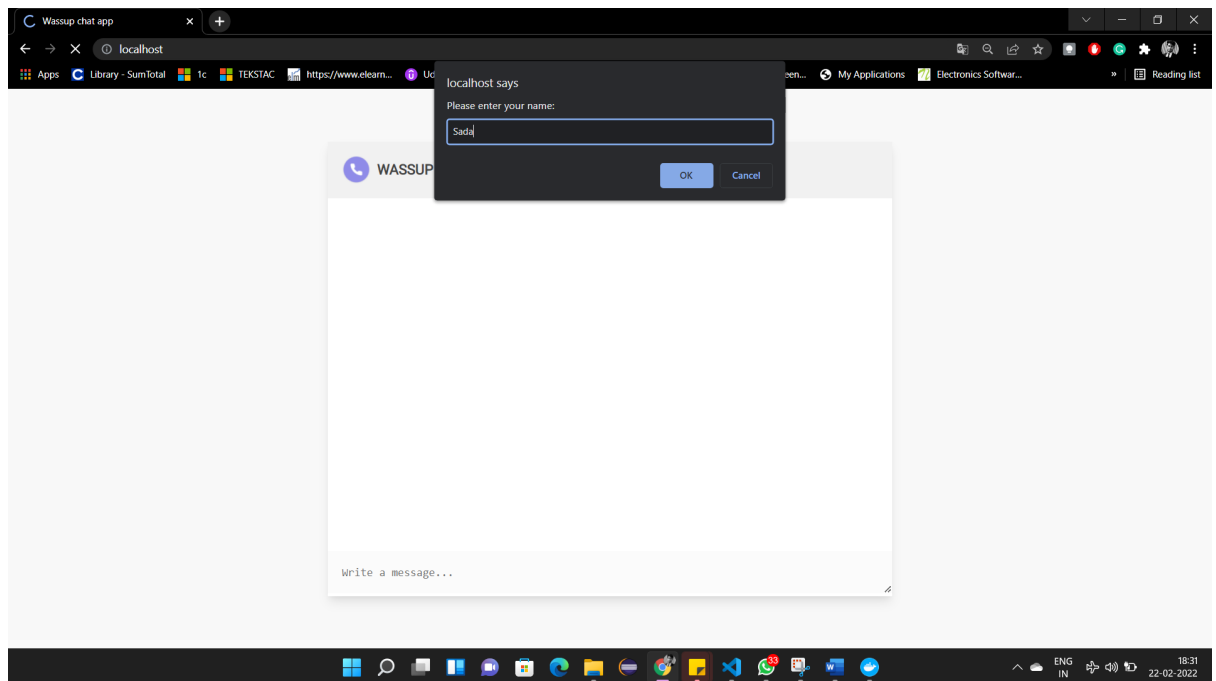
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\ghost\Downloads\realtime-chat-app-master> npm start

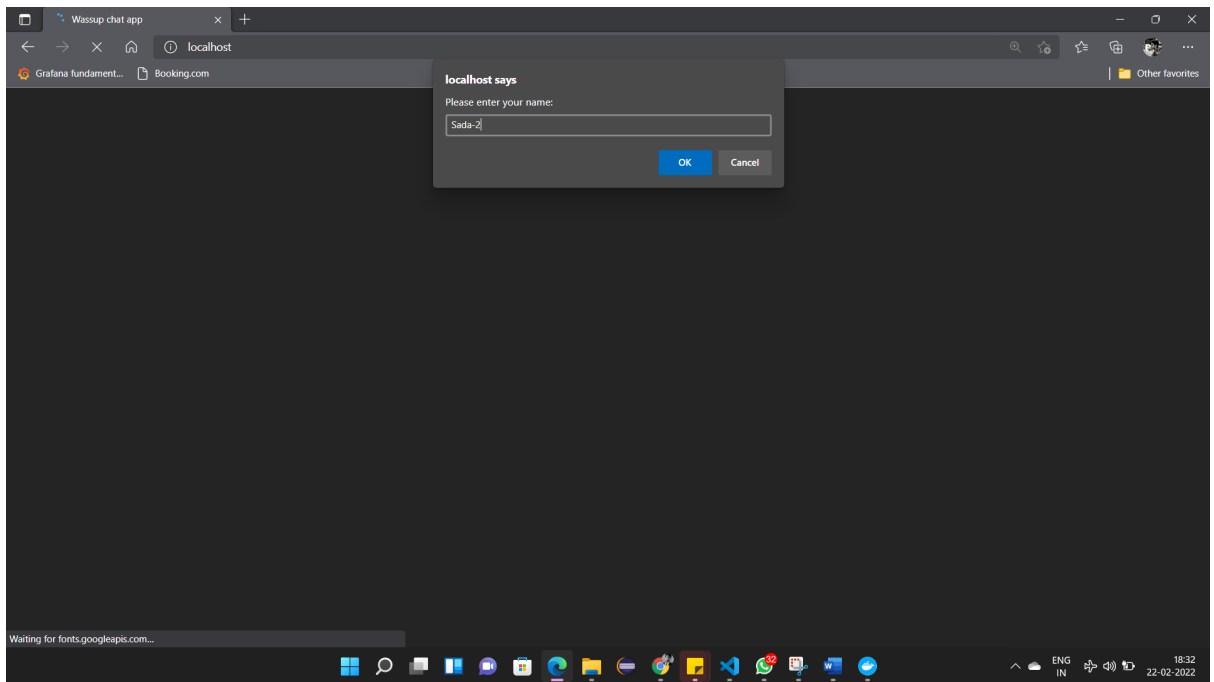
> wassup@1.0.0 start
> node server.js

