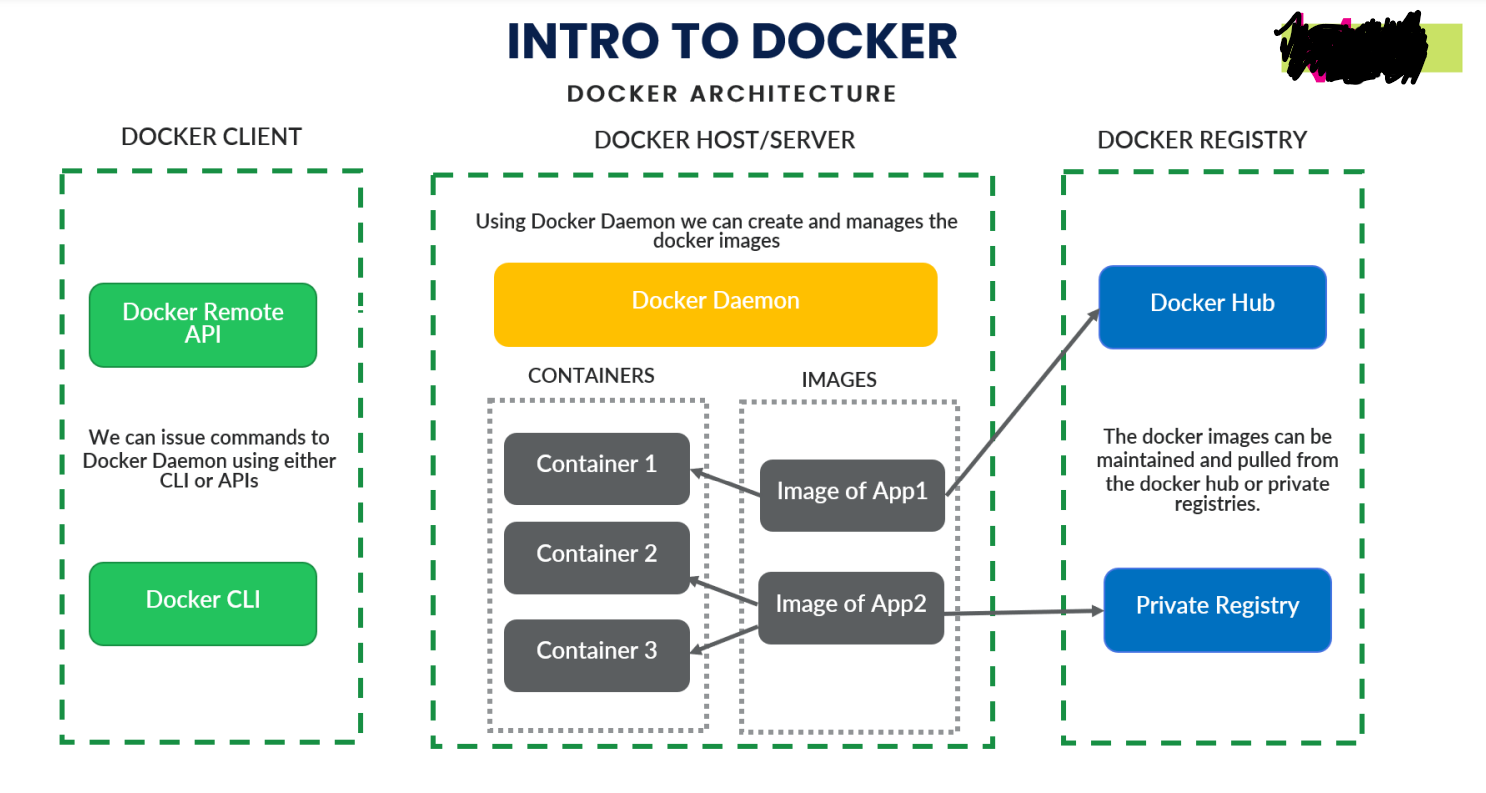
**H] Docker:**

**H1] Docker Architecture:**



**1. Docker Client:**

Docker client uses commands and REST APIs to communicate with the Docker Daemon (Server). When a client runs any docker command on the docker client terminal, the client terminal sends these docker commands to the Docker daemon. Docker daemon receives these commands from the docker client in the form of command and REST API's request.

**2. Docker Server:**

Using docker server we can build or manage images of our application. Here, Image contains **all the business logic that I have written and all the dependencies that are required to run that application**. And from that image we can create our containers, they are for example an image is like our java class and container is an object to that class.

From a single image we can create many containers as per our requirement

Also, once a docker image is been created we can then deploy that same image from our local in to dev/test/prod etc. environment

Without docker image we would have to manually install all the dependencies(lib’s) to run that application

**3. Docker Registery:**

It’s like the github repository where we share our code to different users, docker registery contains of **docker hub** where we can store our docker images as public copy and any person knowing that docker image name can pull that image and start running that image using docker command.

There is also an option of docker **private registery** where we have to pay some money to docker so as our docker image becomes private copy and not an public copy and would be accessed by only authorised persons.

**H2] Docker Installation:**

Windows installation url:

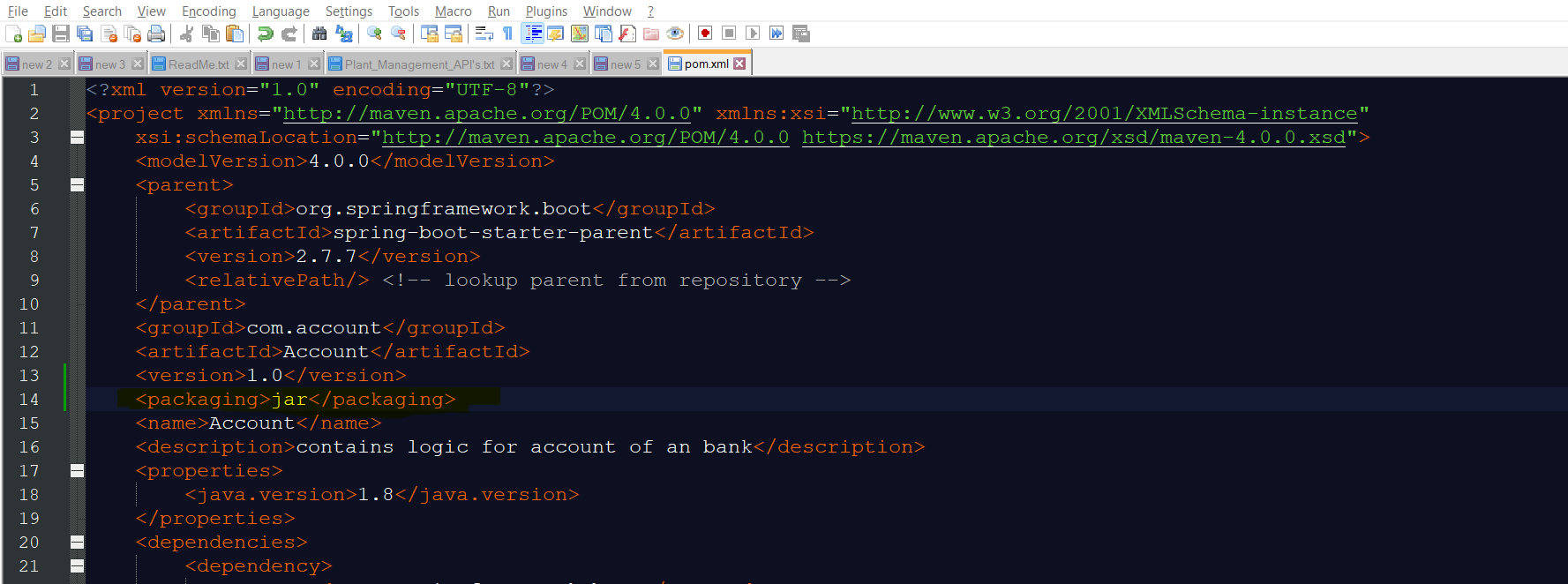
<https://docs.docker.com/desktop/install/windows-install/>

and after that it will automatically install docker in your windows 10 system and will restart windows10 system

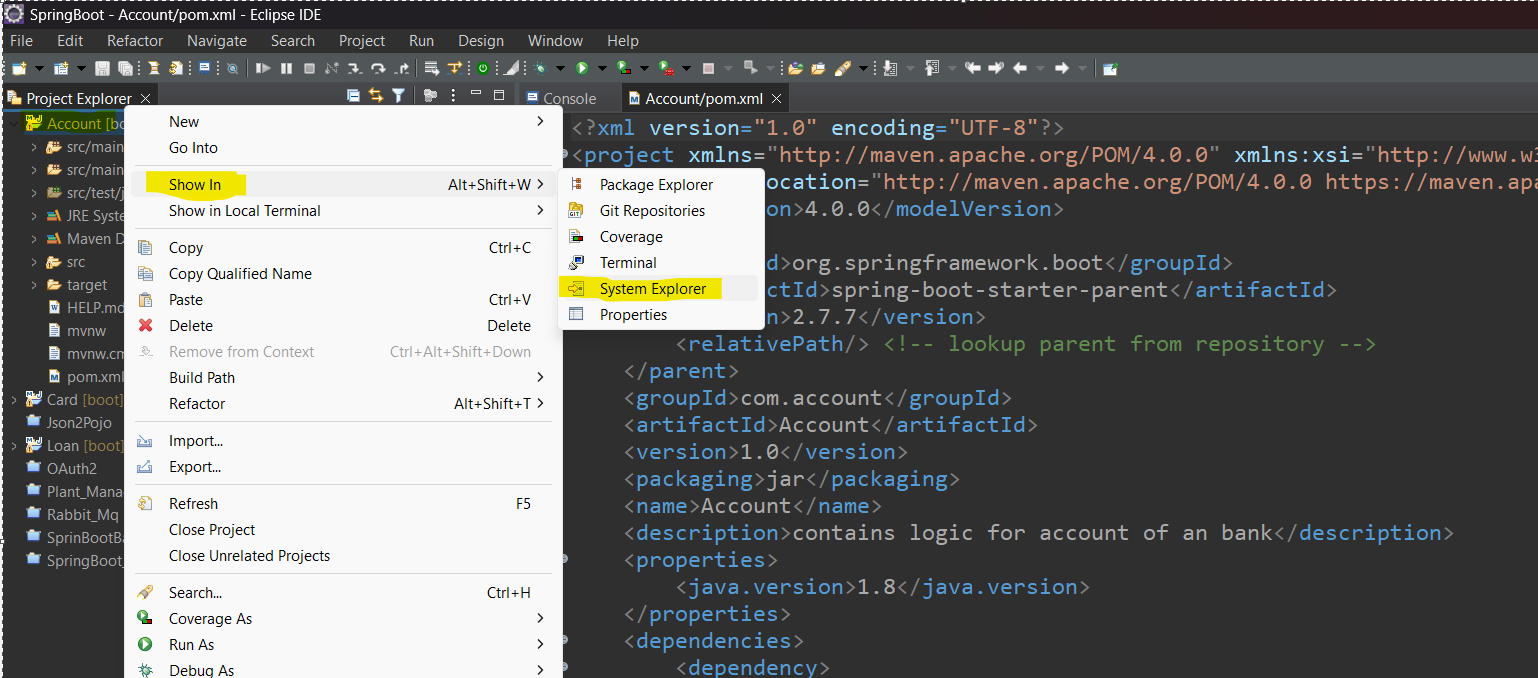
**H3] Creating/Executing a Docker Image and Container:**

**H3.1] Create a jar/war file of your microservice as below:**

S1] Inside pom.xml file in packaging tag type **jar/war** as highlited in below image



S2] Now go to the project directory where your project folder is located as beow:

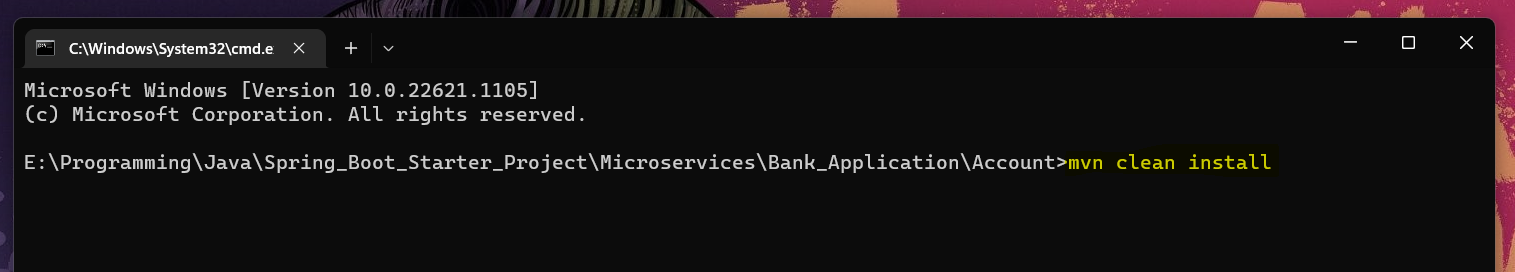


S3] Now open that folder path inside command prompt and run below command in it:

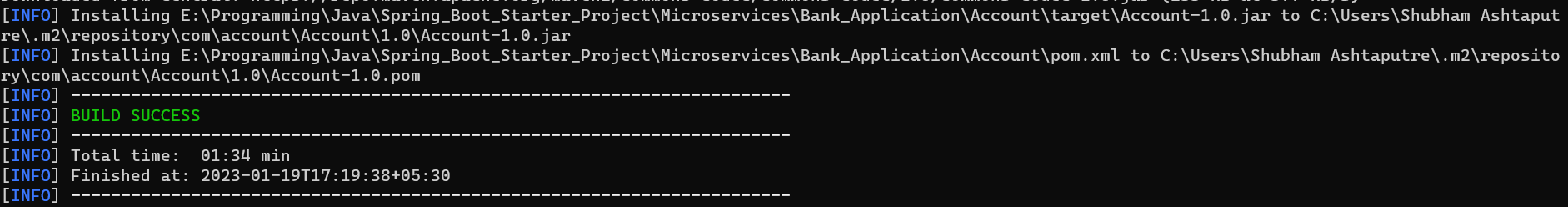
Command: **1] mvn clean install [This command will create a jar file]**

