MICROSERVICES

By:

**Shubham Ashtaputre**

**A]** **MONOLITHIC:**

A1] Monolithic Architecture:

• Single Application

• One Code Base

• One Build System

• Single executional program (ie WAR or EAR file)

• In Enterprise system - an application can become very big

• 10’s of thousands of packages, classes.

A2] Features of Monolithic Architecture:

• Code is stored together

• Typically will use one database

• Code Releases are done as one big version

• Scaling is an all or nothing situation

• If one component needs to increase scale, the whole application needs to scale

A3] Advantages:

• Development is easy - everything is in one project

• Deployment is easy - One app to deploy

• Testing is simplified - One app to test

A4] Disadvantages:

• As the business requirements of Monoliths grow, so does their complexity

• Can lead to anti-patterns - such as Spaghetti Code and Big Ball of Mud design patterns

• Difficult to modify - Even the smallest change will require a full deployment of the application

• Technology Lock In - The monolith becomes tightly coupled to the technology stack

• Difficult to introduce new technologies

• CI/CD difficult

A5] Are Monoliths Bad?

It Depends on much big is your project

**B] MICROSERVICES:**

B1] What Are Microservices?

• Microservices small targeted services

• Each service has its own repository

• Microservices are isolated from other services

• Should not be bundled with other services when deployed

• Microservices are loosely coupled

• When interacting with other services, should be done in a technology agnostic manner

• ie - Restful web services - HTTP / JSON

B2] Microservice Architecture:

• With a Microservice Architecture Applications are composed using individual microservices

• Each service will typically have its own database • Each microservice is independently deployable

• Scaling of individual services is now possible

• CI/CD becomes easier since services are smaller and less complex to deploy

B3] Advantages:

• Easy to understand & develop - Services are smaller and more targeted

• Software Quality - Since services are more targeted and have a limited scope

• Scalability - Independent services can be scaled up and down to the application’s demands.

• Reliability - Software bugs are isolated

• Technology flexibility - Services can be developed using any language or technology stack.

B4] Disadvantages:

• Integration testing can be difficult

• Deployments are more complex. Rather than one application to deploy, you now have many.

• Operational cost with each service - Each service is a small application

• Needs own repo, own deployment process, own database, etc

• Additional hardware resources - Additional services need additional hardware to run on

B5] How ‘Big’ Should a Microservice Be?

• A microservice can be as small as a single API endpoint

• ie - ‘Get Orders’

• A microservice can be several or even dozens of API endpoints

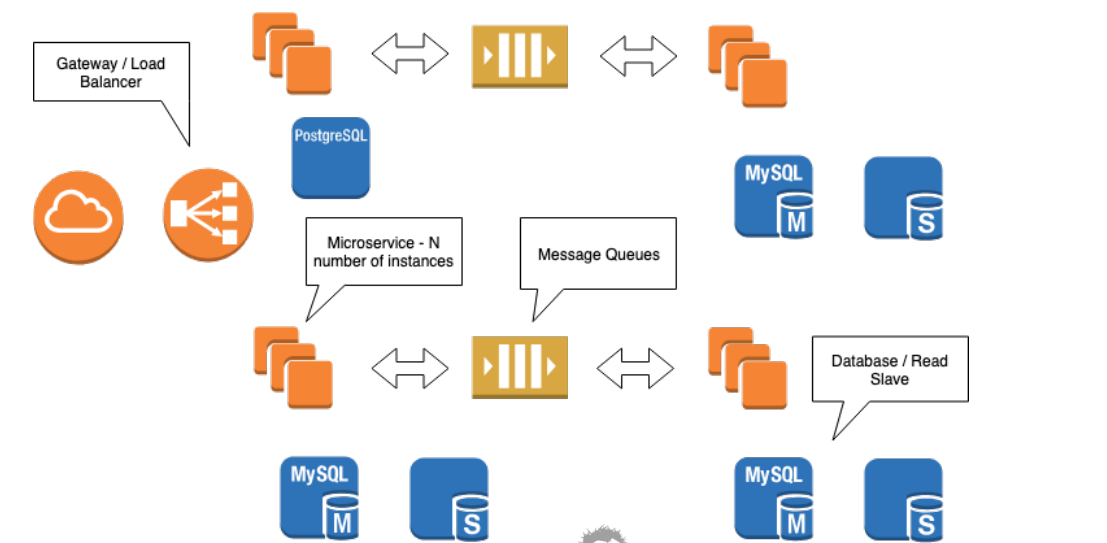
• Answer is a topic of much debate

• Guideline - Amazon’s Two Pizza Team - A microservice should be able to be supported be a team you can fed with two pizzas. (~12 people)

• Scalability - This can also be a consideration in the size of a microservice

• The higher the scalability, the more specialized the service should be

**C] MICROSERVICES ARCHITECTURE:**



C1] Gateway:

• Endpoint that is exposed to other services

• Can be internet for public APIs

• More likely to be internal

• Abstracts implementation of services

• Client calls URL, is unaware of routing taking place to running instance

• Acts as roughly a proxy for network traffic

• Can also act as a load balancer

C2] Service Instances:

• Expect to be running N number of services

• Exact number depends on reliability and load requirements

• Minimum might be 3, for high availability

• Some tools allow you to dynamically scale based on load or anticipated load

• Think Netflix at night

• In evenings, Netflix traffic is 1/3 of US internet traffic

• Netflix will scale up and down with load

C3] Database Tier:

• Typically one database per microservice

• Guideline - not a hard ‘rule’

• Highly scalable services will often have one transactional database

• And one or more read database (replicas)

• Organizations will often have more than one database technology

• Not uncommon to see mix of SQL and NoSQL database technologies

C4] Messaging:

• A common pattern is to expose an API endpoint via a RESTFul API

• Dependent microservices are often message based

• Messages follow an event or command pattern

• Messaging allows for decoupling and scalability

• Messaging can be used to define a work flow

• New Order, Validate Order, Charge Credit Card, Allocate Inventory, Ship Order

C5] Downstream Services:

• Often an action on a microservice will invoke actions on multiple downstream services

• For example, it is rumoured a search on Amazon will invoke over 100 services to return the search results - search, sponsors, your history, logging your search, etc

• Placing a new order might invoke the following:

• Validate Order • Pay Credit Card • Allocate Inventory • Ship Order

**D] ADOPTING MICROSERVICES IN PROJECT:**

D1] Adopting Microservices:

•Often applications will start as monoliths

• Might be because of being older legacy applications

•Or a development choice

• Remember there is a ‘cost’ to splitting into microservices

•It’s not uncommon to start development of an application as a monolith

• Monolithic architectures are well established in companies

• Many companies are just starting to adopt Microservices

D2] Decomposing to Services:

• Decomposing is the process of taking a larger monolithic application and breaking it up into microservices

• Decomposition is more of an ‘art’ than a science

•Strategies you can use:

•By Business Capability - ie Order Service

•By Domain Objects - ie Product Service (services over domain object ‘Product’)

•By action verbs - Payment Service

•By Nouns - Customer Service

D3] Single Responsibility Principle:

• Single Responsibility Principle (SRP) is a term coined by Uncle Bob Martin about object oriented programming.

• SRP says a class should have just one reason to change.

• Meaning your classes should be very specific in what they do

• Do one thing, and do it very well

• SRP can also be applied to microservices

• Do one thing, and do it very well

D4] Microservices and Development Teams:

• Larger organizations might have hundreds of developers

•When possible small teams should be responsible for specific microservices

•This will often lend itself to business functions

•An account team would work on accounting related services

•An Customer Order team would work on Customer Order related services

•An Order Fulfillment team would work on Order Fulfillment related services

•Often you will see a lot of overlap of business domain with the domain of the services

**E] CLOUD:**

E1] What is the Cloud?

• The Cloud is not a physical object, but more of a concept.

• The ‘Cloud’ allows you to use virtual servers and services.

• The ‘Cloud’ abstracts the physical underlying hardware and services

• Amazon Web Services for example:

• Allows provisioning in zones - a physical region made of many data centers, appearing as one

E2] Cloud != Virtual Machines:

• Virtual Machines are easy to provision in cloud environments

• SFG Website runs on a AWS Virtual Machine

• This is not ‘Cloud’ Computing

• Virtualization is not Cloud Based Architecture

• Subject of heated debates!

E3] Cloud Based Architectures:

• Microservices are a key aspect of Cloud Based Architectures

• Cloud based architectures focus on abstraction, redundancy, and avoidance of single point of failures

• For example saving a file to AWS S3

• File is copied to multiple servers, and in multiple data centers before the ‘save’ is confirmed.

• Thus protected from server failure, and even loss of a data center.

E4] Microservices in Cloud Based Architectures:

• Typically multiple instances of microservices are deployed in a cloud environment

• Important to reliability of the application as a whole

• If a service instance is terminated, another running instance can assume the workload

• Netflix as a tool called Chaos Monkey who’s job is to randomly terminate components to ensure there are no single points of failure

E5] Common Microservice Deployment Tools:

• This is very large and diverse area!

• AWS Beanstalk

• AWS ECS / EKS

• Kubernetes

• Docker Swarm

• Red Hat OpenShift

• Cloud Foundry

**F] HTTP:**

• Development of HTTP was started by Tim Berners-Lee of CERN in 1989

• HTTP/0.9 is the Original HTTP proposal by Tim Berners-Lee

• Started as a telnet friendly protocol

HTTP/1.0 - From 1991 to 1995 the HTTP/HTML specifications grew rapidly

• New software known as a “web browser” emerged

• HTTP standards were developed by:

• IETF - Internet Engineering Task Force

• W3C - World Wide Web Consortium

**F1] HTTP/1.1:**

• HTTP/1.1 - Originally released in 1997

• Solved a lot of ambiguities from earlier versions

• Added support for keep alive connections, chunked encoding transfers, byte-range requests, transfer encodings, and request pipelining

• HTTP/1.1 - Updated by RFC 2616 in 1999

• Updated again by RFC 7230 in 2014

• Still in use today

**F2] HTTP/2.0:**

• HTTP/2.0 Standardized in 2015

• Supported by most servers and browsers by the end of 2015

• As of May 2017, only 13.7% of the top 10 million websites supported HTTP/2.0

• Has high level of compatibility with HTTP/1.1

• Transport Performance was a focus of HTTP/2.0

• Improves page load speed by:

• Lower Latency

• Higher Throughput

• Differences from HTTP/1.1 are largely transparent for web developers

**F3] HTTP request methods:**

Request methods, also known as verbs, are used to indicate the desired action to be performed

• **GET** - is a request for a resource (html file, javascript file, image, etc) is used when you visit a website.

• **HEAD** - is like GET, but only asks for meta information without the body.

• **POST** - is used to post data to the server. Typical use case for POST is to post form data to the server (like a checkout form)

• **PUT** - is a request for the enclosed entity be stored at the supplied URI. If the entity exists, it is expected to be updated.

• **POST** is a create request

• **PUT** is a create OR update request

• **DELETE** - Is a request to delete the specified resource

• **TRACE** - Will echo the received request. Can be used to see if request was altered by intermediate servers

• **OPTIONS** - Returns the HTTP methods supported by the server for the specified URL

• **CONNECT** - Converts the request to a transparent TCP/IP tunnel, typically for HTTPS through an unencrypted HTTP proxy

• **PATCH** - Applies partial modifications to the specified resource

**F4] HTTP Safe Methods:**

• Safe Methods are considered safe to use because they only fetch information and do not cause changes on the server

• The Safe Methods are: GET, HEAD, OPTIONS, and TRACE

**F5] Idempotent Methods:**

• Idempotence - A quality of an action such that repetitions of the action have no further effect on the outcome

• PUT and DELETE are Idempotent Methods

• Safe Methods (GET, HEAD, TRACE, OPTIONS) are also Idempotent

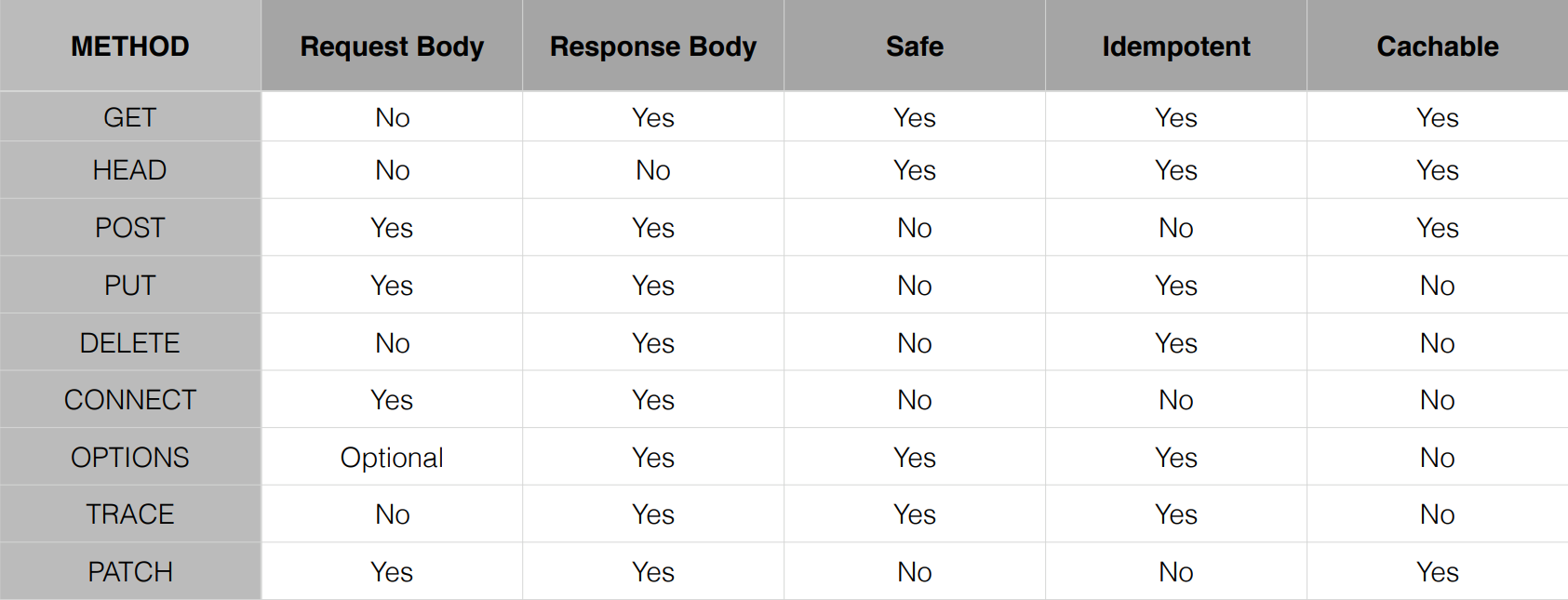
• Being truly Idempotent is not enforced by the protocol

**F6**] **Non-Idempotent Methods:**

• POST is NOT Idempotent

• Multiple Posts are likely to create multiple resources

• Ever seen websites asking you to click submit only once?