Listening on port 80
```

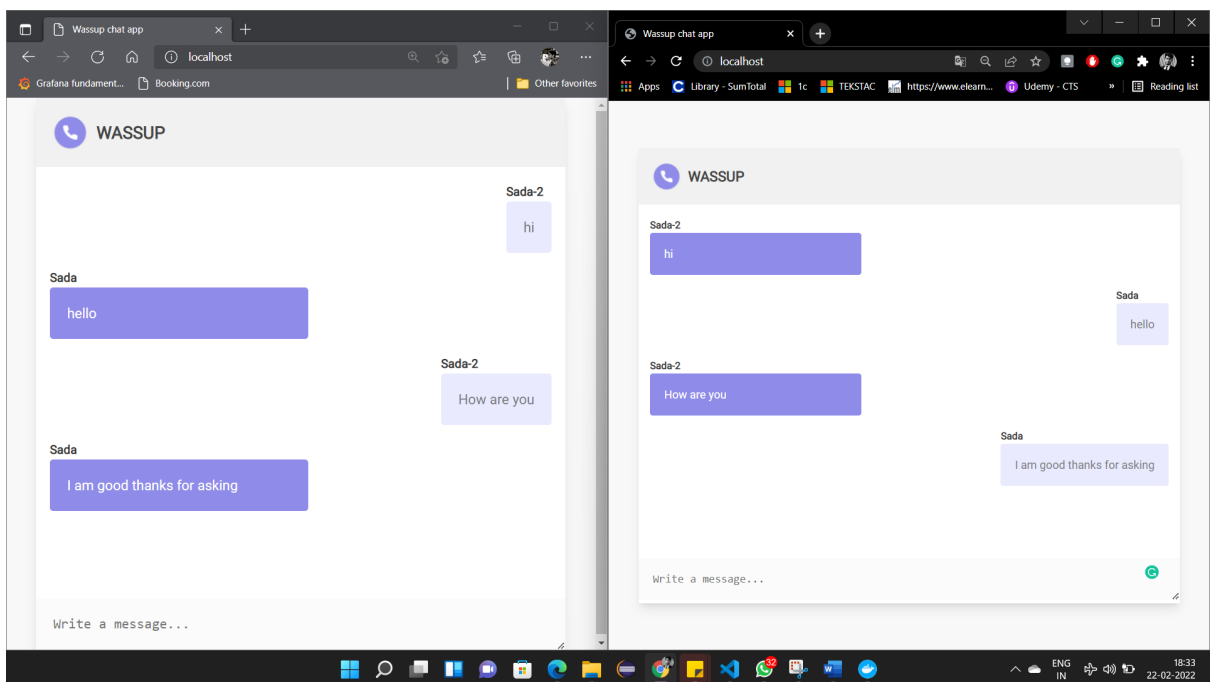
Starting the server



App running locally on the browser

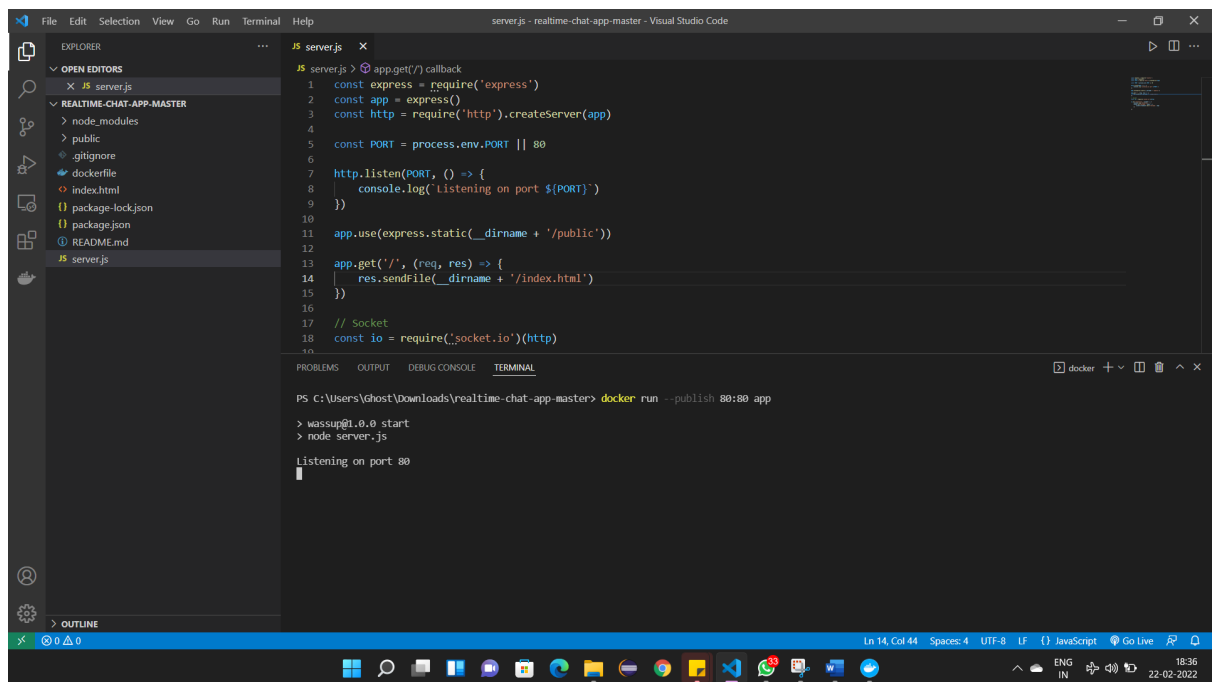


App running from another browser locally with different username



