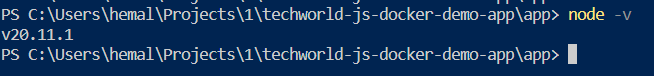
Cloned the appcode files in VS code

<https://gitlab.com/nanuchi/developing-with-docker.git>

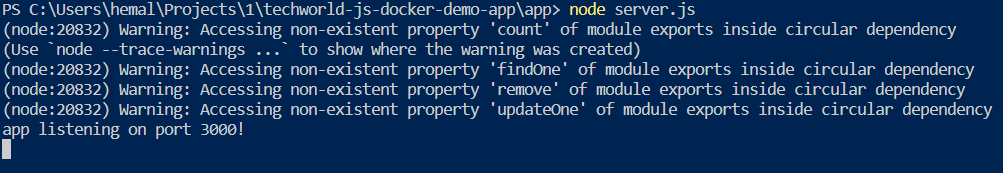
Installed node latest 🡪 <https://nodejs.org/en>



Tried running node server.js 🡪 got error “express module was missing”

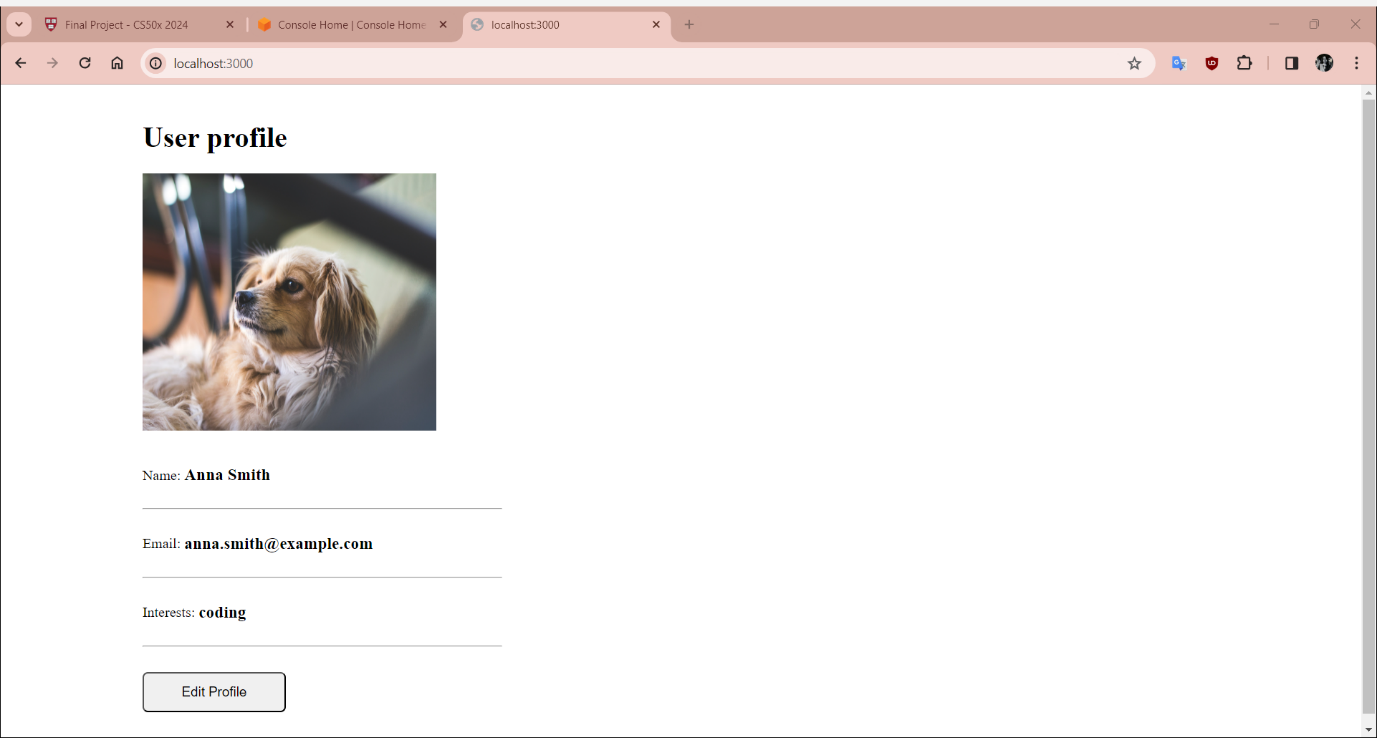
Run 🡪 npm install express

Run 🡪 node server.js



In browser : not able to see UI portion for localhost:3000 ?

Working now

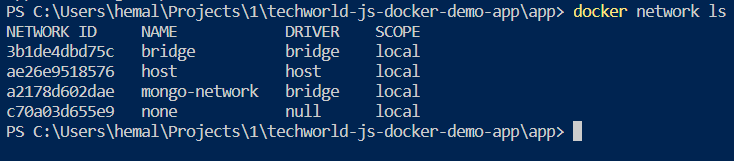
Run 🡪 Docker pull mongo

Run 🡪 Docker pull mongo-express

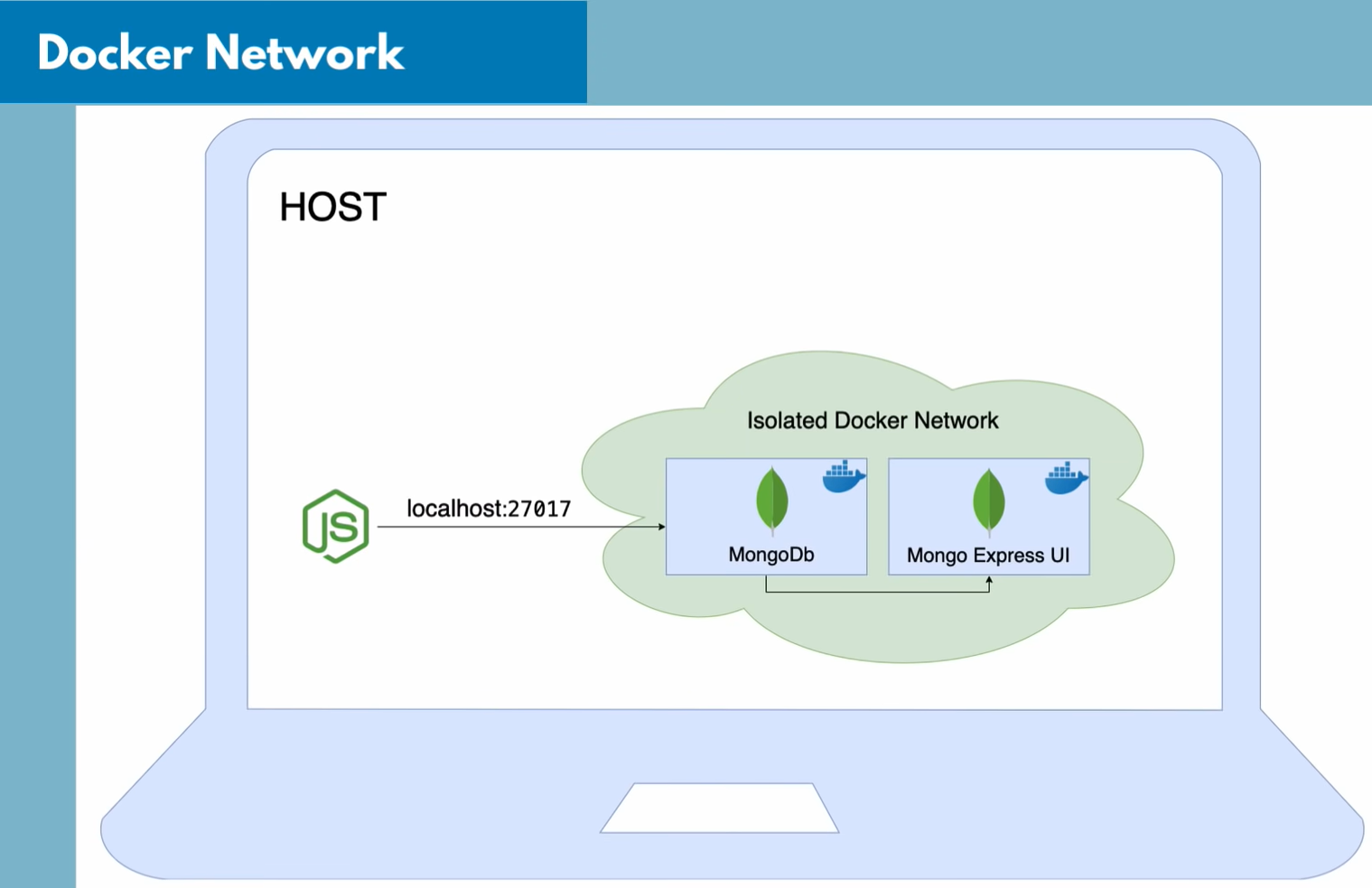
Gets copies of these images for future use

docker network create mongo-network

docker network ls



Create MONGO & Mongo-express containers: they can connect with each other just using the container names since both of them are in the same isolated docker network



docker run -p 27017:27017 -d -e MONGO\_INITDB\_ROOT\_USERNAME=admin -e MONGO\_INITDB\_ROOT\_PASSWORD=secret --name mongodb --net mongo-network mongo

🡪 open port 27017 on local host and bind to mongo container port 27017 (default port of mongodb is 27017)

🡪 -d run the container in detached mode – allows to use terminal while the docker container is running in background

🡪 -e MONGO\_INITDB\_ROOT\_USERNAME

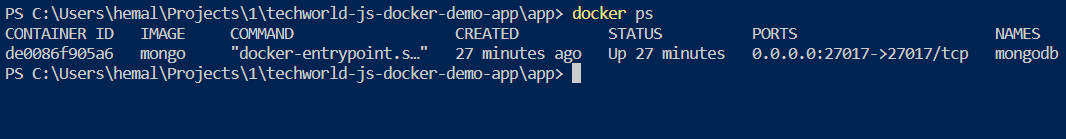
-e MONGO\_INITDB\_ROOT\_PASSWORD

Define root username & password for mongo at startup of the container

🡪 --name – define container’s custom name

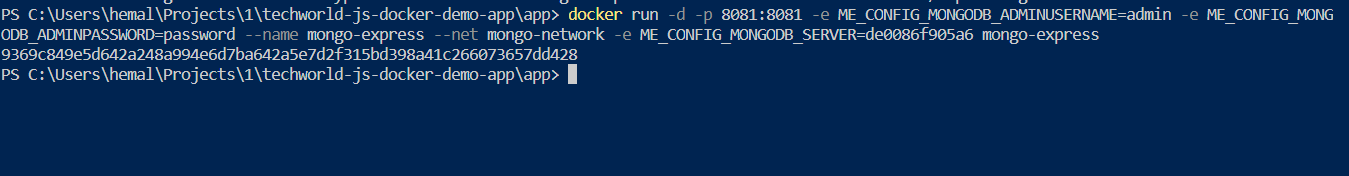
🡪 -- net launch container in the network created earlier