****

**F7] HTTP Status Code:**

• 100 series are informational in nature

• 200 series indicate successful request

• 300 series are redirections

• 400 series are client errors

• 500 series are server-side errors

**F8] Common HTTP Status Codes:**

• 200 Okay; 201 Created; 204 Accepted

• 301 Moved Permanently

• 400 Bad Request; 401 Not Authorized; 404 Not Found

• 500 Internal Server Error; 503 Service Unavailable

**G] REST:**

• Because of their simplicity and versatility, RESTful web services have become the de facto standard for web services.

• REST - Representational State Transfer

• Representation - Typically JSON or XML

• State Transfer - Typically via HTTP

• Established by Roy Fielding from his 2000 doctoral dissertation

**G1] RESTful Terminology:**

• Verbs - HTTP Methods: GET, PUT, POST, DELETE

• Messages - the payload of the action (JSON/XML)

• URI - Uniform Resource Identifier

• A unique string identifying a resource

• URL - Uniform Resource Locator

• A URI with network information - http://www.example.com

• Idempotence - • Wikipedia “Idempotence is the property of certain operations in mathematics and computer science that they can be applied multiple times without changing the result beyond the initial application.” In other words, you can exercise the operation multiple times, without changing the result. **Example**: Refreshing a web page (HTTP GET operation)

• Stateless - Service does not maintain any client state

• HATEOAS - Hypermedia as The Engine of application State

• Wikipedia - “a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other actions that are currently available.”

**G2] Richardson Maturity Model (RMM):**

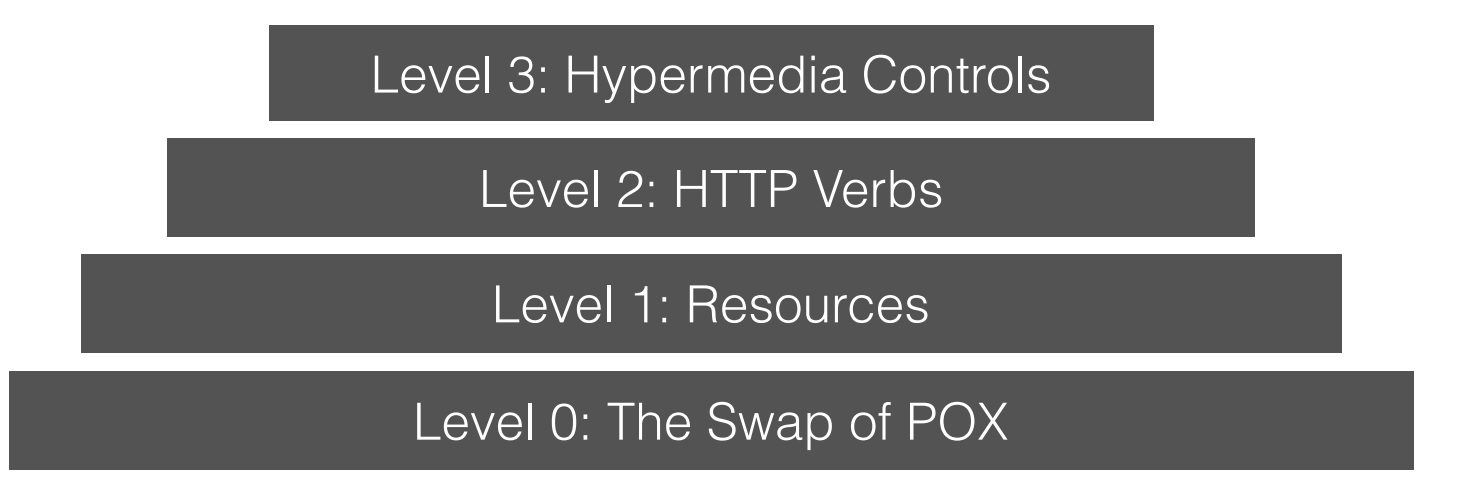
• Established by Leonard Richardson in a 2008 Q-Con Presentation

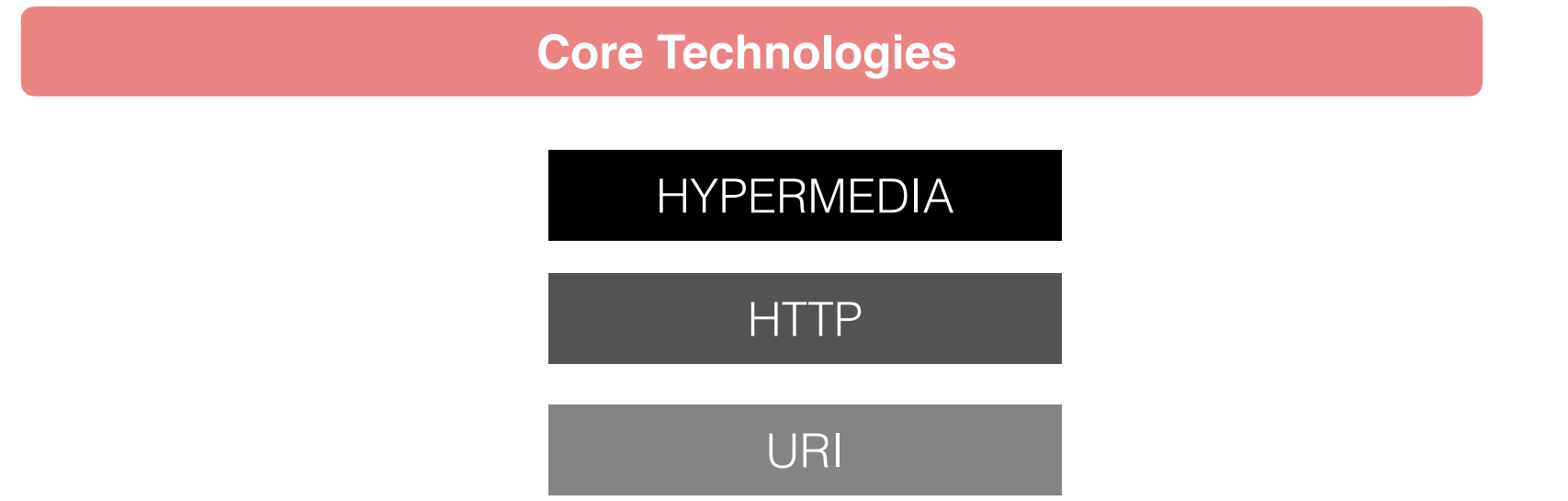
• A model used to describe the maturity of RESTful services

• Unlike SOAP, there is no formal specification for REST

• RMM is used to describe the quality of the RESTful service

**G2.2] RMM Levels:**





**Level 0: Swamp of POX:**

• POX - Plain Old XML

• Uses implementing protocol as a transport protocol

• Typically uses one URI and one kind of method

• Examples - RPC, SOAP, XML-RPC

**Level 1: Resources:**

• Uses Multiple URIs to identify specific resources

• Examples:

• <http://www.example.com/product/1234>

• <http://www.example.com/product/5687>

• Still uses a single method (ie GET)

**Level 2: HTTP Verbs**

HTTP Verbs are used with URIs for desired actions

• Examples:

• GET /products/1234 - to return data for product 1234

• PUT /products/1234 (with XML body) to update data for product 1234

• DELETE /products/1234 to delete product 1234

• Most common in practical use

**Level 3: Hypermedia**

• Representation now contains URIs which may be useful to consumers

• Helps client developers explore the resource

• No clear standard at this time

• Spring provides an implementation of HATEOS

**Summary:**

• Level 1 - breaks large service into distinct URIs

• Level 2 - Introduces Verbs to implement actions

• Level 3 - provides discoverability, making the API more self-documenting

**G3] Spring Framework and RESTful Services:**

• The Spring Framework has very robust support for creating and consuming RESTFul Web Services

• Spring Framework has 3 Distinct libraries for creating RESTful services

• Spring Framework has 2 Distinct libraries for consuming RESTful services

• There are also several popular libraries for creating and consuming RESTful services frequently used with Spring

**G4] Spring MVC:**

• Spring MVC is the oldest and most commonly used library for creating RESTful web services

• MVC - Model View Controller

• Can also be used for Web Applications

• Course focus will be just on RESTFul web services

**G5] Spring WebFlux:**

• Spring WebFlux was introduced with version 5 of the Spring Framework

• WebFlux uses project Reactor to provide reactive web services

• Follows very closely to the configuration model of Spring MVC

**G6] Functional WebFlux - aka WebFlux.fn:**

• Also introduced in Spring Framework 5

• WebFlux.fn is a functional programming model used to define endpoints

• Alternative to annotation based configuration

• Designed to rapidly and simply define microservice endpoints

**G7] Spring RestTemplate:**

RestTemplate is Spring’s primary library for consuming RESTFul web services

• Very mature - been a part of Spring for a very long time

• Highly configurable

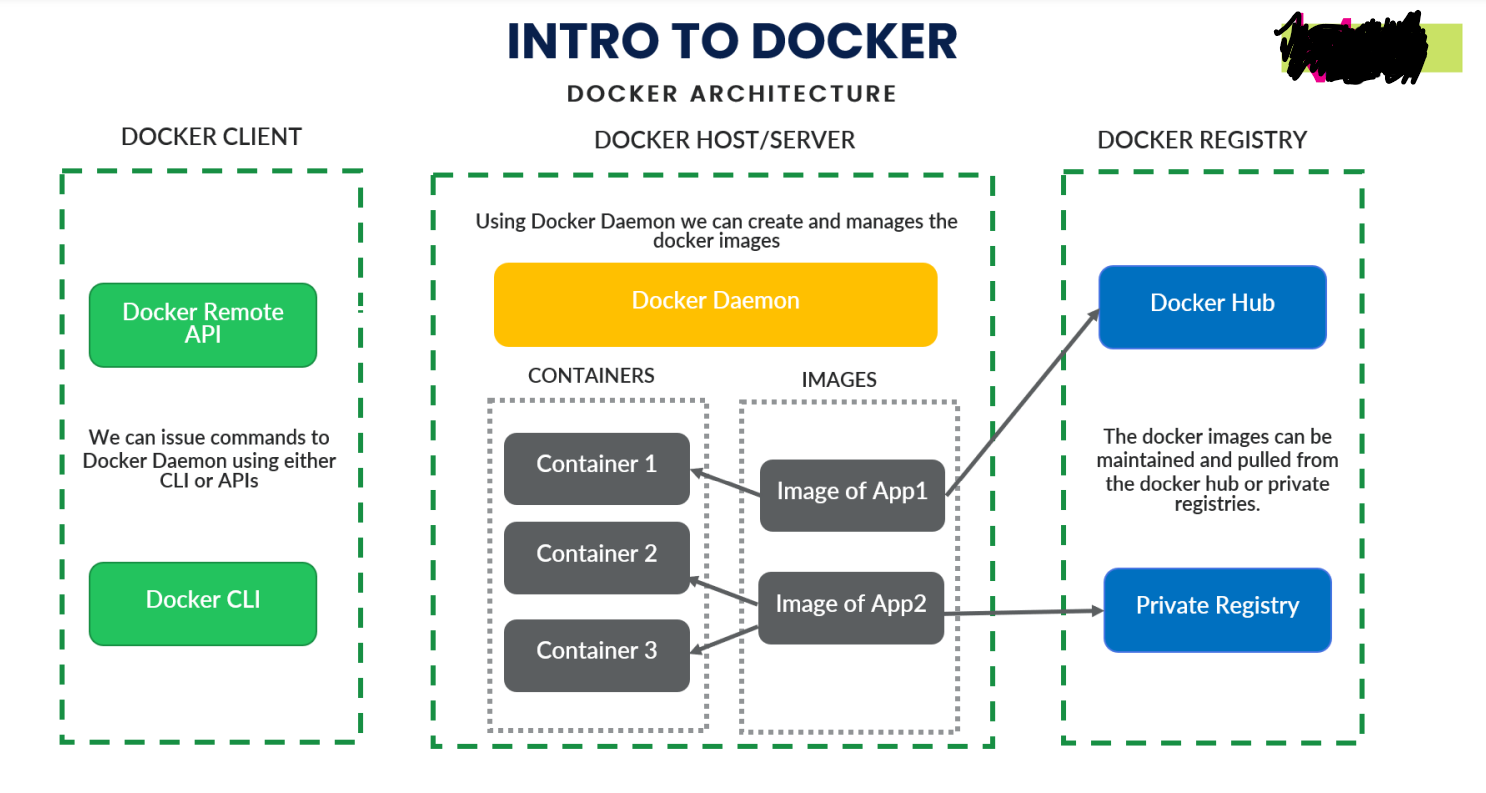
**G8] Spring WebClient:**

• Spring WebClient was introduced in Spring Framework version 5

• This is Spring’s reactive web client.

