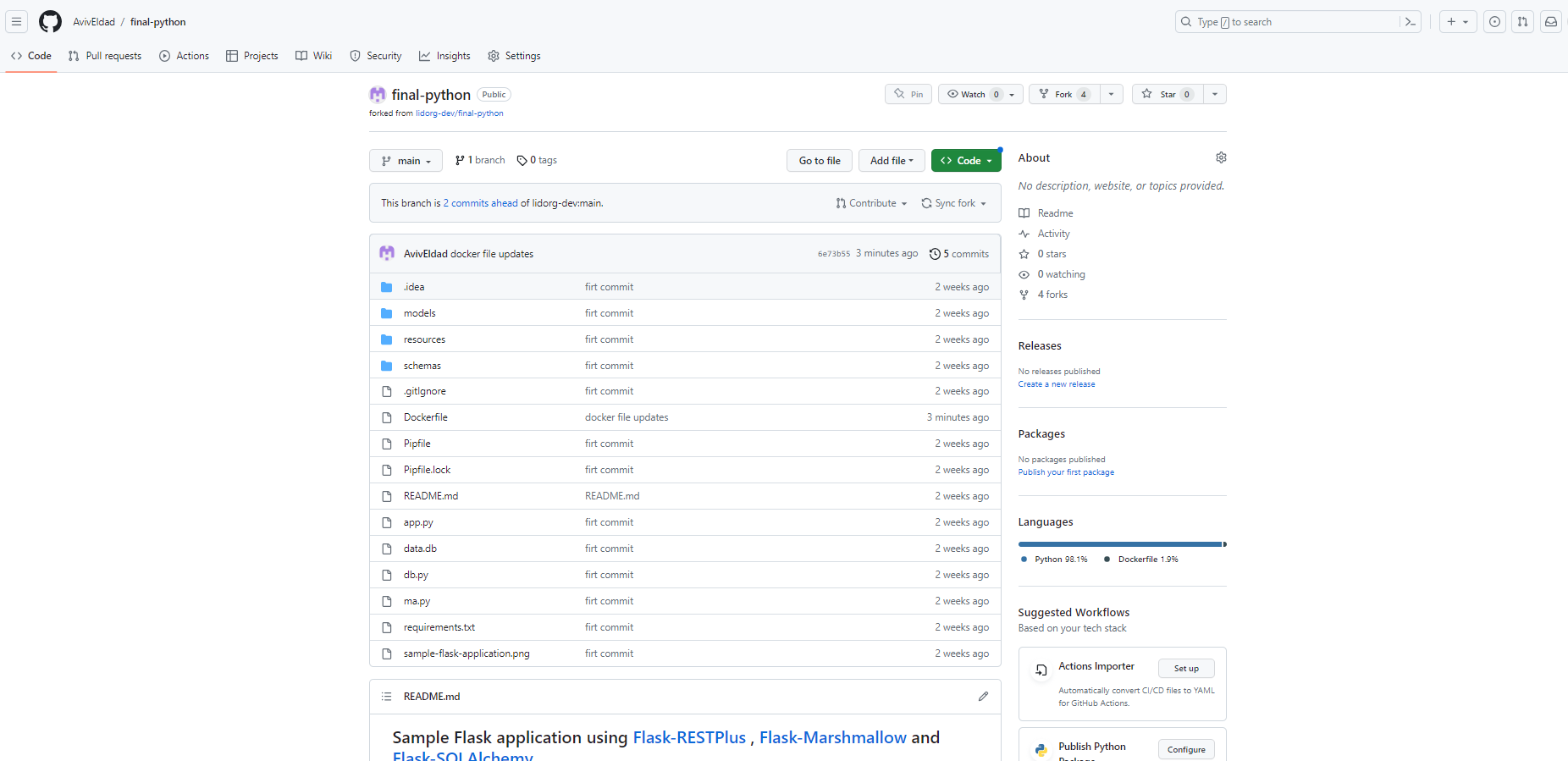
Part A

We chose question number 1.