**2] mvn clean package [This commans will create executable jar file]**

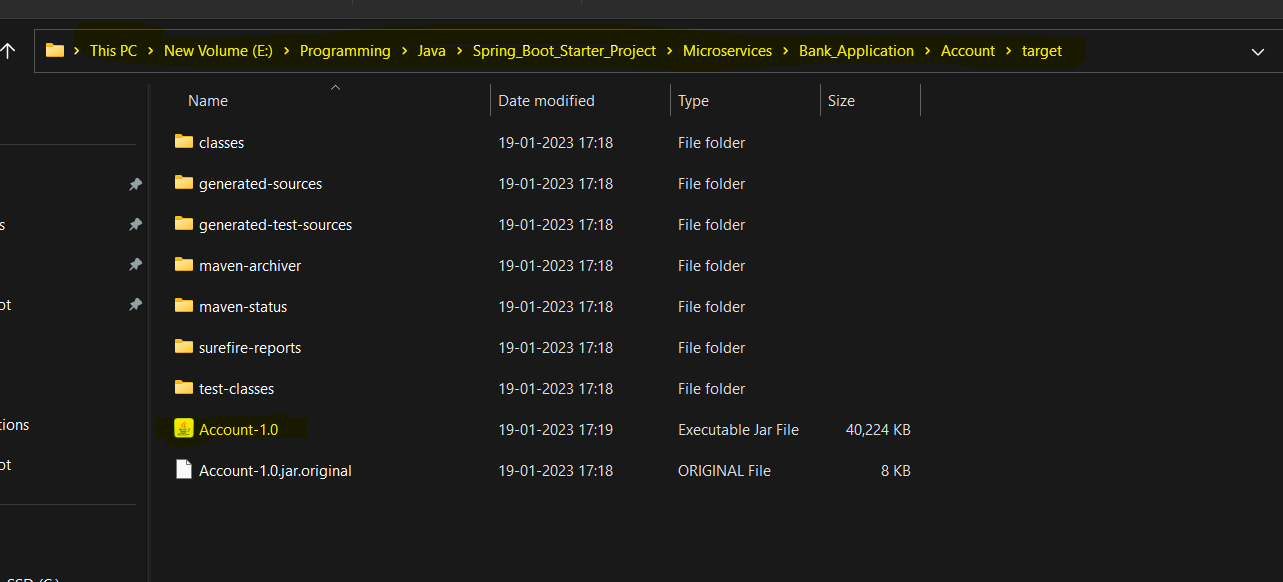
**Generally preffer to create the executable jar file using second command!!!**



This command will create the jar file of our project and indicate once it completed building as below:

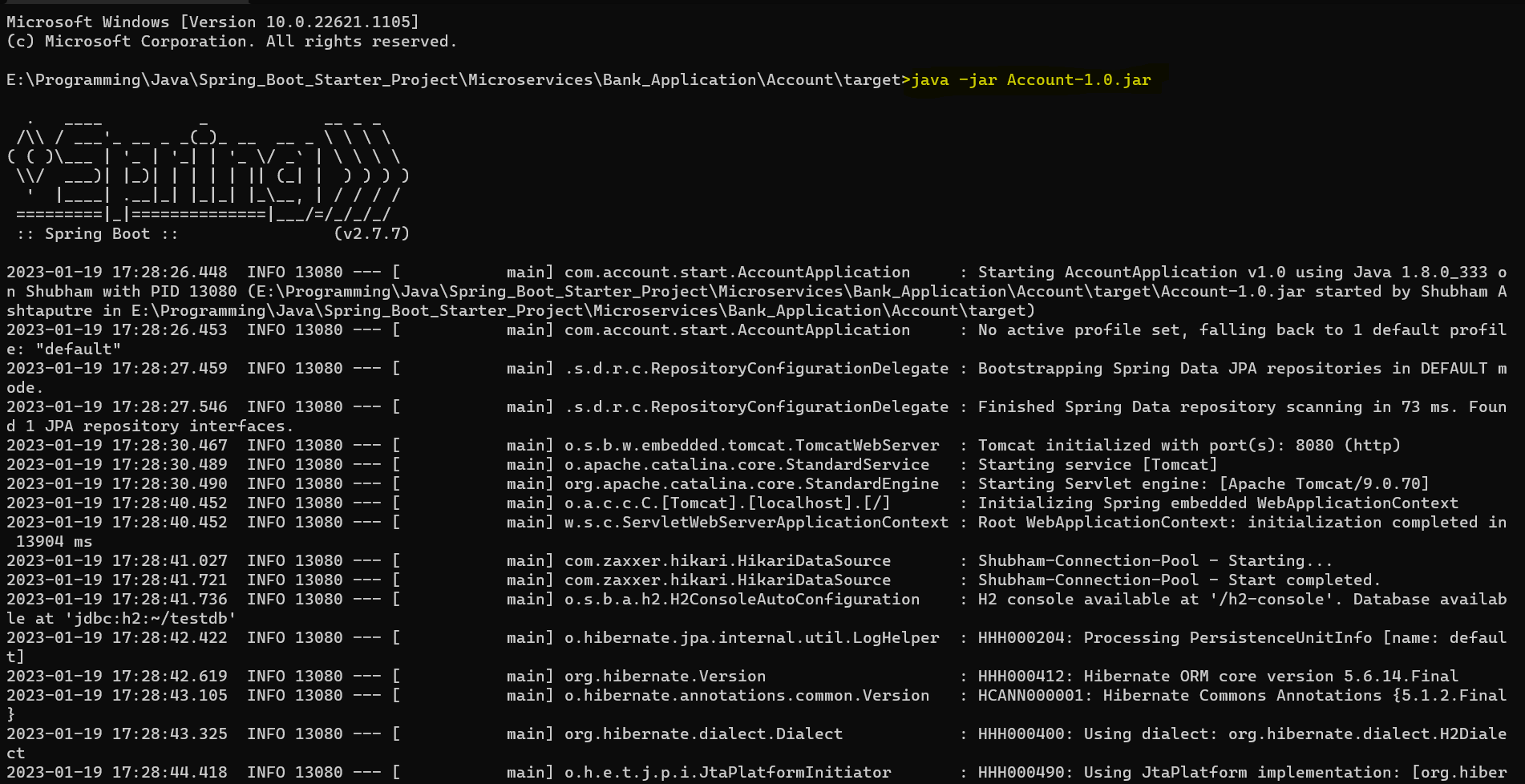


S4] Now you will get your deployed jar file in the target folder of your project as below:



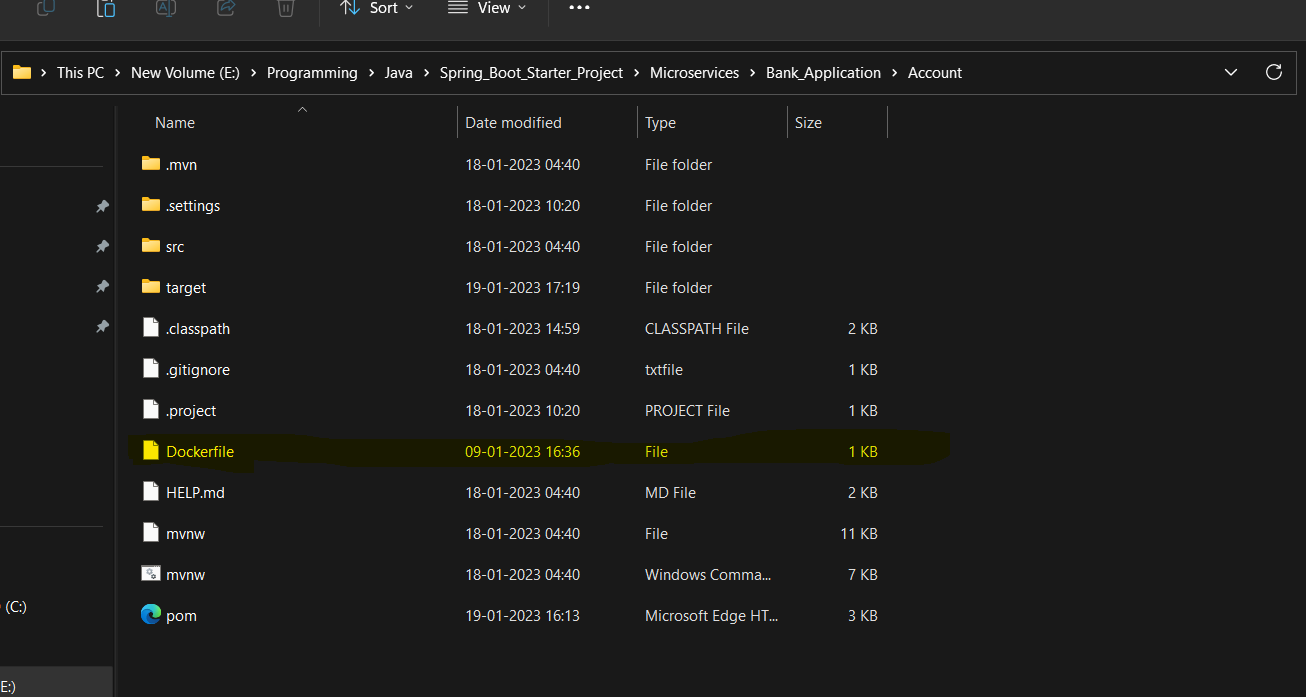
S5] Now you can check whether your jar file is running or not using below command:

**java -jar Account-1.0.jar**

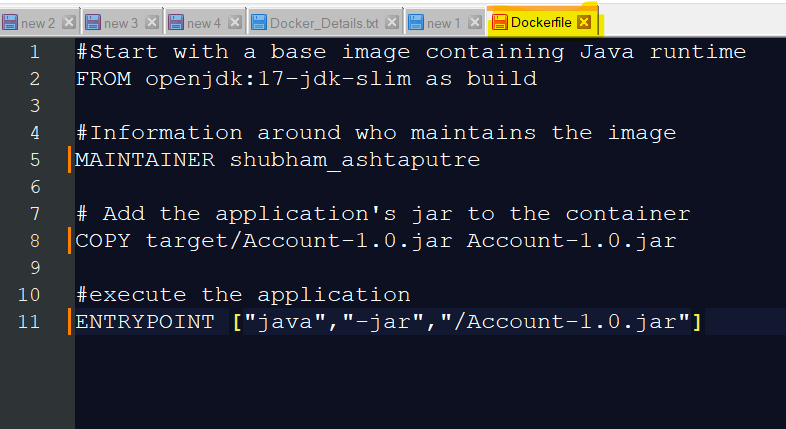


S6] press **ctrl+c** to close that running jar

**H3.2] Create a docker file in your project folder:**



Define the flowing lines in docker file as below:



**FROM openjdk:17-jdk-slim as build**

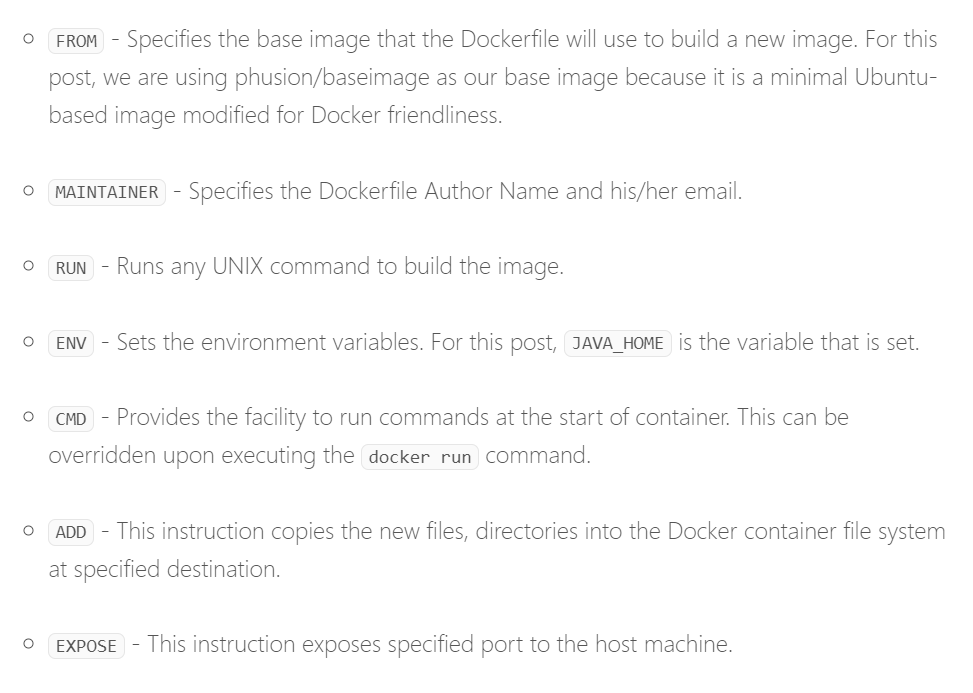
Here the ‘**FROM**’ command indicates what is the base image on which I want to build my application, here in this case I want to build my application on open-jdk-17 but it is not installed on my local system so docker will download it from its own image repository and install it on my system

**COPY target/Account-1.0.jar Account-1.0.jar**

Here ‘**COPY**’ command tells docker to copy the jar file from the defined location in out local system and then paste it into the docker filesystem

**ENTRYPOINT ["java","-jar","/Account-1.0.jar"]**

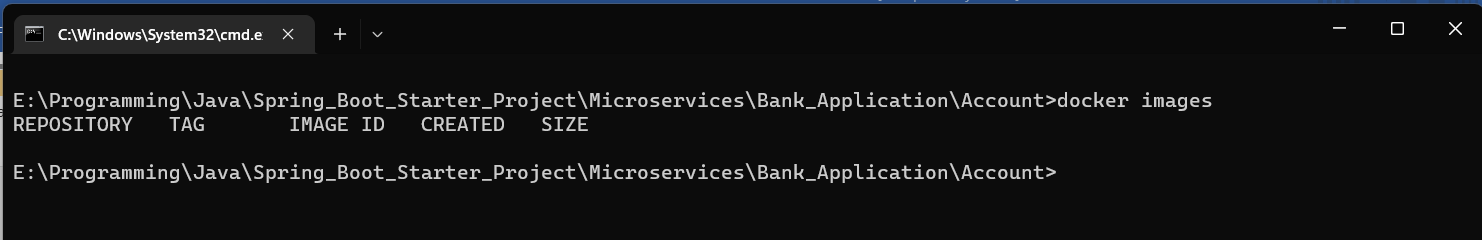
Here, the command entry point tells docker that whenever I want to start my container then start it with the **["java","-jar","/Account-1.0.jar"]** command



**H3.3] Building a docker image:**

Before building a docker image let’s check is there any docker image present inside our project directory as below:

**Cmd: docker images**



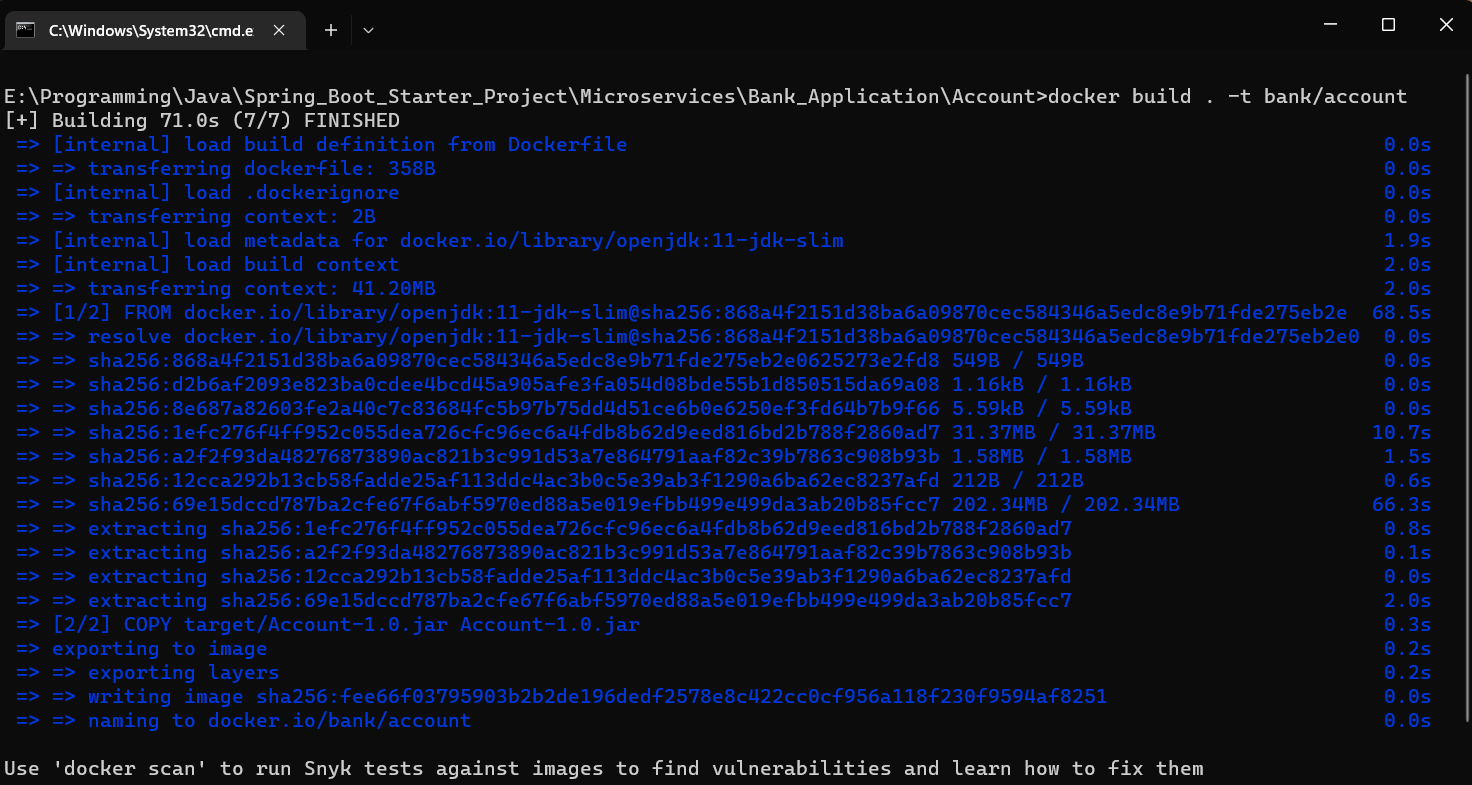
As you can see there is no docker image present inside our project directory

Now, run the below command to create the docker image based on your docker file configuration:

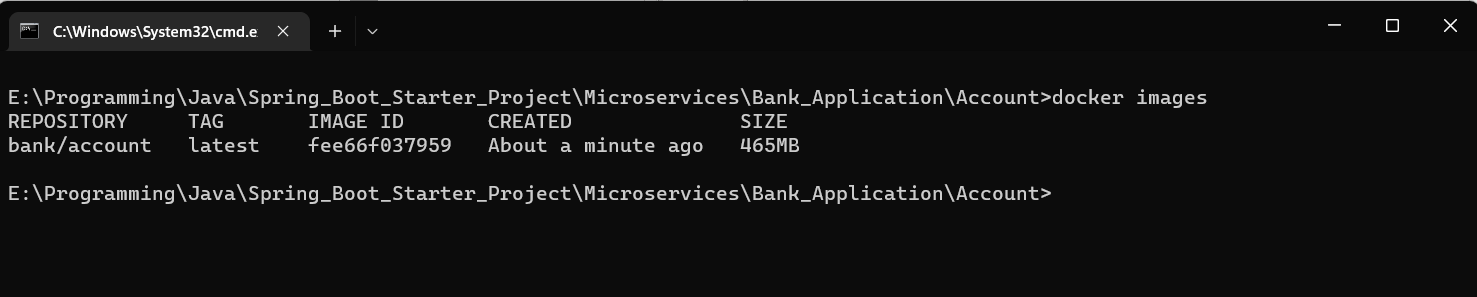
**Cmd: docker build . -t bank/account**

Here, ‘**.**’(**dot**) Indicates the docker file present inside my same project folder location, ‘**-t**’ indicated the tag name that I want to give to my image i.e. ‘**bank/account**’ because docker will by default give **some random number for the image/container** so just to given our own image/container name according to project requirment we use ‘**-t**’ command

After image build is completed it shows as below:

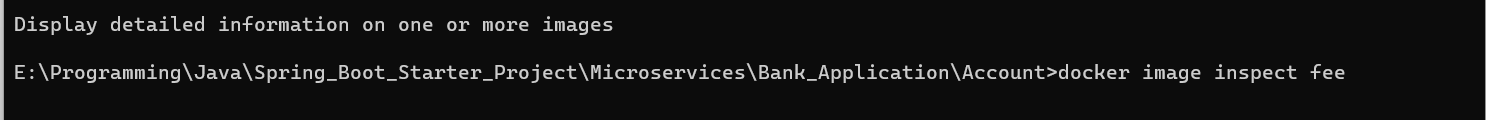


To verify run the below command:

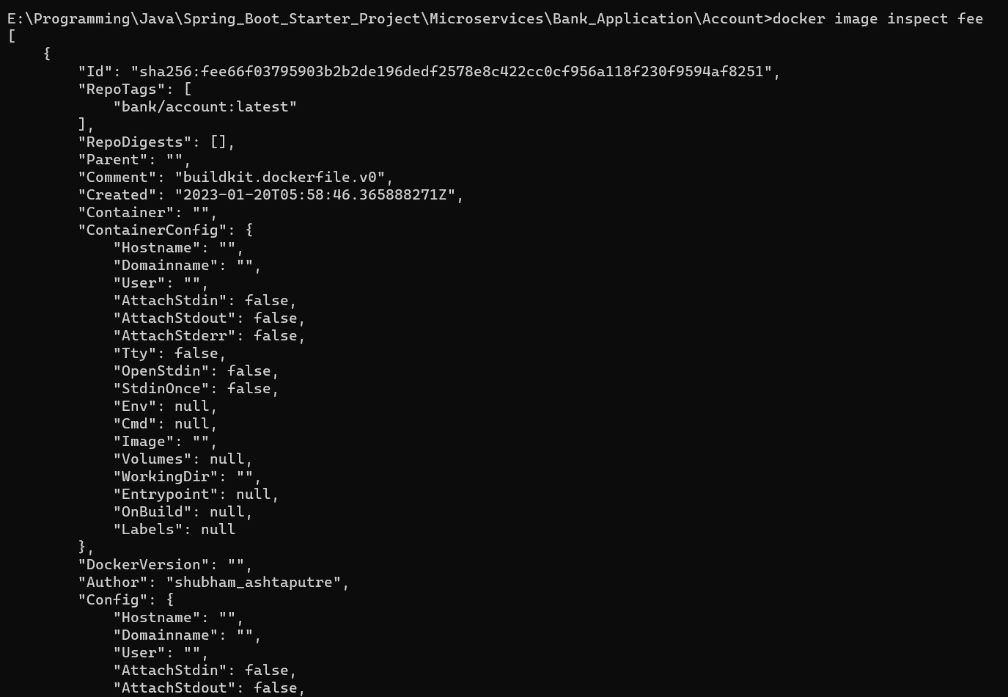


To inspect/ get details about the image run the below command:

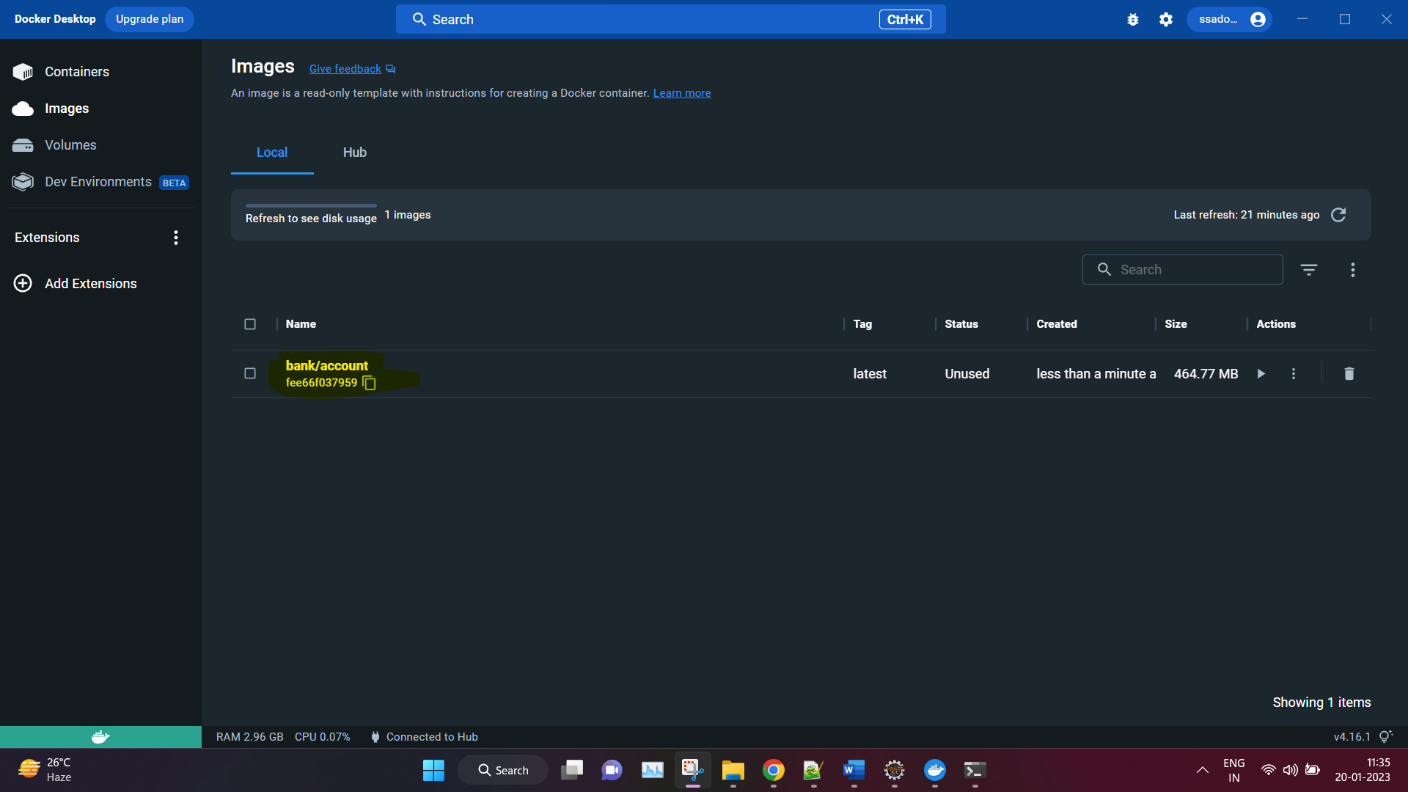
**Cmd: docker image inspect <Image Id>**



Here you don’t need to give full image Id name as docker will figure it out

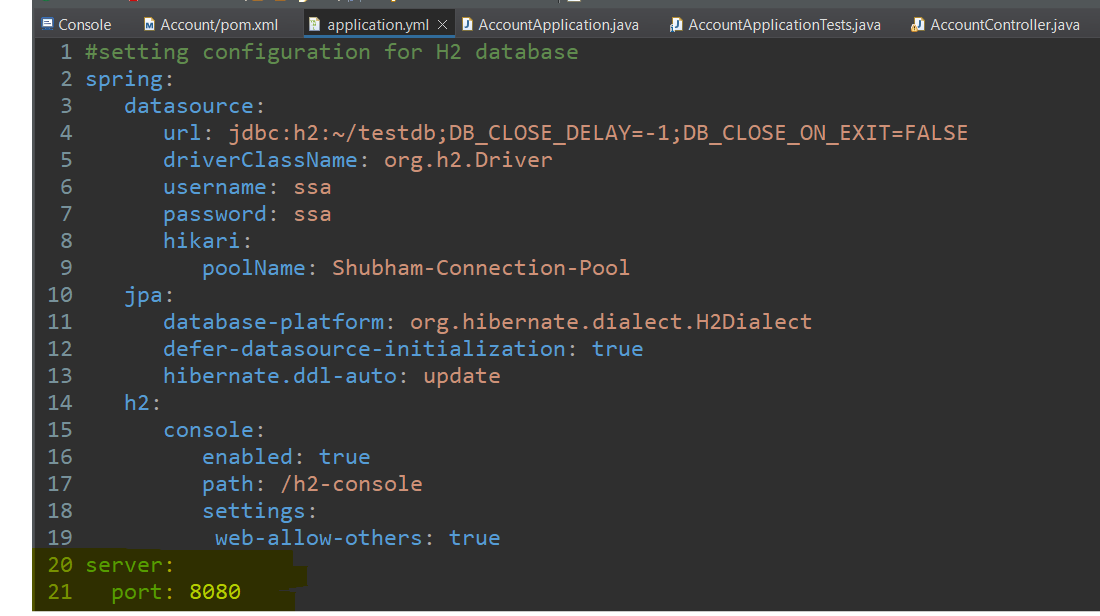


Now inside your docker desktop application also you can see your created image:



**H3.4] Start and Deploy microservice application on docker:**

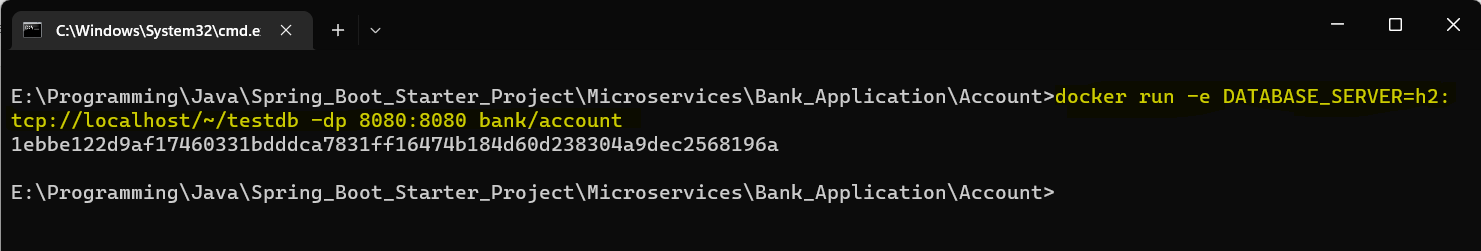
By default, the port on which the server will run is 8080 in docker if you want to change the port number then do changes in projects **application.yml** file as below:



Run the blow command:

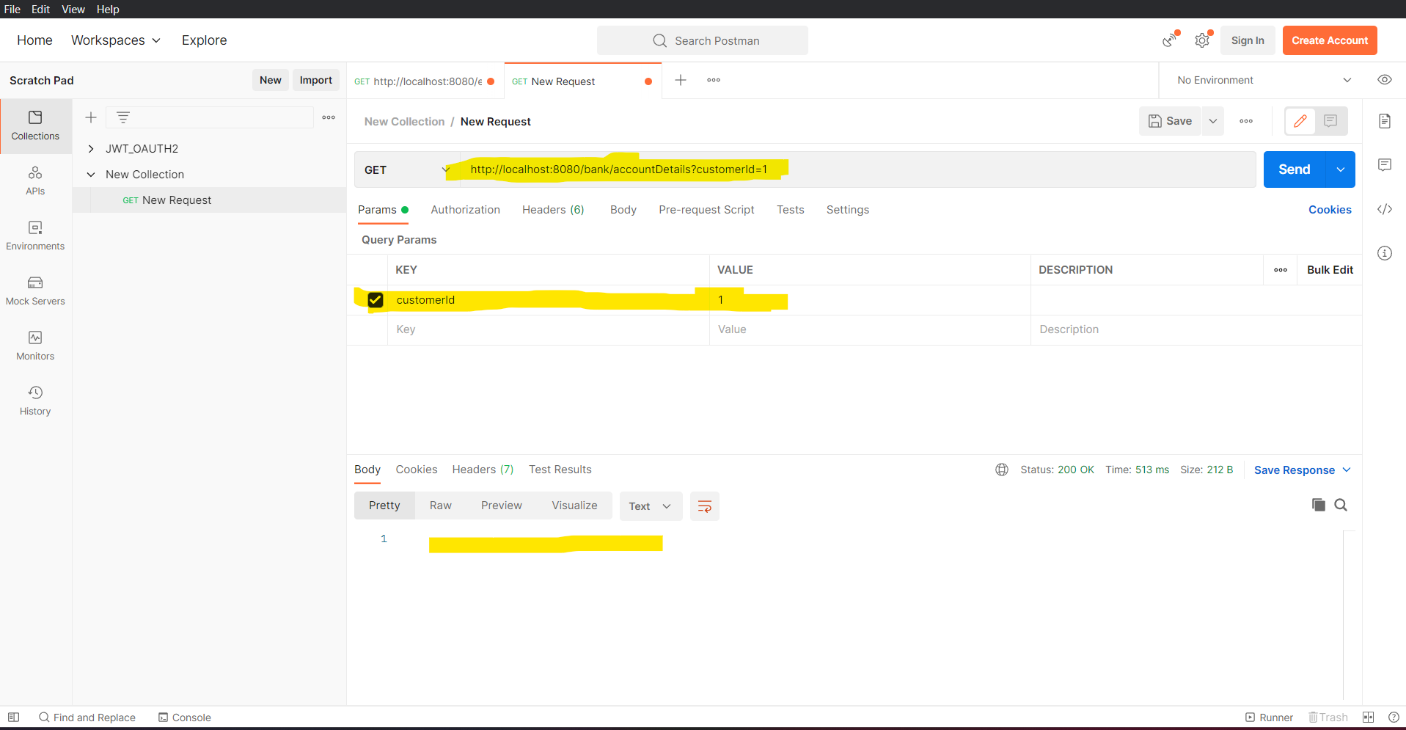
**docker run -e DATABASE\_SERVER=h2:tcp://localhost/~/testdb -dp 8080:8080 bank/account**

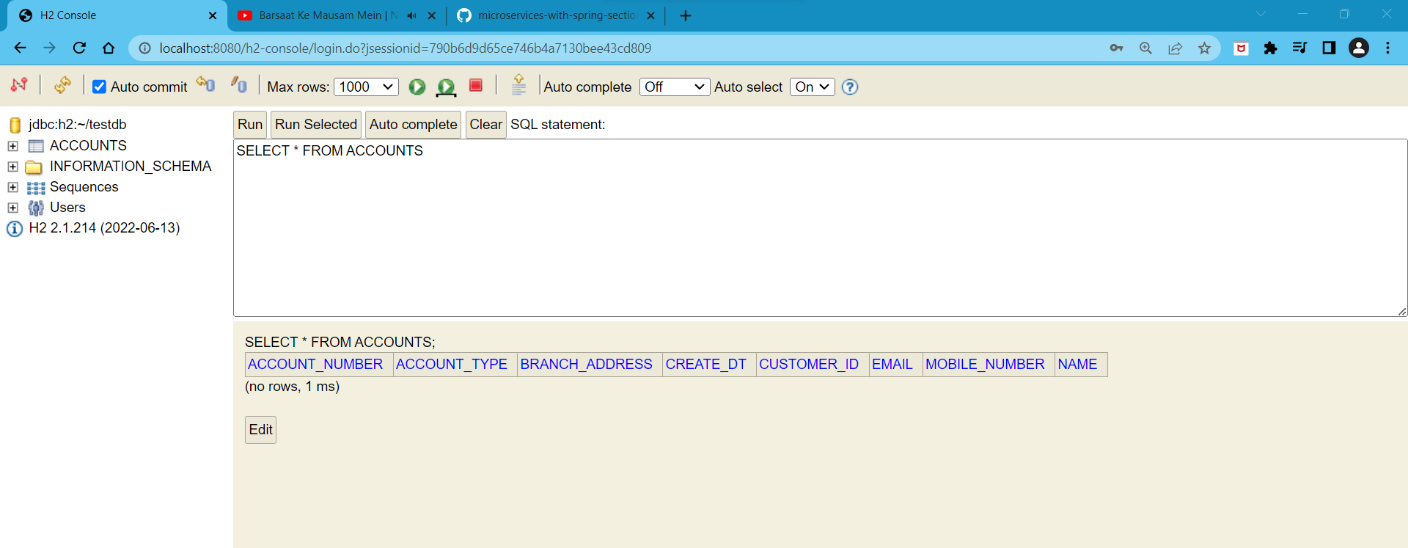
here, **‘-d’** means detach command it will run your container application in background rather than showing you the spring startup console and **‘-p’** means port number on which the application will be hosted in the docker container and mapped to your local system port number as: **<your systems port number>:<docker container port number>**



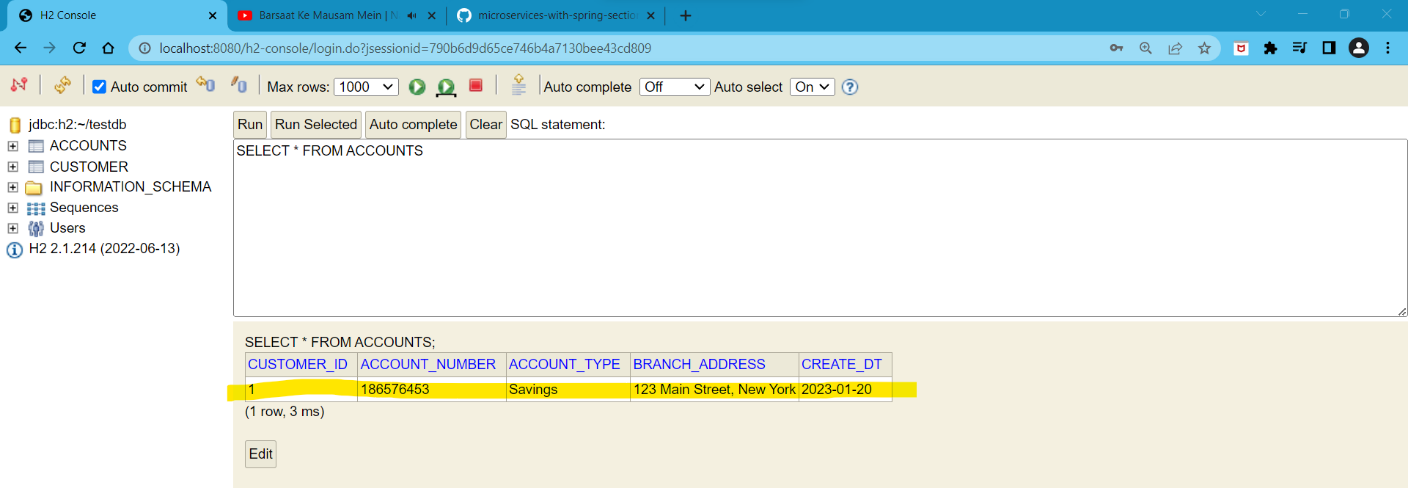
This command will create container for your docker image and start the h2 database server

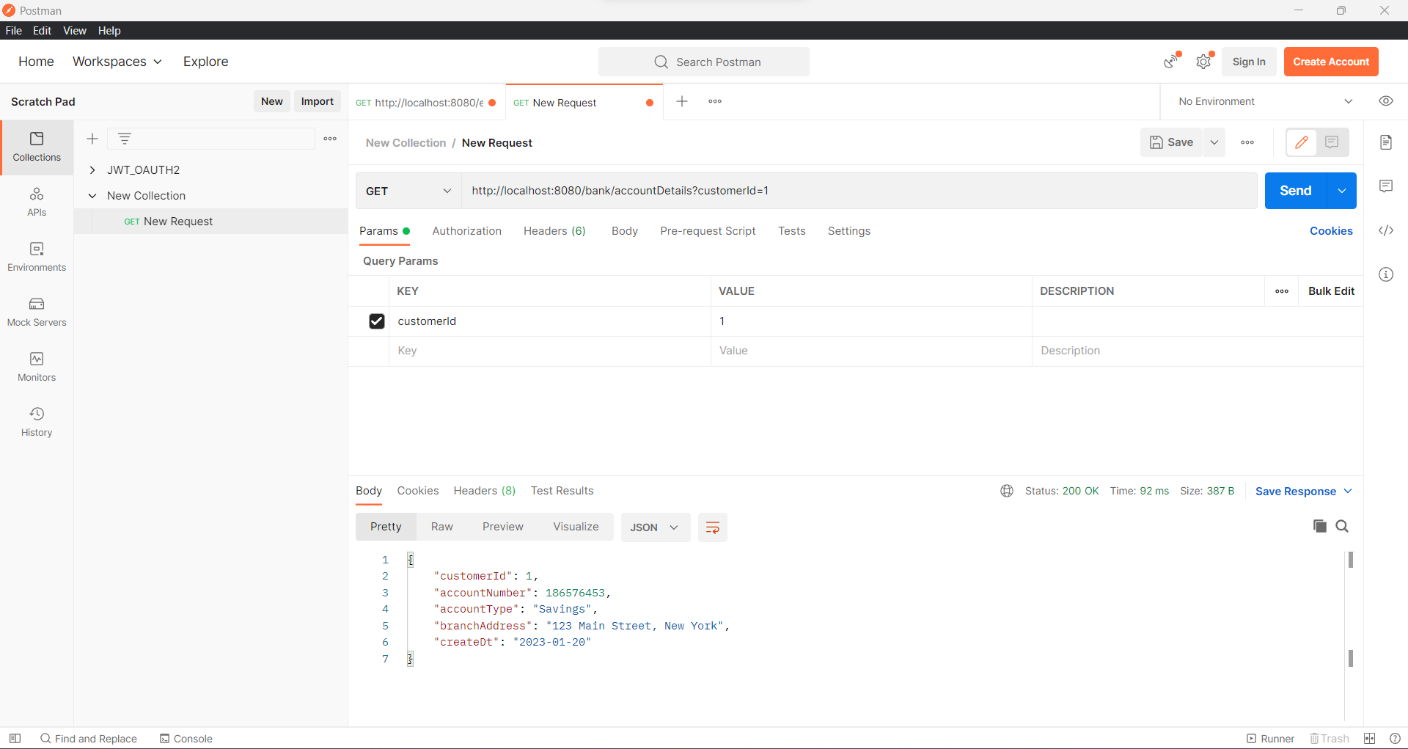
Now test your docker deployed API in postman:

as seen even everything was fine, then also your API didn’t give you any json response. The reason it happend was because in your docker virtual machine the H2 DB with Accounts table was created but with empty data



Now insert data into your H2 container for your Docker container application and now again check in POSTMAN application