**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:

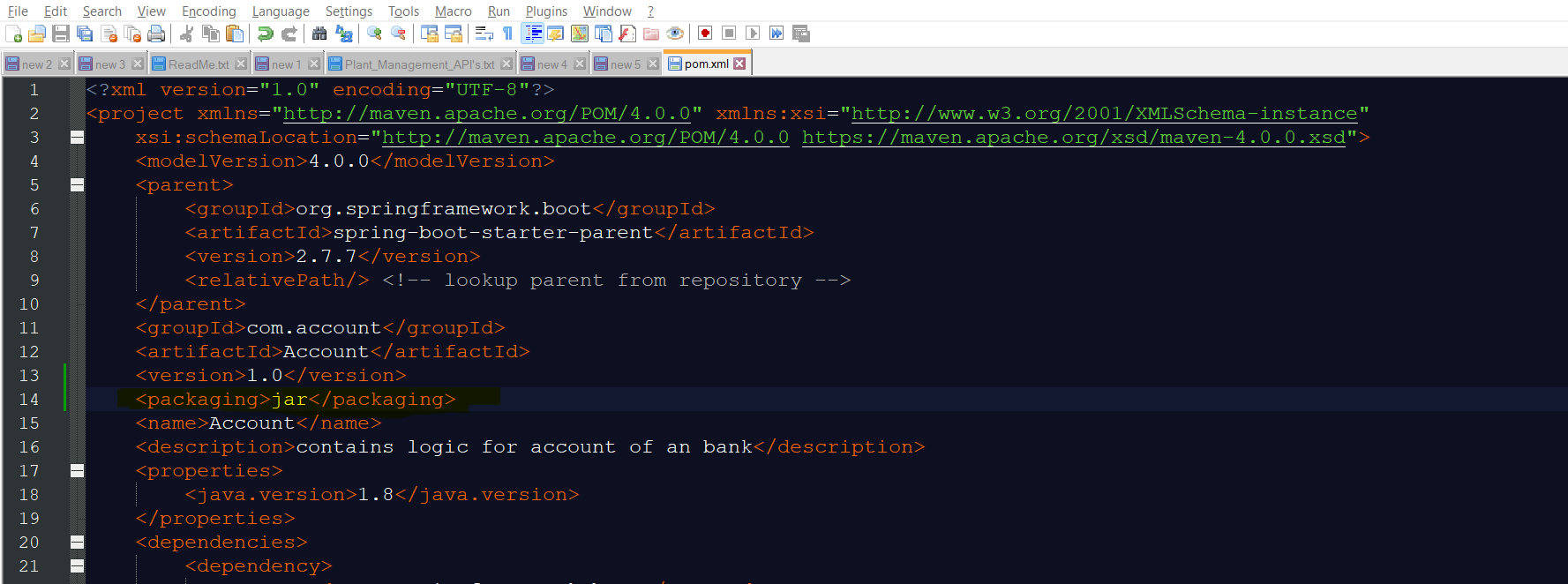
<https://docs.docker.com/desktop/install/windows-install/>

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

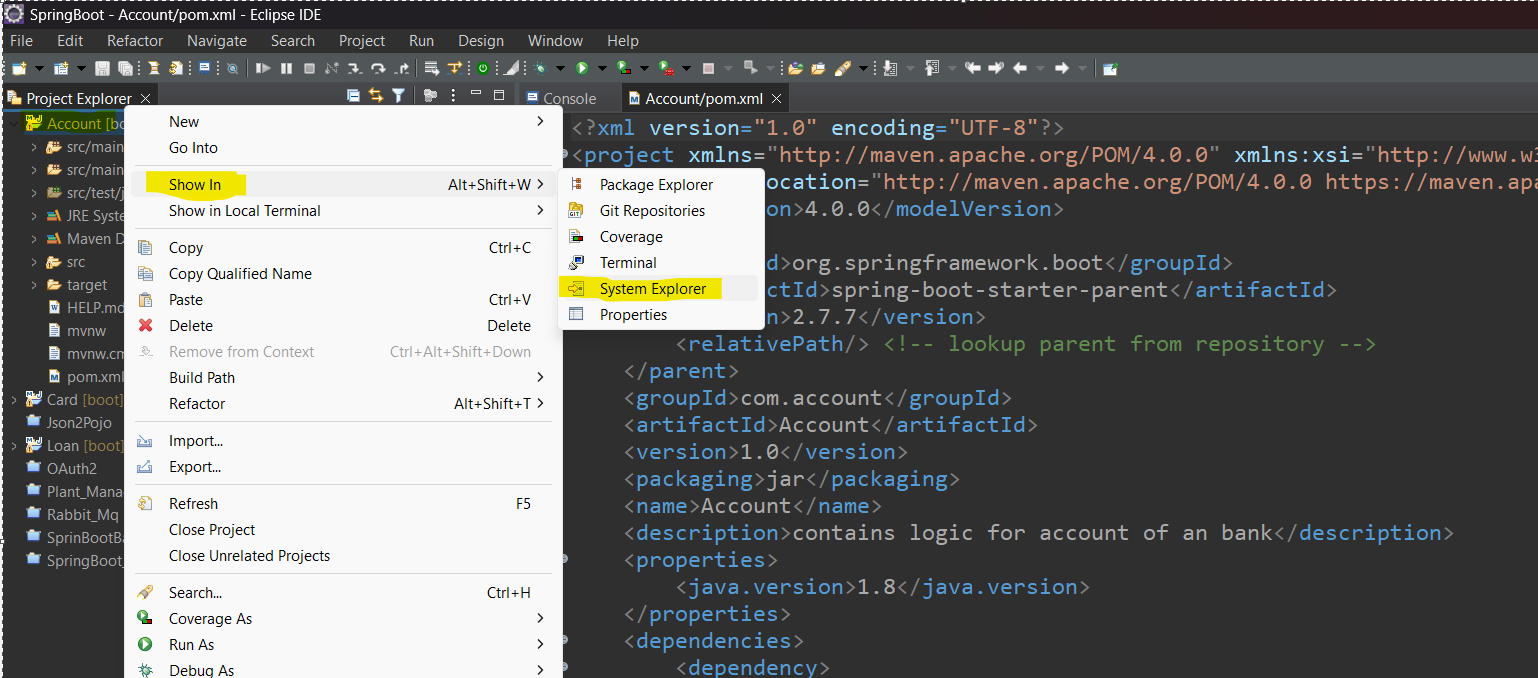
**H3] Creating/Executing a Docker Image and Container:**

**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:

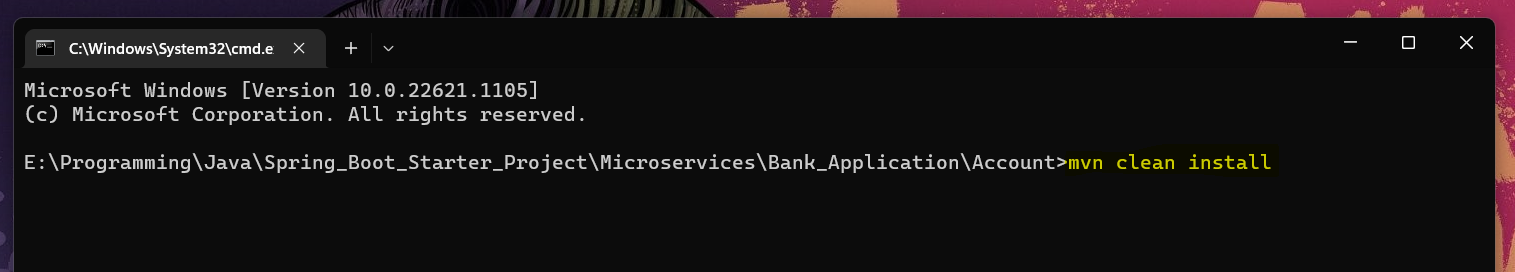


S3] Now open that folder path inside command prompt and run below command in it:

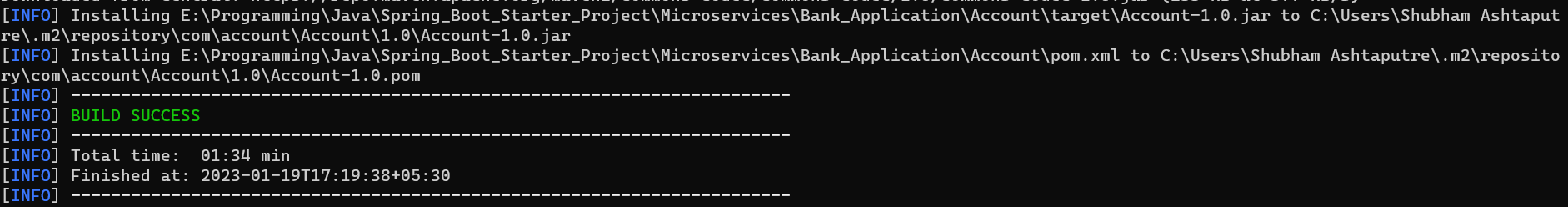
Command: **1] mvn clean install [This command will create a jar file]**

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

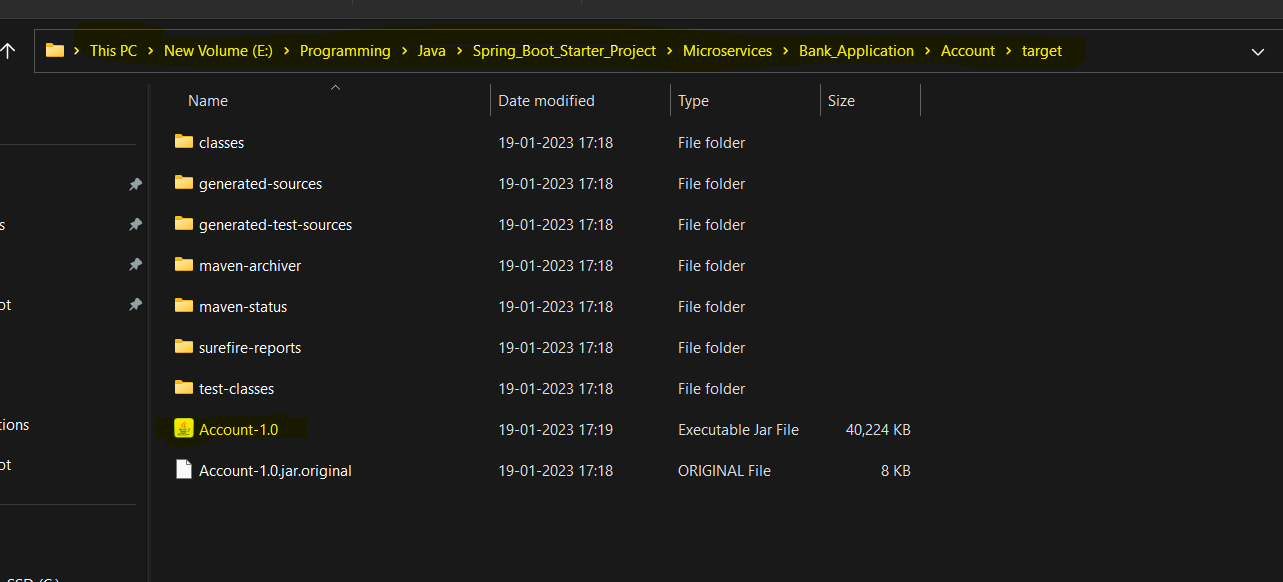
**Generally preffer to create the executable jar file using second command!!!**



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

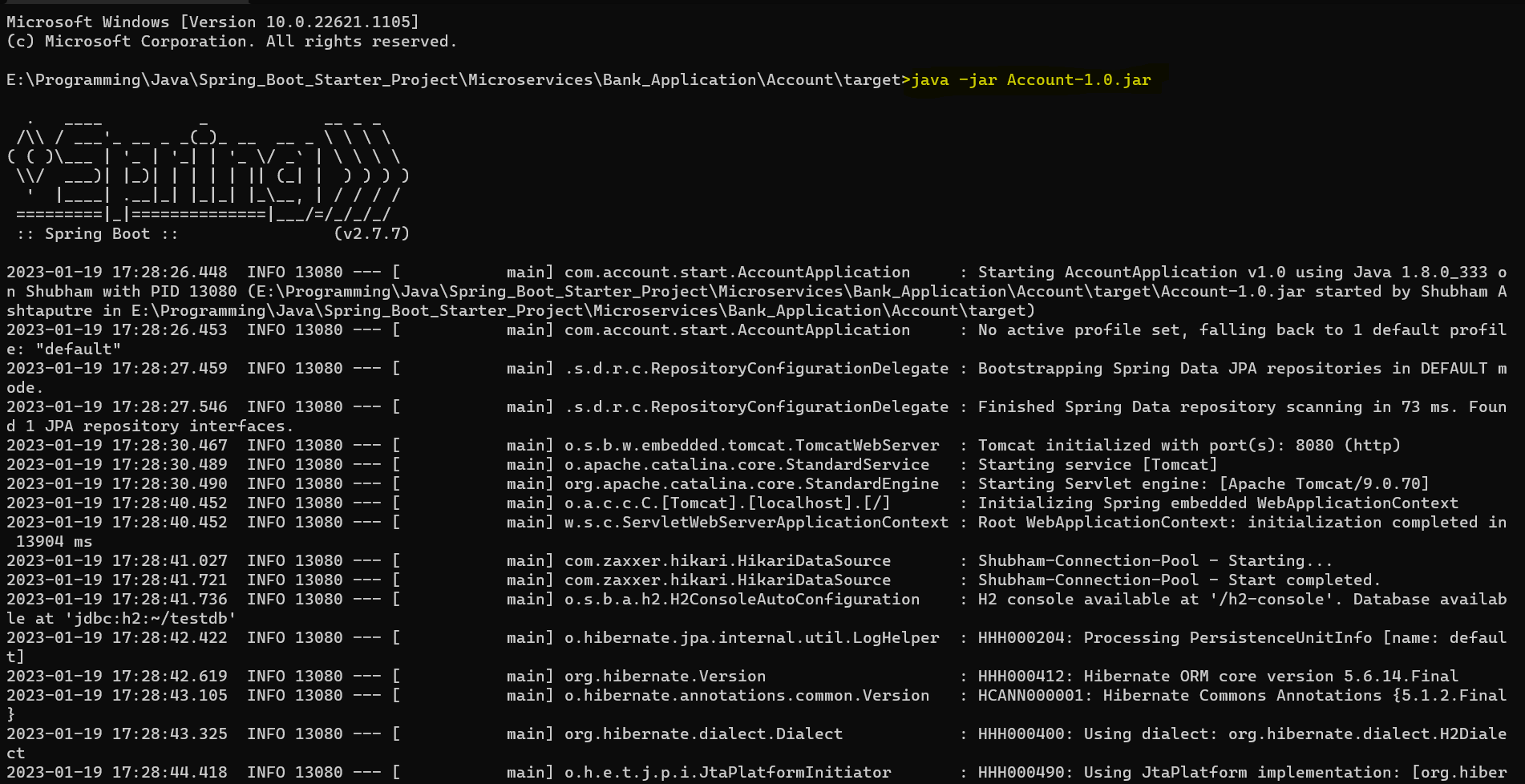


S4] Now you will get your deployed jar file in the target folder of your project as below:



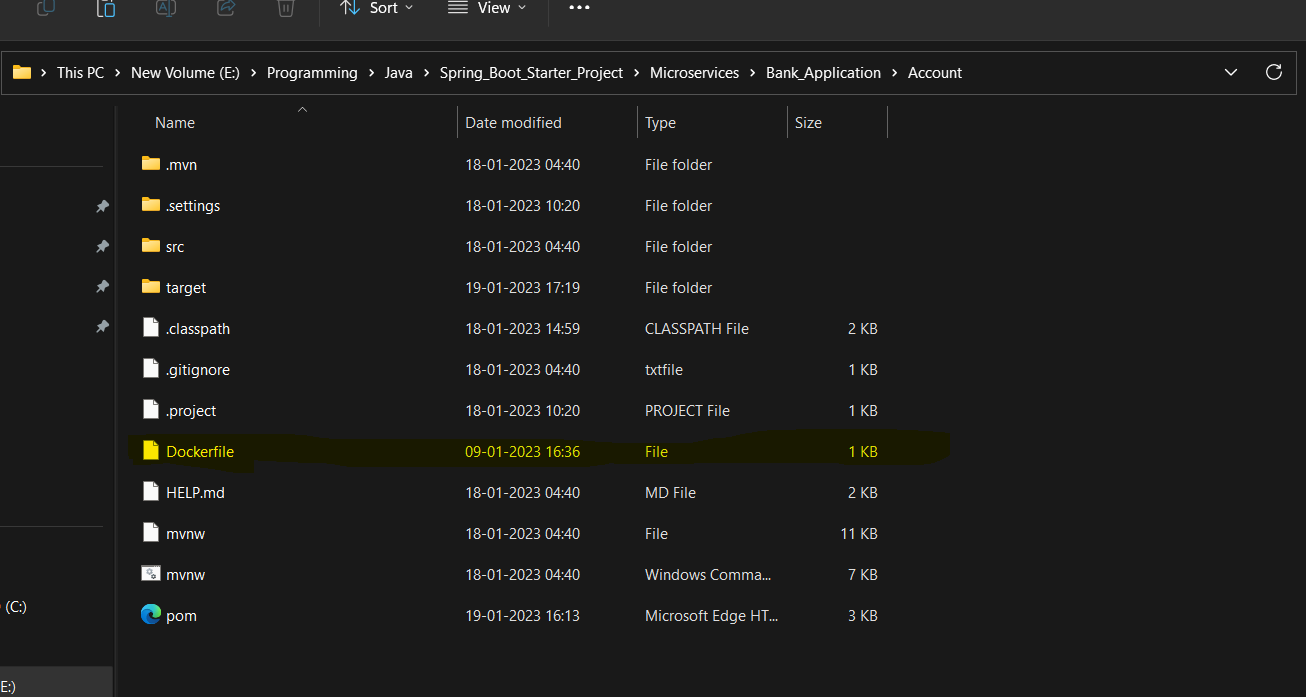
S5] Now you can check whether your jar file is running or not using below command:

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

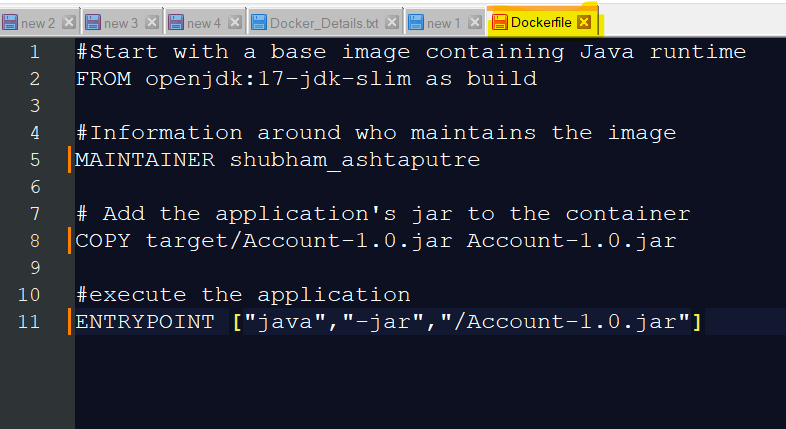


S6] press **ctrl+c** to close that running jar

**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**

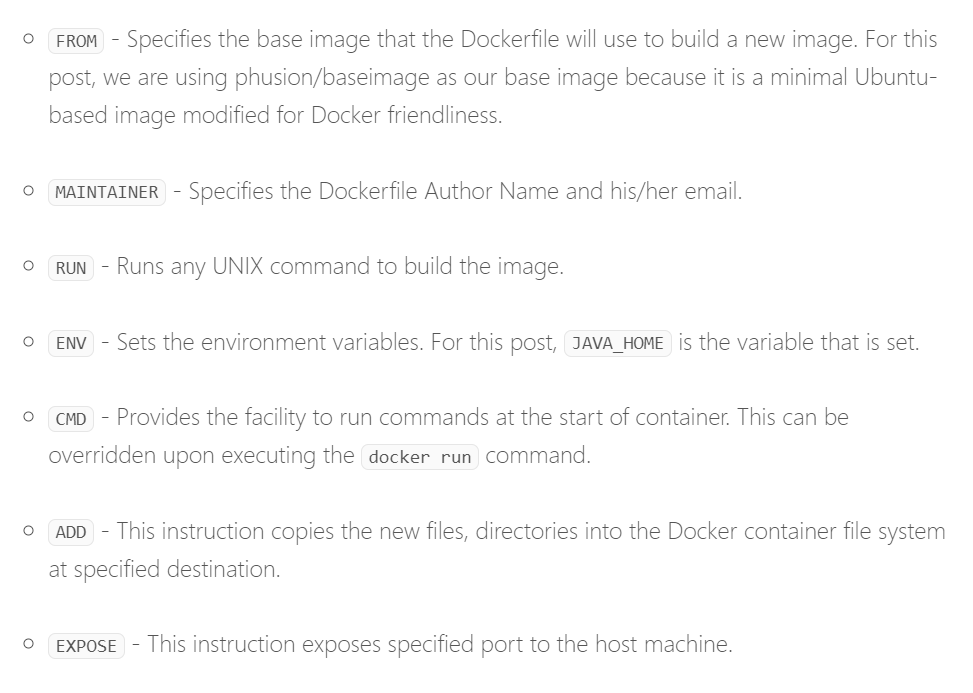
Here the ‘**FROM**’ command indicates what is the base image on which I want to build my application, here in this case I want to build my application on open-jdk-17 but it is not installed on my local system so docker will download it from its own image repository and install it on my system

**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"]**

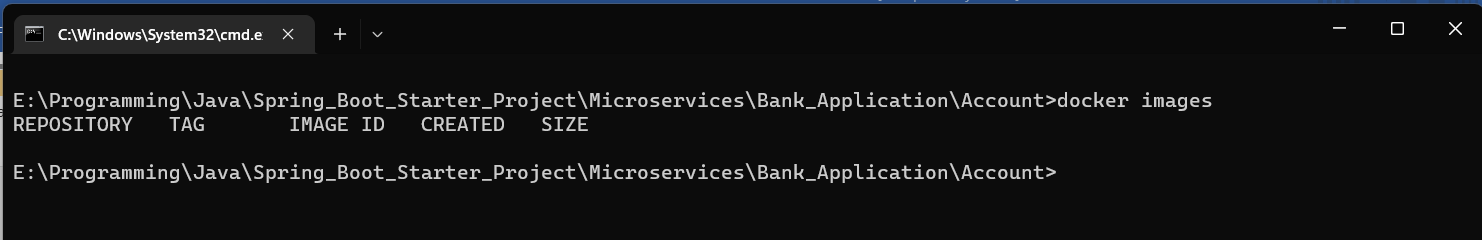
Here, the command entry point tells docker that whenever I want to start my container then start it with the **["java","-jar","/Account-1.0.jar"]** command



**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**



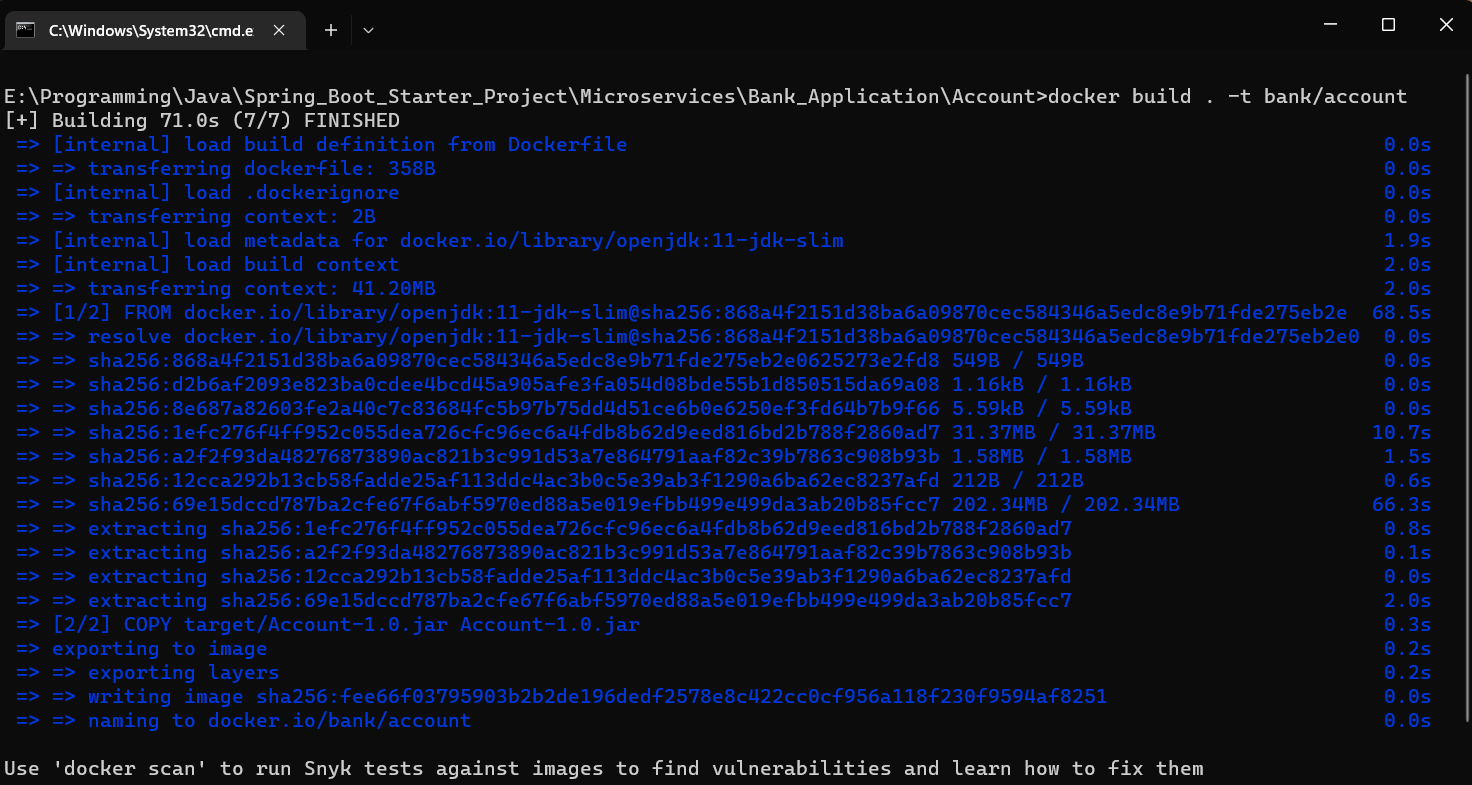
As you can see there is no docker image present inside our project directory

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

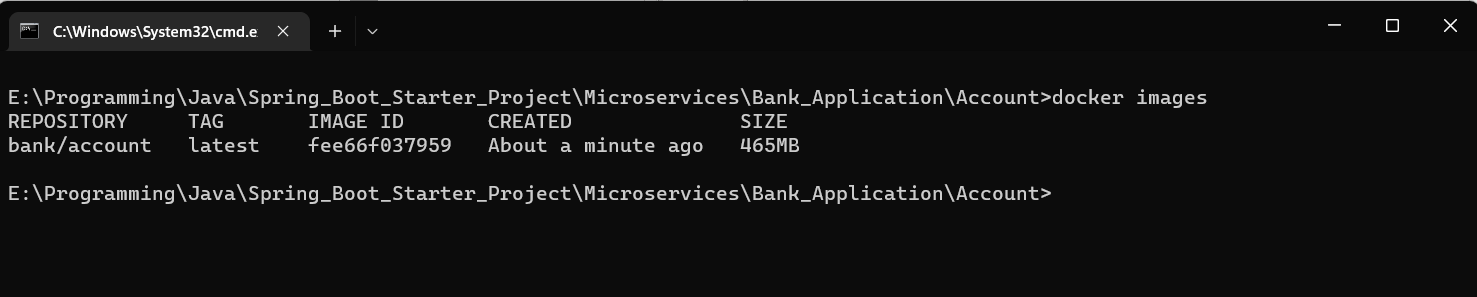
**Cmd: docker build . -t bank/account**

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:

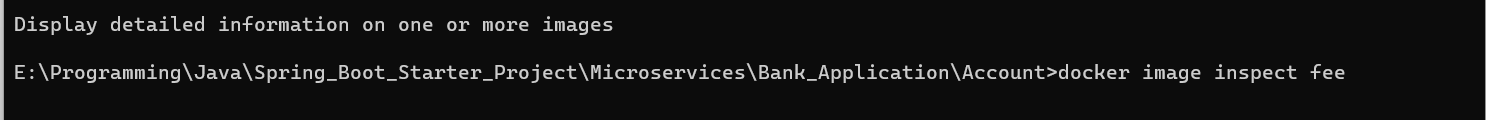


To verify run the below command:

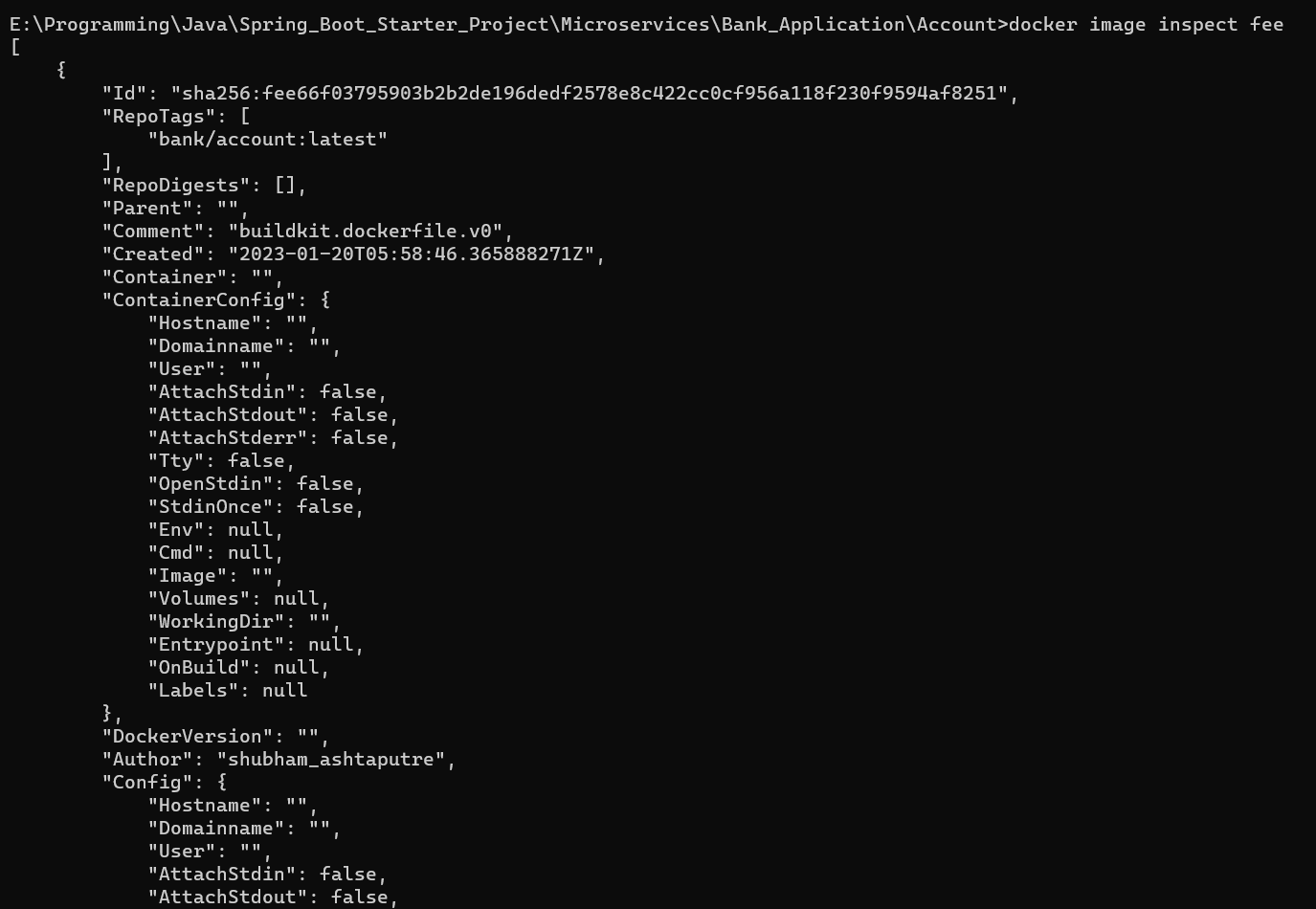


To inspect/ get details about the image run the below command:

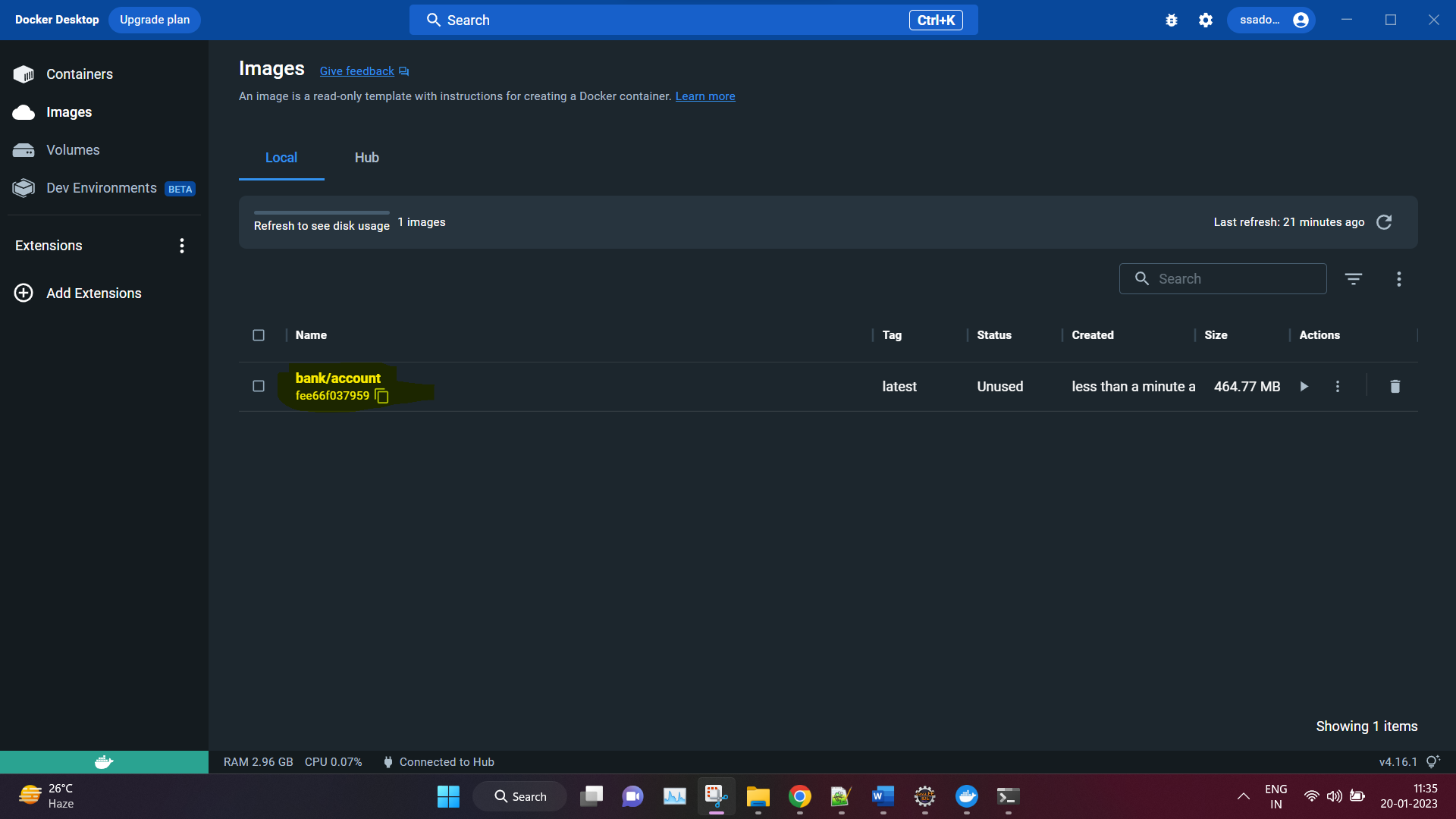
**Cmd: docker image inspect <Image Id>**



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

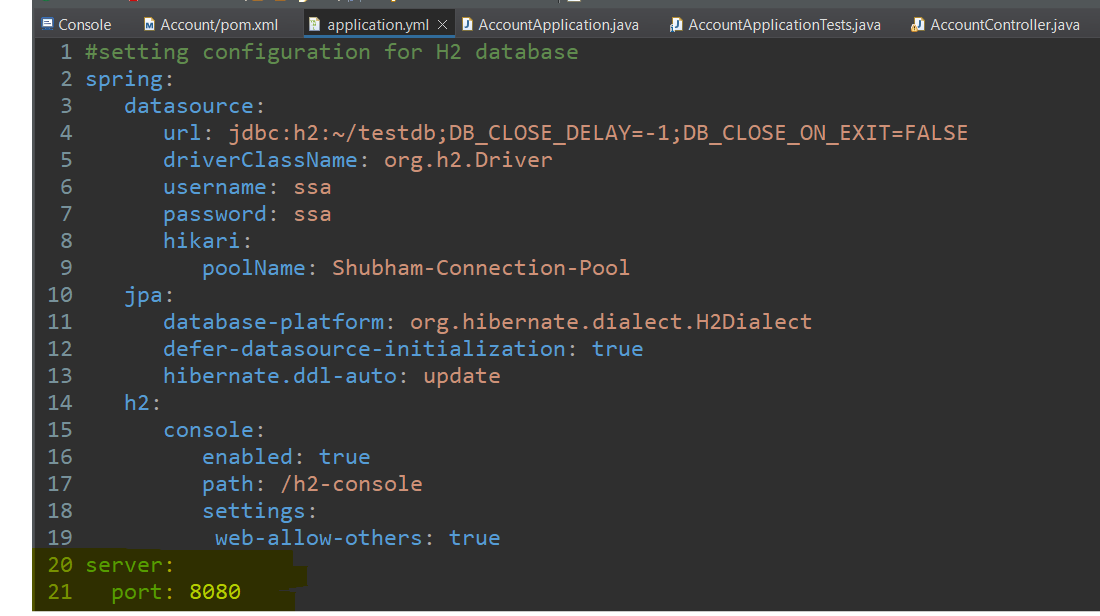


Now inside your docker desktop application also you can see your created image:



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

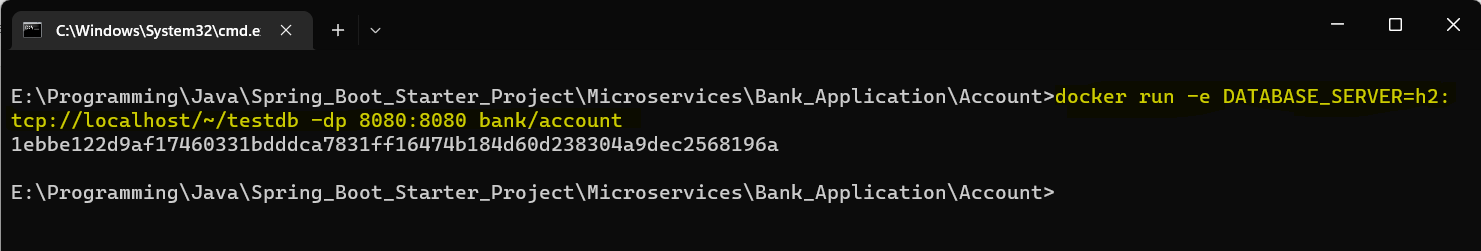
By default, the port on which the server will run is 8080 in docker if you want to change the port number then do changes in projects **application.yml** file as below:



Run the blow command:

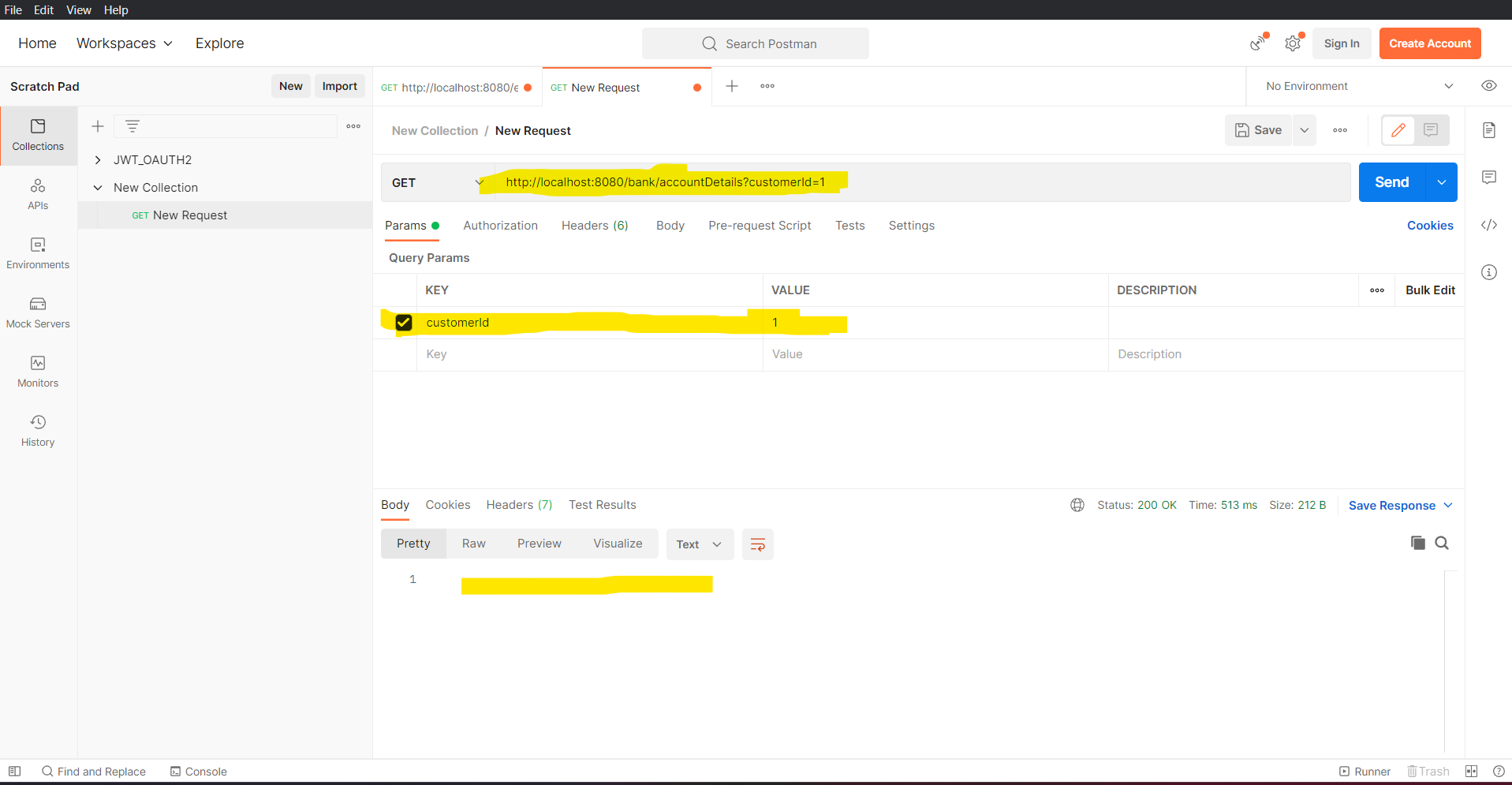
**docker run -e DATABASE\_SERVER=h2:tcp://localhost/~/testdb -dp 8080:8080 bank/account**

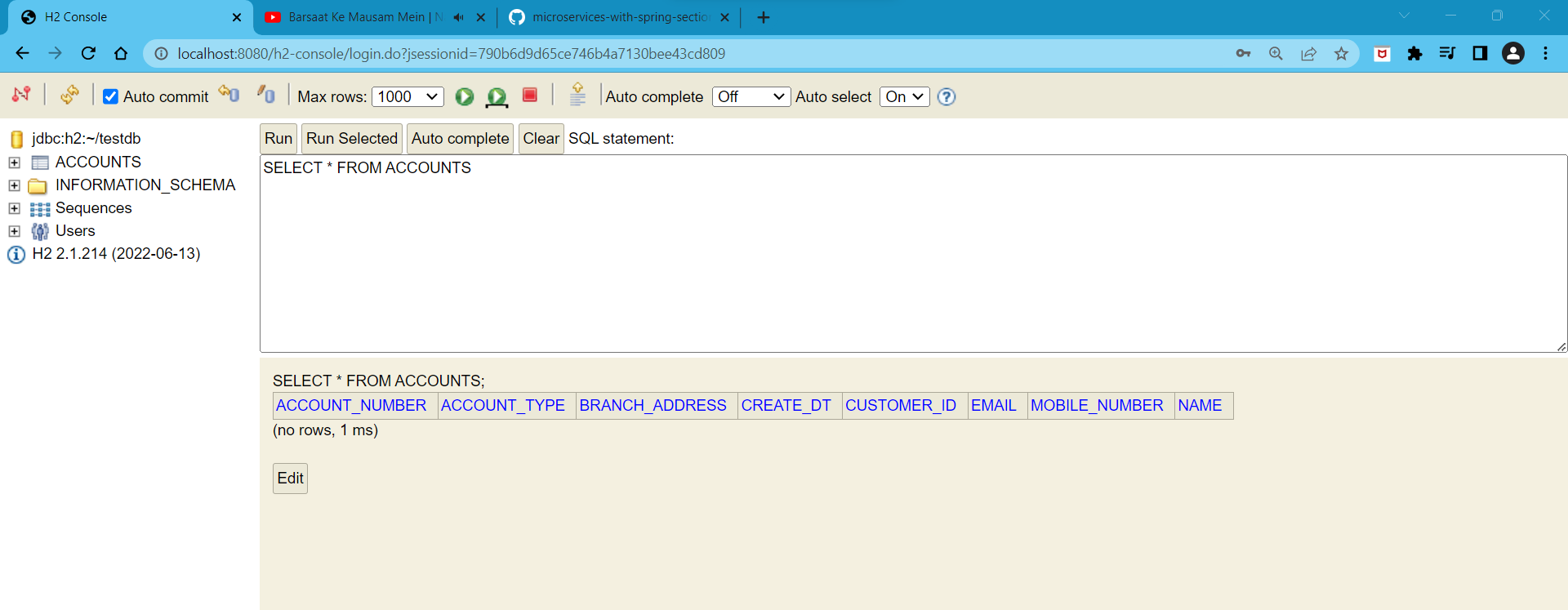
here, **‘-d’** means detach command it will run your container application in background rather than showing you the spring startup console and **‘-p’** means port number on which the application will be hosted in the docker container and mapped to your local system port number as: **<your systems port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



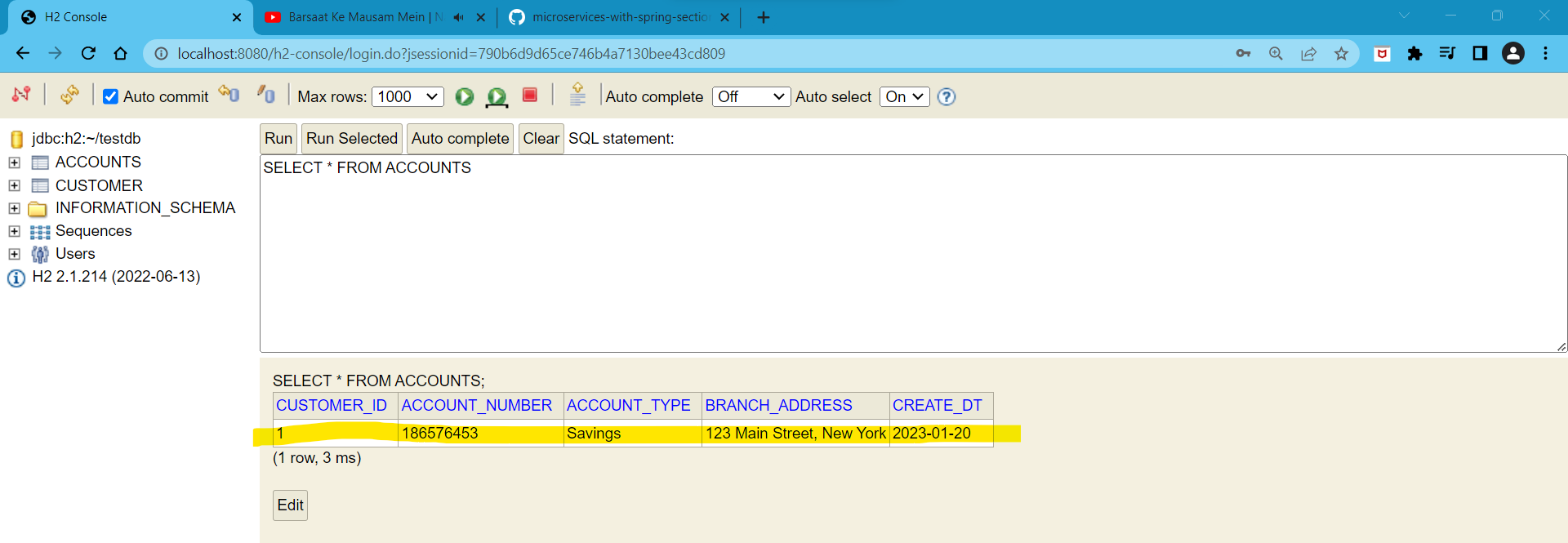
This command will create container for your docker image and start the h2 database server

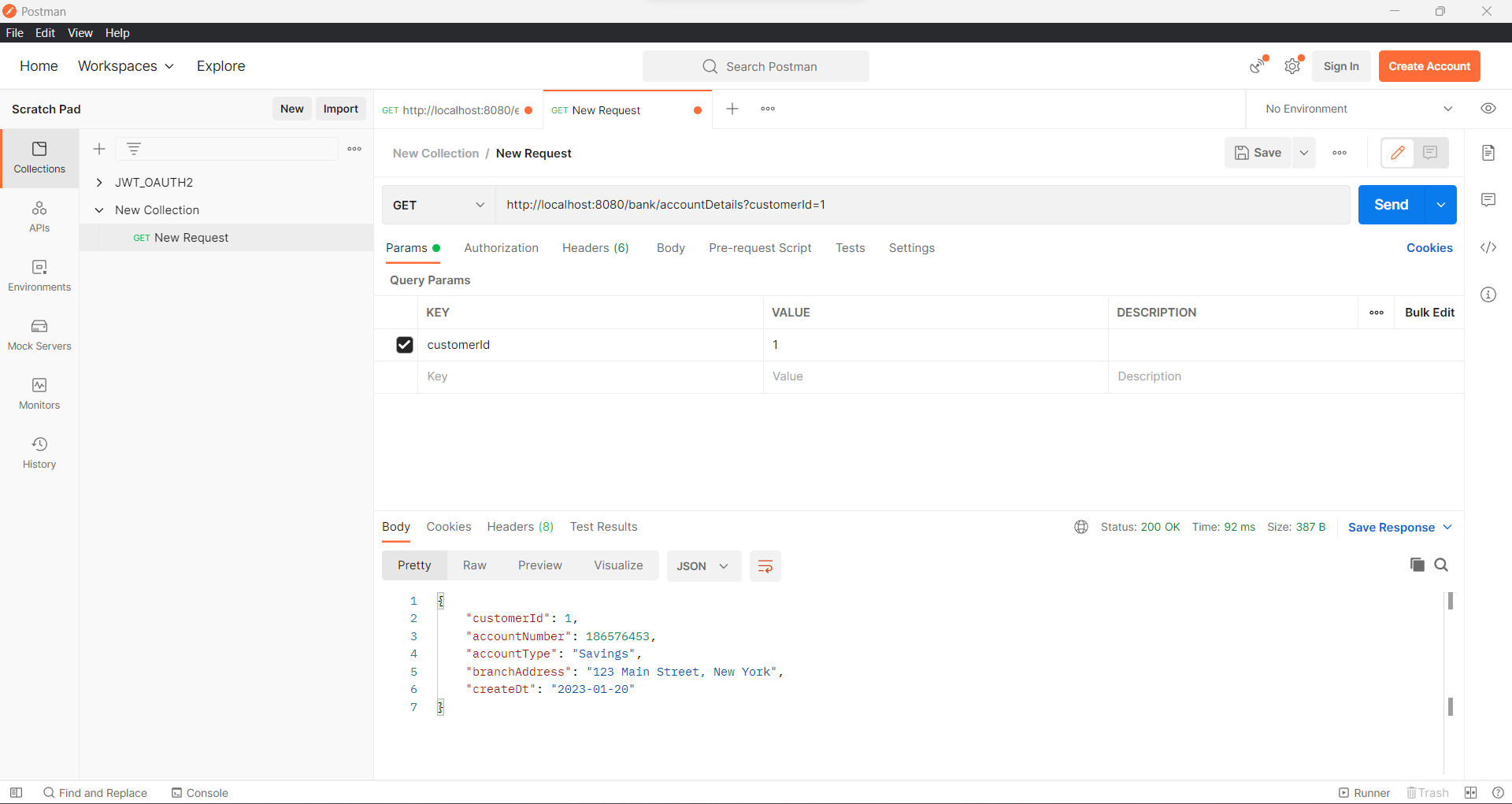
Now test your docker deployed API in postman:

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



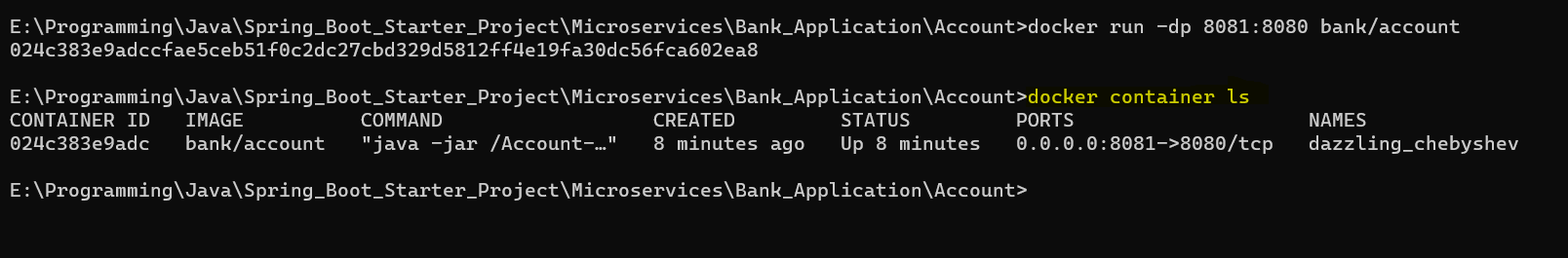


**H4] Docker Commands:**

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

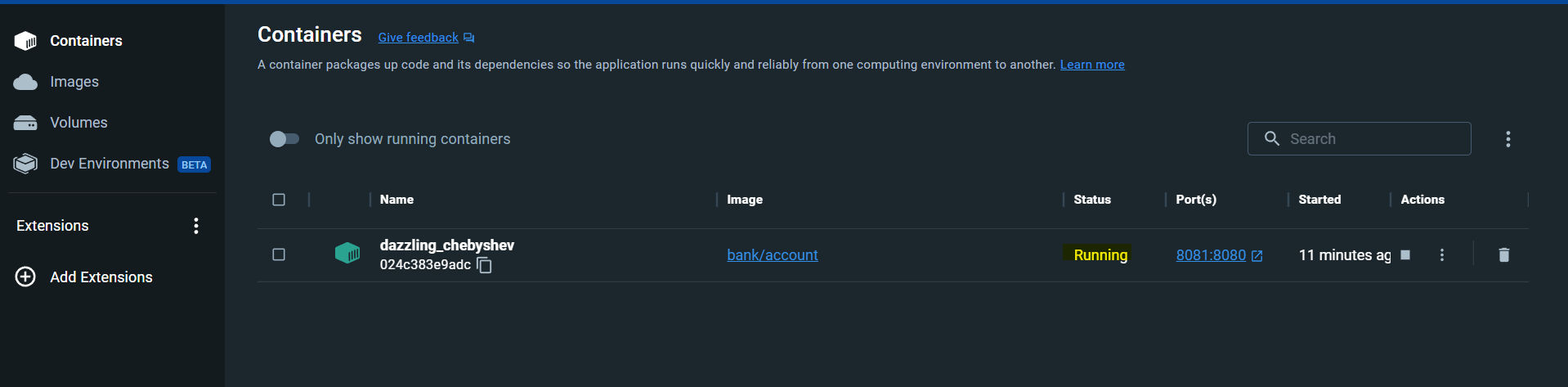
Cmd: 1] **docker container ls** [**see the list of running docker containers**]

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



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

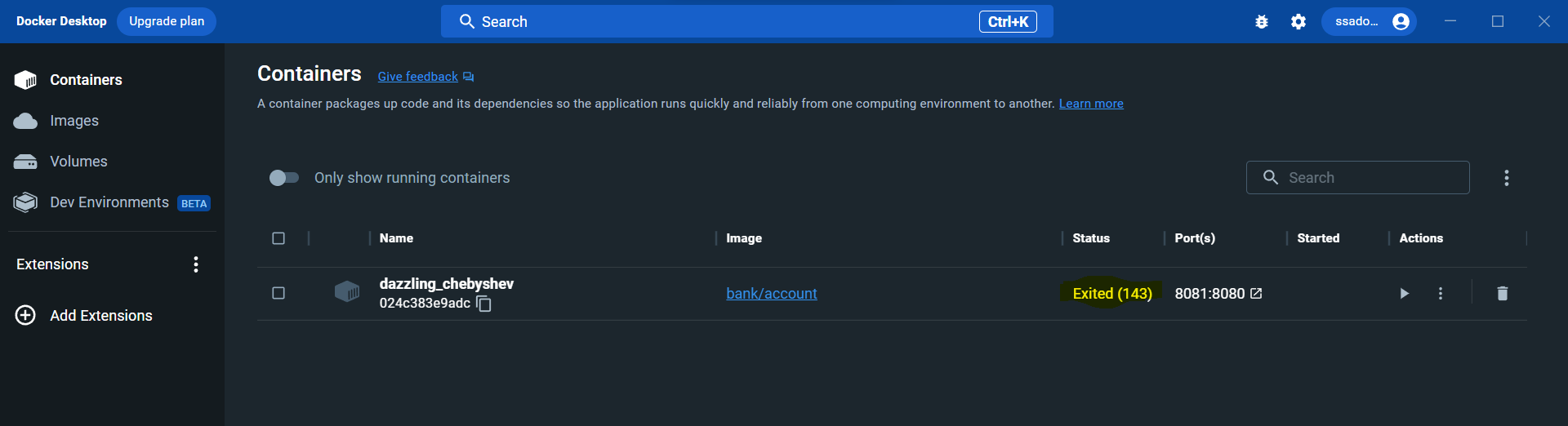
As we can see the docker container is still running:



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

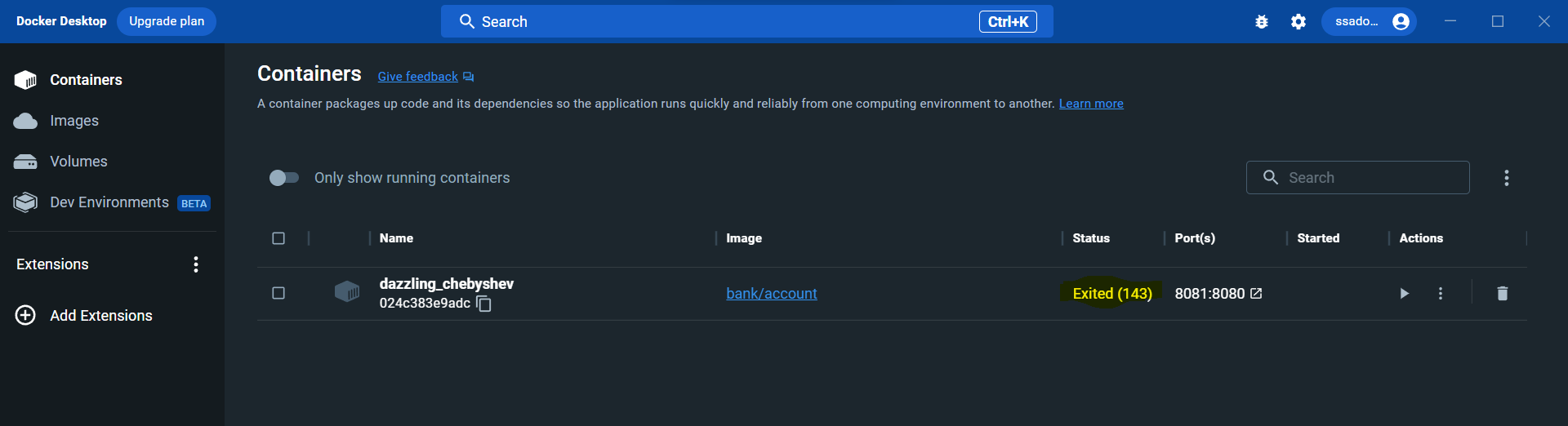
Cmd: **docker stop <Container-Id>**





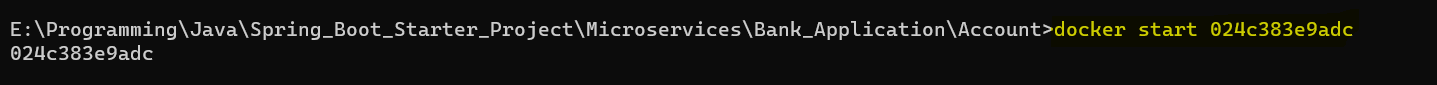
**H4.3] How to again start the existing docker container:**

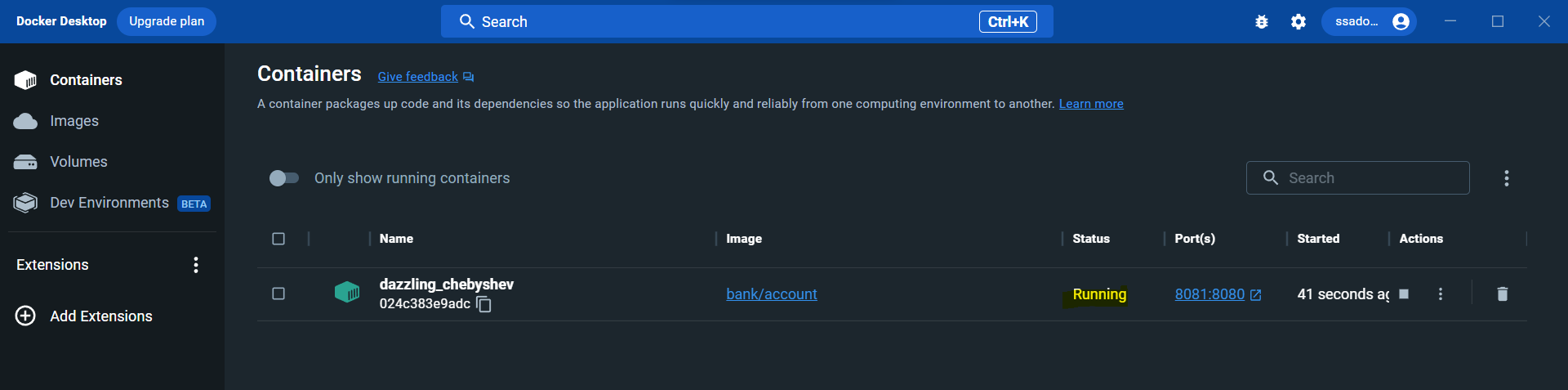
As we can see the existing docker container is now stopped



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

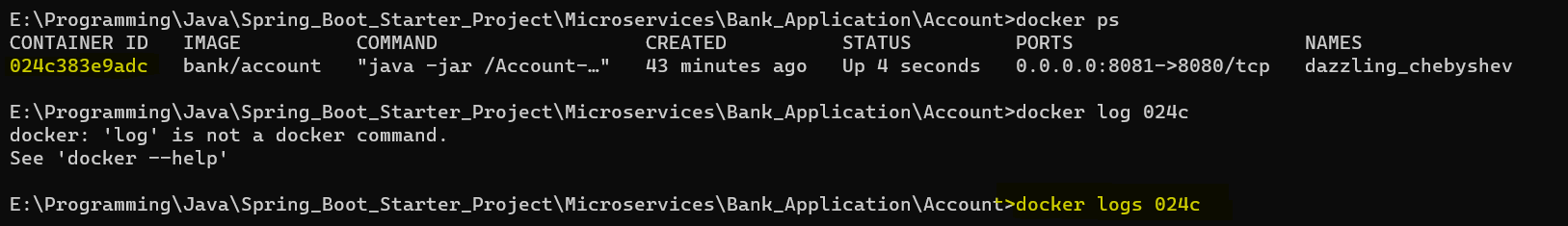
Cmd: **docker start <Container-Id>**





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

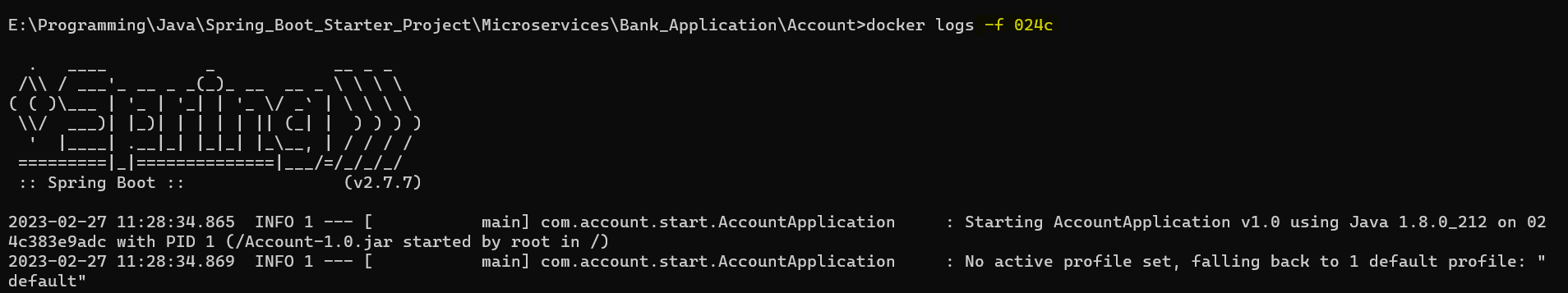
**1] Get static logs of the container:**



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

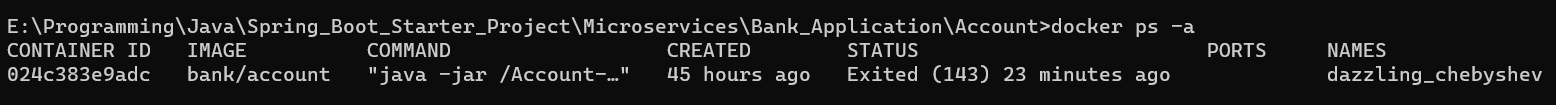


Cmd: **docker logs -f <container id>**

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

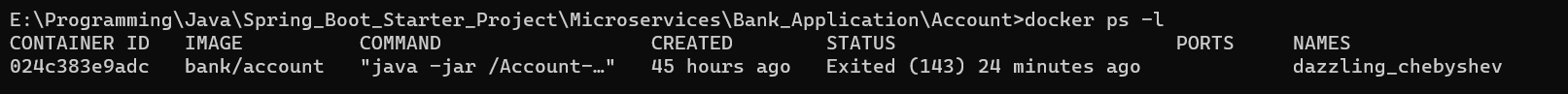
**H4.5] Docker get list of all container whether they are stopped or running:**

Cmd: **docker ps -a**



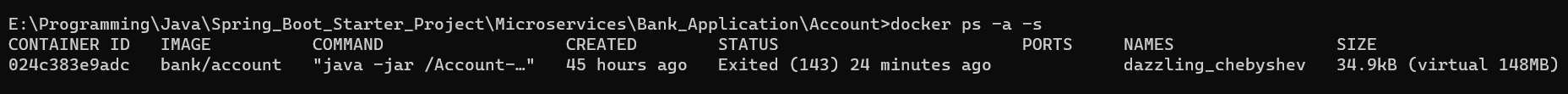
**H4.6] Docker get latest container created details:**

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



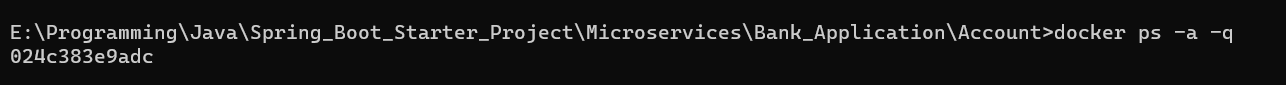
**H4.7] Docker display container size:**

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



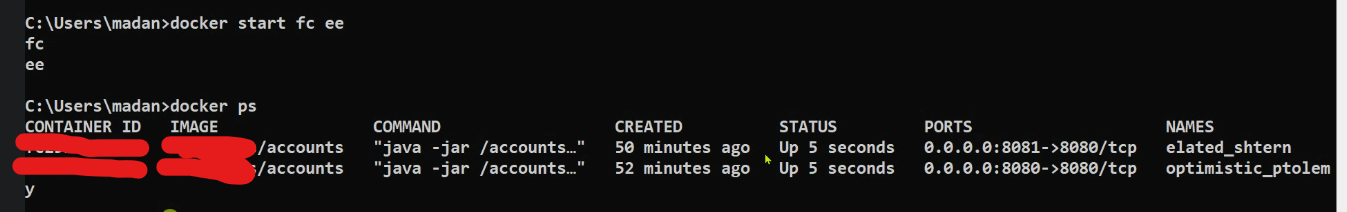
**H4.8] Docker display only container Id’s:**

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



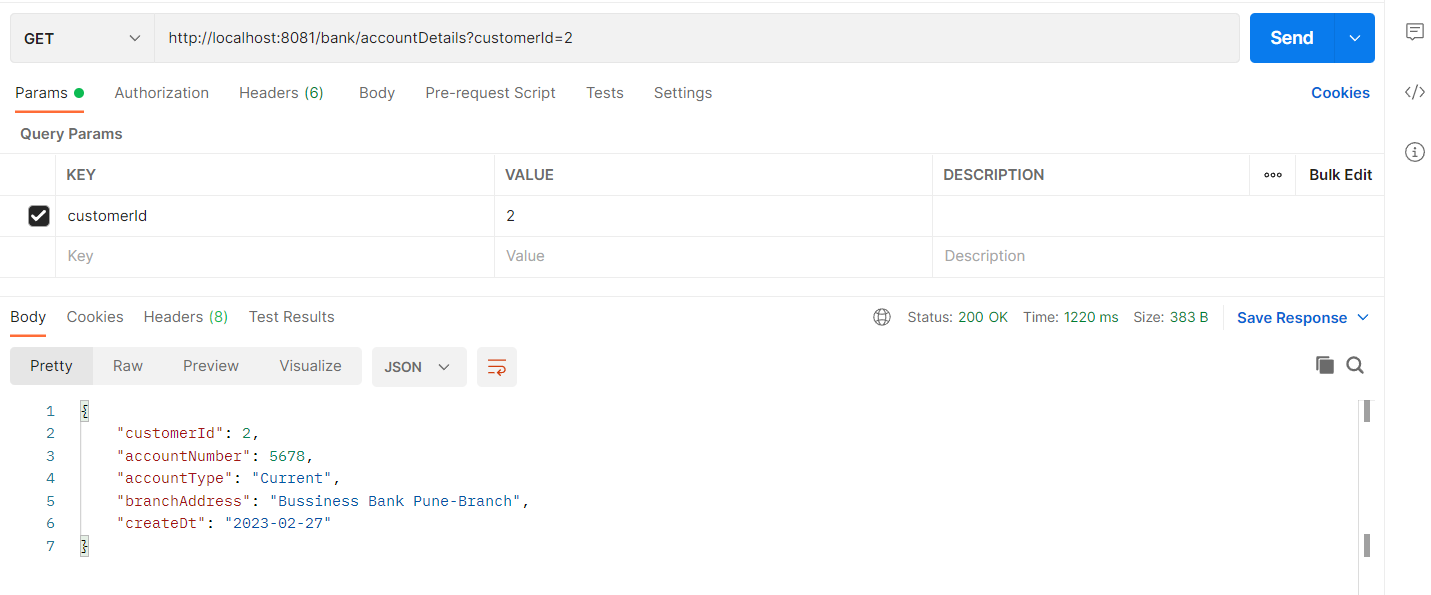
**H4.9] Docker start multiple containers at single time:**

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

****

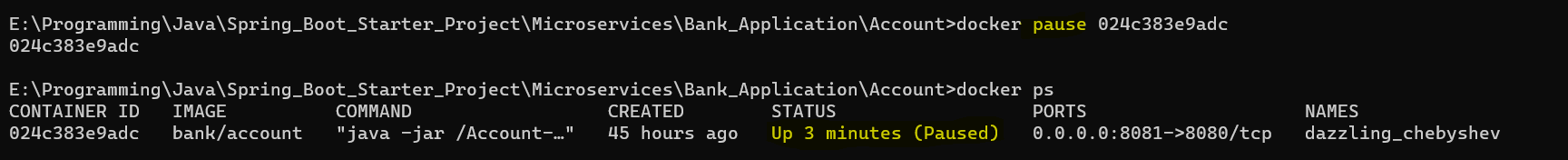
**H4.10] How to stop docker container to accept any request without stopping docker container?**

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

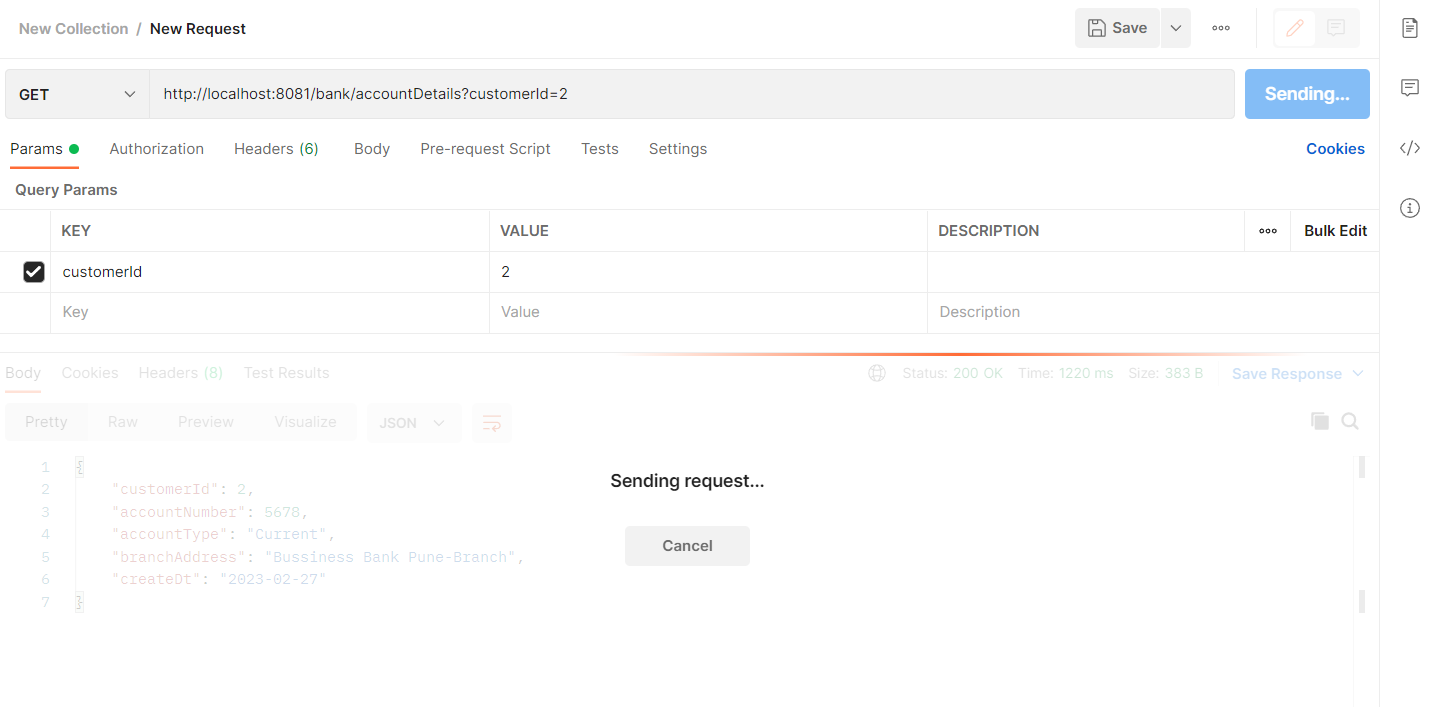


But, now if I want to pause all request’s contained in this container use below command:

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

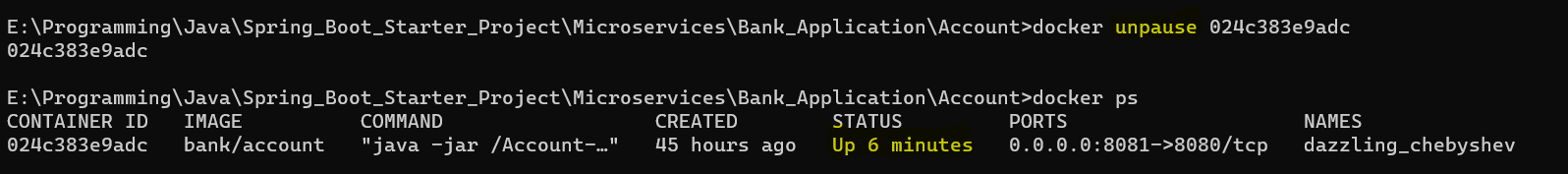


It is not accepting the request



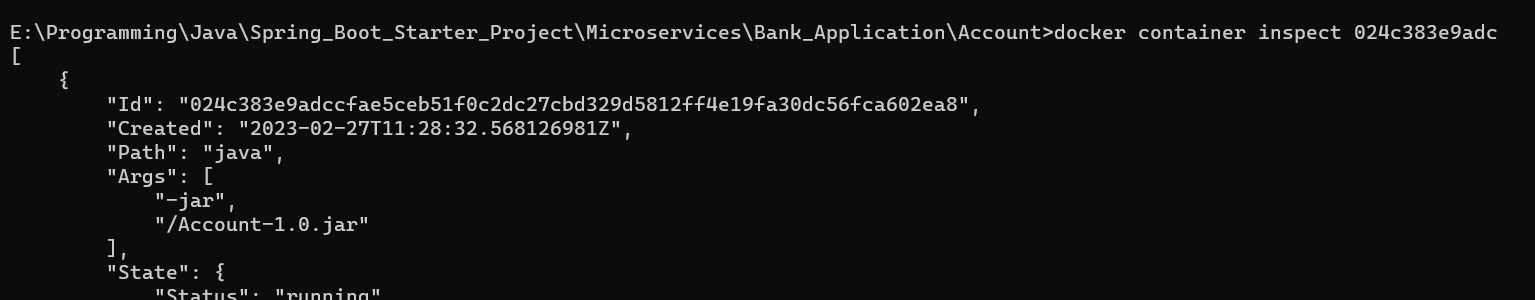
Now to make it work again use the below command:

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 <cont-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>**

H5]