Chat application is running perfectly (local server)

- Phase 2 testing (from docker container)



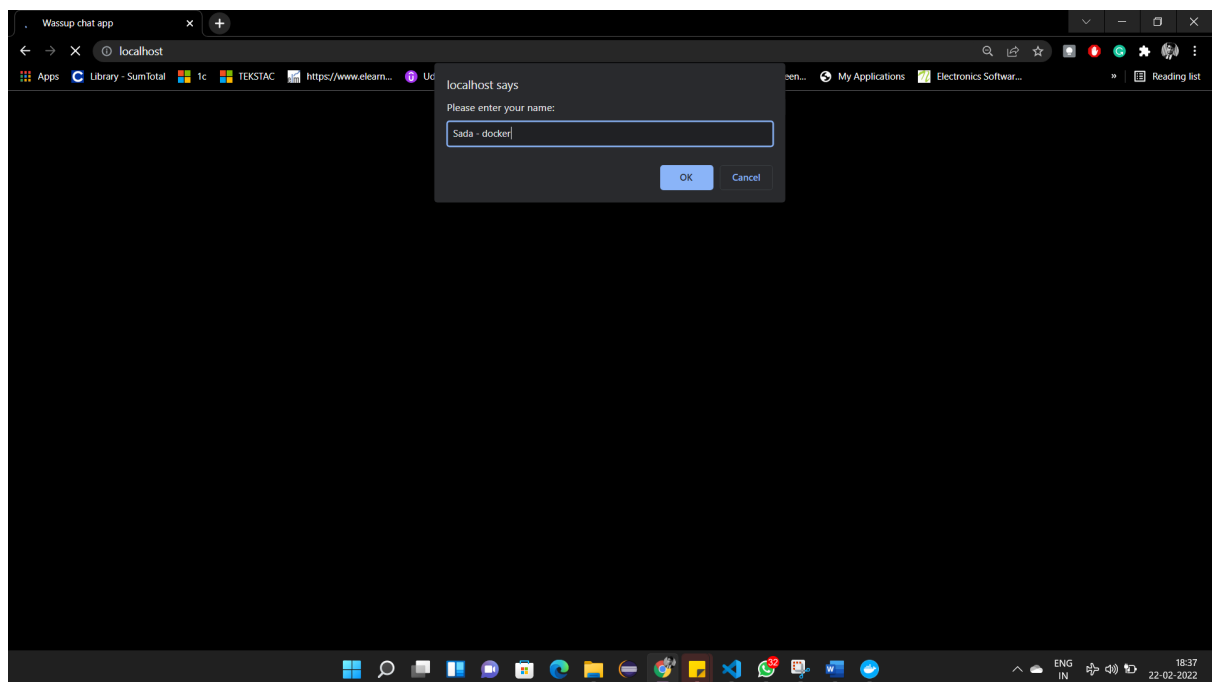
The screenshot shows the Visual Studio Code interface with a file