If launched successfully – we get a containerID

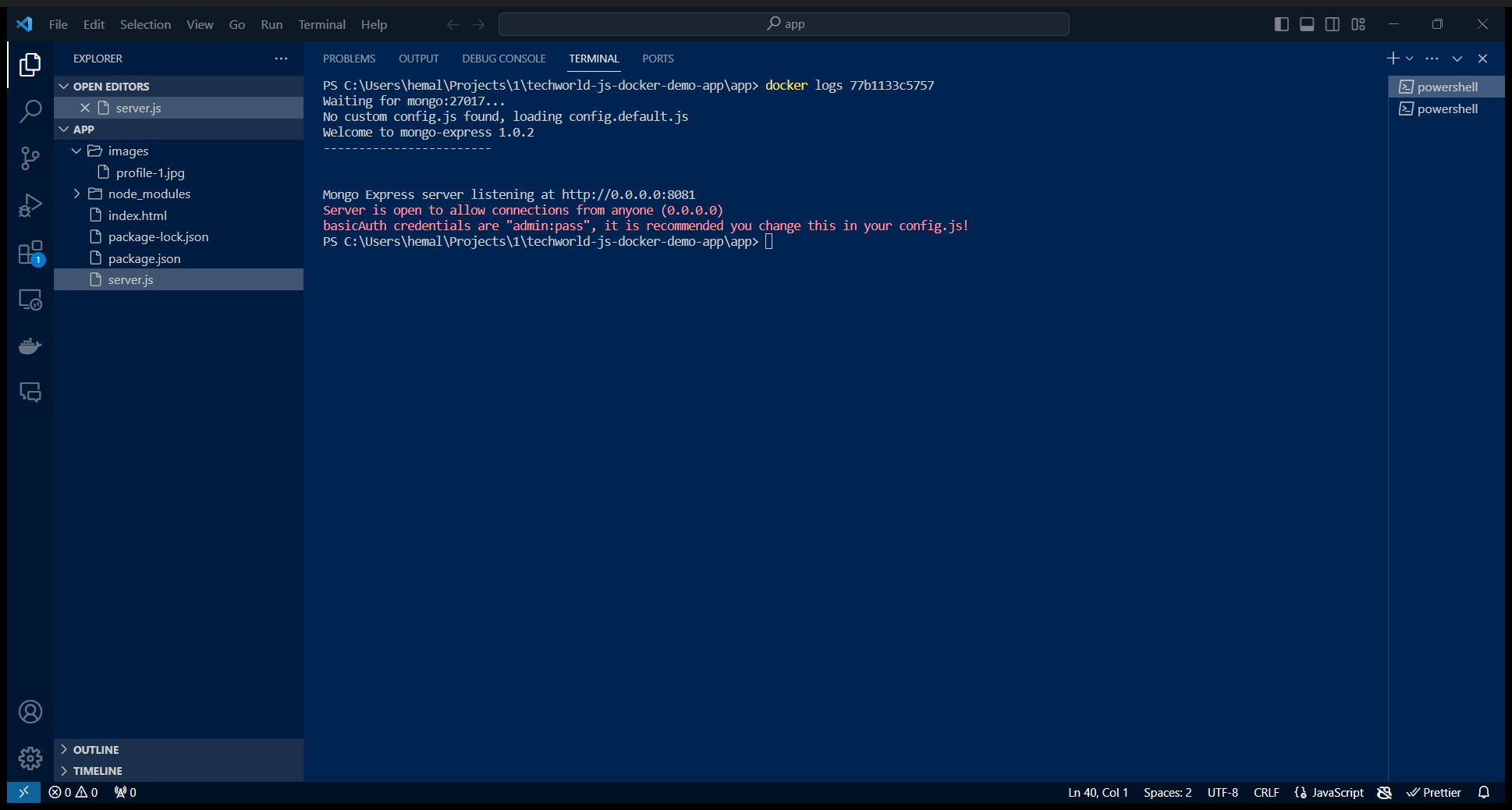


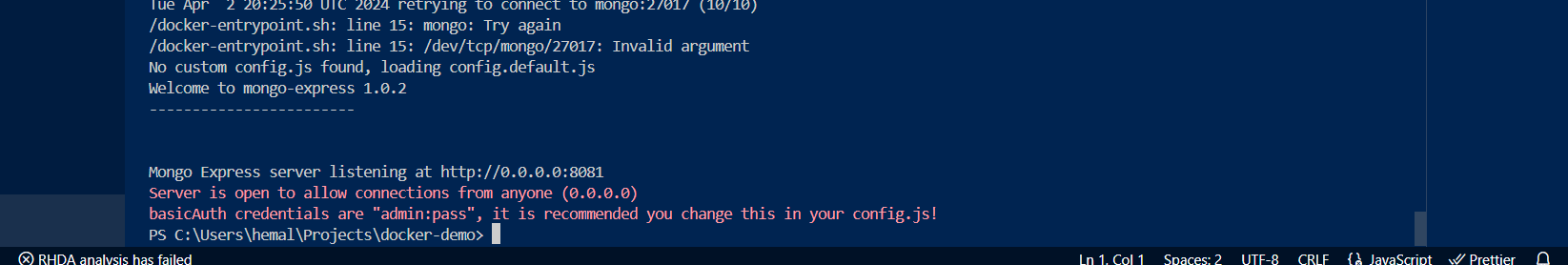
🡪 docker logs containerID

🡪 docker run -d -p 8081:8081 -e ME\_CONFIG\_MONGODB\_ADMINUSERNAME=admin -e ME\_CONFIG\_MONGODB\_ADMINPASSWORD=secret --name mongo-express --net mongo-network -e ME\_CONFIG\_MONGODB\_SERVER=mongodb mongo-express



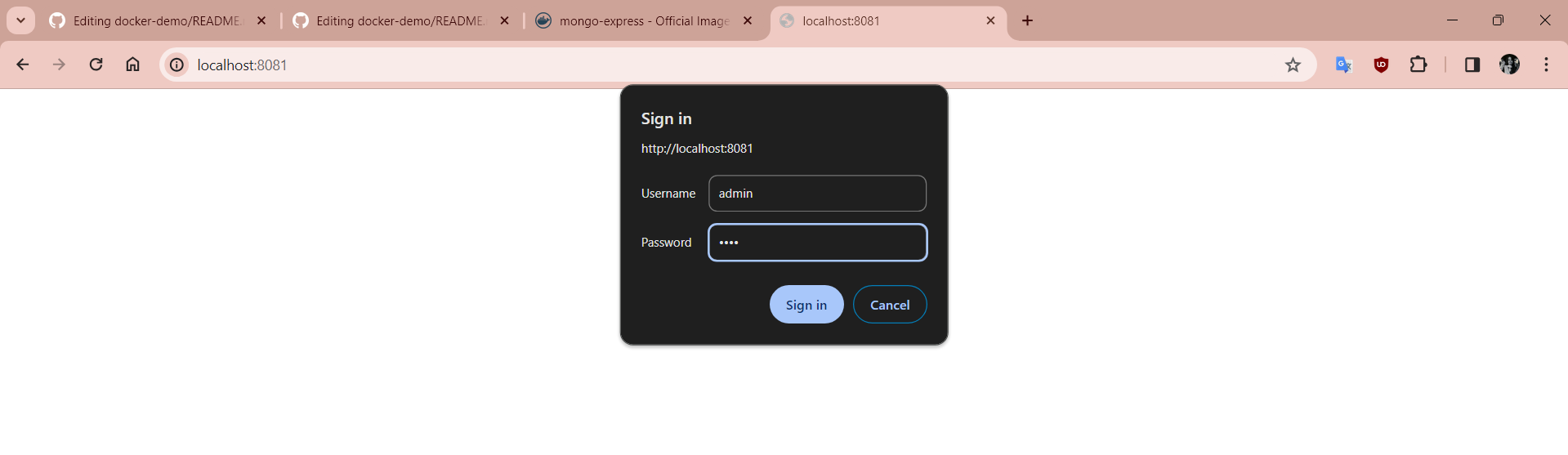
* Open port 8081 on localhost and bind to mongo-express default port (8081)
* Give mongo username & password so that express can connect to mongo db
* Give mongo containerID as env variable: mongoServer

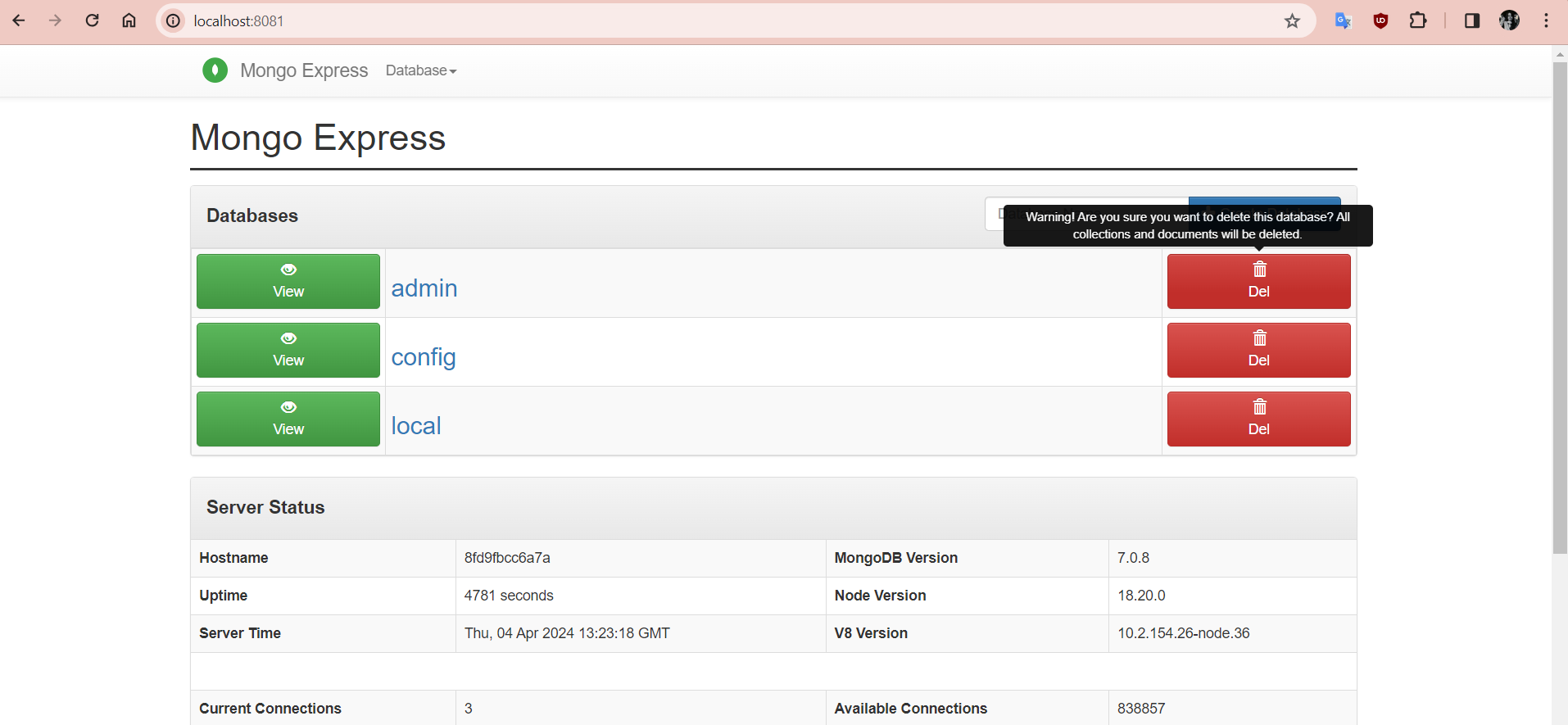




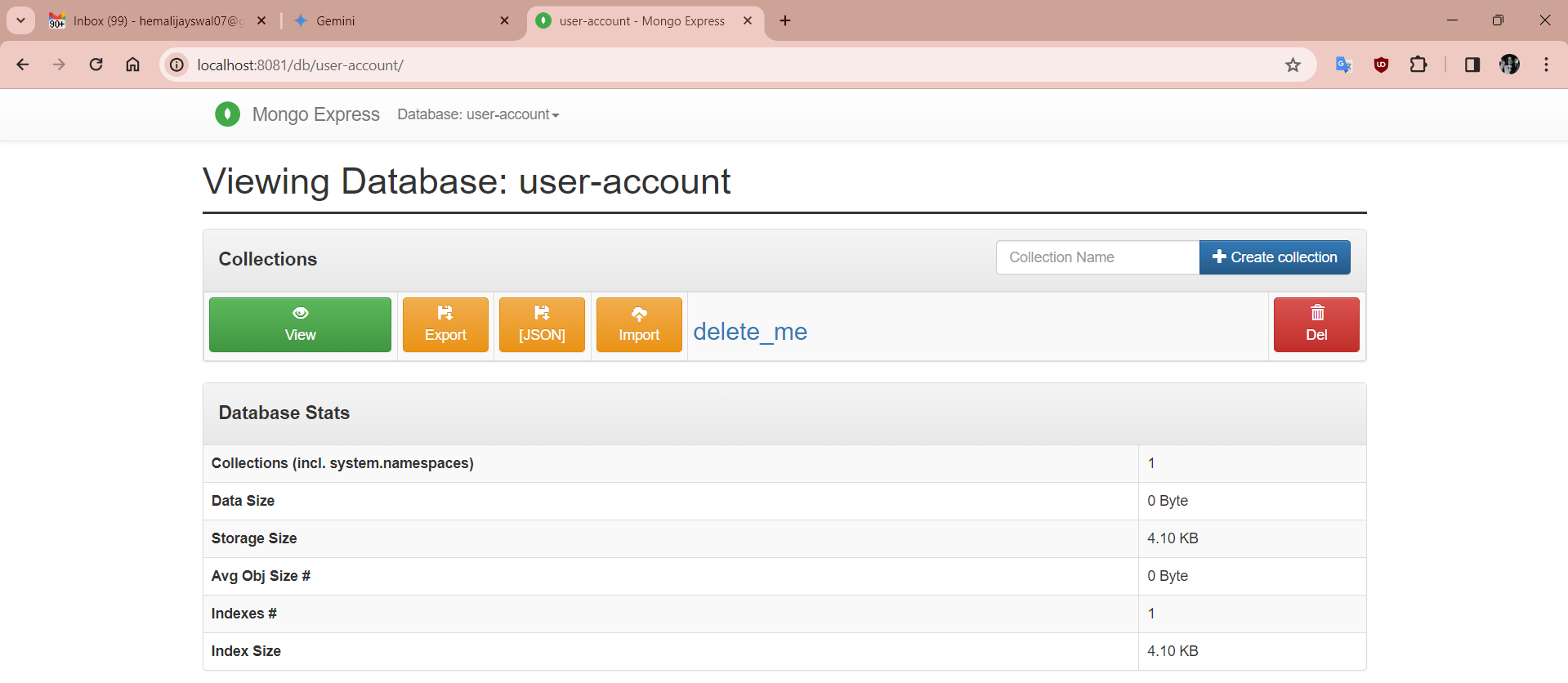
In red font we can see the basic auth credentials => admin and pass

Giving the same in localhost:8081 will launch mongo-express UI



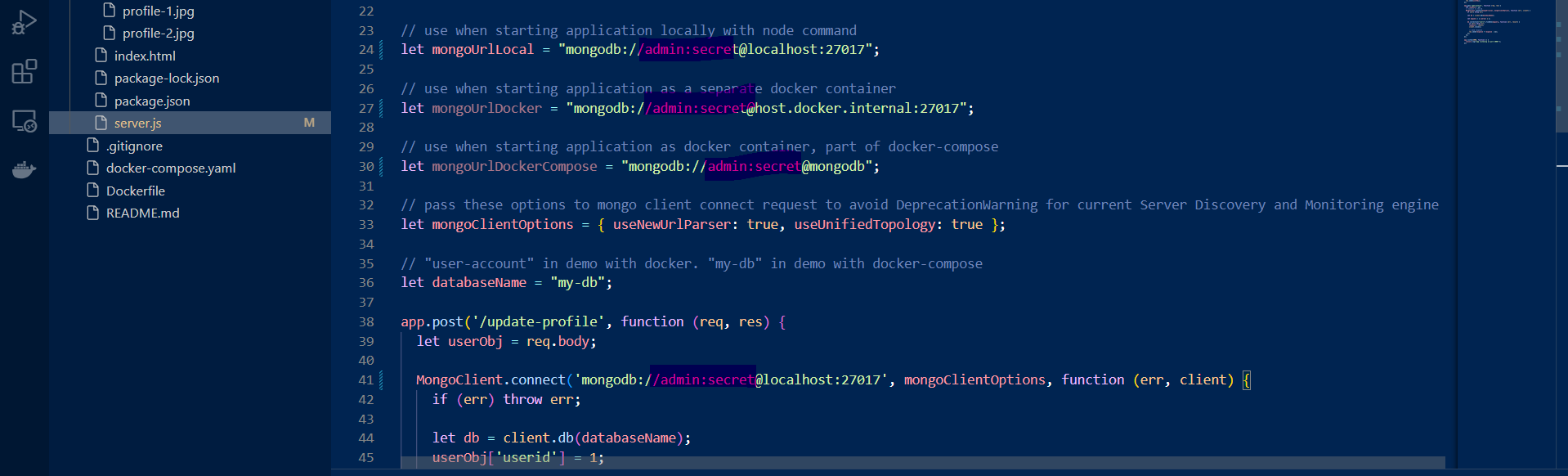


Created new db from ui or can also done with userint db as env variable in startup command for docker container