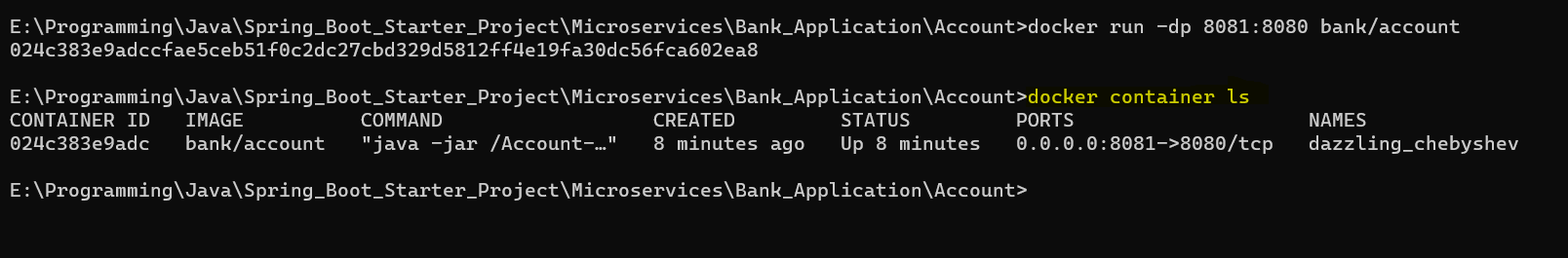


**H4] Docker Commands:**

**H4.1] Get the list of deployed docker containers:**

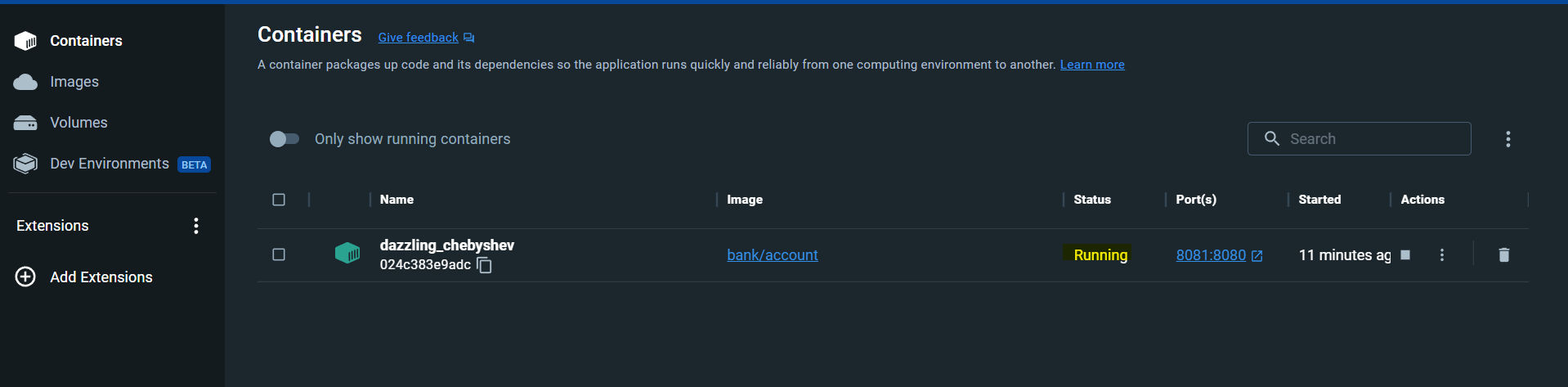
Cmd: 1] **docker container ls** [**see the list of running docker containers**]

2] **docker ps** [**see the list of running docker containers**]



**H4.2] Close the deployed running microservice application container on docker:**

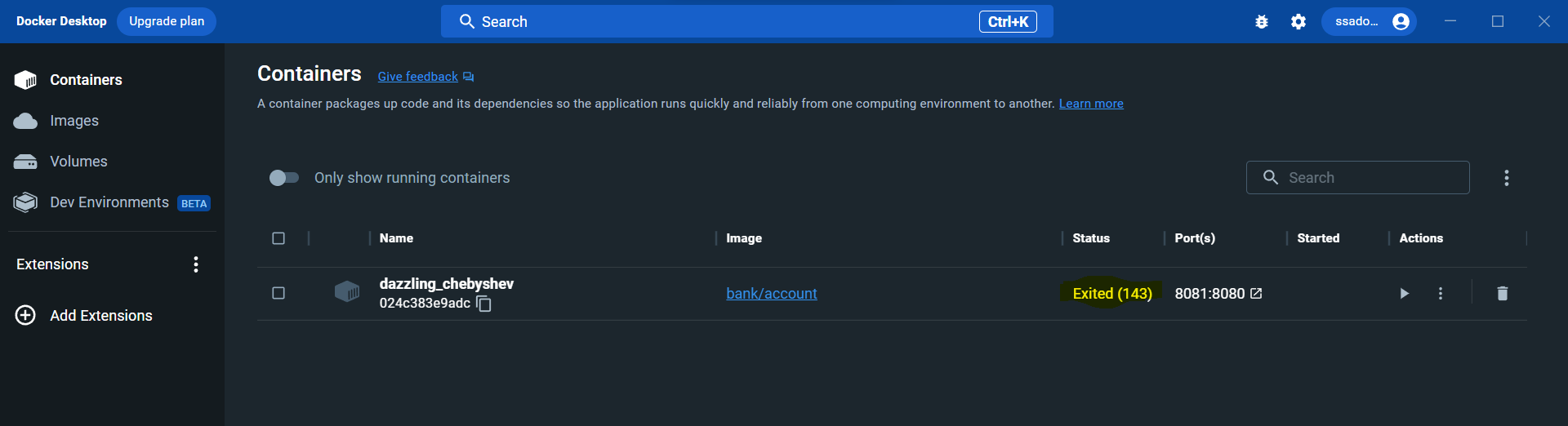
As we can see the docker container is still running:



**Now we will close the running container using command as below:**

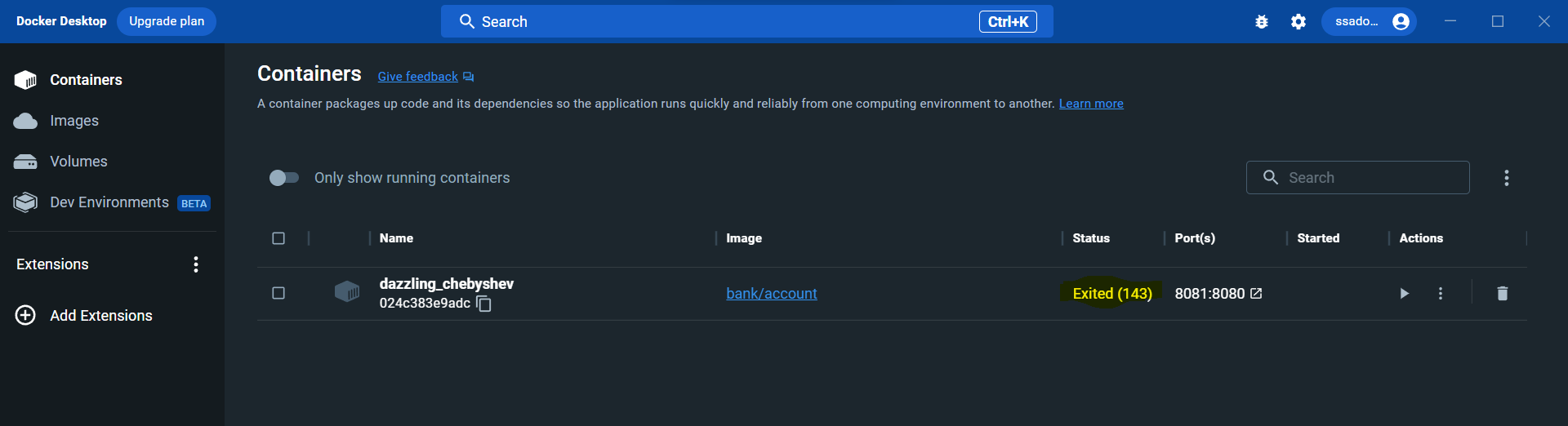
Cmd: **docker stop <Container-Id>**





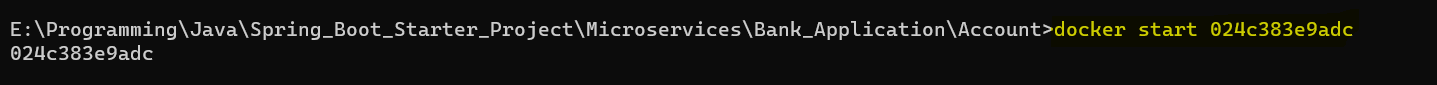
**H4.3] How to again start the existing docker container:**

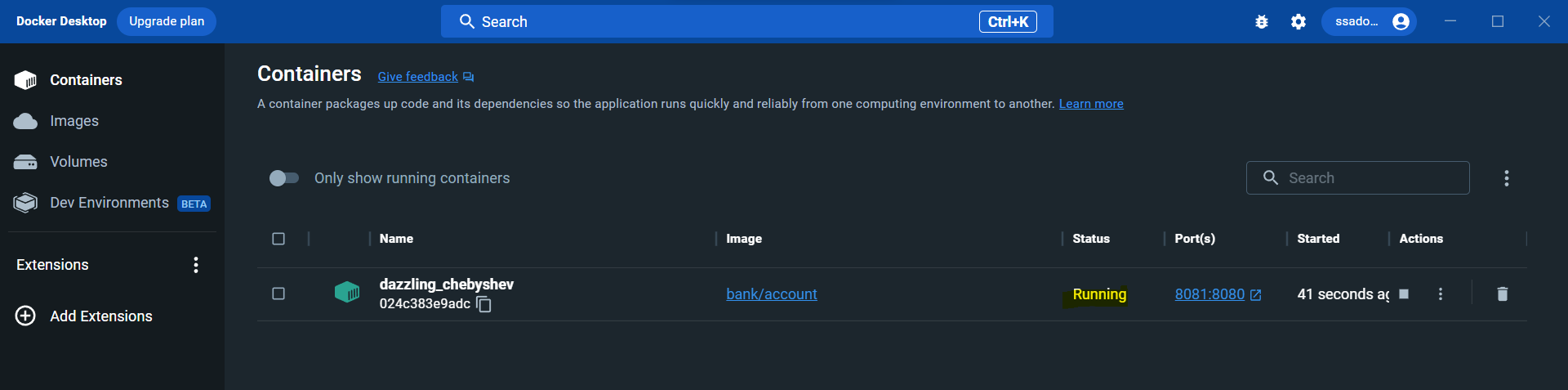
As we can see the existing docker container is now stopped



**Now we will start the closed container using command as below:**

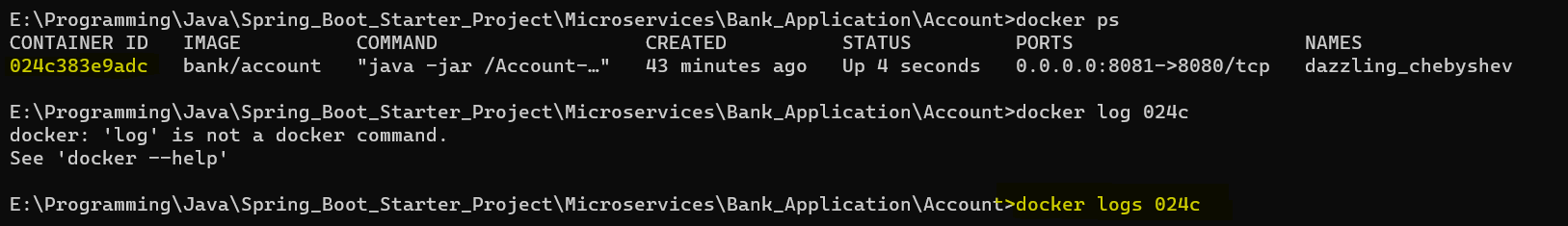
Cmd: **docker start <Container-Id>**





**H4.4] How to get docker container** **logs:**

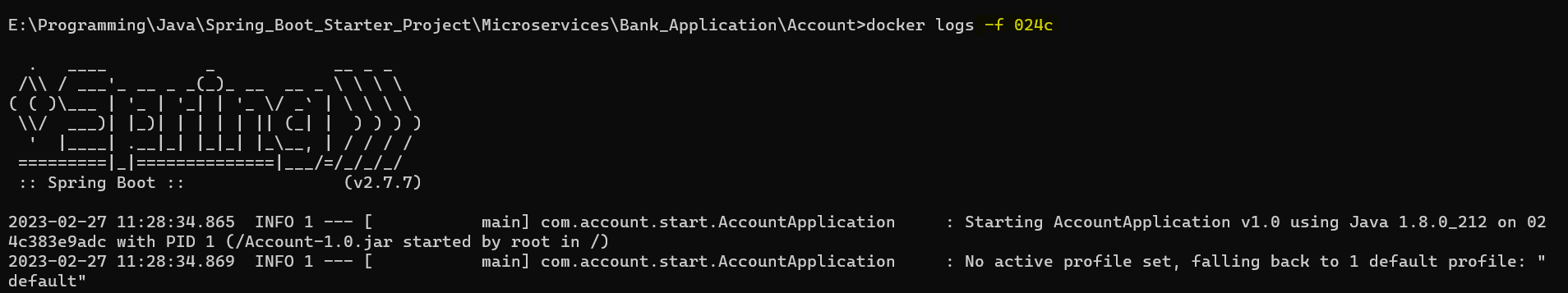
**1] Get static logs of the container:**



Cmd: **docker logs <container id>**

**2] Get dynamic logs of the container:**

Here whenever there will be certain operations on our container or in container application the log values will start to change dynamically

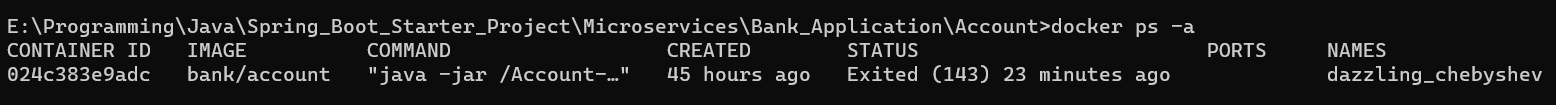


Cmd: **docker logs -f <container id>**

**-f** : Here, ‘ **-f** ’ means follow the container, so log changes dynamically as per the operation on the container

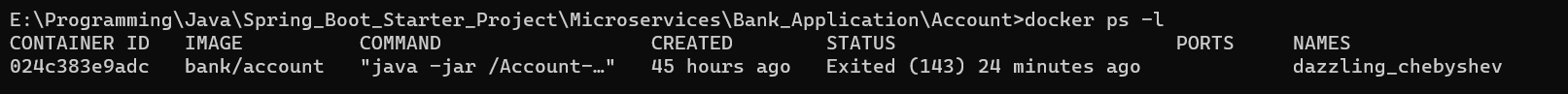
**H4.5] Docker get list of all container whether they are stopped or running:**

Cmd: **docker ps -a**



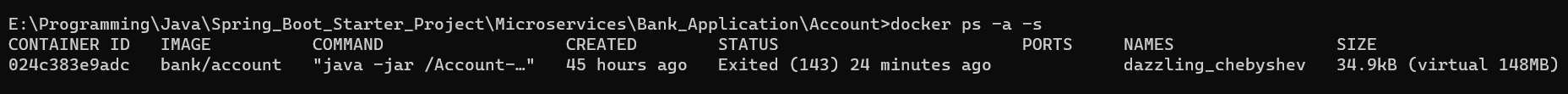
**H4.6] Docker get latest container created details:**

Cmd: **docker ps -a -l**



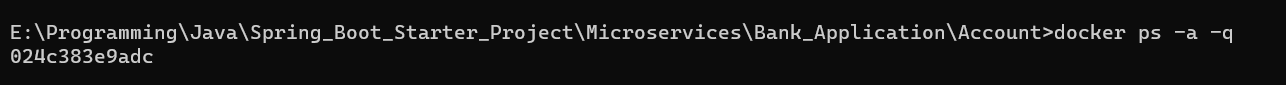
**H4.7] Docker display container size:**

Cmd: **docker ps -a -s**



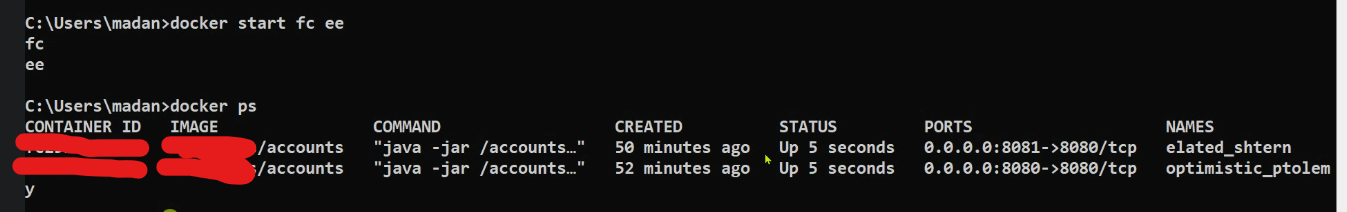
**H4.8] Docker display only container Id’s:**

Cmd: **docker ps -a -q**



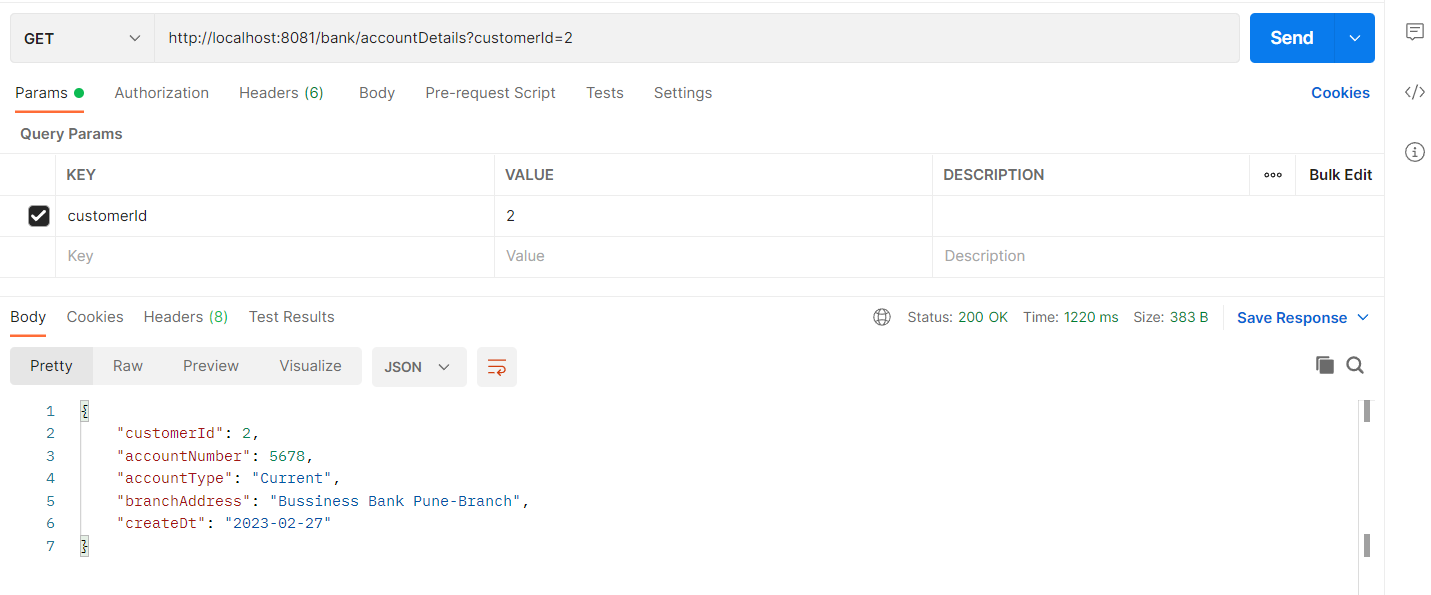
**H4.9] Docker start multiple containers at single time:**

Cmd: **docker start <container-id-1> <container-id-2>**

****

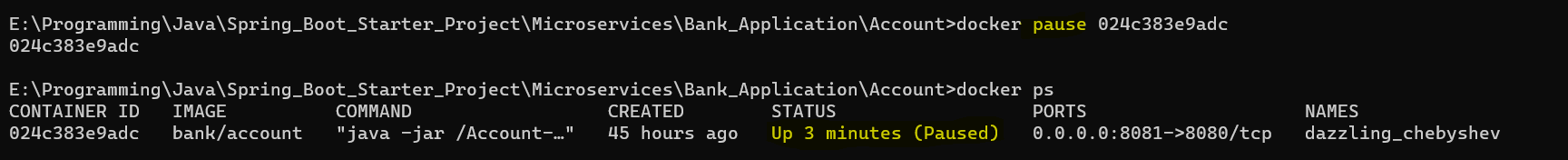
**H4.10] How to stop docker container to accept any request without stopping docker container?**

Here, my docker container is accepting and sending me back response to my request

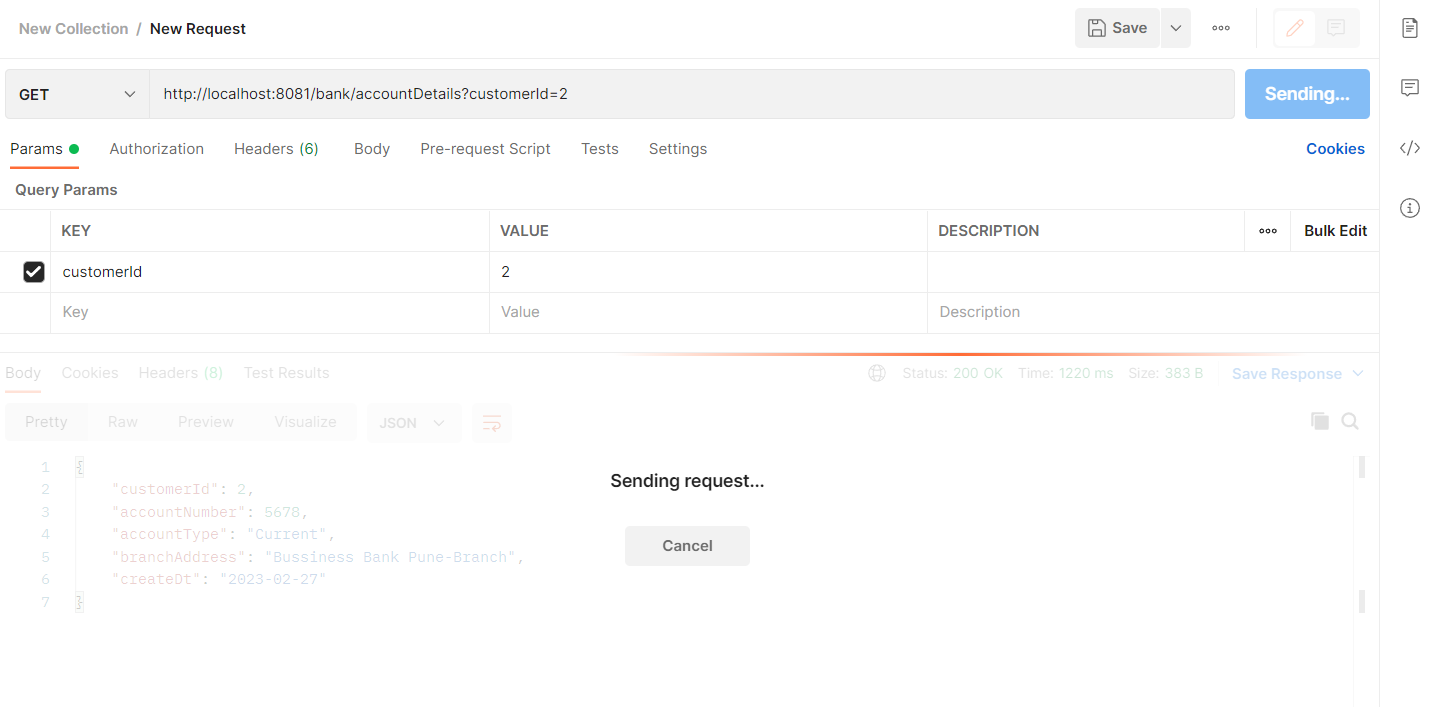


But, now if I want to pause all request’s contained in this container use below command:

Cmd: **docker pause** **<container-id-1>**

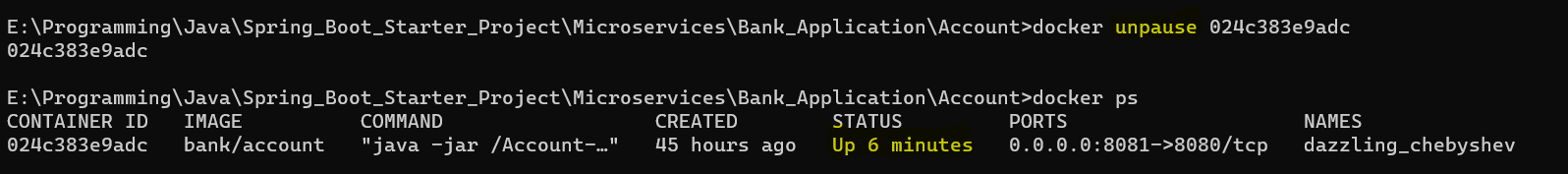


It is not accepting the request



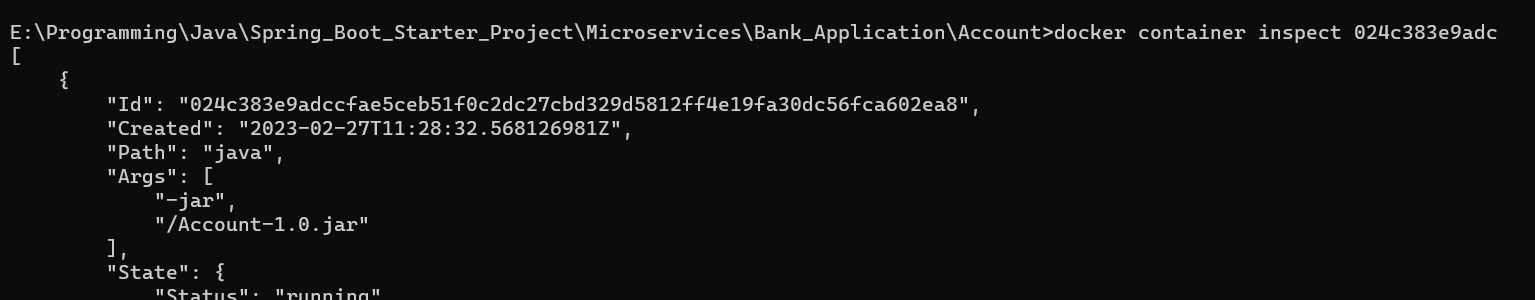
Now to make it work again use the below command:

Cmd: **docker unpause** **<container-id-1>**



**H4.11]** **To get all details of your container use below command:**

Cmd: **docker container inspect <container-id-1>**



**H4.12]** **Difference between docker kill and stop?**

**Docker kill <container-id>** will instantly kill your running docker image without properly shutting it down

**H4.13] How to check our system resource consumption made by all running docker containers?**

Cmd: **docker stats**



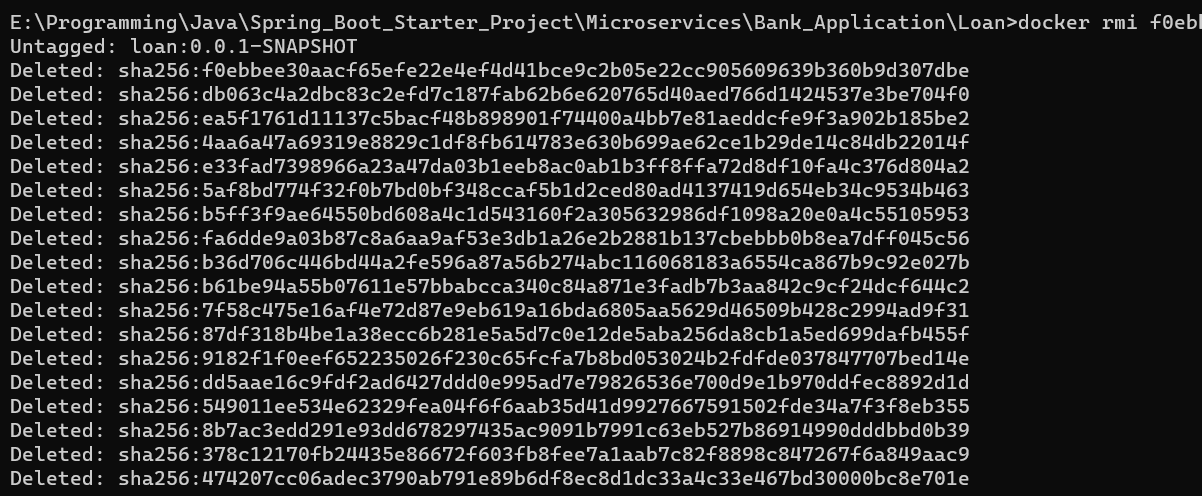
Then press **ctrl+c** to exit from it

**H4.14] how to delete/remove the existing docker container?**

Cmd: **docker rm <container-id>**

**H4.15] how to delete a docker image?**

Cmd: **docker rmi <image-id>**



**H5] Docker Buildpacks:**

**H5.1] What are BuildPacks?**

**H5.1.1]** A Buildpack is a program that turns source code into a runnable container image without precisely defining the steps, as in the case of Dockerfiles. Instead, it detects the language and converts the source code into a runnable container image. There are buildpacks for Ruby, Go, Node. js, Java, Python, and more.