1. We forked the original repo to our GitHub account. The screenshot provided is our repo, including the docker file:



1. We will explain about Docker: Docker is a technology that allows you to package and run applications in a consistent and isolated environment. It simplifies the process of deploying software by creating containers, which are like virtual machines but more lightweight. A Docker container is a standalone, executable unit that includes the application and all its dependencies, making it easy to run the same application on different machines without worrying about compatibility issues. A Docker image is a blueprint for creating containers, containing all the necessary files, libraries, and configurations to run the application, making it portable and enabling easy sharing and distribution of applications.
2. After the fork, we cloned the repo to our local environment (VScode) and created the docker file. We wrote the instructions in the file and then pushed it to the repo in GitHub.

A screenshot of a computer

Description automatically generatedDocker file screenshot:

1. Docker file instructions explanations:

From - the base image from docker Hub. We chose python:3.7-slim-buster version.

Maintainer - The maintainer information for the image. We set our names.

Workdir - the working directory inside the container. This will be the directory where subsequent commands will be executed.

Copy - Copies all the contents of the repo directory into the /app directory of the container.

Run - run the command pip install to install all the Python dependencies specified in the requirements.txt file.

Expose - Informs Docker that the container will listen on port 5000.

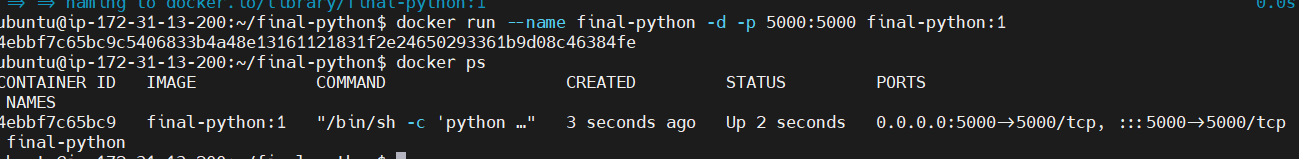
Cmd - Specifies the command to run when the container starts. In our case, it will run app.py from Python.

The Docker file defines the steps and configuration required to create a consistent and reproducible Docker image for the application. The instructions are used to build a Docker image.

1. After we created the Docker file, we started an EC2 machine on our AWS account. We create a machine with Ubuntu 22.04 , and **opened the traffic for port 5000**. We used MobaXtrem to use the machine and install docker on it. We created a final-python directory and copied all the files to the new directory. We run the command: “docker build -t final-python:1 .” to build the image based on our docker file. The -t flag is to give a name to the image, and the number is the version.
2. After this we run the command: “docker run --name final-python -d -p 5000:5000 final-python:1” to run a new container. The –name flag is used to provide a name for the container. The -d flag is used to run the container in detached mode, so it will run in the background.

The -p flag is used to publish the ports and map a port from the container to the host. We are mapping port 5000 to port 5000. In the end, the image name to run (final-python:1)

After running, we check our container is up and running. We used the ‘docker ps’ command to see the running containers:

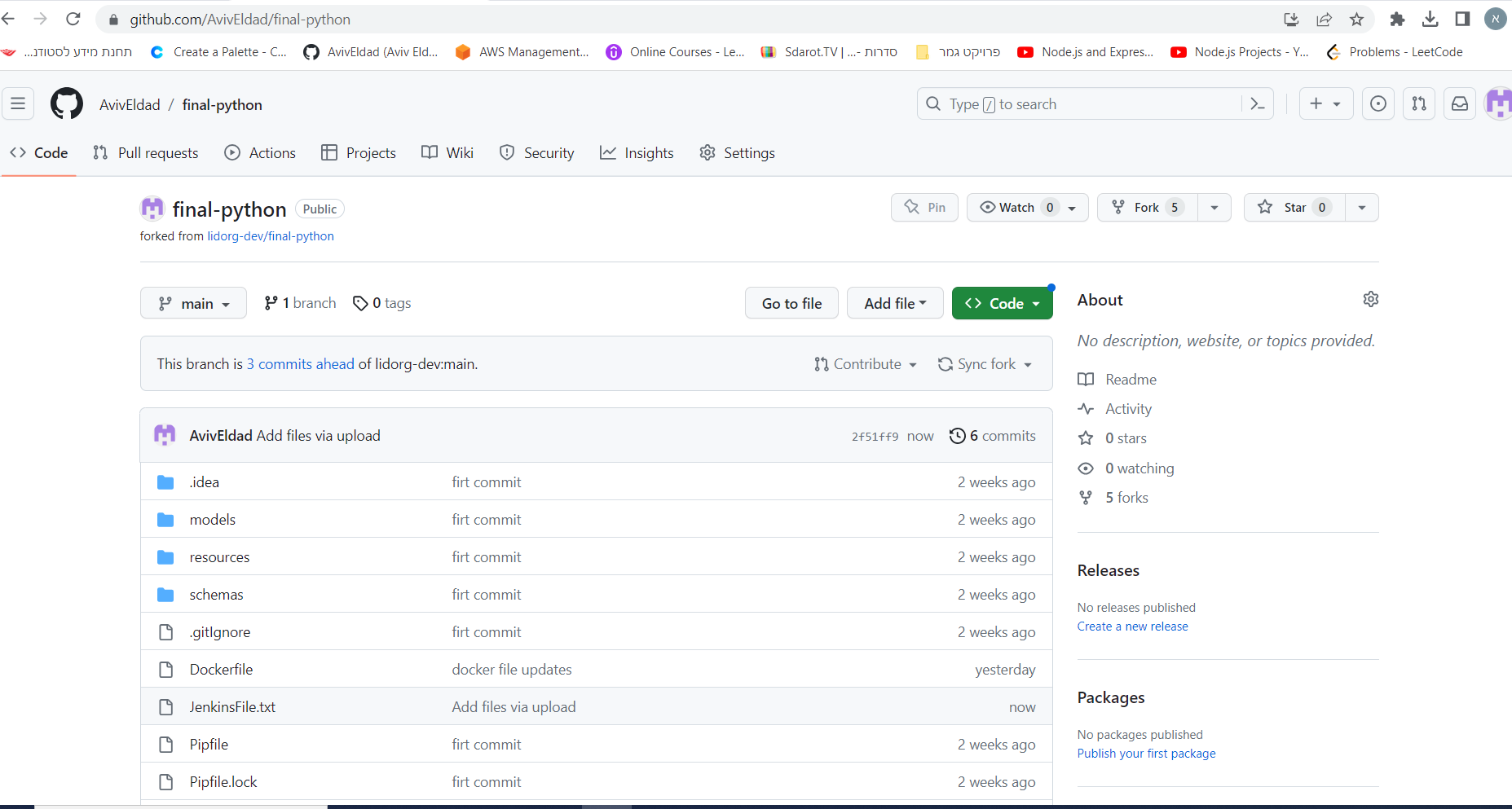


1. A screenshot of a computer

   Description automatically generated with medium confidenceAs we can see, the container is up and running. We can access the app in the browser, where we need to use the machine IP, port 5000, and the path that was given in the repo example (/api/doc):