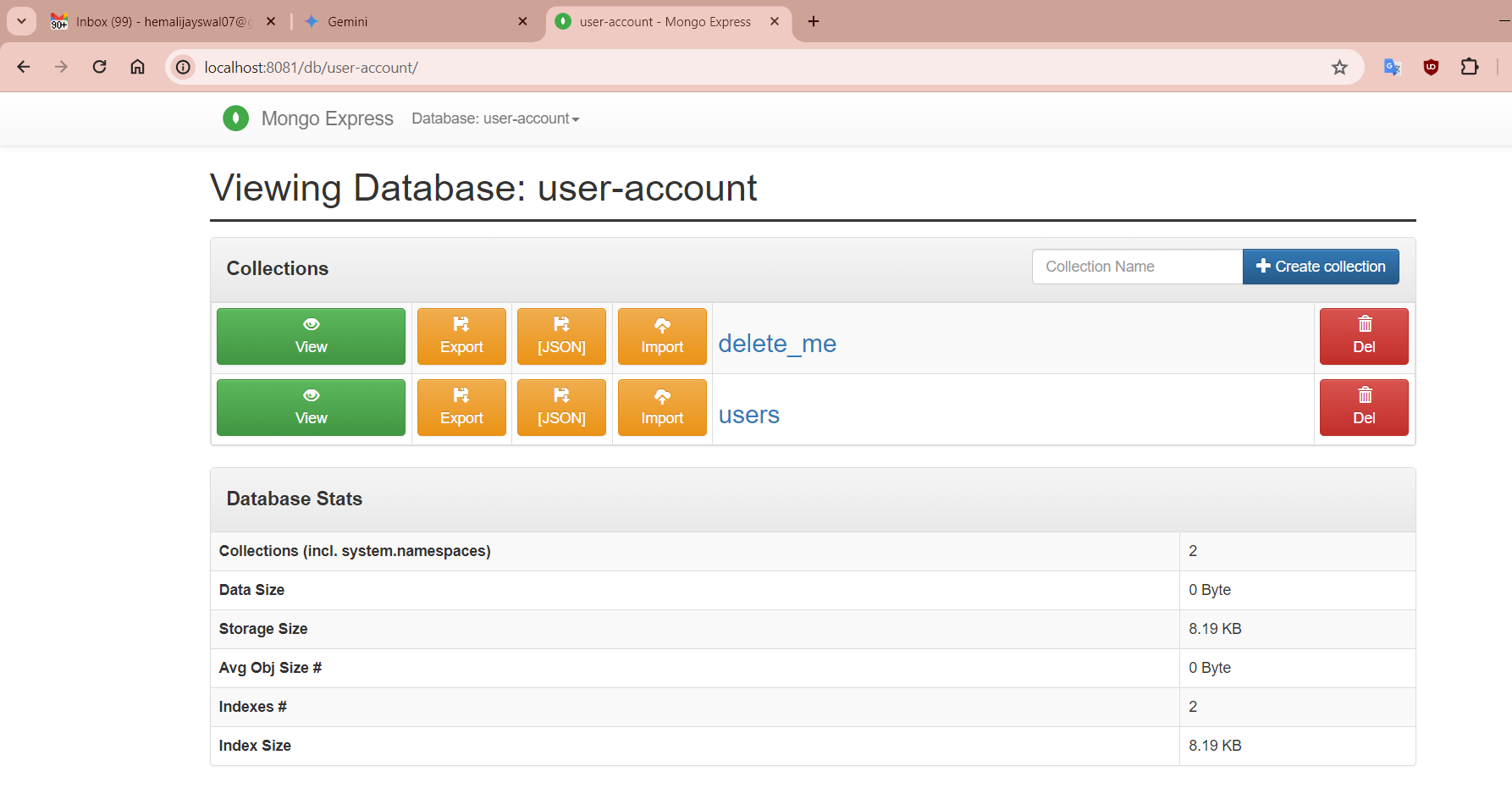


Now, we need to connect to this database by giving username and password from the previous command in the code base (not a good practice)

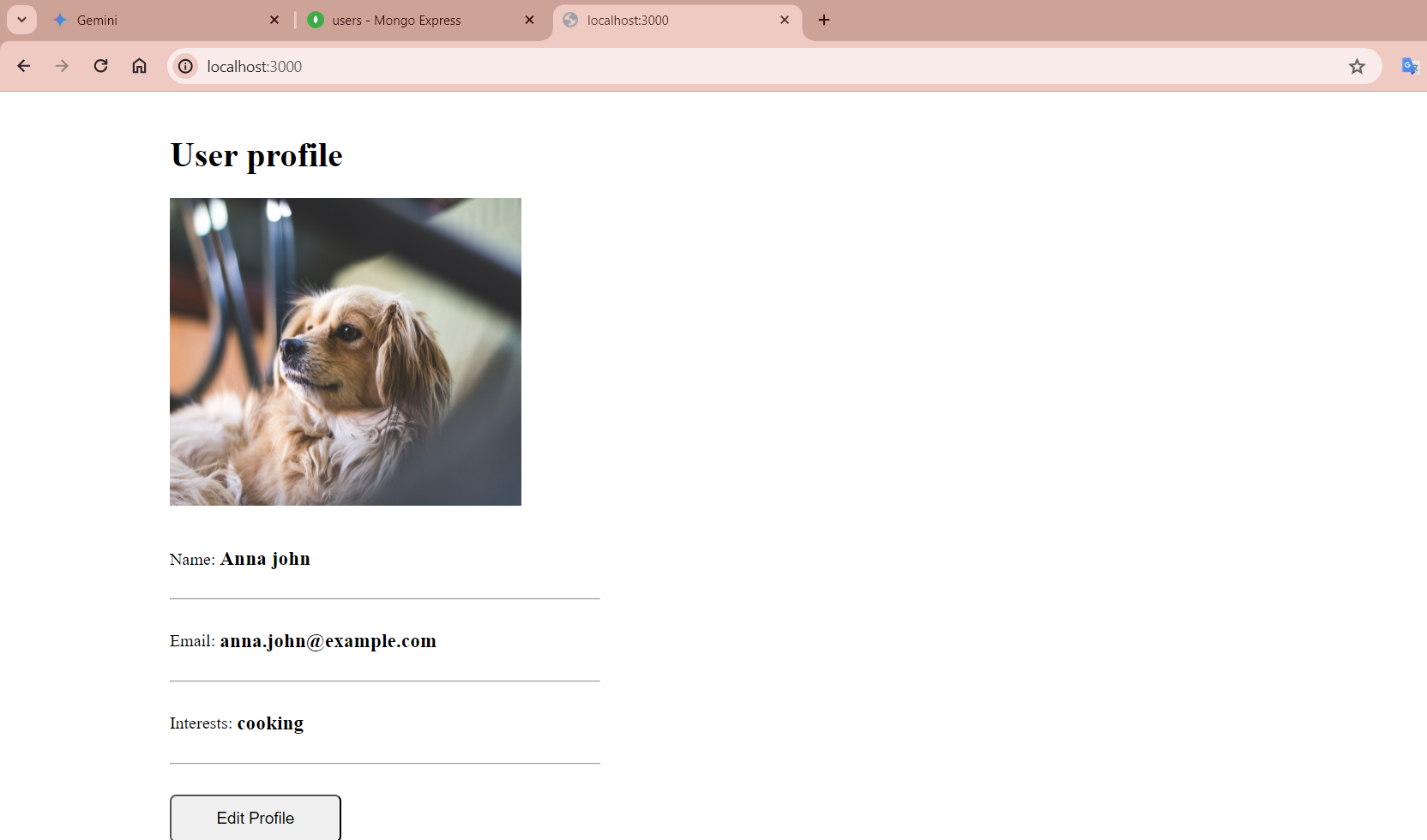
4 places in code

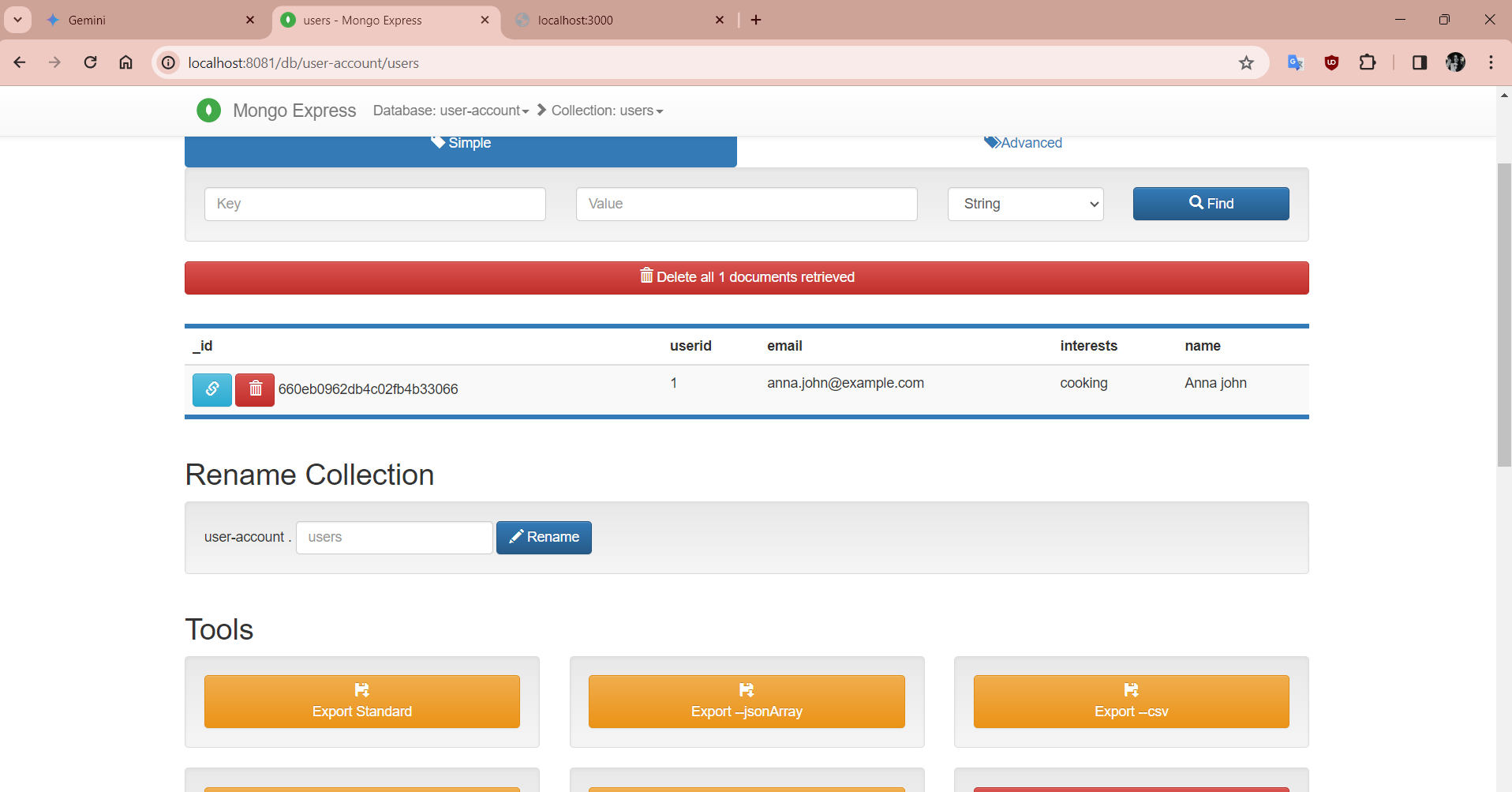


Create a collection in mongo-express UI: users



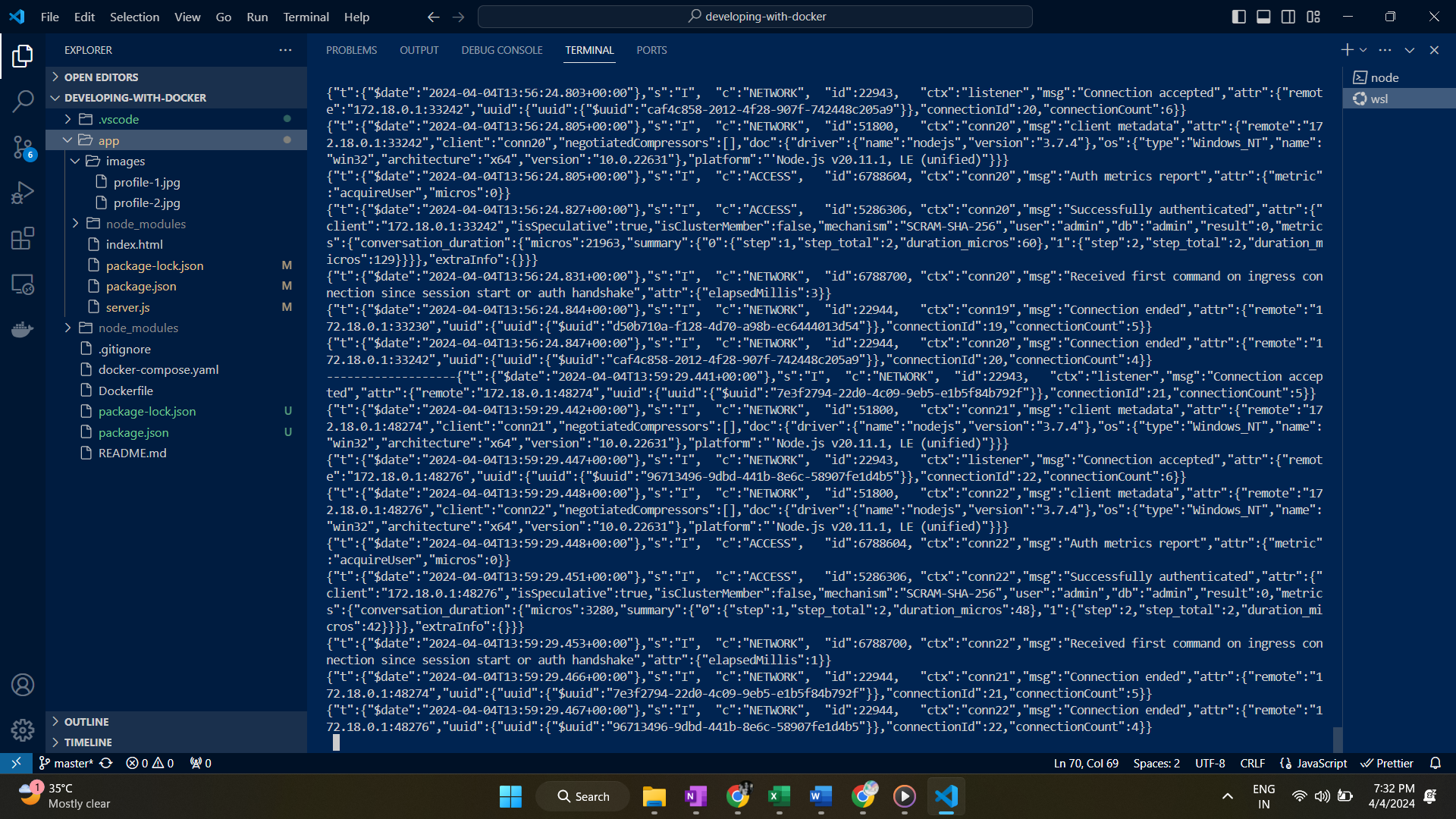
Now try to add user from UI the same should be visible in the mongo-express DB