**H5.1.2]** In short the steps that we have performed in **“H3] Creating/Executing a Docker Image and Container”** to manually create docker image is been automated using the Buildpacks

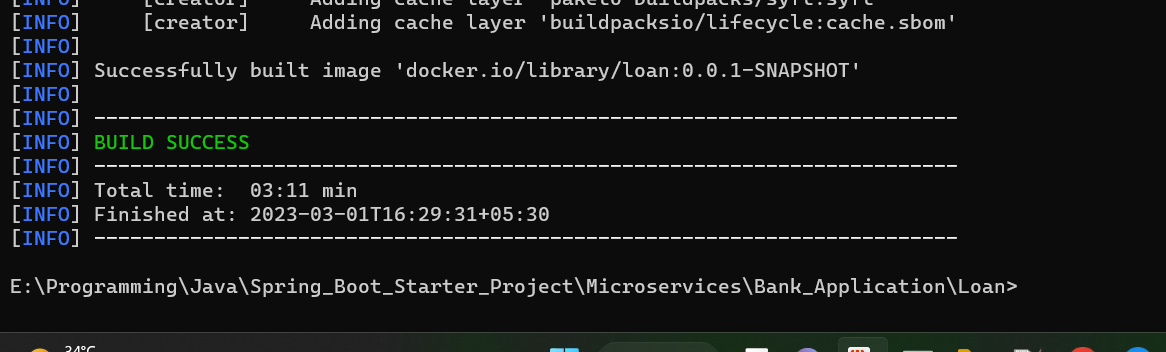
**H5.1.3]** And this Buildpack is a concept like microservices

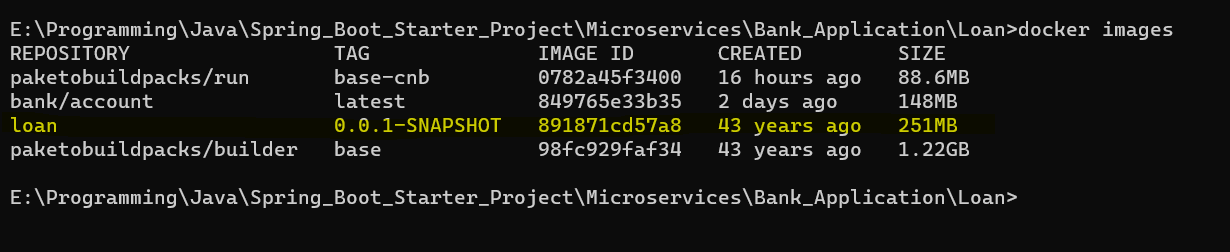
**H5.2] Which buildpack do springboot internally uses?**

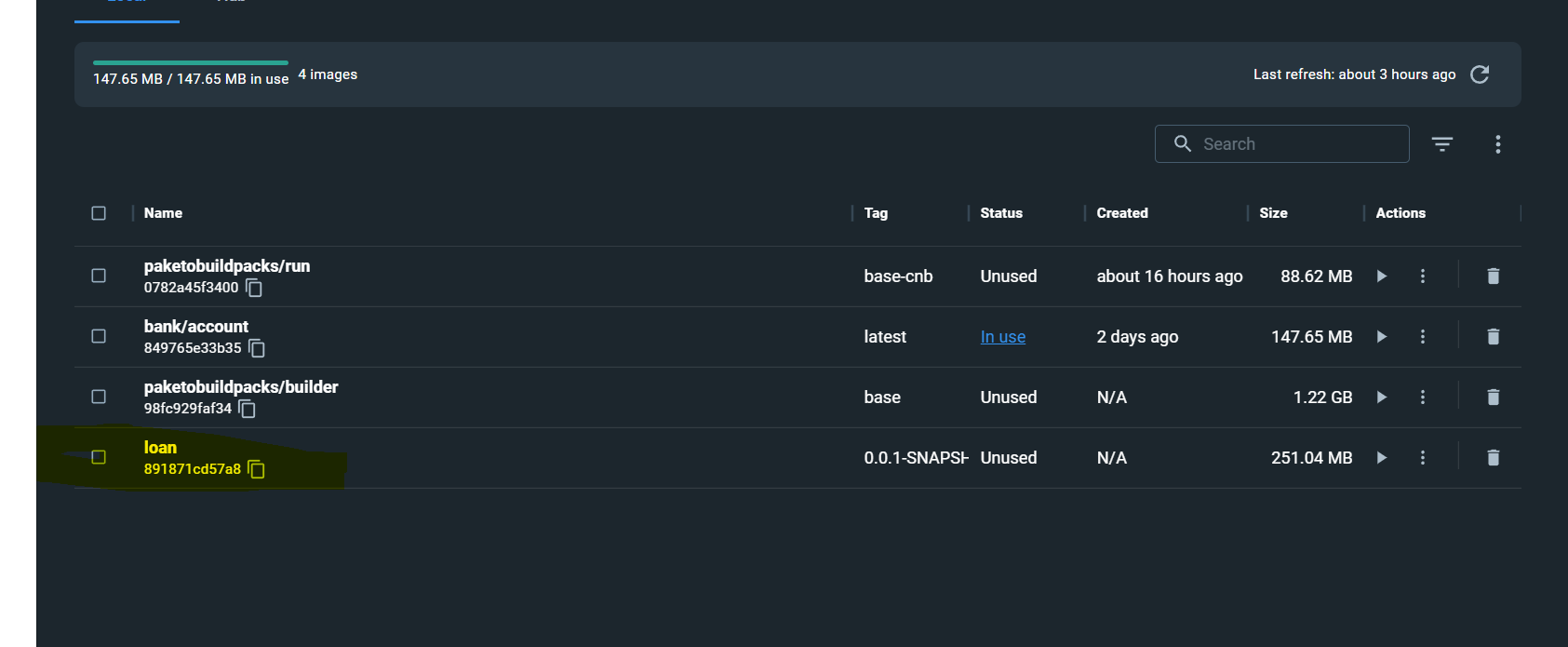
It uses **PACKETO-BUILDPACK**

**H5.3] Command to create docker image using buildpack:**

Cmd: **mvn spring-boot:build-image**







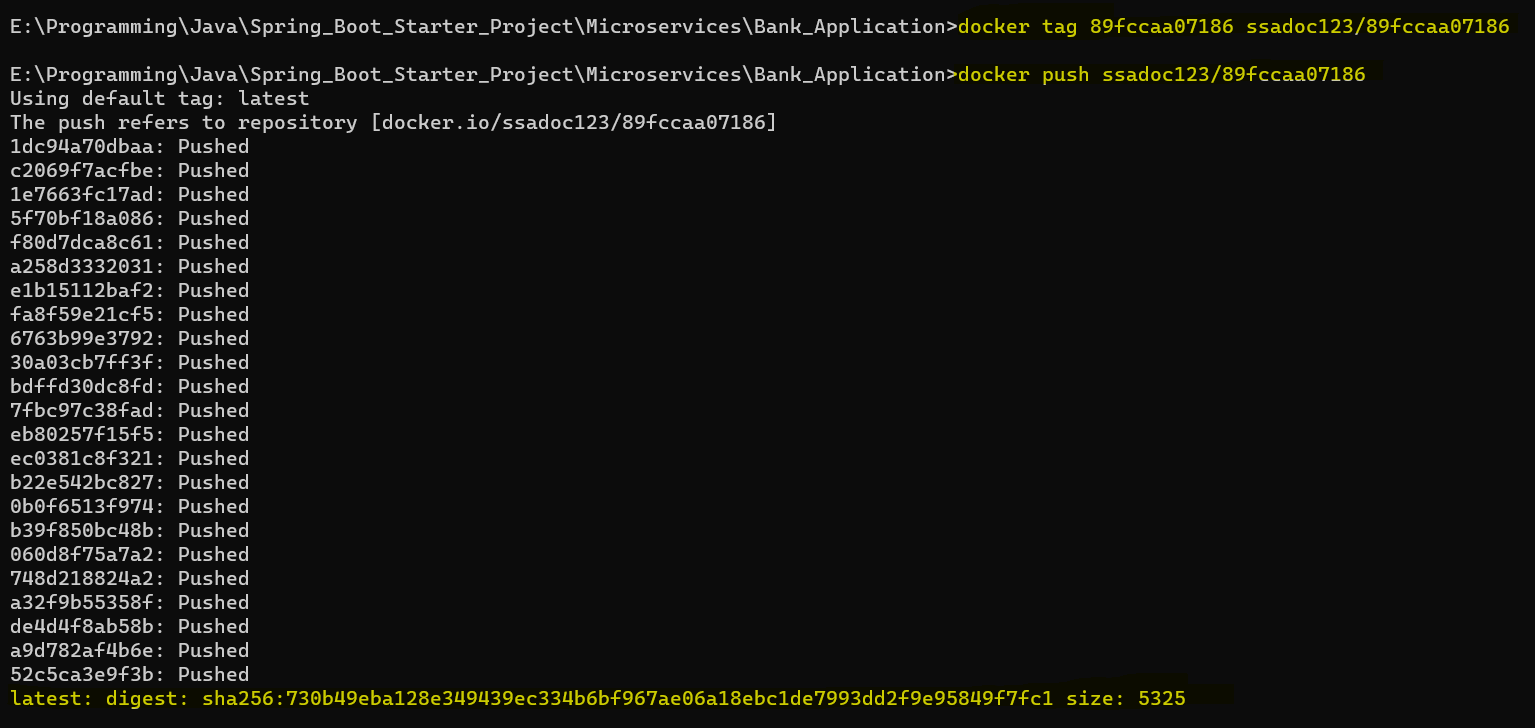
**H6] Pushing Docker images from our Local repository to Docker hub repository**

**Cmd:**

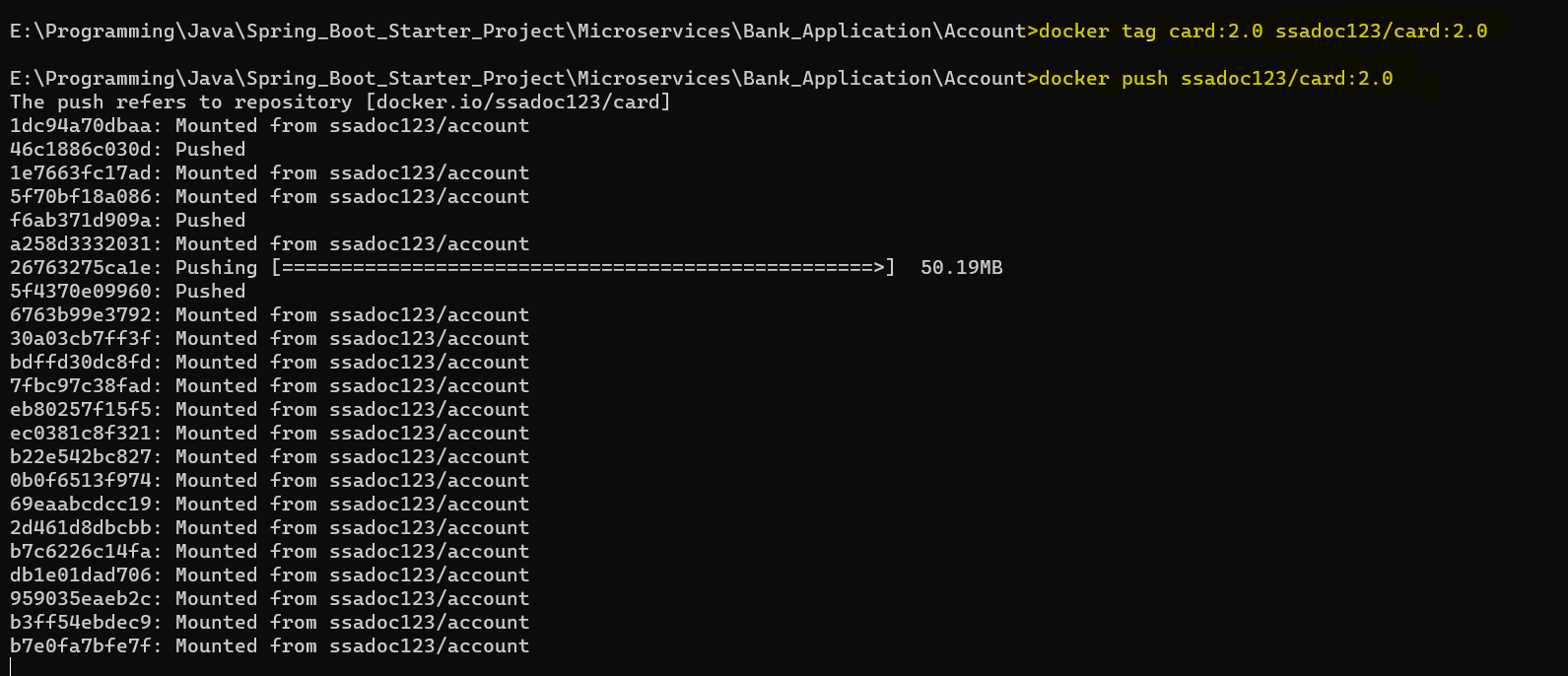
1] docker login docker.io

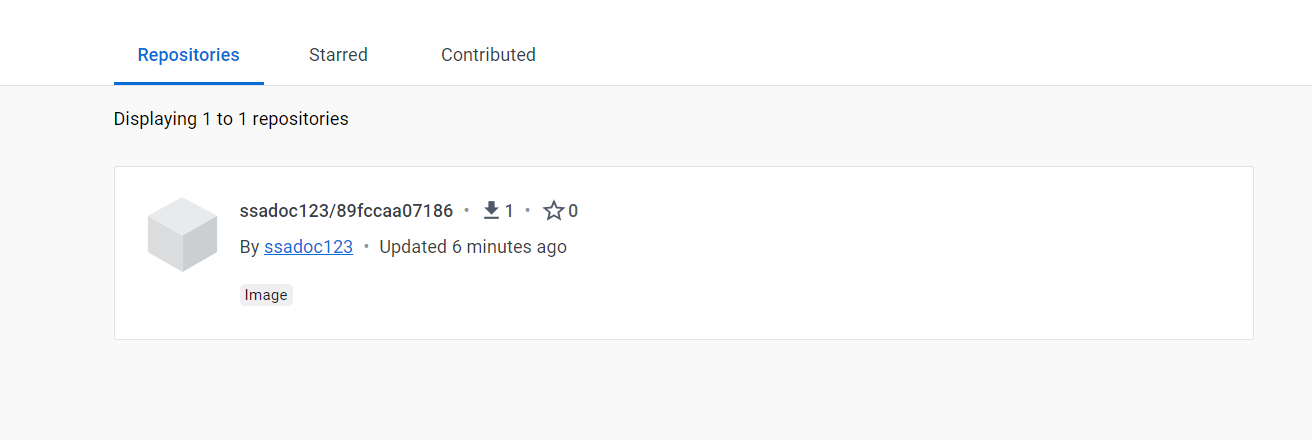
2] docker tag <image-id> **YOUR\_DOCKERHUB\_NAME**/<image-id>

3] docker push **YOUR\_DOCKERHUB\_NAME**/<image-id>



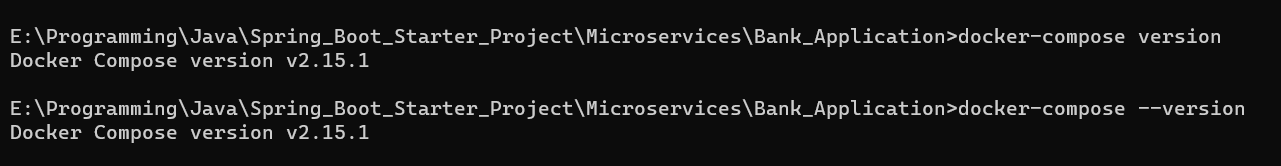
Or



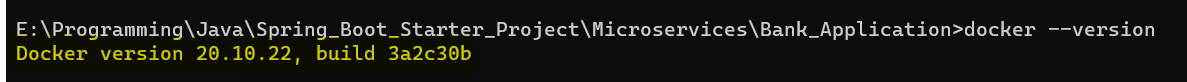


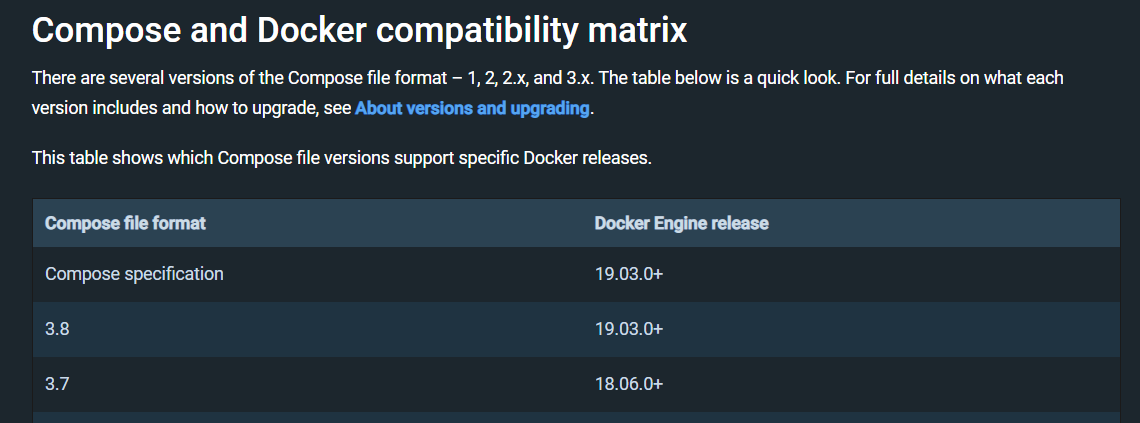
**H7] Docker Compose**

H7.1] Check if docker compose exists or not:



H7.2] Check for docker version





H7.3] Supported docker file name:

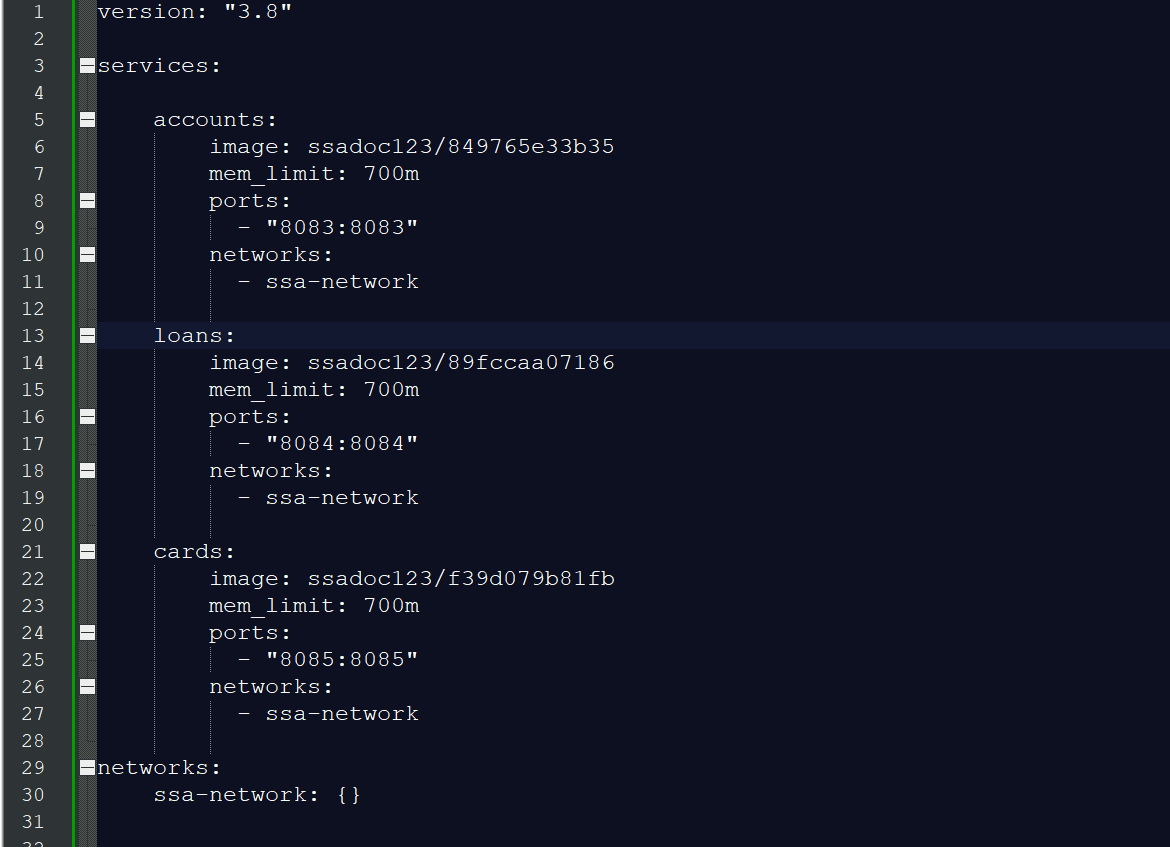
**docker-compose.yml**,

**docker-compose.yaml**,

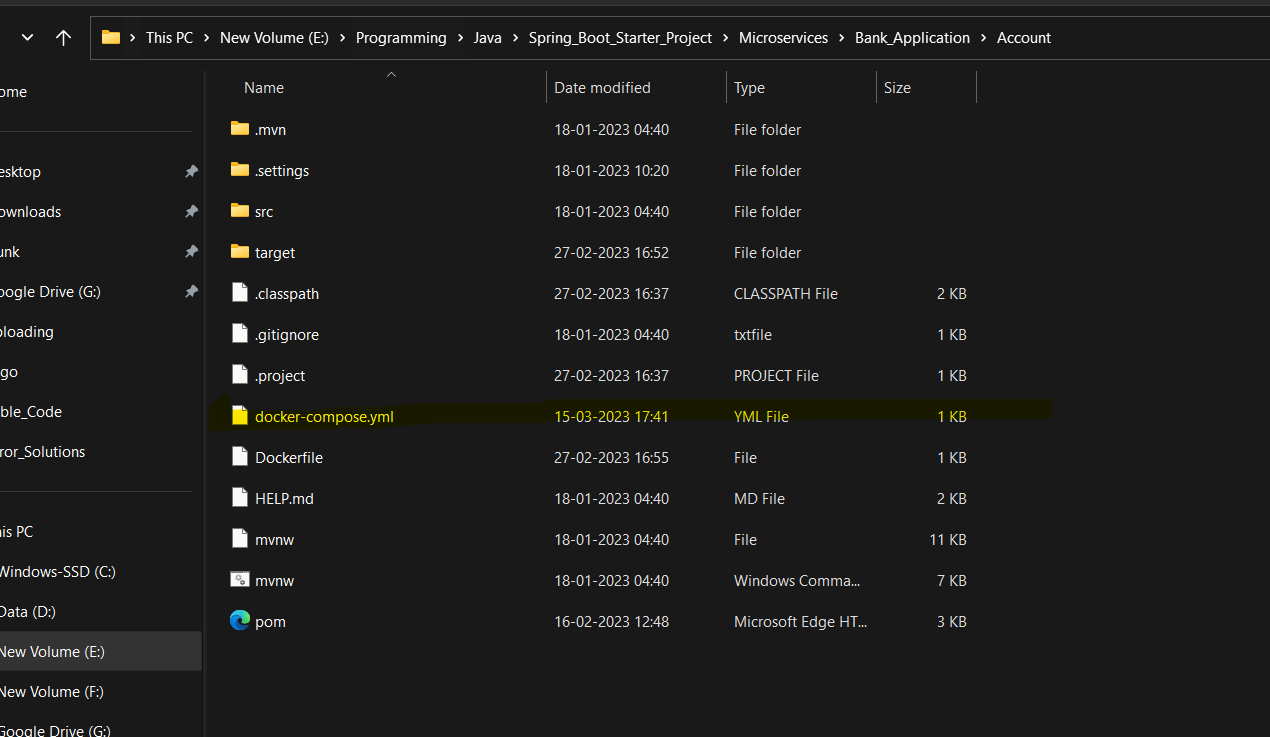
**compose.yml**,

**compose.yaml**

H7.4] Create a docker compose file as **docker-compose.yml**



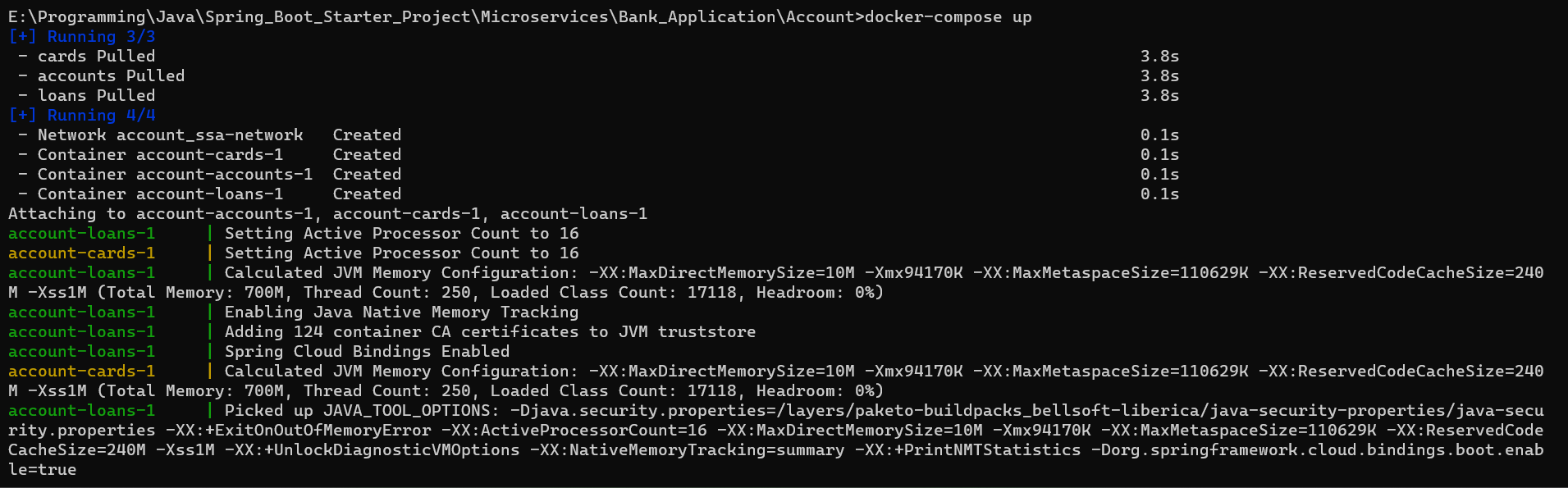
H7.5] Place that compose file in any on project folder as below:



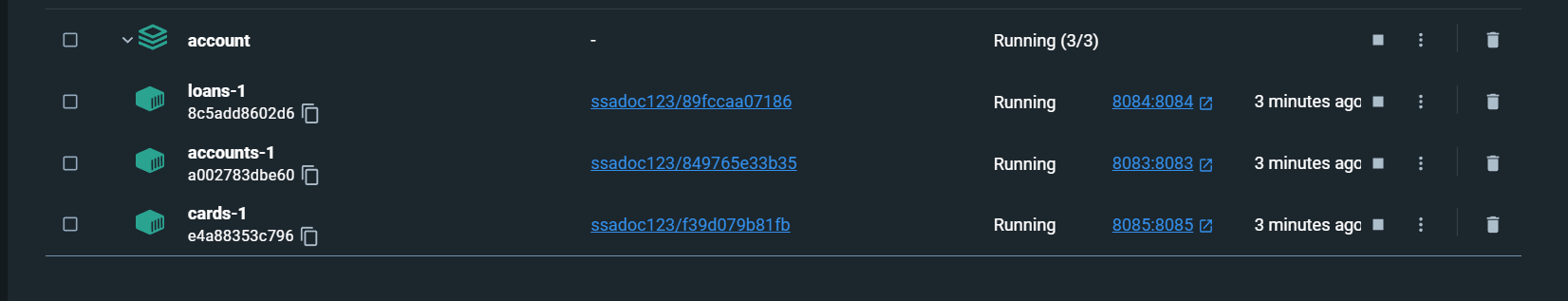
H7.6] Run the below command:

Cmd:

Docker-compose up



Your containers got created and running as below:



H7.7] Stop a docker container:

Cmd:

**docker-compose stop**



H7.8] Start a docker container:

Cmd:

**docker-compose start**