explorer on the left displaying the project structure for 'serverjs'. The main editor shows the 'server.js' file with the following code:

```
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)
```

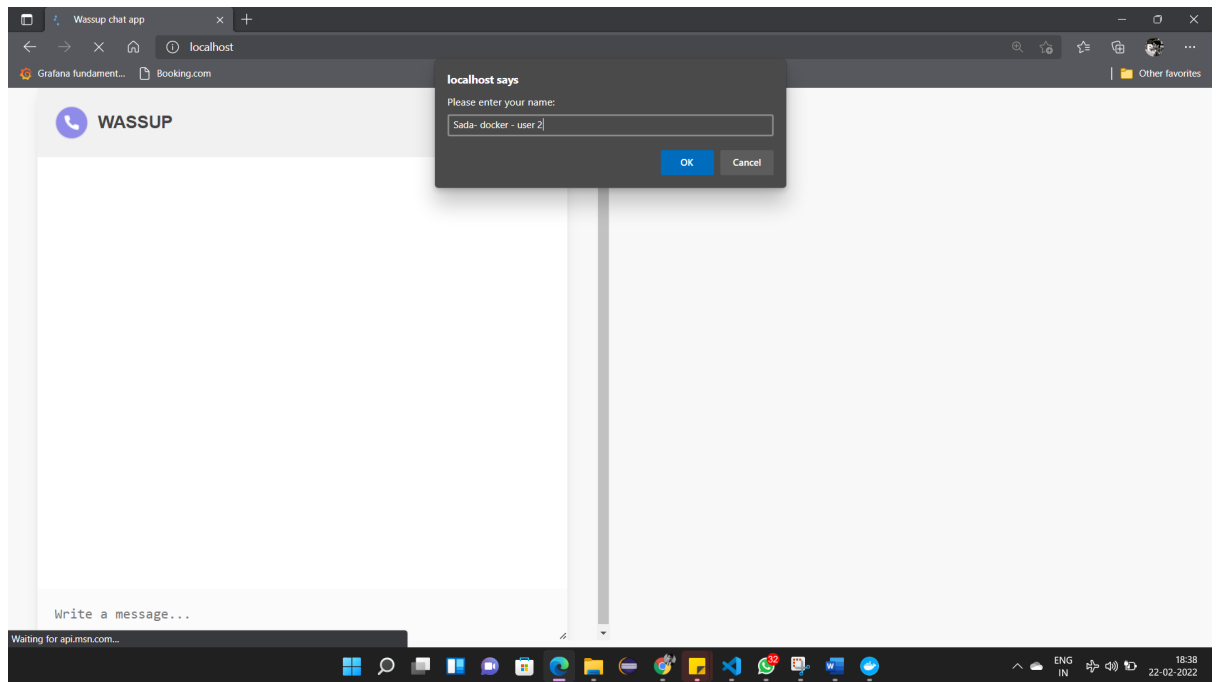
The terminal window at the bottom shows the following commands and output:

```
PS C:\Users\ghost\Downloads\realtime-chat-app-master> docker run --publish 80:80 app
> wassup@1.0.0 start
> node server.js