Docker logs mongo | tail 🡺 to view the last logs

Docker logs mongo -f 🡺 to stream the logs



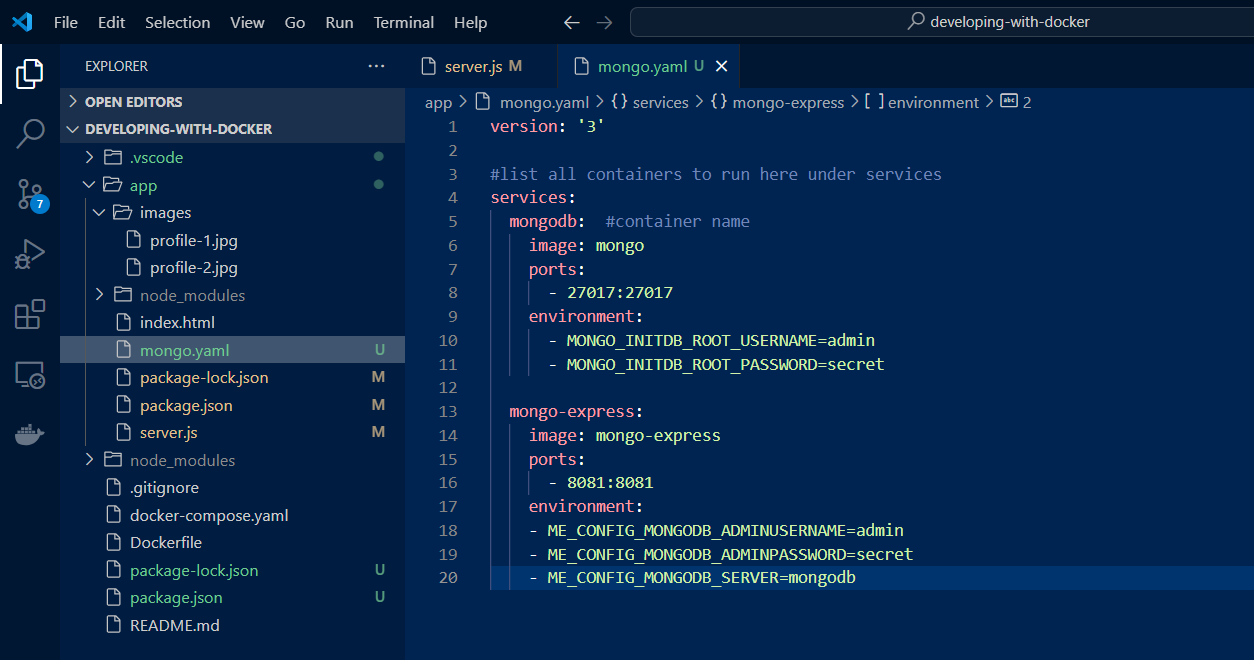
So, now we have a fully functional app serving user profile in the frontend using javascript and nodejs in backend up and running

Here, we have given many docker commands to get this app running 🡪 too much manual work!!

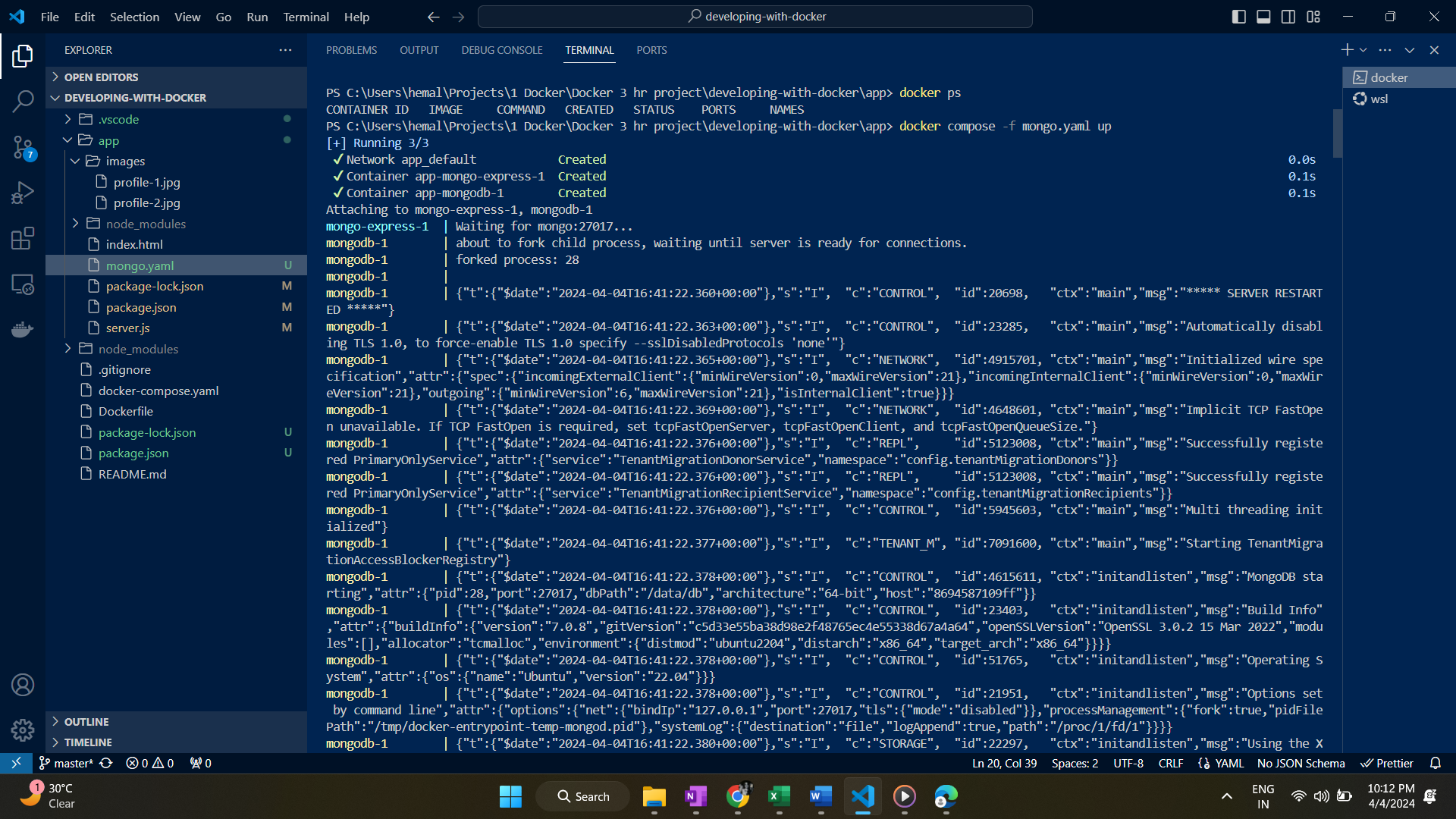
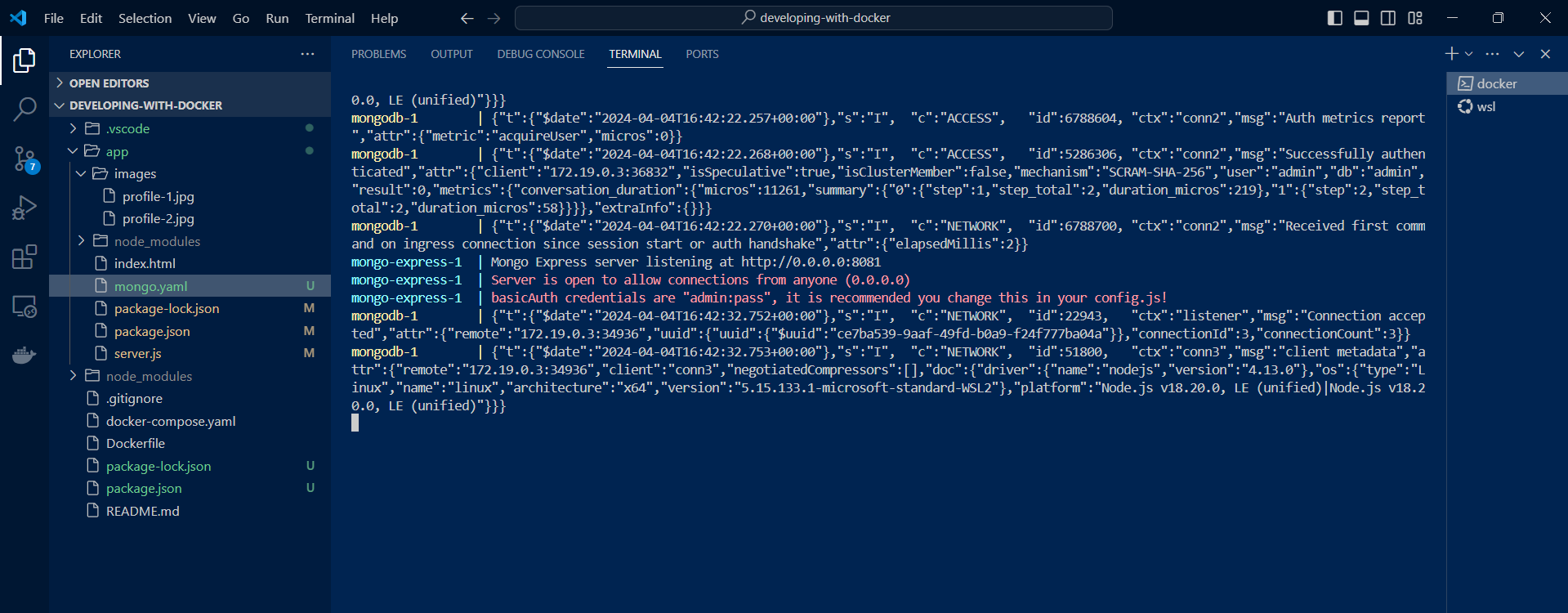
We can automate this using docker compose file

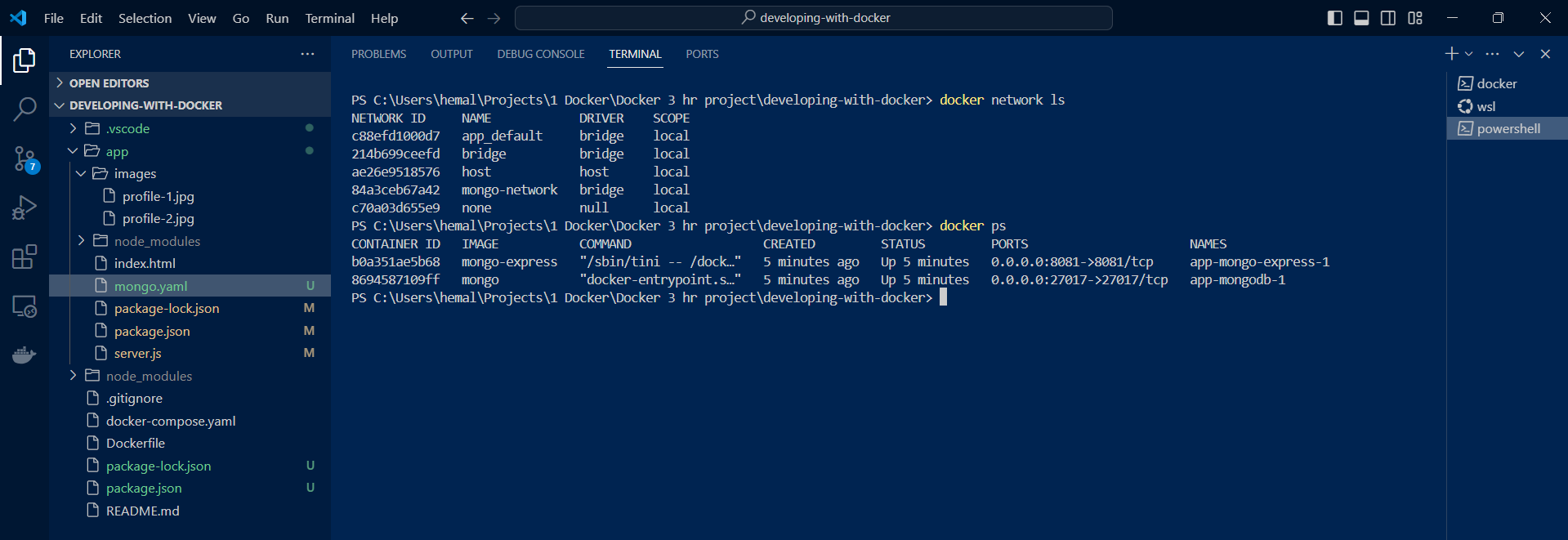
If we have bunch of containers to work with, we probably want to automate the deployment and starting of applications

**Docker compose**: allows to running multiple docker containers with all the configuration easily than with individually passing docker run commands



Running this file from terminal : docker compose -f mongo.yaml up

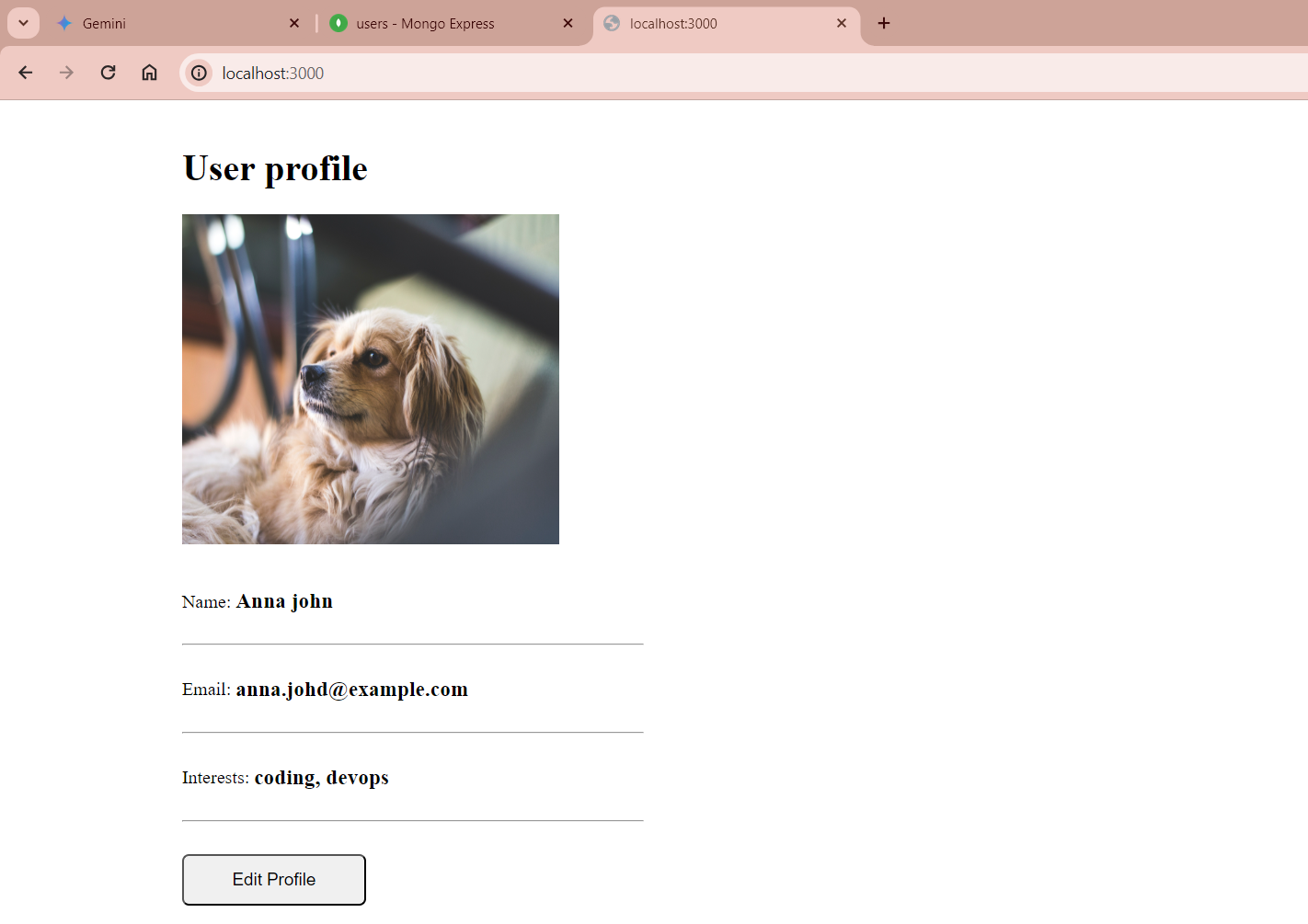
 

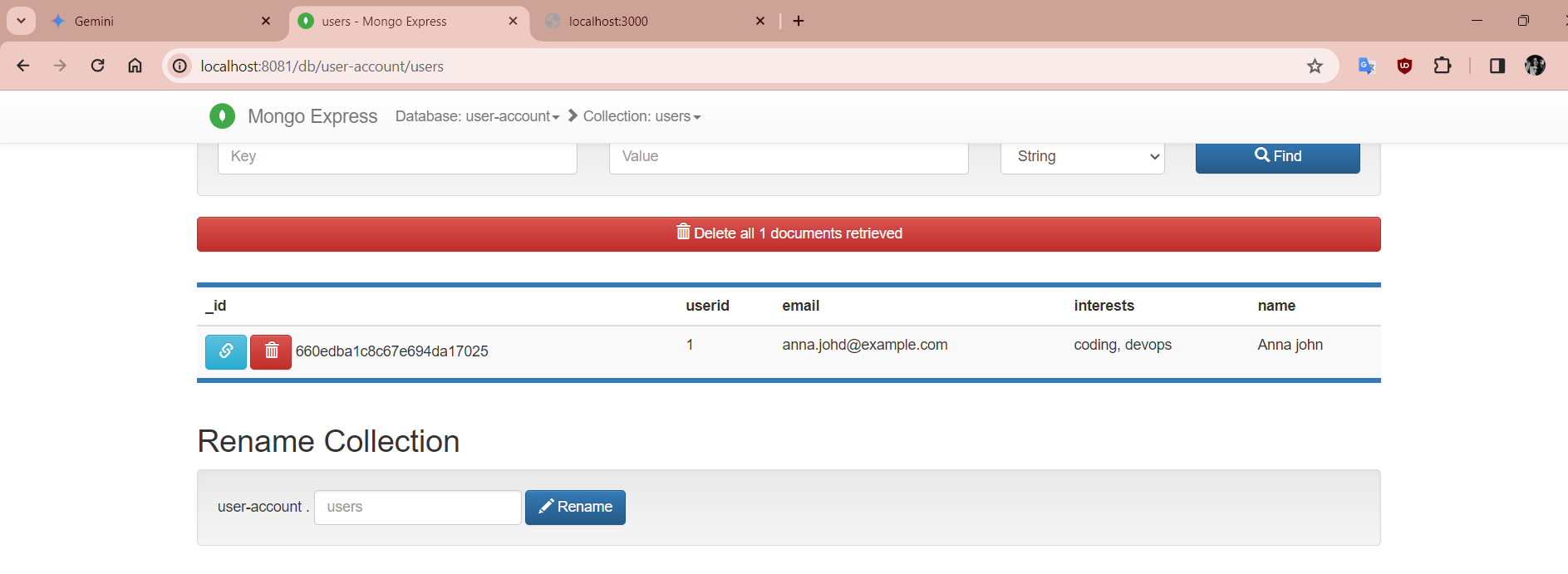


Since both containers are started at once, logs are mixed for both, we can add waiting logic in yaml file when one container depends on another as here

Also the previously created db from mongo-express ui is gone because we don’t have data persistence so when containers are restarted all data is lost. This can be prevented using data volumes

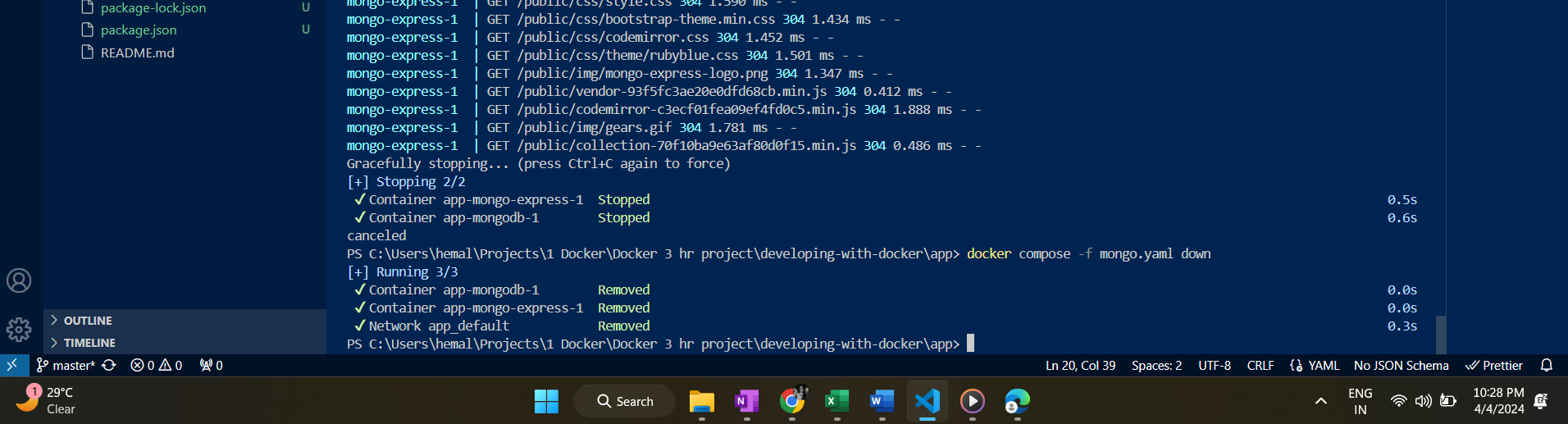
starting the app again now





Stopping the containers: docker compose -f mongo.yaml down

The network created is also deleted

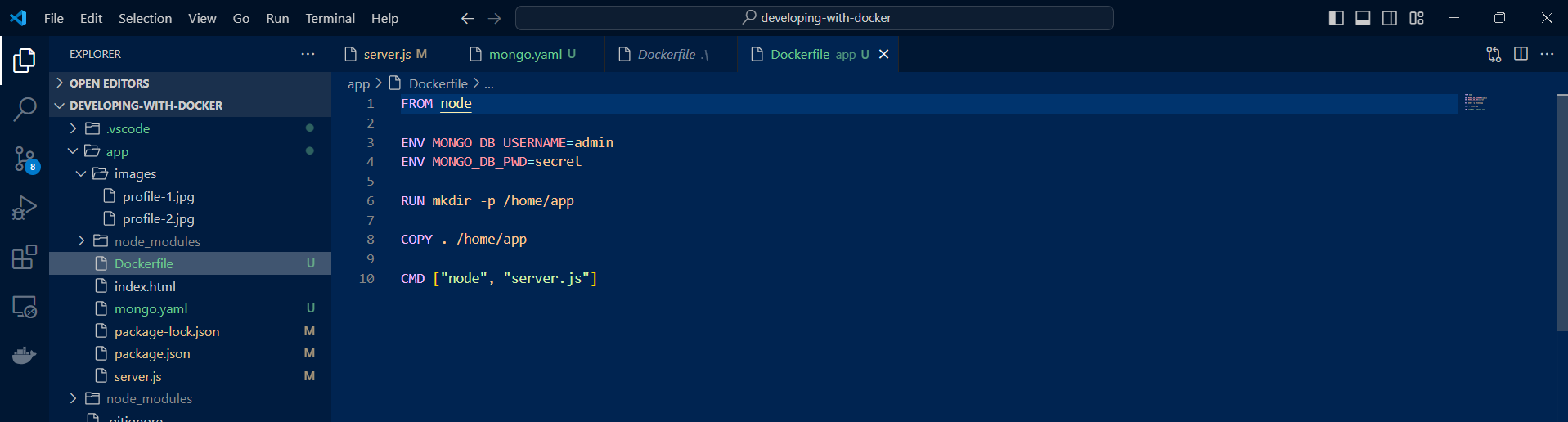


Creating Dockerfile & building the image:

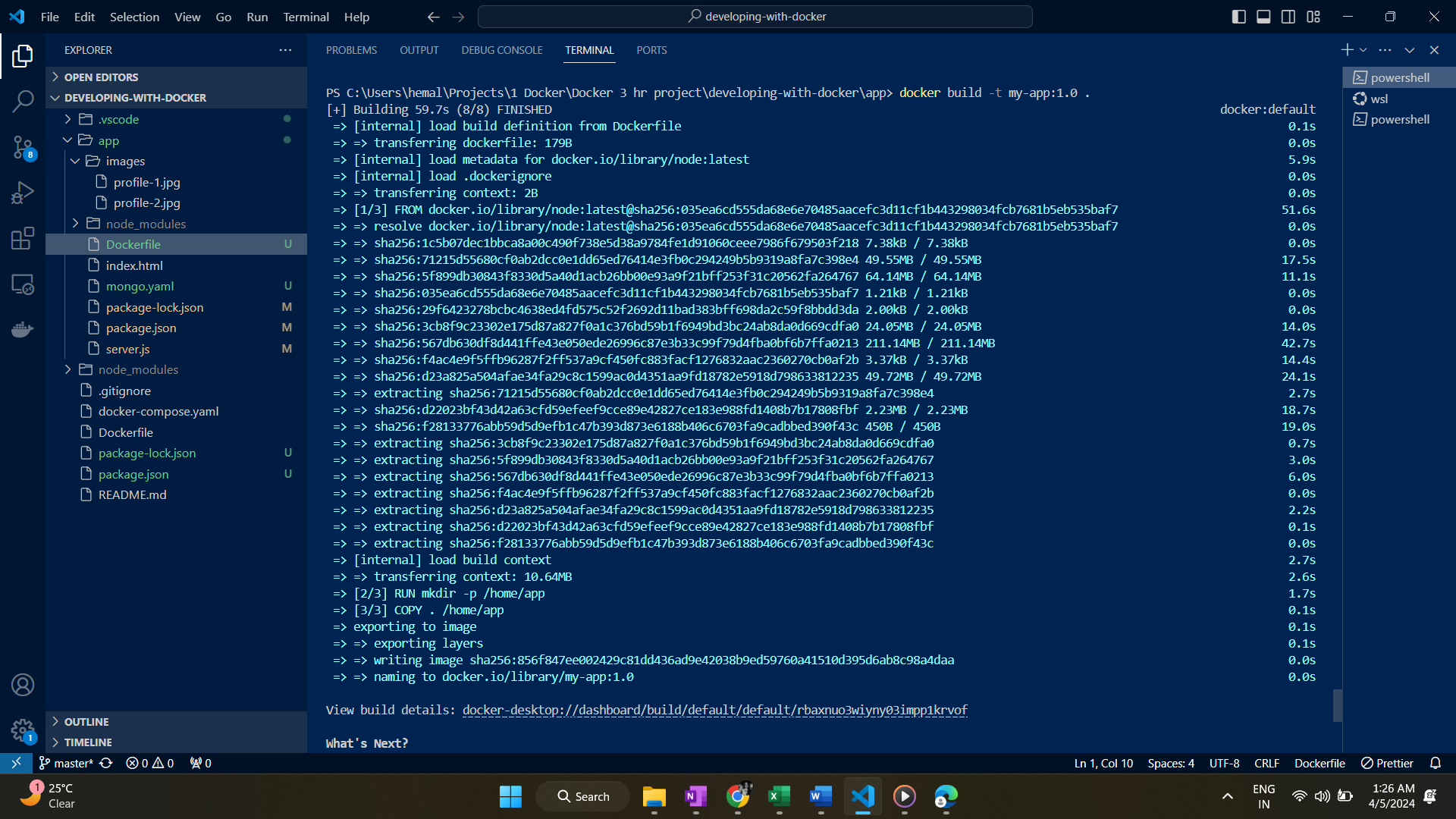
To deploy this app on dev, test or prod server, we need to package this app along with the dependencies into a single artifact 🡺 Docker file

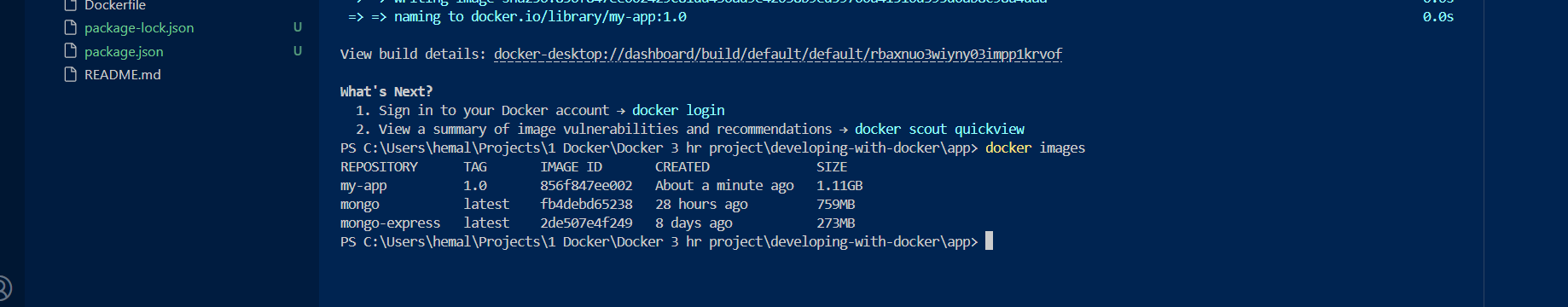
Now we are going to simulate what Jenkins does locally:

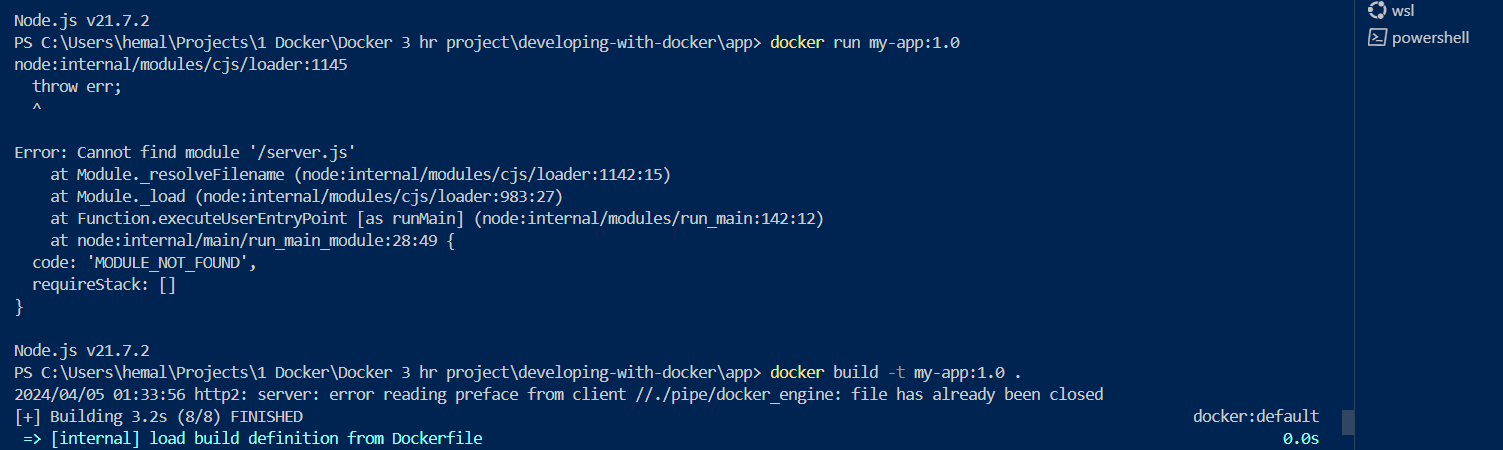
When we commit changes to git 🡪 a CI that runs (Jenkins), that builds the app -> packages app to a docker image 🡺 pushes to registry



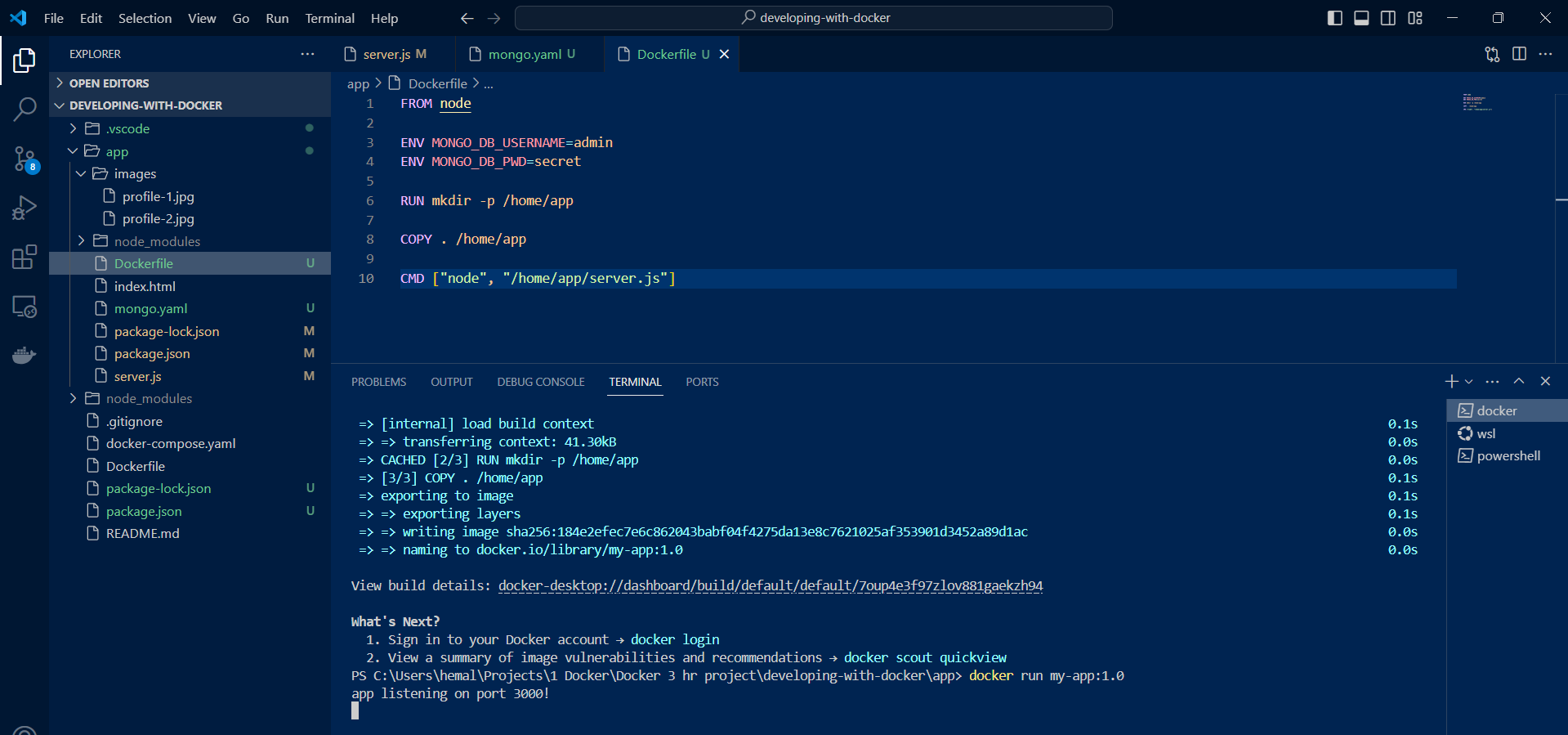
Run: docker build -t imageName:tag location-of-docker-file

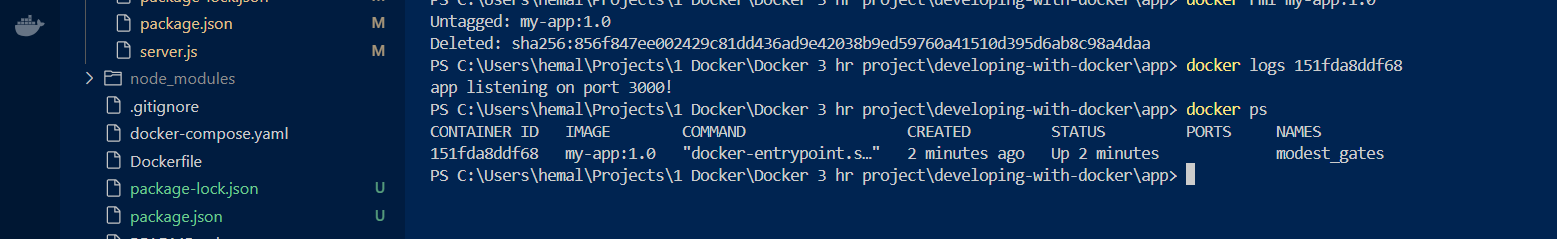


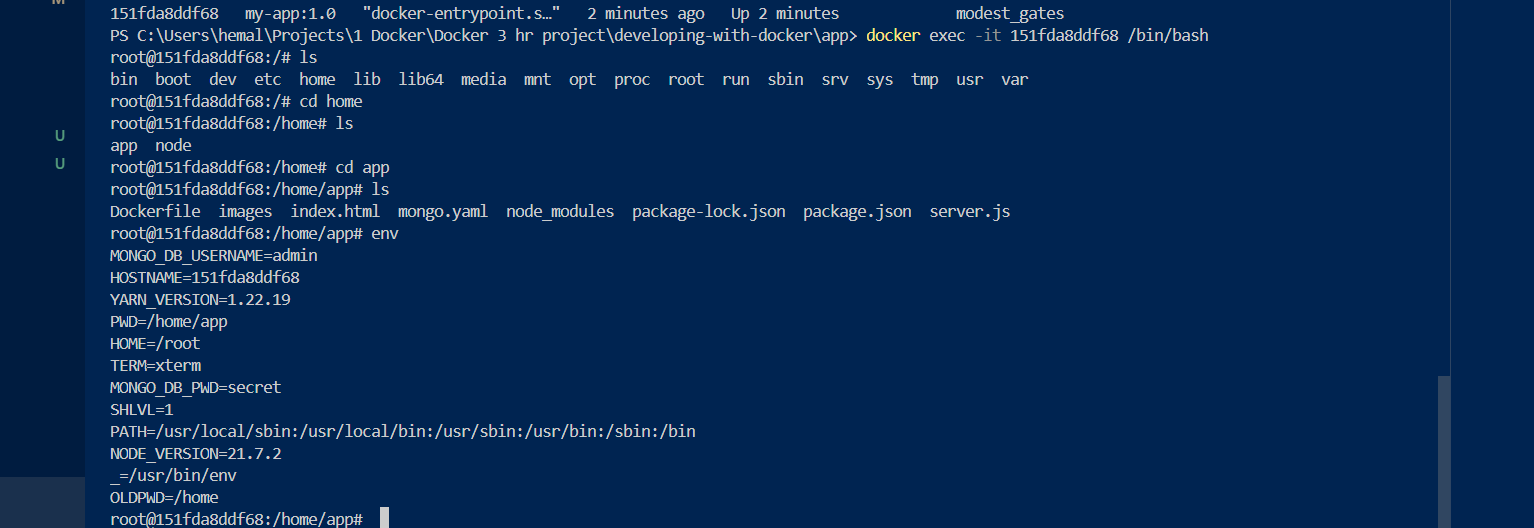




Make sure to give proper path in cmd path to find the app source file to start the app







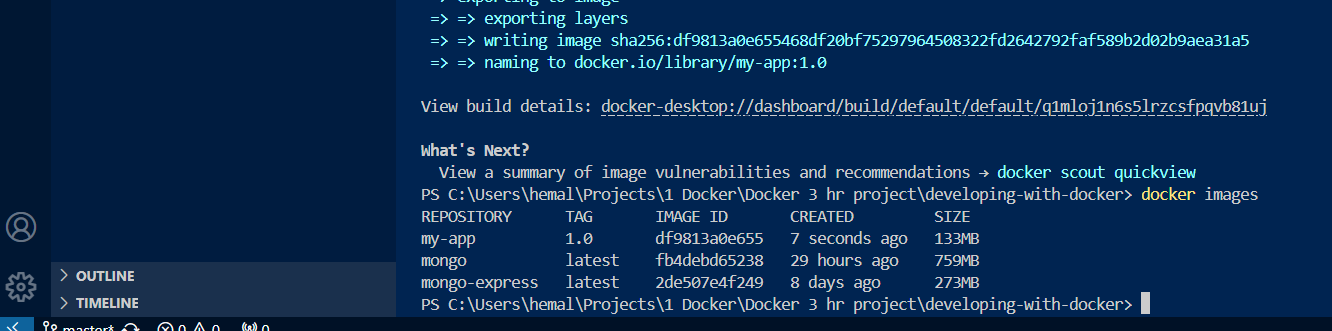
All the files in app folder got copied in the container

Pushing image to AMAZON ECR – private registry

* Login to amazon console and head to Amazon ECR

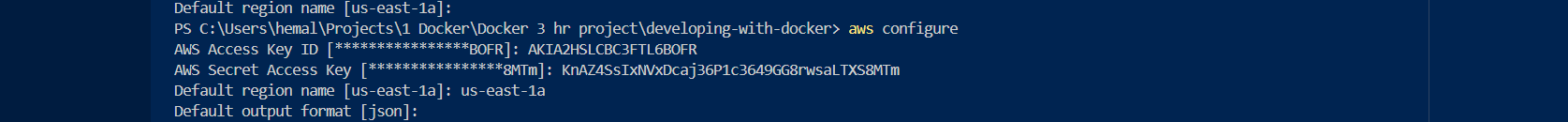
In ECR, we have to create a repo per image, unlike docker hub where we can store multiple images in a repo

In the repo, we can store different versions of the same image



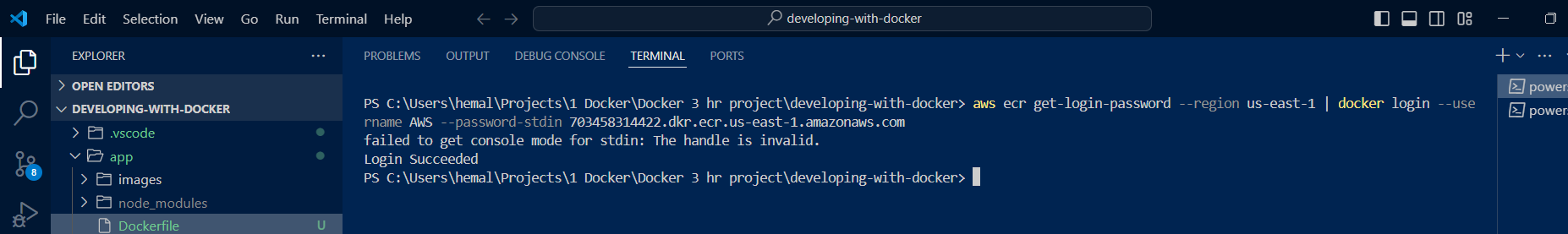
We want to push this image to ECR

* First authenticate to your aws account using aws configure command



Retrieve an authentication token and authenticate your Docker client to your registry.