Part B

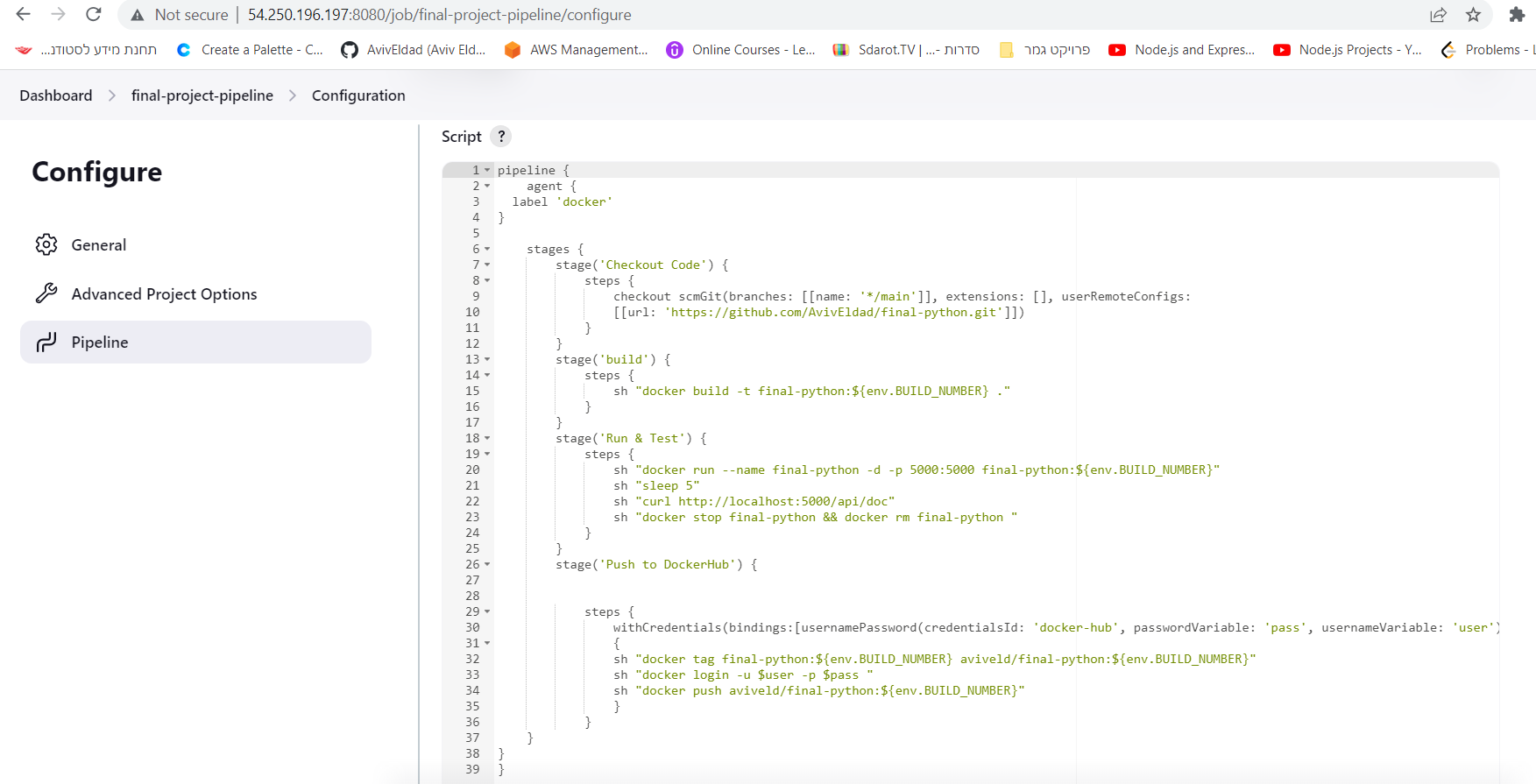
We are using the same repo, we added a Jenkins file to it:



We will explain Jenkins: Jenkins is an automation server that helps streamline the software development process. It acts as a central hub for managing tasks like building, testing, and deploying applications. It allows you to automate repetitive tasks and create workflows for continuous integration and delivery. With Jenkins, you can easily monitor the progress and status of your development projects, collaborate with team members, and ensure the timely and reliable delivery of software. Essentially, Jenkins simplifies and automates various aspects of the software development lifecycle to improve productivity and efficiency. Jenkins can help automate the process of building and running Docker containers based on Dockerfiles.

For this section, we needed two machines. One that will run the Jenkins and one that will be the agent.

We used the machine from exercise 3, where we had all the necessary configurations (user, plugins, credentials, etc.)

1. We entered the first machine and run Jenkins. We opened the browser and entered the Jenkins user.
2. Before we could run the pipeline, we needed to do preparation:
3. Add our credentials to docker hub so the pipeline will be able to push the image to docker hub.
4. Run a second machine that will be the agent. **The agent is responsible for carrying out the tasks specified by the Jenkins pipeline**. On the second machine, we installed docker and java (because Jenkins runs with Java), and created Jenkins directory.
5. Create a new agent through manage nodes and clouds. We created an agent with the name docker and got the commands to connect the agent to Jenkins. We entered the commands in the agent machine and got connected.
6. We opened all the necessary ports in the machines.
7. After all the preparations are done, we created a new pipeline with the name “final-project-pipeline”.
8. We wrote the Jenkins file code in the script box:

Code explanation:

1. The code defines a pipeline with 4 stages, where the agent that will run it will be ‘docker’
2. The first stage is **Checkout Code**: This stage checks out the code from our Git repository (https://github.com/AvivEldad/final-python.git) branch main.
3. **Build**: This stage builds a Docker image with the tag final-python:${env.BUILD\_NUMBER}. The env.BUILD\_NUMBER variable represents the Jenkins build number.
4. **Run & Test**: This stage runs a Docker container named final-python based on the built image. It exposes container port 5000 to the host port 5000 using the -p flag. It then waits for 5 seconds to let the container set up, performs a curl request to http://localhost:5000/api/doc to test the running container, and finally stops and removes the container using docker stop and docker rm commands.
5. **Push to DockerHub**: This stage tags the built Docker image as aviveld/final-python:${env.BUILD\_NUMBER}. It uses the docker login command the credentials we provided in the Jenkins credentials. Finally, it pushes the tagged image to our DockerHub using the docker push command.

A screenshot of a computer

Description automatically generatedWe can see the pipeline after we build it:

And the image in our docker hub repo (with tag 1 as the build number):

A screenshot of a computer

Description automatically generated with medium confidence

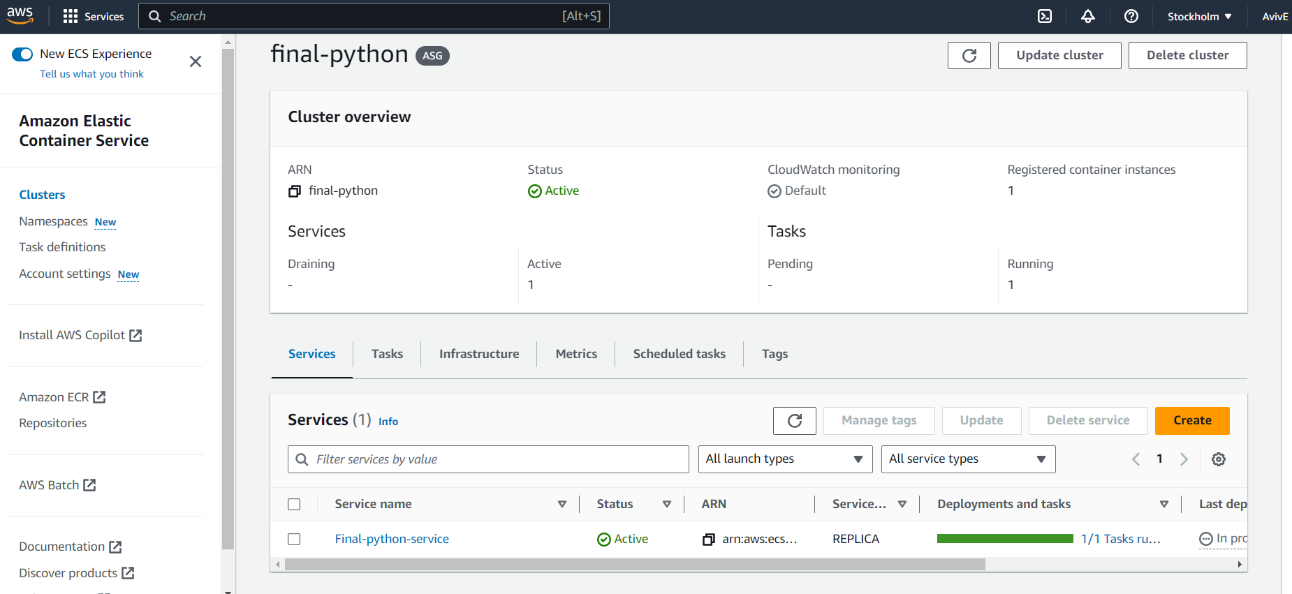
Part c

We chose question number 5.

ECS helps us run and manage Docker containers. It takes care of the underlying infrastructure, allowing you to focus on deploying and scaling your applications easily.

To do so, we first need to create some configurations:

1. **Cluster**: A cluster in ECS represents a group of EC2 instances (We chose this option) tasks that provide the computing resources for running the containers. It's like a pool of servers managed by ECS, where the containers are scheduled and executed. The cluster created for us an EC2 machine.