Listening on port 80
```

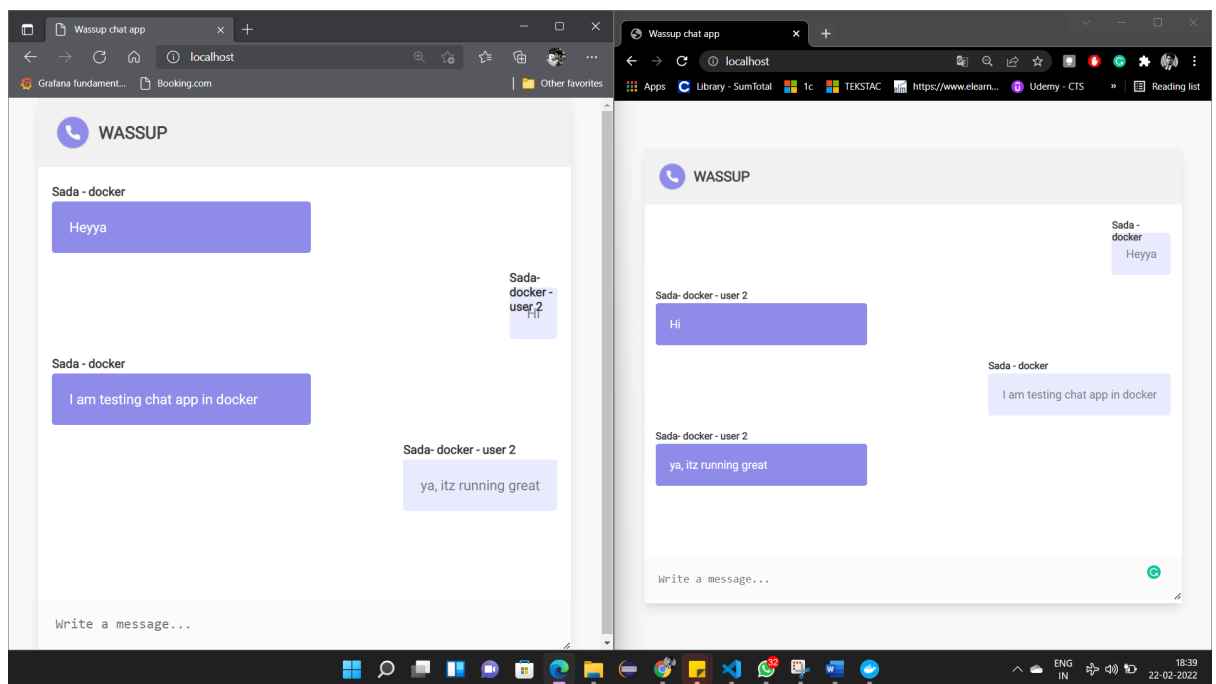
Docker run with port mapping



User 1

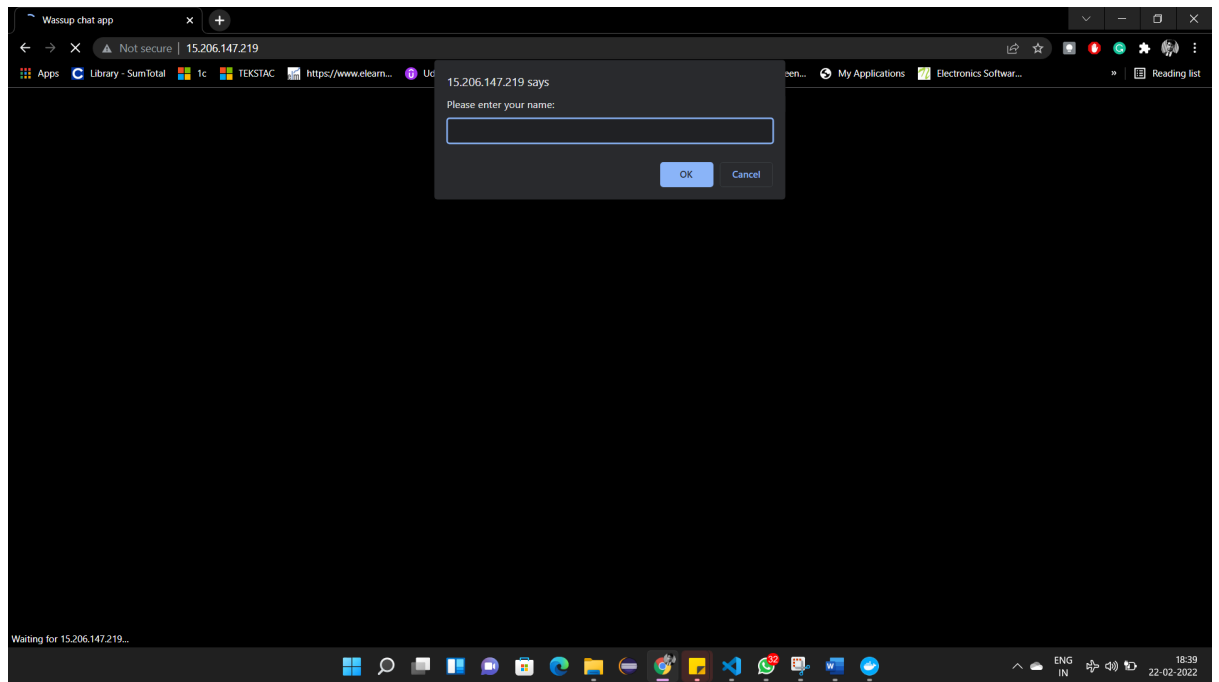


user 2 from another browser

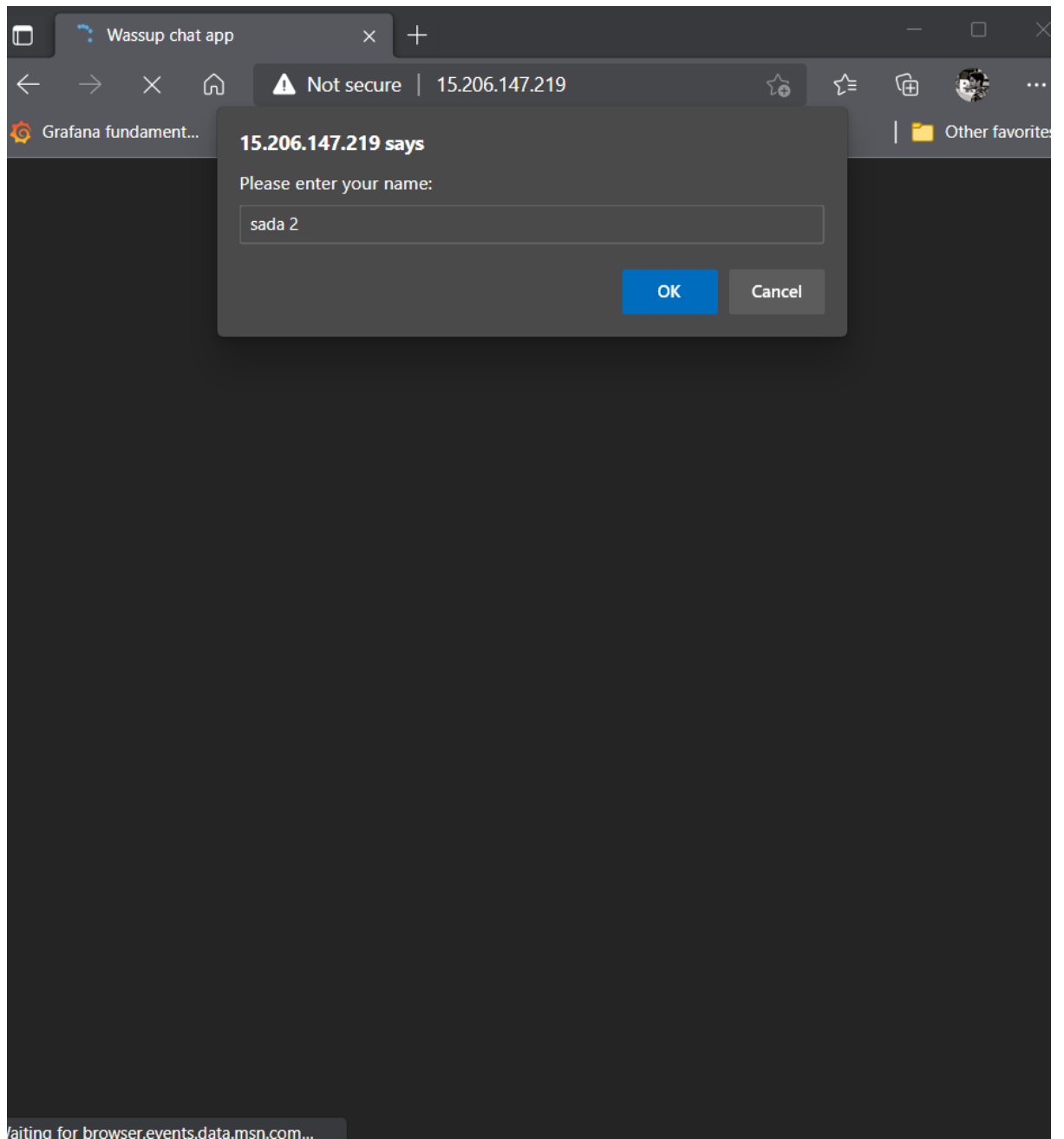


Chat app working from docker container as well

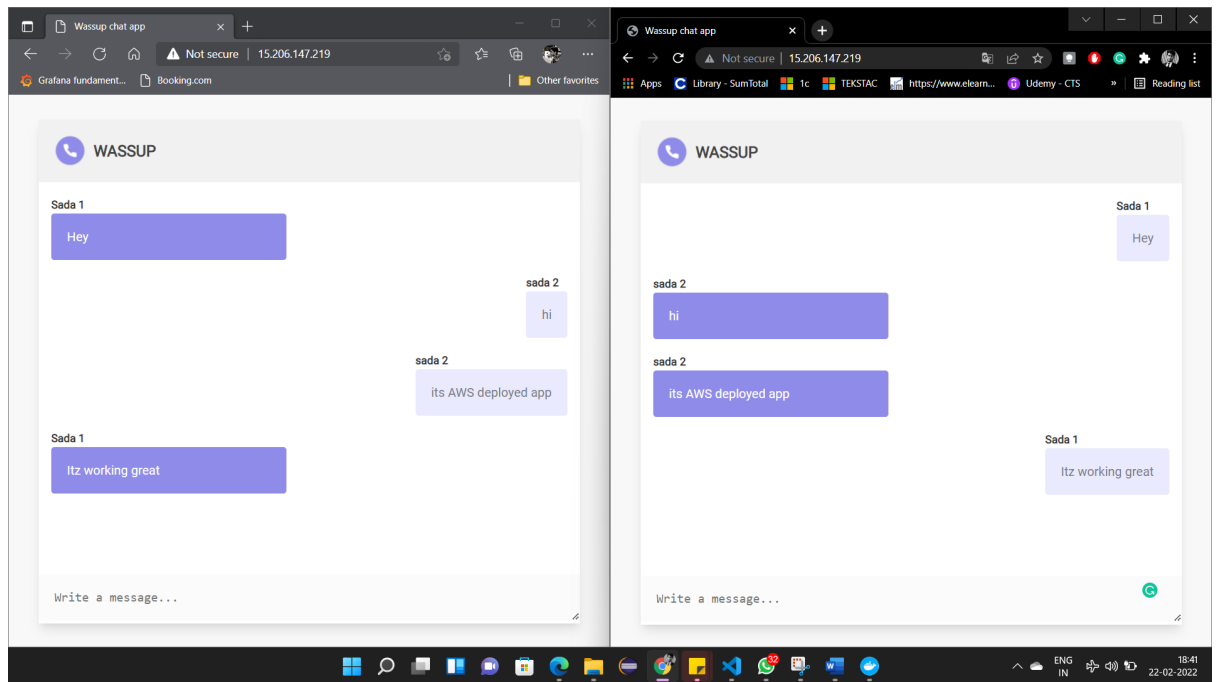
- Test phase 3
Deployed app



User 1 from one browser



User from browser 2



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