Use the AWS CLI:

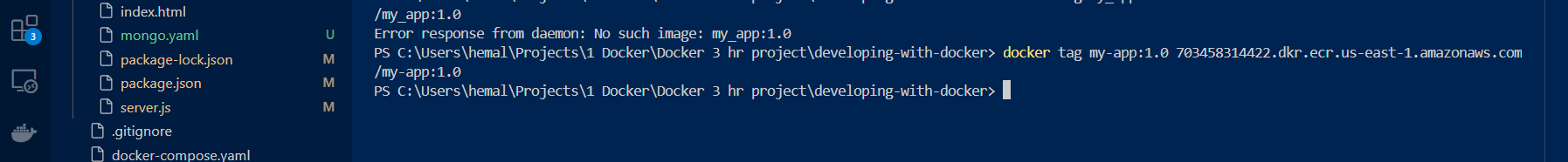


Build your Docker image using the following command. For information on building a Docker file from scratch, see the instructions [here](http://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html). You can skip this step if your image has already been built:

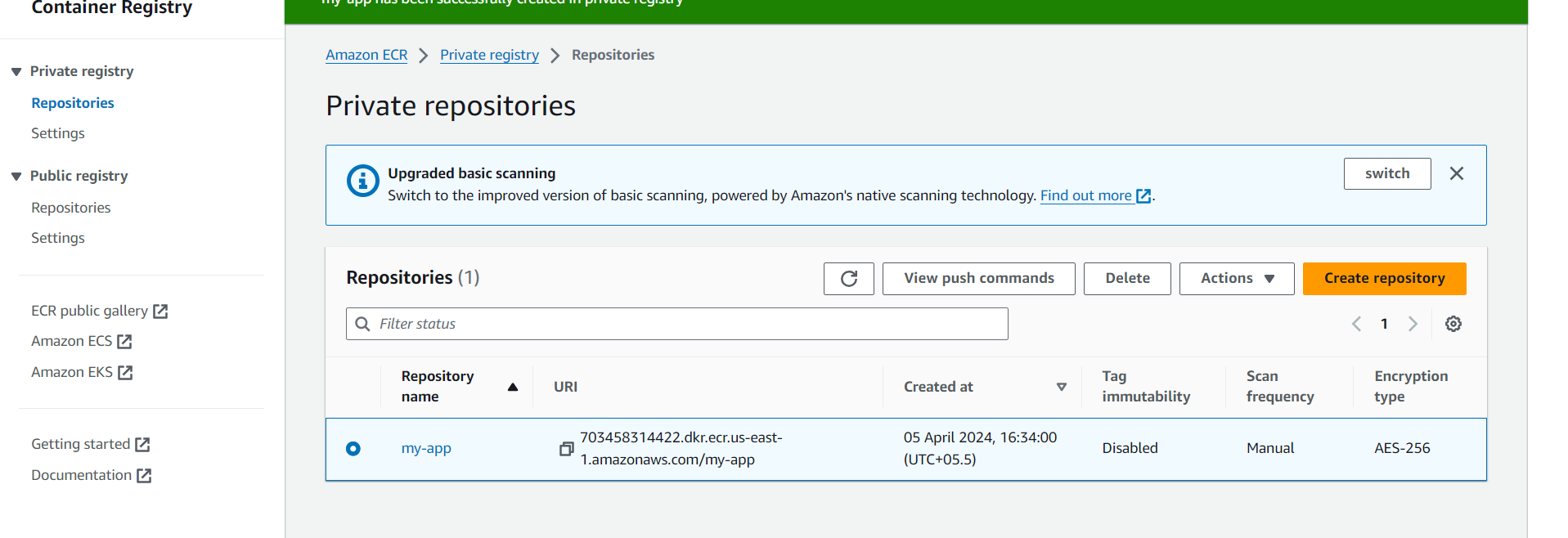
docker build -t my-app:1.0

After the build is completed, tag your image so you can push the image to this repository:

docker tag my-app:1.0 703458314422.dkr.ecr.us-east-1.amazonaws.com/my-app:1.0



Run the following command to push this image to your newly created AWS repository:



We have created repo with same name as image name

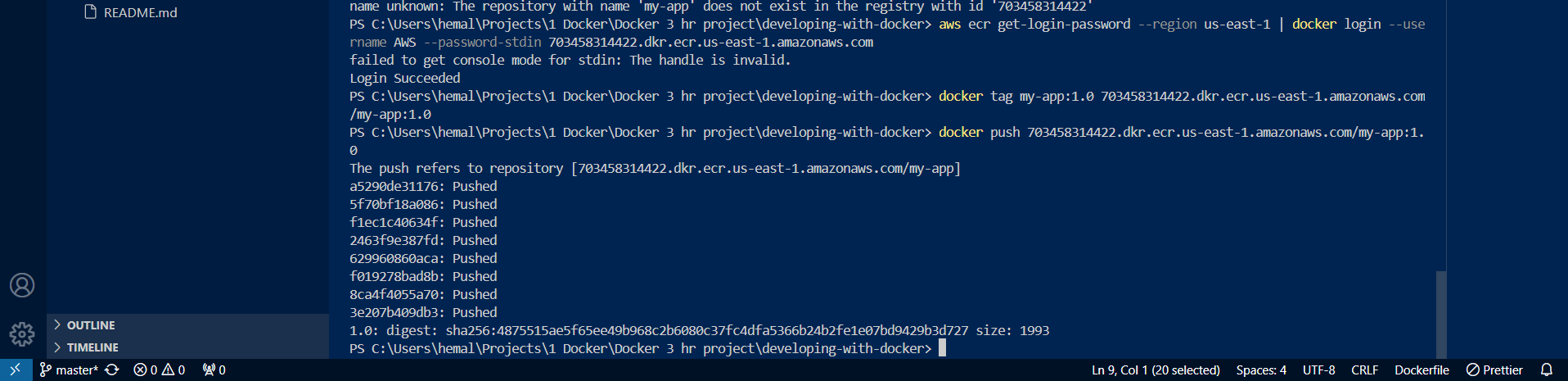


Image pushed to private repo

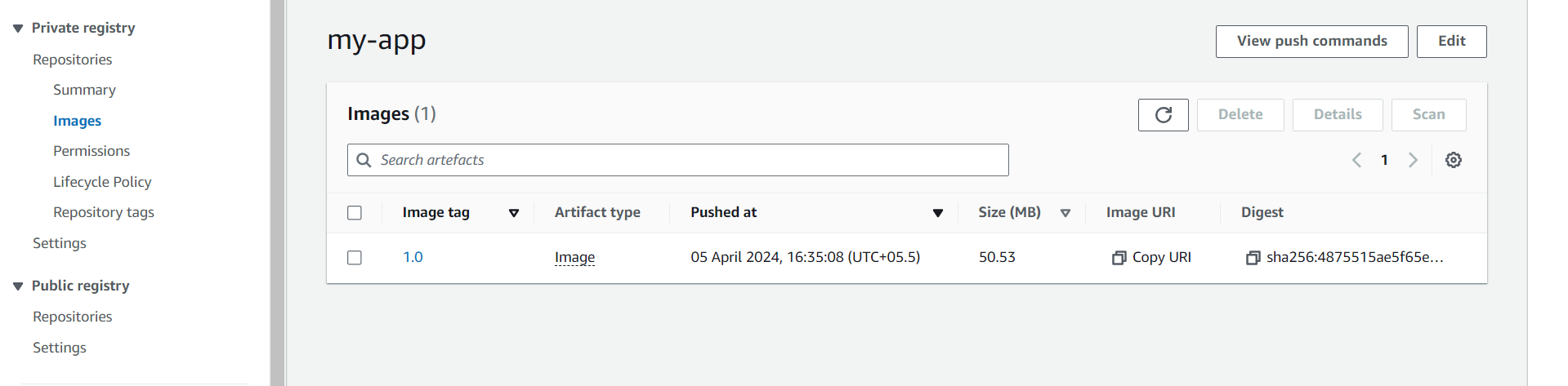
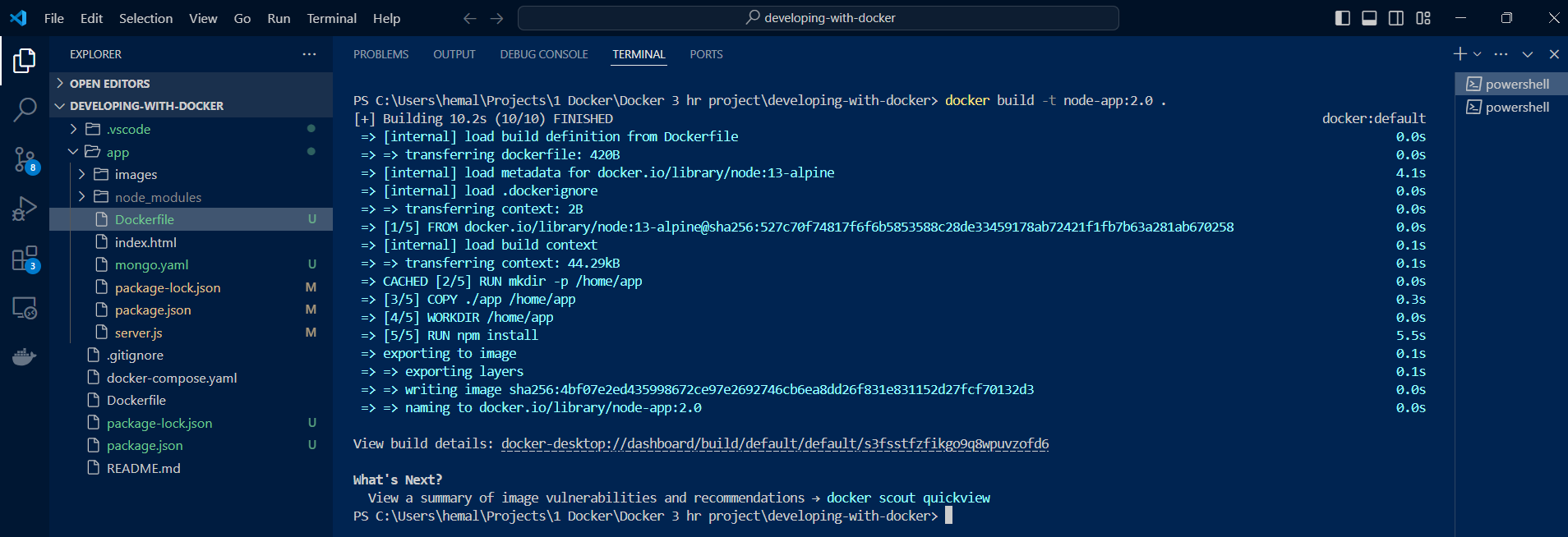
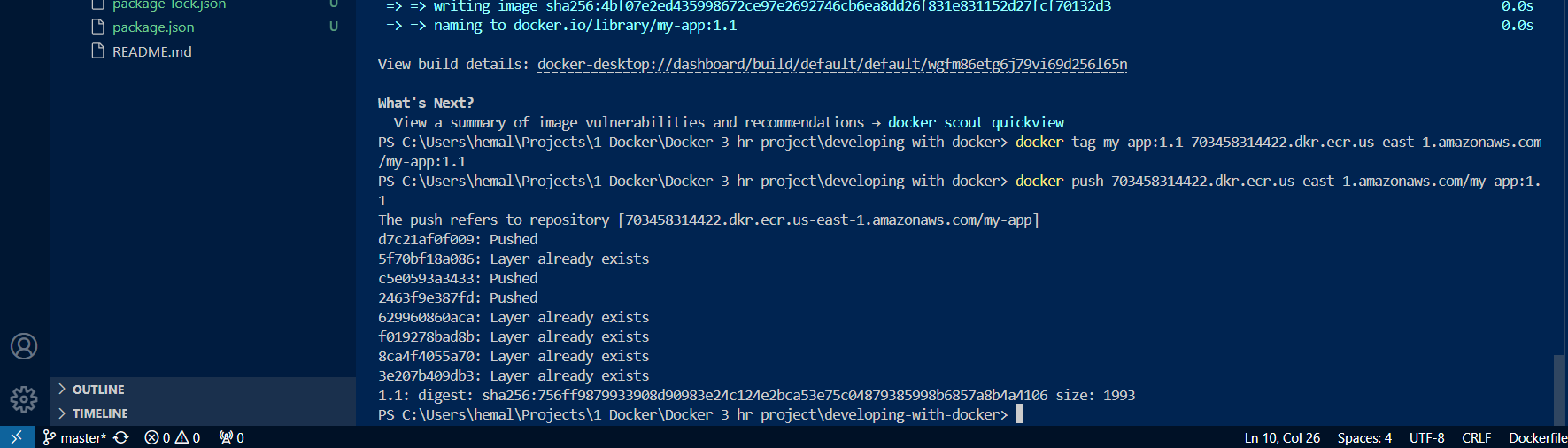


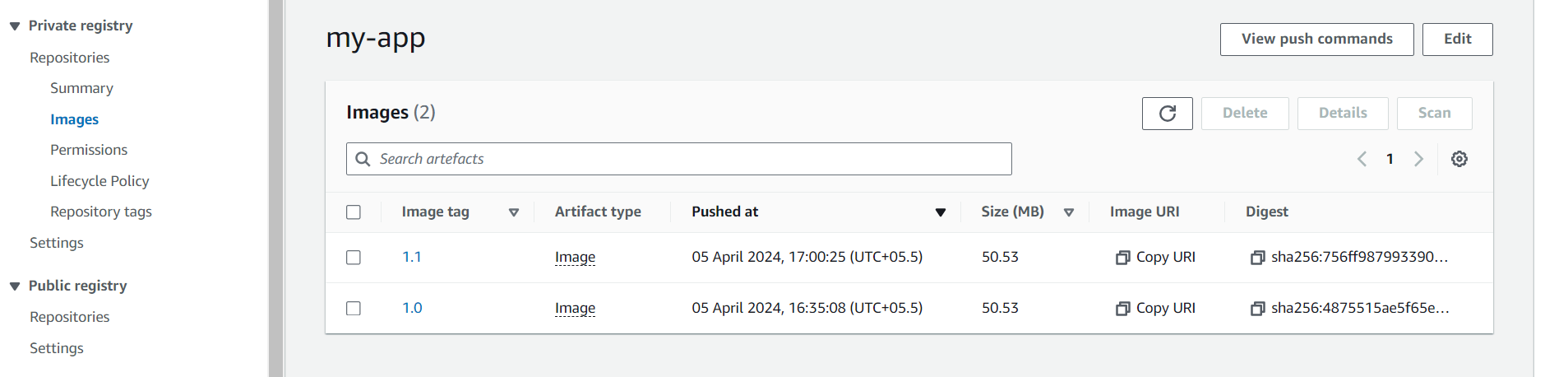
Image pushed successfully to ECR

* Making some changes to app source code or docker file 🡺 rebuild the image => push to ECR with updated version



Pushing to ECR the new version



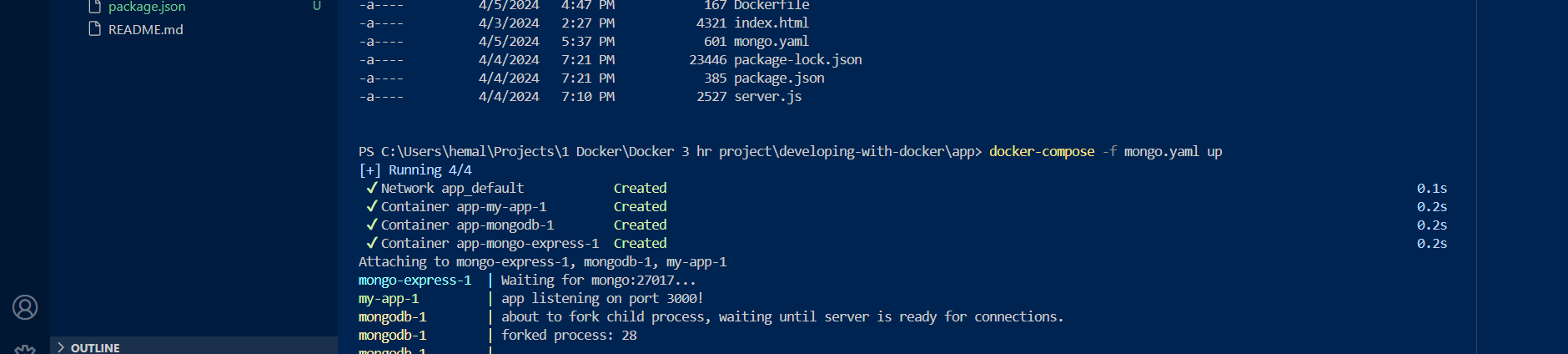


Deploy the app on server now..

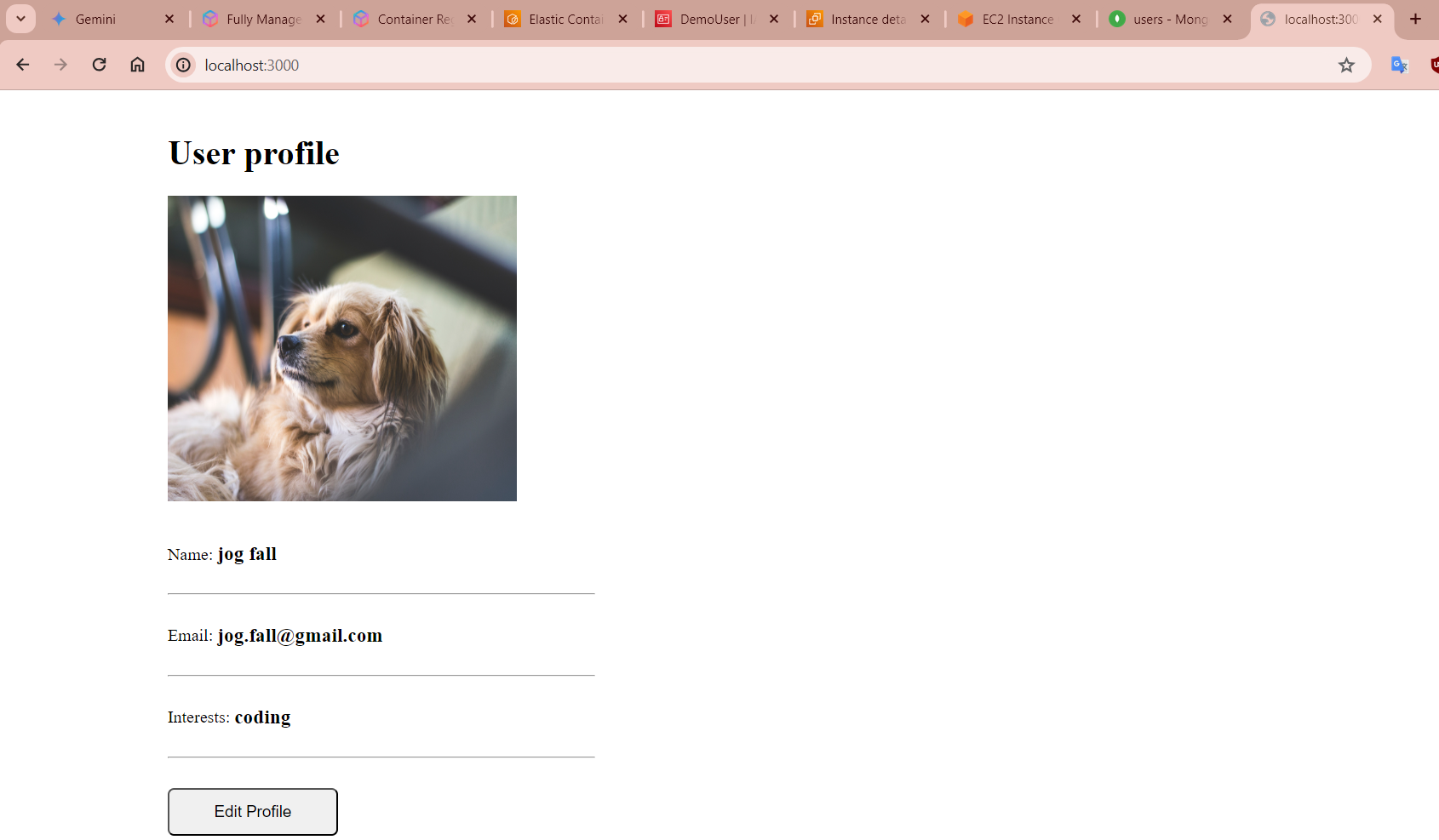
Now this server has to pull the app image from private ECR repo, we first need to provide docker login so that server can authenticate to ECR

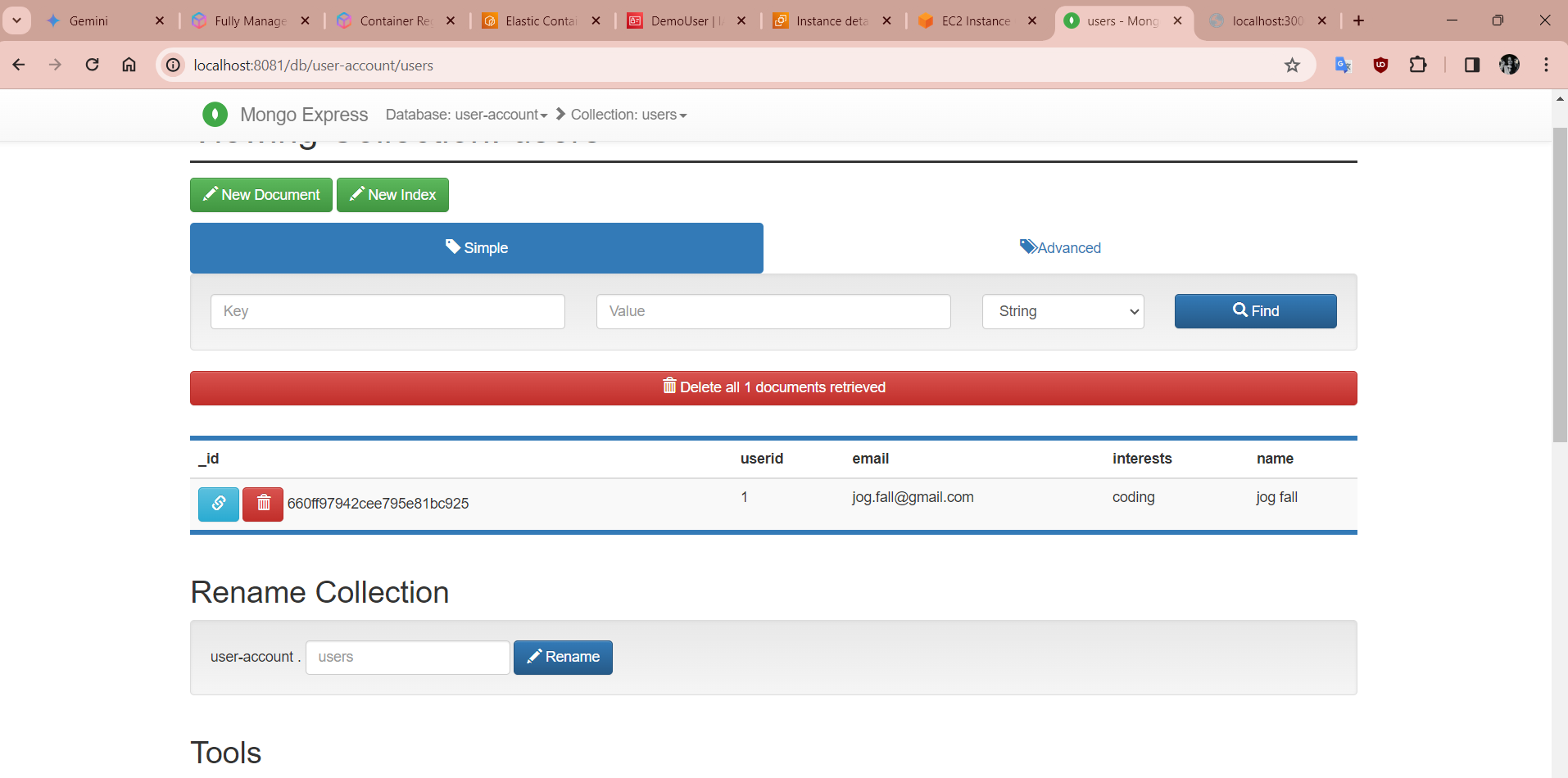
* Install aws cli and docker on the server
* Authenticate to aws : aws configure

Now run the compose file having my-app added as container name



Run the app , and test it out

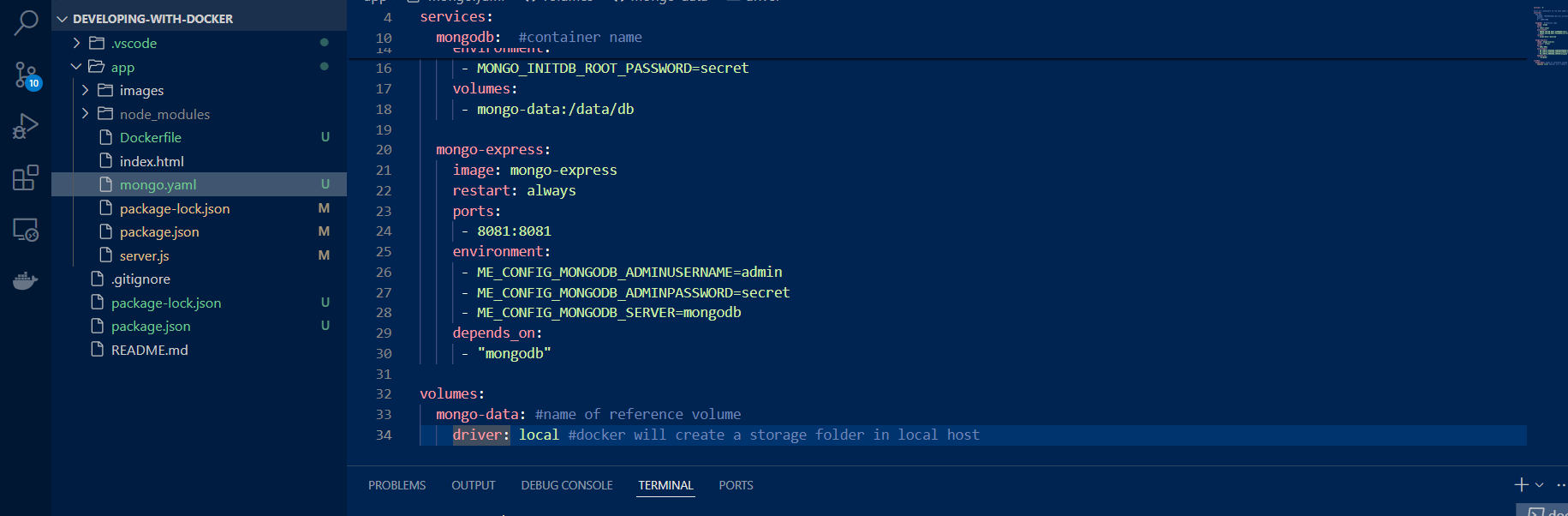




Docker Volumes:

When we restart a container, all the data stored in mongodb is lost!

So mounting Named volumes to containers, will mount a folder in host to the containers for persisting data when containers are restarted



Database and user info persists in UI even after restarting the containers