We created a cluster for our exercise:

1. After the cluster is running, we need to define the task definition. A task definition is like a blueprint that describes how the containers should run. It specifies the Docker image, resource requirements, networking, environment variables, and other settings needed for running your containers. We specified the images from our docker hub, and the relevant ports (5000) for it.

The task definition screenshot:

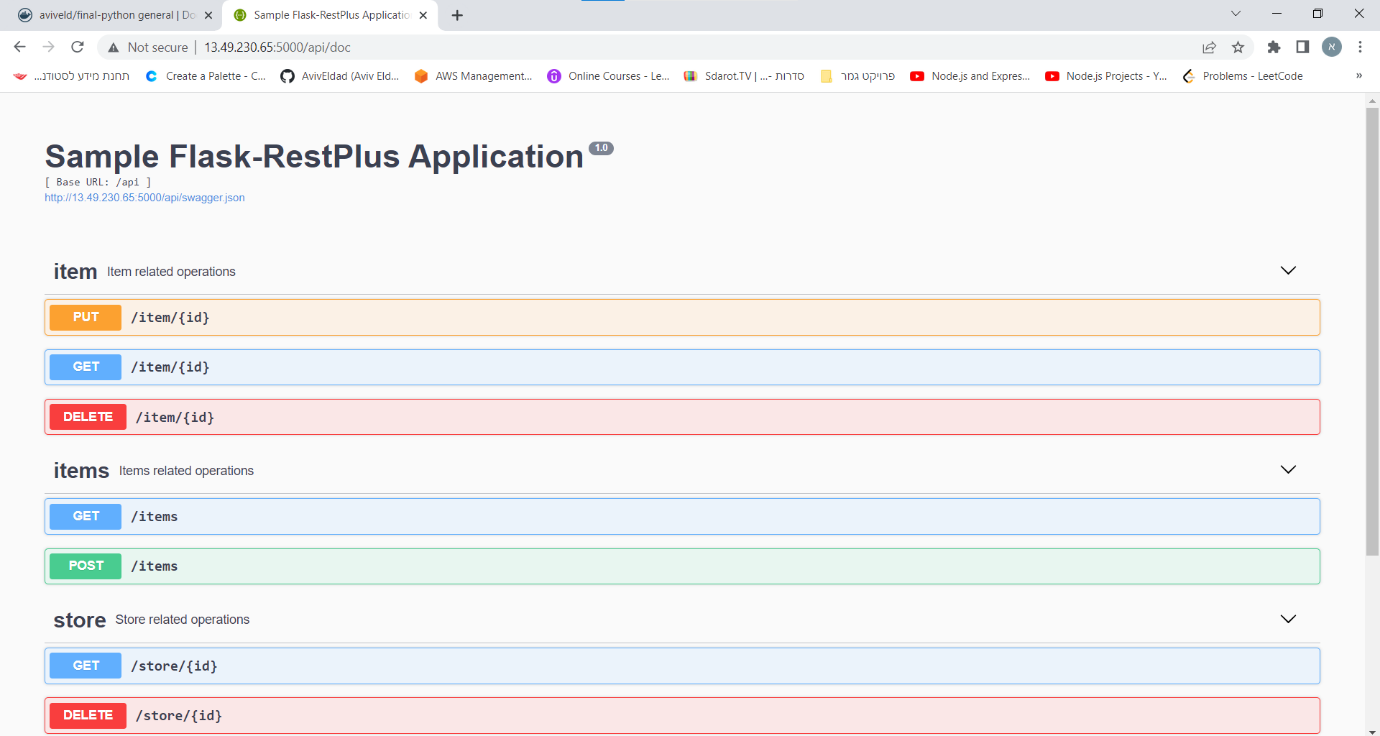
A screenshot of a computer

Description automatically generated

1. After the task definition is running, we need to create a service for it. A service is a higher-level construct that allows us to define and manage the desired number of tasks (containers) to run and maintain availability. It ensures that the specified number of containers are always running and automatically handles scaling and load balancing.

A screenshot of a computer

Description automatically generated with medium confidenceWe created a service for the task. As we can see, the service is running:

After all of this, we have an app in the air. We opened the browser with the EC2 IP (the machine the cluster created for us) and entered the port and path to see the app: