**Importance of docker**

**Deployment, Portability and scalability of Microservices:**

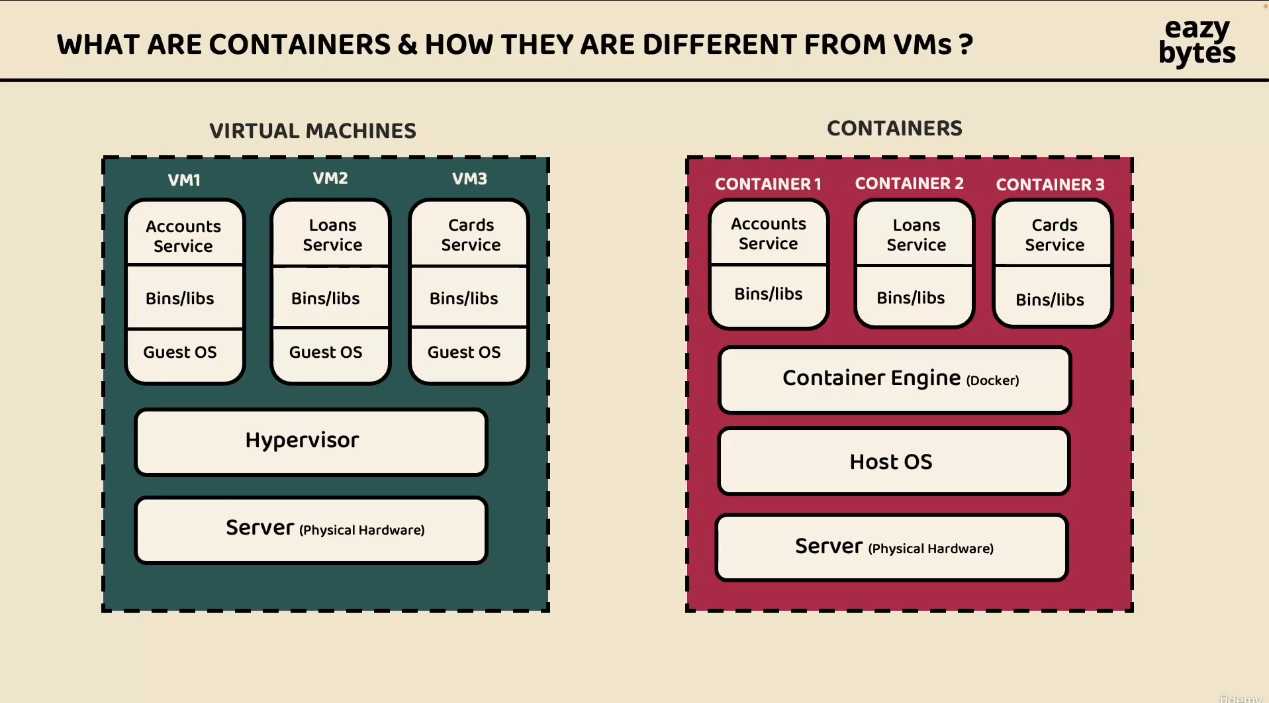
Developer -> GitHub Repository or another version control system -> From there we need to deploy into the development environment -> Once dev build is stable, we need to move to the UAT environment, SIT environment or QA environment -> Once the testing is completed, we need to deploy Production replica environment -> Once the pre-production testing is completed eventually the deployment as to be done inside the production environment.

**Deployment** – How do we deploy all the tiny 100s of microservices with less effort and cost?  
**Portability** – How do we move our 100s of microservices across environments with less effort, configurations and cost?  
**Scalability** – How do we scale our applications based on the demand on the fly with minimum effort and cost?

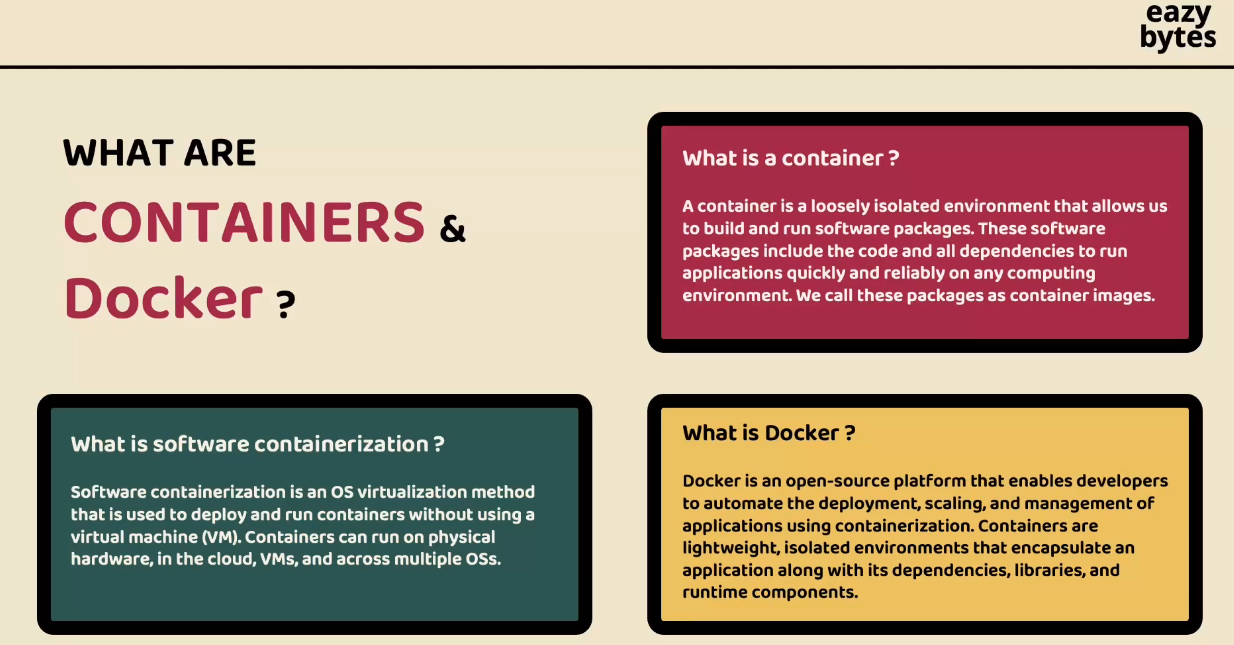
So, these are the three challenges that we face whenever we try to build microservices with a traditional monolithic mindset.

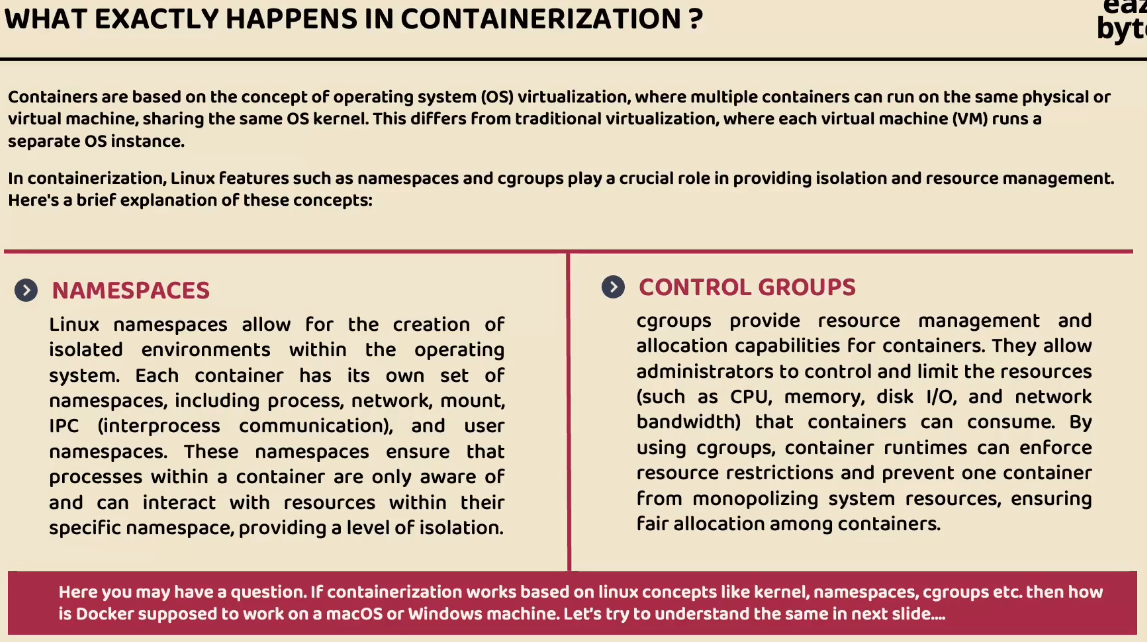
To overcome the above challenges, we should **containerize** our microservices. Why?  
Containers offers a self-contained and isolated environment for applications, including all necessary dependencies. By containerizing an application, it becomes portable and can run seamlessly in any cloud environment. Containers enabled unified management of applications regardless of the language or framework used.

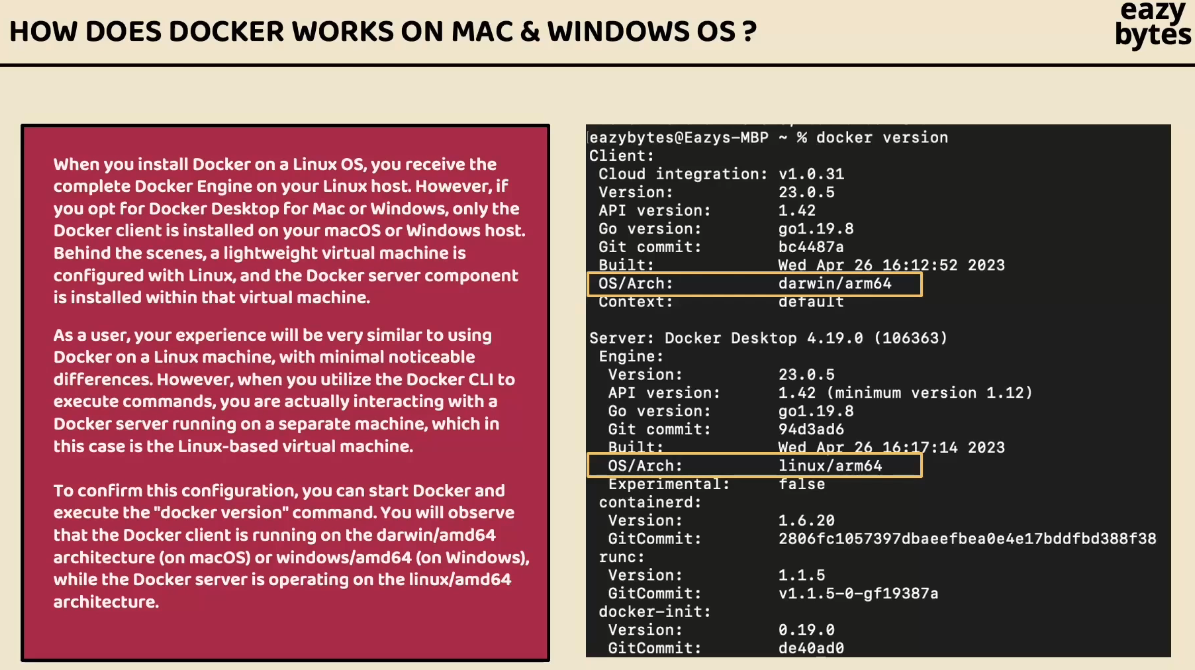
How do we containerize our maven application?  
-> **Docker** is an open-source platform that “provides the ability to package and run an application in a loosely isolated environment called a container”.

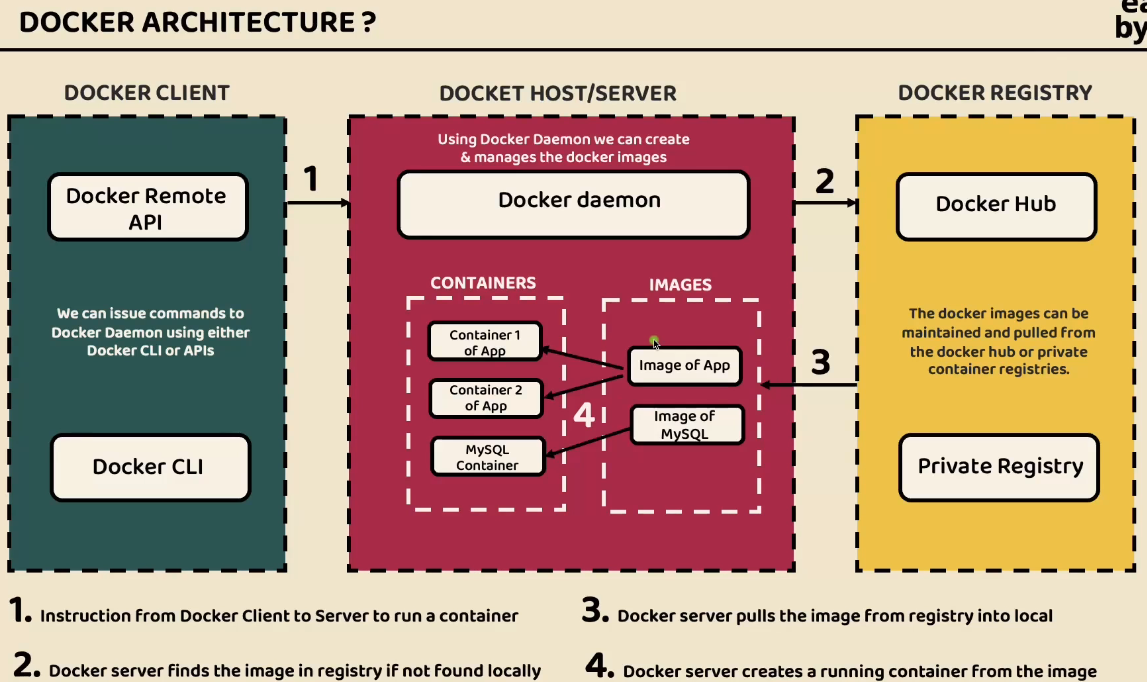


Containers they are going to become a light weight, creating a container destroying a container or restarting a container is going to take only few seconds. So, this is the very first advantage we have container compared to virtual machines.  
The other advantage is that we have with container is inside the same virtual machine they can have a separate isolated virtual environment.  
Deployment is very easy because it is a lightweight component. The main difference of container and virtual machines is containers they don’t need the Guest operating system not hypervisor to assign the resources, instead they will use the container engine which is Docker.









**Install Docker in our local system:**

1. <https://www.docker.com/> visit the official docker website. Install the application based on the operating system. Based on our operating system we have to download.

Docker credentials – Email used: [varunkumarkm44@gmail.com](mailto:varunkumarkm44@gmail.com) and username used: varunkumarkm43

1. Need instructions to install docker visit this page - [https://docs.docker.com/desktop/?\_gl=1\*h620h1\*\_gcl\_au\*MTg3NzgyNjA4LjE3MjgyODk1MzY.\*\_ga\*MjAxMTMwODkyNC4xNzI3MTU4ODE4\*\_ga\_XJWPQMJYHQ\*MTcyODI4OTUzNS4yLjEuMTcyODI4OTcxMC4yMy4wLjA](https://docs.docker.com/desktop/?_gl=1*h620h1*_gcl_au*MTg3NzgyNjA4LjE3MjgyODk1MzY.*_ga*MjAxMTMwODkyNC4xNzI3MTU4ODE4*_ga_XJWPQMJYHQ*MTcyODI4OTUzNS4yLjEuMTcyODI4OTcxMC4yMy4wLjA).

If we want to store the docker images in private access the we need to upgrade our plan.

1. Once we installed the docker and created the docker account -> Need to click on the docker icon available inside our local system -> click on the dashboard. -> Now our local docker is connected to the docker hub repository that we have created -> We can also confirm running the command inside the terminal **docker version** -> This command shows all the versions related details.
2. Inside the mac operating system we try to create the docker internally it is going to create a small and light weight Linux virtual machine, and inside that Linux VM docker server is going to be install.

**Generate Docker Images in the Microservice:**

By converting our microservices into docker images we are going to make them very light weight which will make them suitable to overcome challenges related to the deployment, portability, and scalability.

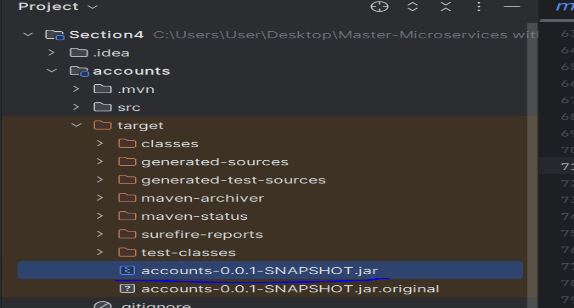
To generate docker images from our microservices, we will explore the below three different commonly used approaches. We can choose one of them for the rest of the course.

1. **Dockerfile -> accounts microservices**: We need to write a docker file with the list of instructions which can be passed to docker server to generate a docker image based on the given instruction.
2. **Buildpacks -> loans microservices:** Buildpacks (<https://buildpacks.io>), a project initiated by Heroku and pivotal and now hosted by the CNCF. It simplifies containerization since with it, we need to write low-level dockerfile.
3. **Google Jib -> Cards microservices:** Jib is an open-source java tool maintained by google for building docker images of java applications. It simplifies containerization since with it, we don’t need to write a low-level dockerfile.

These are the main three different approaches commonly used to generate docker image from a spring boot web application.

1. **Dockerfile approach:**

Dockerfile will give lot of flexibility to us, if we have any custom requirements while we are generating a docker image we can achieve all of them with the help of dockerfile approach.

* **accounts microservices**
* Goto the accounts microservices -> pom.xml file -> after version element we need to add <packaging>jar<packaging> element after <version>0.0.1-SNAPSHOT</version> this tag, this tells to our maven want to package our web application in a jar format. Similarly, we can add war format as well <packaging>war<packaging> but jar is a good format that we can follow whenever we want to generate a docker image.
* Delete all the existing combined classes of accounts microservices as of now there is no jar file inside the target folder, there are only compiled classes and folders available -> so, delete the content inside the target folder.
* Goto our section4 local folder location -> open the account service -> open the terminal at this folder choosing the option new terminal.
* Run a maven command **mvn clean install** -> before that maven is install inside our system and we setup the maven home path.
* This is the link to install maven <https://maven.apache.org/download.cgi> For **Windows**: Download the **Binary zip archive**. -> Extract the folder in our local.
* **Set MAVEN\_HOME and add Maven to PATH**: -> **Search** for **Environment Variables** in the Windows search bar. -> Click on **Edit the system environment variables** > **Environment Variables**. -> **Create a new MAVEN\_HOME variable**: -> Under **System Variables**, click **New**. -> **Variable Name**: MAVEN\_HOME -> **Variable Value**: C:\Program Files\Apache\Maven (or the path where you extracted Maven). -> **Add Maven to the PATH**: -> Under **System Variables**, find and select the Path variable, then click **Edit**. -> Add a new entry: %MAVEN\_HOME%\bin -> **Click OK** to save the changes.
* Use the command in CMD **mvn -version** this will give which version of maven is installed in our system.
* Now go to the accounts microservices directory use **mvn clean install** – using this command we are telling to maven to compile our spring boot application with the name accounts -> when use this command behind the scenes -> maven is going to compile accounts microservices and it will also do a small basic unit testing and if the unit testing is successful, it will be going to generate a jar file into the target folder.
*  The jar file was created around 54 mb, we call this jar as a fat jar inside spring boot, because inside this fat jar it is going to have all kind of dependencies except java runtime, so it has all the spring boot library, it will have embedded tomcat server all the dependencies related to accounts microservices is going to be available inside this jar except java libraries.
* So based on the combination of the <artifactId>accounts<artifacatId> <version>0.0.1-SNAPSHOT</version> and version the jar name will be derived.
* **mvn spring-boot:run** – When we try to run this command, behind the scenes inside our pom.xml since we have a plugin <artifactId>spring-boot-maven-plugin</artifactId> this plugin is looking for a jar inside our target folder and using the same jar it is going to start our web application.
* **Use control c** – Automatically our application stops executing.

Now we are going to use java command instead of a maven command since inside docker image we don’t want to unnecessarily install the maven.

**java -jar target/accounts-0.0.1-SNAPSHOT.jar** – This is the command we execute, whenever try to run this java command we can see the same effect like our web application will start at the port whichever we mentioned in our application.

**Step – 1**: Right click on our accounts microservices folder -> select the new file option the file name has to be **Dockerfile** exactly the same name. -> Because is look for the exact file name this not have any extinction. -> In this **Dockerfile** we need to write instructions that can be used to generate a docker image for my accounts microservices.

In order to run our accounts microservice any time whenever we want to run our accounts microservice the very first basic requirement that we have is to install JRE or java inside any system. That’s we should communicated to the docker saying that our accounts docker image as a dependency on java for the same we are going to use a keyword **From** -> using this from command we are telling to docker that might docker image as dependency on some other docker image so please import this base image into our docker image whenever we try to generate a docker image of our accounts microservice. To this **from** command we need provide the base image inside this scenario the base image is java for the same.

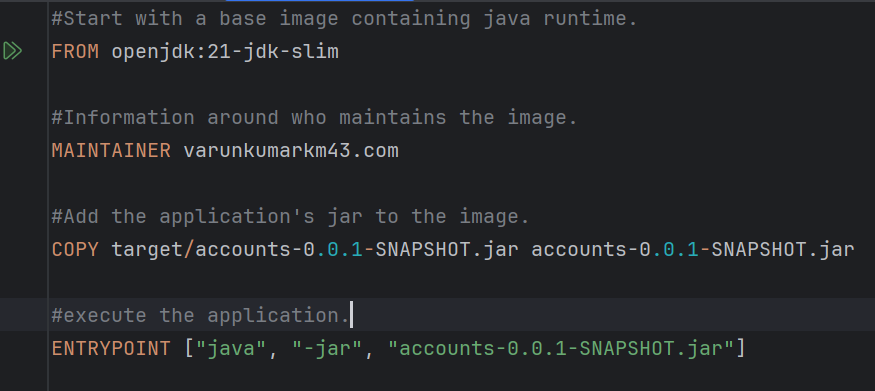
**Form openjdk:21-jdk-slim**

**openjdk –** This is the image name. We can search **openjdk** in docker hub it will shows   
**21-jdk-slim –** This is the tag name; it is very similar to java version.

**MAINTAINER** – In the latest version of Docker, MAINTAINER is deprecated in Favor of LABEL. Using MAINTAINER, we can provide of who is maintaining of this docker image.

**COPY** – using this command we pass our jar file; we are trying to say take the jar file present inside the target folder and copy the same jar into our docker image. When we generate a docker image it is going to have all the dependencies in a packaged form.

**ENTRYPOINT** – With is command telling to generate a container to my docker image please execute so and so command.



Using this Dokerfile, we can generate a docker image.

**Step – 2**: Inside the terminal execute the **docker version** command this will make shore the docker is install also started properly inside our local system.

docker build -t Since our dockerfile is present inside same folder location of accounts from where we are running this command, we don’t need to provide an any file name or any folder path but if a dockerfile is present inside some other folder the we need to provide that exact path. **-t** this indicates tag, after this tag what is the docker image name that we want to consider for our accounts microservice. So, whenever we trying to give a docker image name we need to follow a standard format, the format is first we need to make sure we are providing the **docker account username**. So, the docker account username is varunkumarkm43. Post that what is the name that we want give for our accounts microservice. That name is **accounts** and what is the version and tag name is s4.

**docker build . -t varunkumarkm43/accounts:s4** – So this is the command used to docker server is trying to create a docker image. Once it’s created a docker image we can able to see the naming of our docker image **docker.io/varunkumarkm43/accounts:s4**

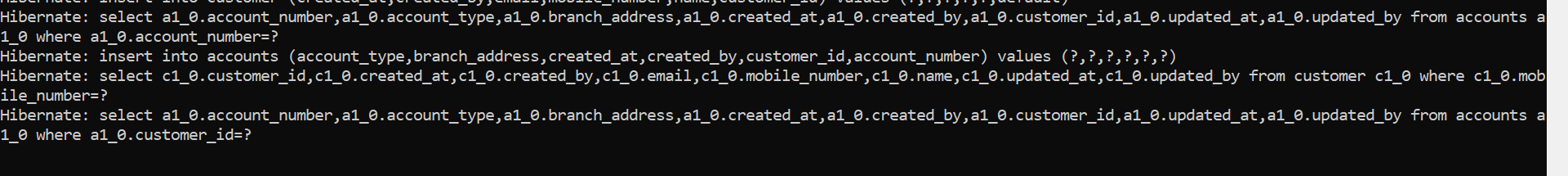
**docker images** – This is the command helps us to list our all the images right now inside our local system.  
REPOSITORY TAG IMAGE ID CREATED SIZE  
varunkumarkm43/accounts s4 1d2dea3b97ec 5 minutes ago 497MB

**docker inspect image and imageId 1d2d**If we try to inspect this image by running a command docker inspect image and imageId 1d2dea3b97ec this is the image id we don’t have to mention the entire imageId instead we can mention the initial 3-4 characters   
[  
 {  
 "Id": "sha256:1d2dea3b97ec1f75988d1a730e36f4b33e1da890d65164defe3e93f870662d6d",  
 "RepoTags": [  
 "varunkumarkm43/accounts:s4"  
 ],  
 "RepoDigests": [],  
 "Parent": "",  
 "Comment": "buildkit.dockerfile.v0",  
 "Created": "2024-10-29T05:54:26.458246431Z",  
 "DockerVersion": "",  
 "Author": "varunkumarkm43.com",  
 "Config": {  
 "Hostname": "",  
 "Domainname": "",  
 "User": "",  
 "AttachStdin": false,  
 "AttachStdout": false,  
 "AttachStderr": false,  
 "Tty": false,  
 "OpenStdin": false,  
 "StdinOnce": false,  
 "Env": [  
 "PATH=/usr/local/openjdk-21/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",  
 "JAVA\_HOME=/usr/local/openjdk-21",  
 "LANG=C.UTF-8",  
 "JAVA\_VERSION=21"  
 ],  
 "Cmd": null,  
 "Image": "",  
 "Volumes": null,  
 "WorkingDir": "",  
 "Entrypoint": [  
 "java",  
 "-jar",  
 "accounts-0.0.1-SNAPSHOT.jar"  
 ],  
 "OnBuild": null,  
 "Labels": null  
 },  
 "Architecture": "amd64",  
 "Os": "linux",  
 "Size": 497296782,  
 "GraphDriver": {  
 "Data": {  
 "LowerDir": "/var/lib/docker/overlay2/c80de1746f86bed97689da0b72d19d53c0accce444728a5bb33aabdd98f0e903/diff:/var/lib/docker/overlay2/bf16baba69016c3d451a08b3661362c7a6975877c5e5cc830aa216a89fdfa994/diff:/var/lib/docker/overlay2/a784ef2464ef648cb23c12d7de36b17e1baff31422b562dcd0bd9399dcc31868/diff",  
 "MergedDir": "/var/lib/docker/overlay2/tmd6s4wo9jalqwb049c3p240i/merged",  
 "UpperDir": "/var/lib/docker/overlay2/tmd6s4wo9jalqwb049c3p240i/diff",  
 "WorkDir": "/var/lib/docker/overlay2/tmd6s4wo9jalqwb049c3p240i/work"  
 },  
 "Name": "overlay2"  
 },  
 "RootFS": {  
 "Type": "layers",  
 "Layers": [  
 "sha256:d310e774110ab038b30c6a5f7b7f7dd527dbe527854496bd30194b9ee6ea496e",  
 "sha256:0ac7ecf8a41c6c682f3194a046f3be4db6950df016790c9dec3fb429dec54ec6",  
 "sha256:659a8c4ba776d3d4d1006ee50db0eb93133ea3acddf224bd9036e965510e698f",  
 "sha256:d6f7aaf8e7da9256871d41227854b4f60457dccaacb4b75026f0daffdcbb9b96"  
 ]  
 },  
 "Metadata": {  
 "LastTagTime": "2024-10-29T05:54:26.773583224Z"  
 }  
 }  
]

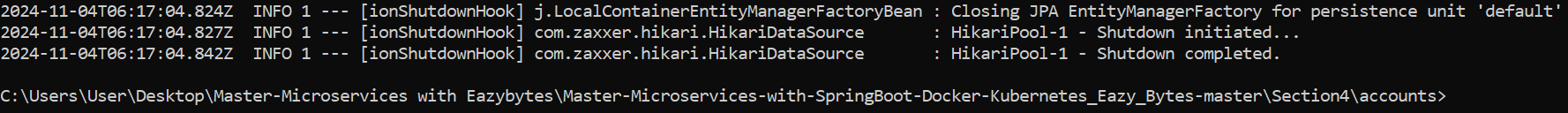
This is the way docker image has been created and we can go to the docker desktop click on the images there is a name accounts, tag is s4 -> click on that name which will shows all the layers available inside our docker image. This confirm our docker image successfully created based upon the dockerfile that we have provided, now we are ready to convert this docker image into a docker container.

**Step -3:** How to run a docker container from our docker image.

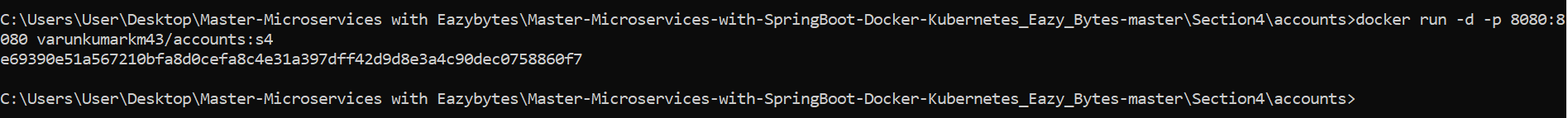
* Using **docker run** command we can run a container from our docker image, this **docker run** command is similar to new operator inside java just like with the help of new operator we can create any number of instances or objects of our class very similarly with the help of docker run command we can create any number of containers from our docker image.
* After giving **docker run -p** p indicates port with the help of port we need to provide a port mapping, all our docker containers going to start inside their own isolated network and we can’t access the services inside our docker network because since it is deployed inside it’s own private network, so that’s why not to get them access from the external network like in our local system or from any other system we need to expose the explicitly with this help of port mapping.
* That’s why we are giving a port number 8080:8080 **docker run -p 8080:8080** with this what we are telling to our docker container is our docker container is going to start 8080 our accounts microservices also running the port 8080 that’s why we need to mention the port at which container is going to start with the help of this second port whereas the first port mapping which is highlighted with that we are telling to the docker please expose the container to the outside of the docker network at the port 8080. So, anyone who want to communicate with these containers from outside of the docker network they have use the port which is 8080.
* Once we provide these port mapping, we need provide that what is the docker image name so from which docker image we want to create a docker container so that docker image is **varunkumarkm43/accounts:s4** this is the complete docker image name.
* Finally, we execute this command **docker run -p 8080:8080 varunkumarkm43/accounts:s4** Once execute this command our docker container is being started which means our accounts microservice behind the scenes is getting started at the port 8080 and with this we are able to successfully started our docker image as a docker container.
* Once the above command running successfully afterwards, we can’t execute any other commands.



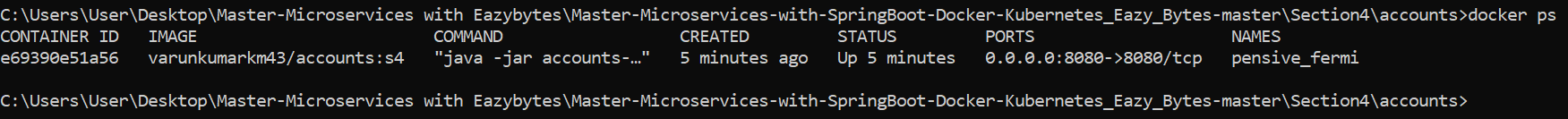
* That’s why we can start the container in a detached mode for that we have to kill this container or stop this container by pressing **control c** so with that our container will stop.



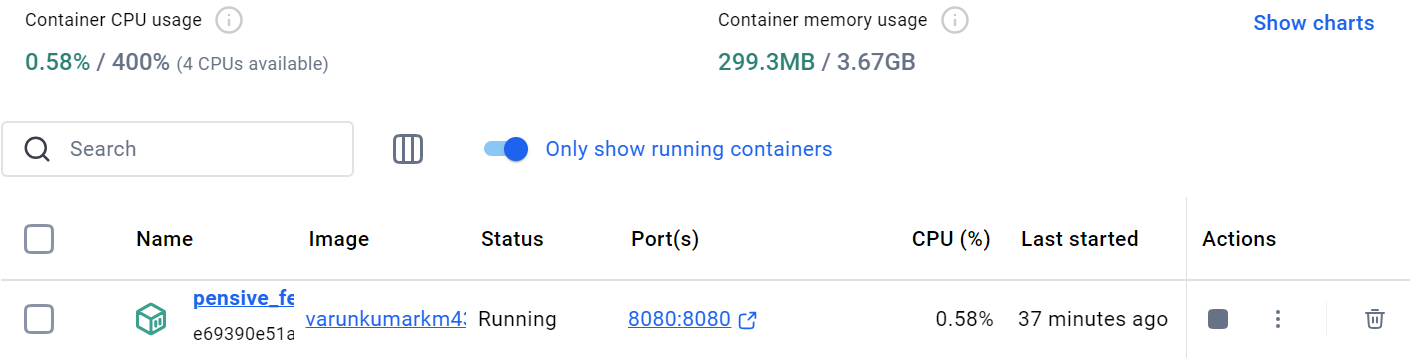
* After this we need to execute the same command **docker run -p 8080:8080 varunkumarkm43/accounts:s4** inside this command just after the run going to add a new flag which is **-d** this d indicates detached mode, so we are telling to our docker server to run the container behind the scenes in a detached mode and we don’t want to see any logs inside our terminal **docker run -d -p 8080:8080 varunkumarkm43/accounts:s4** Once we execute this command we can see container id is displayed **e69390e51a567210bfa8d0cefa8c4e31a397dff42d9d8e3a4c90dec0758860f7** like this is the container id behind the scenes it is running now we can also run any other docker commands inside our terminal.



We can see all the existing running containers with the help of **docker ps** execute this command we will see what are the existing containers that are right now running inside our system.



**docker ps -a** – Using this command going to display all type of containers that right now are in stopped status



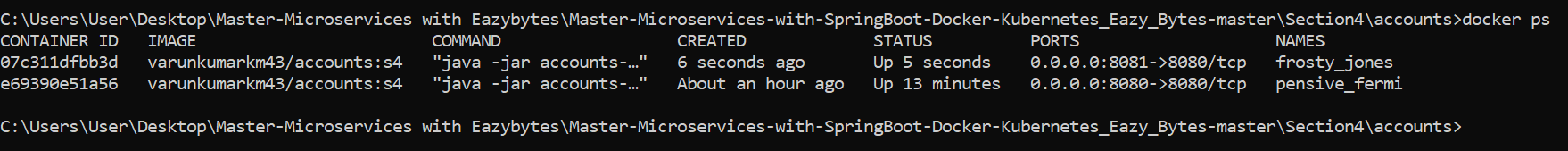
This is the running container showing inside the docker, we try to click on the running container like that is the random name that is given to our container by our docker server so we can try to click on this, we will be able to see they are:

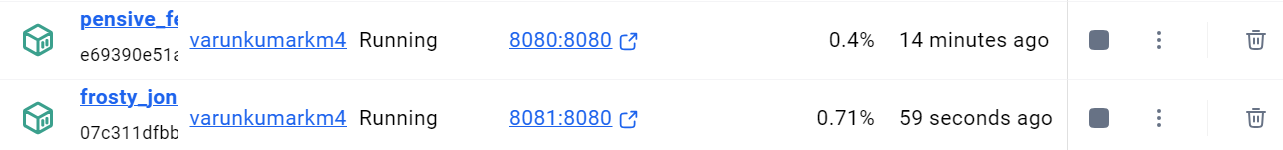
* **Logs** - logs of our container, if we try to invoke our REST Apis and if the generate any logs we can see them here.
* **Inspect** – we will be able to see the details about our container what is the java version, what is the java home, and what is the port where it is being started.
* **Files** – we will be able to see all files inside our docker image.
* **Stats (statistics)** – which will show the CPU usage, memory usage, and other related resources usages from our container.
* **Terminal or Exec** – when we click on the terminal, we are actually inside the docker running container. Whenever we want to run some commands inside our running container we can always come to this terminal and executes some commands.

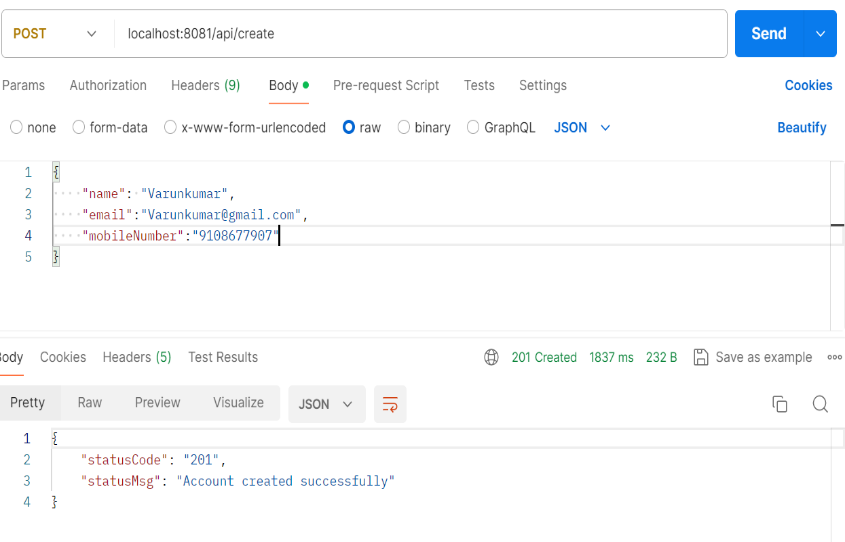
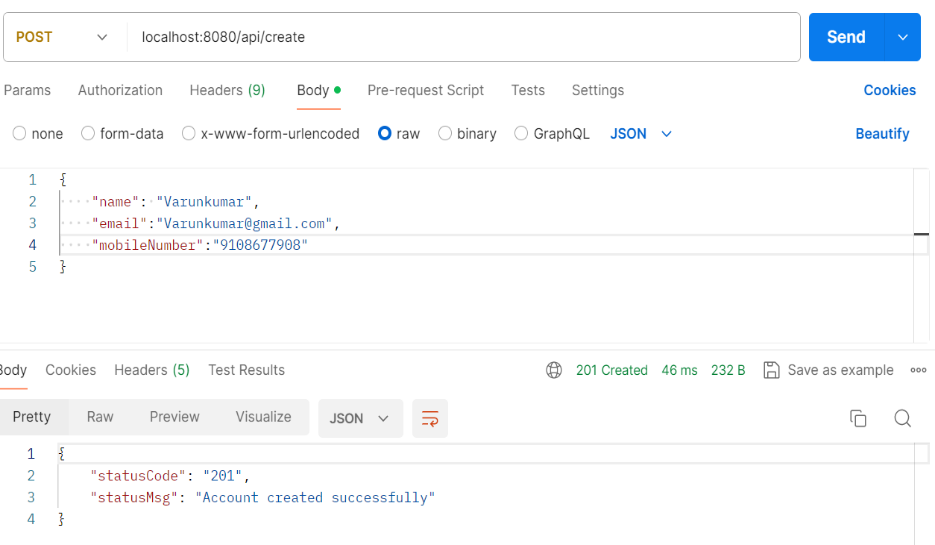
We can start restart our container another way like stop the container manually in the docker and copy the container id e69390e51a567210bfa8d0cefa8c4e31a397dff42d9d8e3a4c90dec0758860f7and we can run a command in the terminal **docker start e69390e51a567210bfa8d0cefa8c4e31a397dff42d9d8e3a4c90dec0758860f7** container id. This is the container id we can also give 1st four digits or 1st four characters as well. So as soon as run this command we should be able to see the existing container might have started. Wea can also stop the same container using the below command. **docker stop e69390e51a567210bfa8d0cefa8c4e31a397dff42d9d8e3a4c90dec0758860f7**

8080 – This port will expose our local system we can’t run the same port, because our local system everything is a same network and that’s why we can’t reuse the same port number :8080 – This port number indicate the port number where our container is going to start at inside the docker network. So, this container and the second container will have their own isolated network which means they can use the same port number and it will not create any issues.

**docker run -d -p 8081:8080 varunkumarkm43/accounts:s4** – this command is starting the different container.





Like we can see there are two running containers, now we can also validate these containers by invoking few REST Apis. This way we can create any number of containers, this is help us to deployment, portability, and scalability issues, we can see how easily going to scale one instance to another instance. We can create any number of instances, coming to the portability the same docker image can be taken by anyone as long as they have the docker install inside the system. The same application is going to work very similarly like in the local system, with this docker image they don’t have download and install any JDK, spring boot, Maven all the setup is already included our docker image we just have to run the docker run command and do a port mapping and provide our docker image name with that the portability issue also resolved and coming to the deployment the same docker image we can deploy inside our local system, inside a virtual machine, cloud server etc. wherever we are trying to deploy the command is going to the same, the docker image is going to same, nothing is going to change and this will make very easy in terms of deployment whenever we trying to move our microservice from one environment to other.

**Running a spring boot app as a container using Dockerfile:**

Steps to be followed: These are the main steps to create docker image using Dockerfile.

1. Run the maven command “**mvn clean install**” from the location where pom.xml is present to generate a fat jar inside target folder.
2. Write instructions to docker inside a file with the name **Dockerfile** to generate docker image. Sample instructions are mentioned on the left-hand side.
3. Execute the docker command “**docker build -t varunkumarkm43/accounts:s4”** from the location where Dockerfile is present. This will generate the docker image based on the tag name provided.
4. Execute the docker command “**docker run -p 8080:8080 varunkumarkm43/accounts:s4”.** This will start the docker container based on the docker image name and port mapping provided.

So, if their unused containers are running so that the memory and storage usage inside our system will be used properly otherwise our system is going to get hang when we are using the system is very low ram and storage.

**Disadvantages and defects of the Dockerfile approach:**

1. In order to write a Dockerfile we need to be expert of docker concept, we need to use some commands and tags, but in real time projects the application will large, this simple docker file may not be work. Convert our microservices into docker image we need to learn lot of docker concept, need to learn lot of instructions and how to provide them with the help of docker file, so there is a lot of learning involved for the developer. Why should developer learn everything about docker because he is not a deveops person not a platform team member, so that’s why this option is not good for the developers.
2. Developers are always looking for an approach where the docker image is going to be generated automatically without writing any low-level instructions inside dockerfile. That’s where we have solutions like **buildpacks** and **google jib** came into picture.
3. **Buildpacks approach:** [**https://buildpacks.io/**](https://buildpacks.io/)

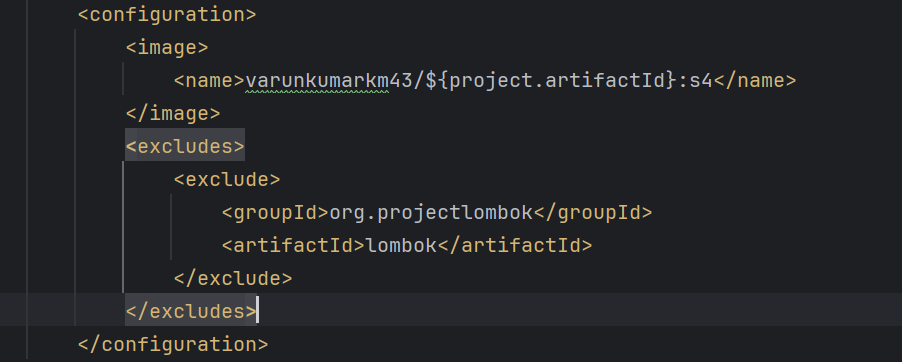
Using buildpacks we can transform our application code into docker image that can run on any cloud there is no need of writing low level instructions to the help of docker file with a single maven command we can generate a docker image very easily. So, this buildpacks is developed by **Heroku**.

Buildpacks is a framework or an eco-system or a concept based upon this ecosystem we have **paketo** buildpacks we can use for java-based applications apart from java it is also going to support many other languages.

**Generate Docker image of loans microservices with Buildpacks:**

**Step – 1**: Goto the loans microservice pom.xml file under the version element tag we have to add **<packaging>jar<packaging>** and we need to make sure we have a plugin related to the maven inside our pom.xml with the help of this spring boot maven plugin we can generate a docker image and behind the scenes spring boot maven plugin is going to leverage buildpacks and paketo to generate a production ready docker image.

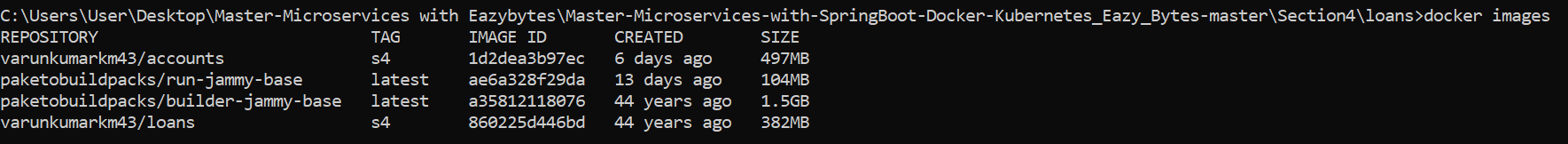
But in order to generate a docker image first we need to provide what is the name that we want to consider for the docker image that we are going to generate. That’s why inside the configuration we need to invoke a tag which is **image**. So, inside this image tag we need to invoke one more tag which is **name.** under this name tag we need to provide image name. That docker image name is very similar to the accounts microservices name.



**Step – 2**: Take the command prompt (CMD) in the loans microservices, here we need to issue a maven command **mvn spring-boot:build-image** so with this command we are telling to the maven to generate a docker image of our spring boot microservice by leveraging buildpacks. So, behind the seances it going to leverage the buildpacks. If we are doing this very first time it is going to take some mins because it has to download all the buildpacks or paketo related libraries and images inside our local system.

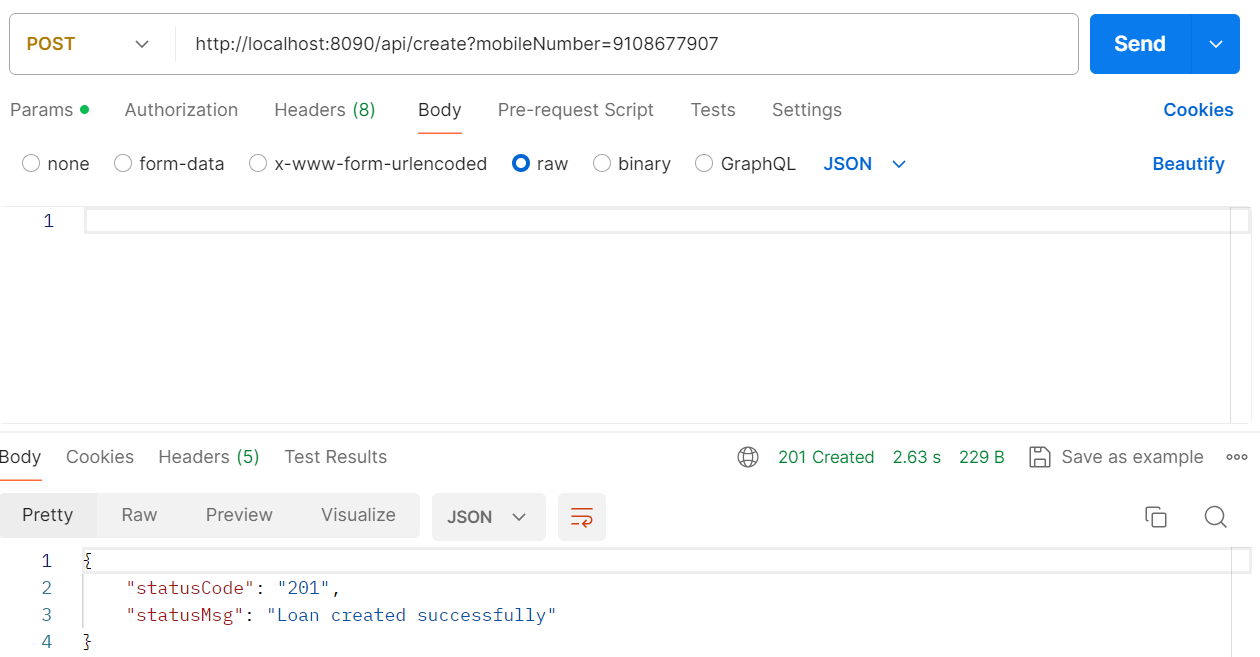
**Step – 3**: If we see for accounts microservice we have use the docker file, size of the accounts docker image is 497MB, because we don’t know how to follow the best standards, how to cash multiple layers, how to compress our multiple components inside docker image, we don’t know all those standards and the docker concept. But with the help of buildpacks there is lot of improvement in terms of the size of the docker image it reduces 497MB – 382MB so, this one of the grate advantages. If we use the product called buildpacks.

We can also see paketobuildpacks docker image is downloaded so this image is going to do all the work of generating a docker image for our spring boot microservice and this is for 1.5GB. This is a very heavy image; we don’t have to take this into our deployment we just need only the loans docker image.



**Step – 4**: Now we can try to run the loans docker image as a container for the same the command is very similar; we are executed by the accounts microservice. **docker run -d -p 8090:8090 varunkumarkm43/loans:s4** – Execute this command the docker container will get created behind the scenes, in case see any warning we can ignore that.

After that will check the Rest Apis in the postman tool, this confirms our container successfully executed.



Cloud native buildpacks offer an alternative approach to Dockerfiles, prioritizing consistency, security, performance, and governance. With Buildpacks, developers can automatically generate production-ready OCI images from their application source code without the need to write a Dockerfile.

Steps to be followed:

1. Add the configurations like mentioned on the above image inside pom.xml file. Make sure to pass the image name details.
2. Run the maven command “**mvn spring-boot:build-image**” from the location where pom.xml is present to generate the docker image without the need of Dockerfile.
3. Execute the docker command “**docker run -p 8090:8090 varunkumarkm43/loans:s4**”. This will start the docker container based on the docker image name and port mapping provided.
4. **Google Jib**: <https://github.com/GoogleContainerTools/jib>

Google jib offer an alternative approach to Dockerfiles, prioritizing consistency, security, performance, and governance. With Jib, developers can automatically generate production ready OCI images from our application source code without need to write a Dockerfile and even local Docker setup.

**Step - 1**: We have gone to this GitHub repository -> scroll down there are some instructions how to use Google jib containerize our java applications.

Note – Jib is going to work only for java applications where as buildpacks is going to work as many other famous languages like ruby, phyton, node js etc. whereas jib is strictly for java applications. Since we are building java based microservices we can use this jib without any issues.

**Step – 2**: We have an option **Maven** and **Gradle**, currently for the cards microservices we are using Maven -> click on maven -> it will redirect to the jib maven plugin is available. -> we can scroll down here there are details how to get started.

1. Try to delete the existing contents of target folder.
2. Mention the packaging details under the version element tag **<packaging>jar</packaging>**
3. Inside cards microservice pom.xml file we need to paste the plugin and make sure copy the plugin details into the jib GitHub repository, which is help us to generate a docker image.

<plugin>

<groupId>com.google.cloud.tools</groupId>

<artifactId>jib-maven-plugin</artifactId>

<version>3.4.4</version>

<configuration>

<to>

<image>myimage</image>

</to>

</configuration>

</plugin>

1. Image name we are followed similar to the loans and accounts microservices **varunkumarkm43/${project.artifactId}:s4** this is the image name for the cards microservices.
2. Once its completed, we can scroll down the GitHub repository we have to execute this command **mvn compile jib:dockerBuild** – use this command in our local server setup to generate a docker image execute this command inside our cards microservice terminal.
3. It scans all the details inside our pom.xml file and it is going to generate a docker image for our cards microservice. And **this jib is going to be faster than buildpacks.** Because jib create an image quickly compare to the buildpacks.
4. Check the status using the command **docker images** we got a docker image with the name cards and this also is having a size of 322MB which is very similar to buildpacks. This confirms jib also doing the good job it following all the standards and providing a production ready docker image.
5. We can see the created value which 43 years ago and 53 years ago for both jib and buildpacks, this is not a bug this is feature provided by them. The reason why they are using old age is they will have a starting date which is somewhere in 1970 or 1960 which they are use whenever we try to generate a docker image.
6. Since our cards microservices is going to at 8001 we have to mention this port mapping, image name and tag name. **docker run -d -p 8001:8001** **varunkumarkm43/cards:s4** so using this command run the cards microservices docker image. The new container was started we can check into the docker desktop.
7. We have to check our postman tool as well response we getting or not.

**Running a spring Boot app as a container using Google Jib:**

1. Add the configurations like mentioned above plugin element tag inside the pom.xml file. Make sure to pass the image name details.
2. Run the maven command “mvn compile jib:dockerBuild” from the location where pom.xml is present to generate the docker image without the need of Dockerfile.
3. Execute the docker command “**docker run -d -p 8001:8001** **varunkumarkm43/cards:s4”** This will start the docker container based on the docker image name and port mapping provided.

**One more grate advantage using google gib compare to buildpacks:**

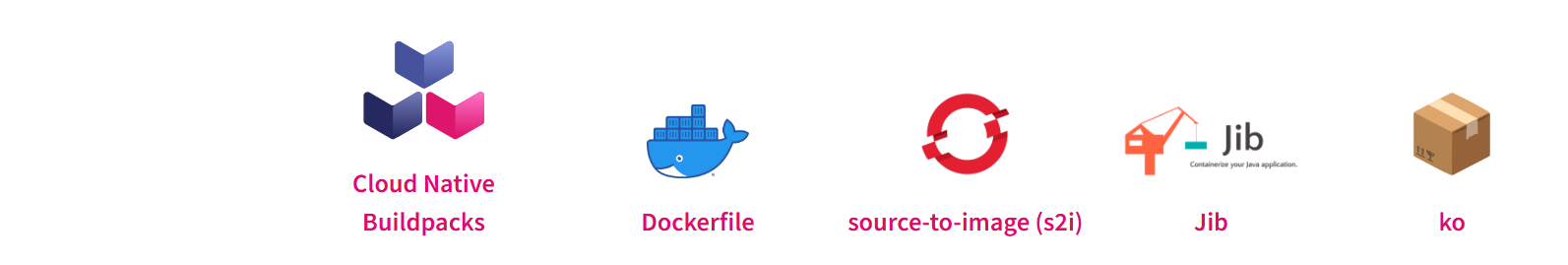
Whenever we using google jib, we can generate a docker image even if we don’t have a docker installed inside our system that is the beauty of google jib. So, if we don’t want to install docker inside our local system but at the same time we want to convert our application to a docker image then we can use a command which is **mvn compile jib:build** when we use this command it is going to generate a docker image from our application and same will be pushed into the remote repository, because if we are not running docker inside a local system that means there is no place for our jib to generate the docker image. That’s why it will look for the remote repository details where it has to store. So, whenever we try to push our docker image into the remote repository it may need some credentials like our docker, amazon, GCP etc. provide those details it will always look that google jib documentation we can knows how to provide.

This approach is going to helpful, but the project where the developers are not responsible to generate docker image. The simply responsible to push the code into the GitHub repository. Inside the GitHub repository using CI/CD tool like Jenkins, GitHub actions, or any other tool we can write the scripts to generate the docker image from our application and inside our CI/CD server we don’t have to install this heavy software which is docker in such scenarios this is going to be helpful.

**We looked at 3 most commonly used approaches to generate a docker image of our microservices. So, that which approach is better and which approach we likely use?**

There is no better approach every approach as its own advantages and disadvantages so based upon our scenario, we need to select one of them approaches.

But in this course, we don’t have to use dockerfile approach, because we are the developers we don’t have to learn all the concepts of dockerfile and follow all the best practices manually we don’t want to take the burden instead we can really on this open-source platforms like buildpacks and Google jib, inside these two approaches again we have a confusion like try to understand the advantage and disadvantages of them. So, inside the **buildpacks.io** website click on the features tab and scroll down there is a comparison table provided by the buildpacks team among the various approaches we can use to generate a docker image. Can we see here buildpacks is the winner in most of the scenarios.





The approach that we are going to use **Google jib** in this course. There are three different reasons to use google jib approach inside this course.

1. Jib is going to take very less time to generate a docker image it is going to super quick compare to build pack. Also going to take less memory inside our system.
2. Right now, our microservice is only java-based service we don’t have any intension to develop microservices based upon the other languages. That’s why we are not going to choose buildpacks. But in real project where we have multi-level language microservice please follow the buildpacks.
3. The last reason is buildpacks have some issues in the mac operating systems. There are some official defects are also logged inside buildpacks and the buildpacks team is trying to fix them in the coming months or it may take years as well.

As of now we have the docker images generated or available inside our local system, if we keep them inside our local system then obesely this is not going to make scenes, because the images have to be eventually deployed into our production server or inside a dev server or QA server. To achieve this deployment in higher environment just like how we push our code into the GitHub very similarly we need to push our docker images into a remote repository provided by the docker hub or any other private registries available inside AWS, GCP, Azure so there are many private registries available even from GitHub there is private registry. But in this course, we are going to store our all docker images inside the remote repository provided by the docker hub. For the same we need to push our docker images from local system into the remote docker hub repository.

The command that we need to run to push our docker images into the docker hub repository is **docker image push docker.io/varunkumarkm43/accounts:s4   
docker image push docker.io/varunkumarkm43/cards:s4  
docker image push docker.io/varunkumarkm43/loans:s4**

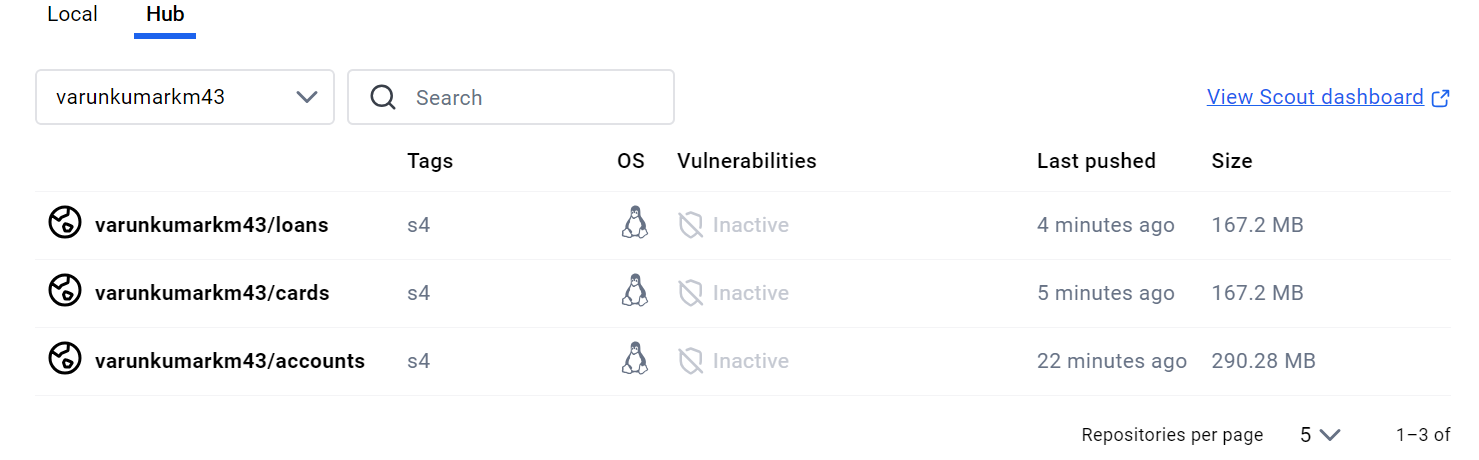
So, this is the complete command that we need to use to push the image from our local system to the remote repository.

So, here we didn’t give our credentials to push into the docker hub there is a password for our account inside the docker hub but we have not provided that password or username anywhere inside our command, here what our CLI will do we are already logged in into docker desktop with the same username **varunkumarkm43** it is going to fetch our username and password details from the docker desktop. So that CLI and docker desktop there are going to work together and based upon the credentials that we have used to login into our docker desktop to same it is going to be utilize while trying to push our image into a remote repository.

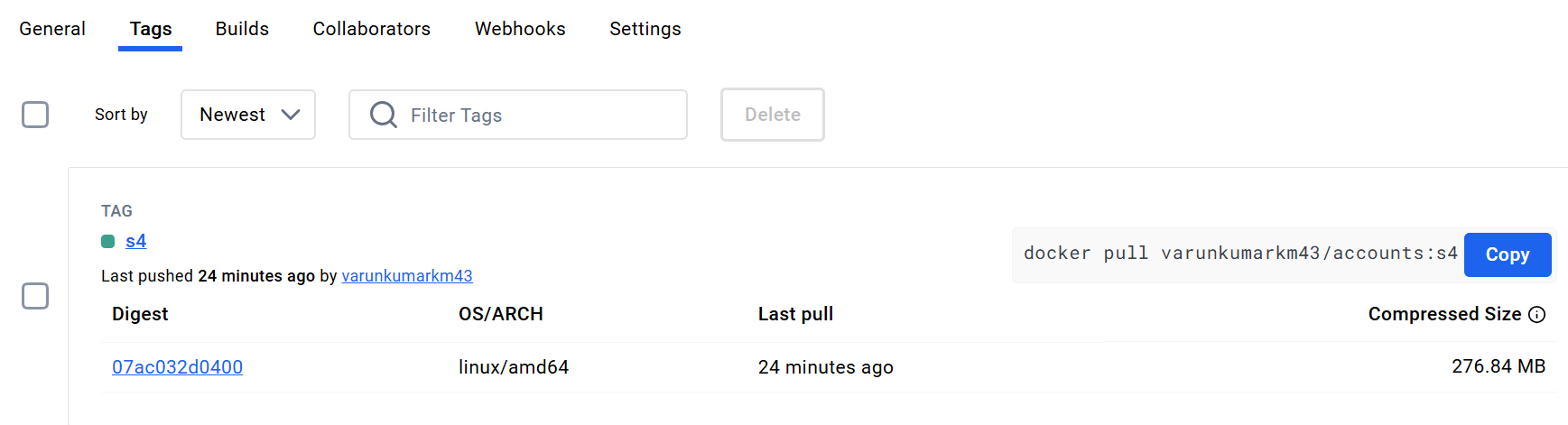
If we are not logged in into our docker desktop or if there are no login details configured inside the local system then definitely, we will get an error saying that login failed or access denied.

To validate if these images are really pushed into the remote repository or not, we have to go to the docker desktop dashboard in the **hub** tab we ned to check the details. We can also validate these details going into the **docker hub website.**

This is very similar to GitHub private repository and public repository so this was very similar to GitHub, if we want to change this docker image from public view to private view then we need to -> Go to the setting -> Here under the visibility setting make this a private. Like with the basic personal plan we can have only one image as a private repository, but we are not going to make it has private. These images can make it has public.



We want to see the tags in the docker hub in the docker hub website -> tags we can see



As of now we have only one tag which is s4, if anyone want to use this docker image with this tag, they can simply execute this command **docker pull varunkumarkm43/accounts:s4** ->EX: Goto the docker desktop first we try to delete the cards microservice container and image -> then execute this command in the command prompt this will download that image from the remote repository into our local system.

**Introduction to docker compose:**

As of now we have three different images like accounts, cards and loans these are the docker images of three microservices that we have built so far. To run our microservices we need to convert this docker images into containers with the help of docker run command. So, if we want to start our three microservice we need an issue with docker run command three different times with the image name and the port mapping. What if we want to start multiple instances of our microservices in such scenarios giving this docker run command manually for each microservice and for each instance it is difficult and time-consuming process. Suppose if we have 100s of microservices we need to issue this docker run command 100 different times which is not a good option.

To overcome this challenge, we have another component inside docker eco-system with the name **docker compose**. <https://docs.docker.com/compose/> - This is the details about the docker compose.

Docker compose is a tool for defining and running multi-container docker applications. With Compose, we can use a YAML file to configure our application services. Then, with a single command, we create a start all the services from our configuration.

Compose wors in all environments like production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of our application.

* Start, stop, and rebuild services
* View the status of running services
* Stream the log output of running services
* Run a one-off command on a service

The key feature of compose that make it effective are:

* Have multiple isolated environments on a single host
* Preserves volume data when containers are created
* Only recreate containers that have changed
* Supports variables and moving a composition between environments.

This way it has many advantages, so whenever we install docker desktop inside our system by default docker compose related component also will be install we can also validate the same by running a command **docker compose version** This will print the what is the docker compose version **Docker Compose version v2.29.2-desktop.2** so this is the version of docker compose. This also confirm that docker compose is installed our machine.

If not install we have Go to this link <https://docs.docker.com/compose/> -> click on the Manuals tab -> install docker compose -> overview -> based on the operating system we need to install docker compose.

Now to get started with the docker compose, we need to write a configuration file and inside this configuration file we need to define all the details about our docker images and how we want to start them.

Step -1: we need to create a configuration file into our accounts microservice -> we can create this file any location, but since we want to check in this file into the GitHub repository and trying to create inside this accounts microservice maven project. Right click on the accounts -> create new file -> file name will be **docker-compose.yml** extension should be in yml because we are going to provide all our configuration details to the docker server using yml format. -> Here we want to define all our service details.

services:  
 accounts:  
 image: "varunkumarkm43/accounts:s4"  
 container\_name: accounts-ms  *if we don't give this container name when we are trying to create docker image random name will be given to our container by the docker server.* ports:  
 - “8080:8080”

When we create a port initially, we are using (-) this hyphen indicates this is the single element inside an array. If we need multiple port mapping by pressing enter, we can create.

Now whenever our container is being created, we want to make sure that it has a maximum memory allocation beyond that we don’t want to assign that because inside our local system we have a minimum ram we are going to create multiple containers inside our course we should make sure we are restrict our containers to a maximum memory.

deploy:  
 resources:  
 limits:  
 memory: 700m

With this we are telling to this accounts microservice at a max we want to put provide 700mb memory beyond that our docker server not going to assign.

If we see there is a hierarchy that we followed inside yml file services is the parent and under this services there is a child with the name accounts and under the accounts we have details like image, container name, ports mapping and what are the deployment instructions under the deploy we have resources under we have limits and under we have memory.

We copy same thing and same position for creating a loans and cards microservices

services:  
 accounts:  
 image: "varunkumarkm43/accounts:s4"  
 container\_name: accounts-ms   
 ports:  
 - "8080:8080"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
   
 loans:  
 image: "varunkumarkm43/loans:s4"  
 container\_name: loans-ms  
 ports:  
 - "8090:8090"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
   
 cards:  
 image: "varunkumarkm43/cards:s4"  
 container\_name: cards-ms  
 ports:  
 - "8001:8001"  
 deploy:  
 resources:  
 limits:  
 memory: 700m

Now if we can try to start our microservices using this docker compose file all these services are going to start with the single command.

As of now there is no dependency between these microservices they are all going to start in different isolated network what if we have a dependency between this microservices or what if in future we have a scenario where these microservices need to talk with each other in that scenario it is not going to work by default because all this services are going to get created inside their own network, so to make shore that the intercommunication is working we need to tag all these microservices under a same network for that we have to do is:

Just after the deploy instructions mention networks after we can define any number of networks this mention for all the services. Make sure this presence in the same position. We can come to the place where the services are mentioned in the very first line, here we need to invoke a root element inside our docker compose, the root element is networks: eazbank just like how we have services at the root level similarly we can also define networks. Mention the networks name as same name as networks post that eazybank: following by the driver: “bridge” with this we are telling to the docker server please create a network with the eazybank with the driver bridge since we are using same network inside our accounts and loans microservices and the cards our docker server is going to establish a bridge using which all our microservices that we can communicate each other.

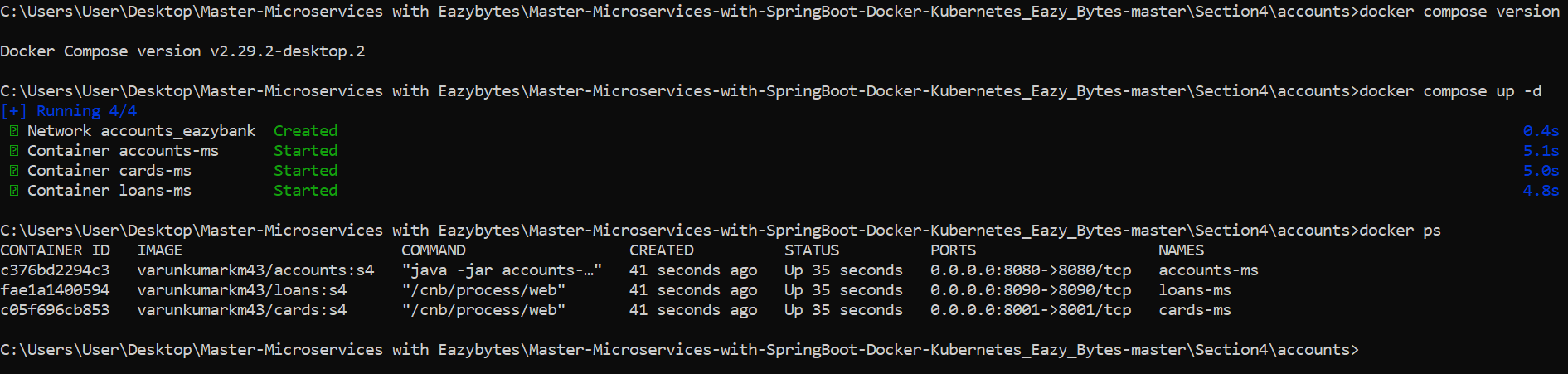
So, this way we have define all our instructions inside this docker compose yml file. Right now, our docker compose file is ready. The below mentioned is final docker compose file.

services:  
 accounts:  
 image: "varunkumarkm43/accounts:s4"  
 container\_name: accounts-ms  
 ports:  
 - "8080:8080"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - eazybank  
  
 loans:  
 image: "varunkumarkm43/loans:s4"  
 container\_name: loans-ms  
 ports:  
 - "8090:8090"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - eazybank  
  
 cards:  
 image: "varunkumarkm43/cards:s4"  
 container\_name: cards-ms  
 ports:  
 - "8001:8001"  
 deploy:  
 resources:  
 limits:  
 memory: 700m  
 networks:  
 - eazybank  
networks:  
 eazybank:  
 driver: "bridge"

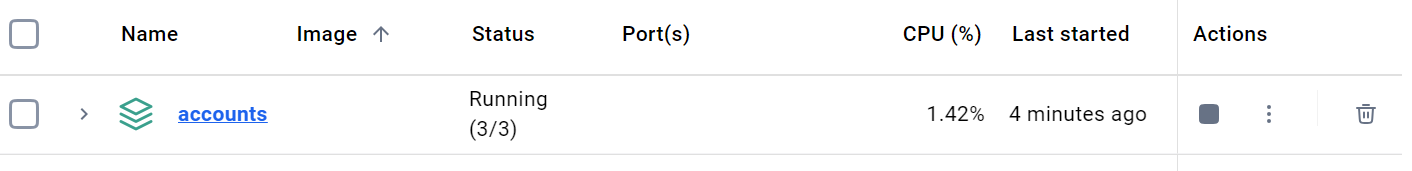
**Running all microservices containers using Docker Compose command:**

Using the **docker compose up** command, with this command we should be able to start at all our microservices that we have define inside our docker compose yml file, but we make sure we are running this command from the local where our docker compose yml file is located.

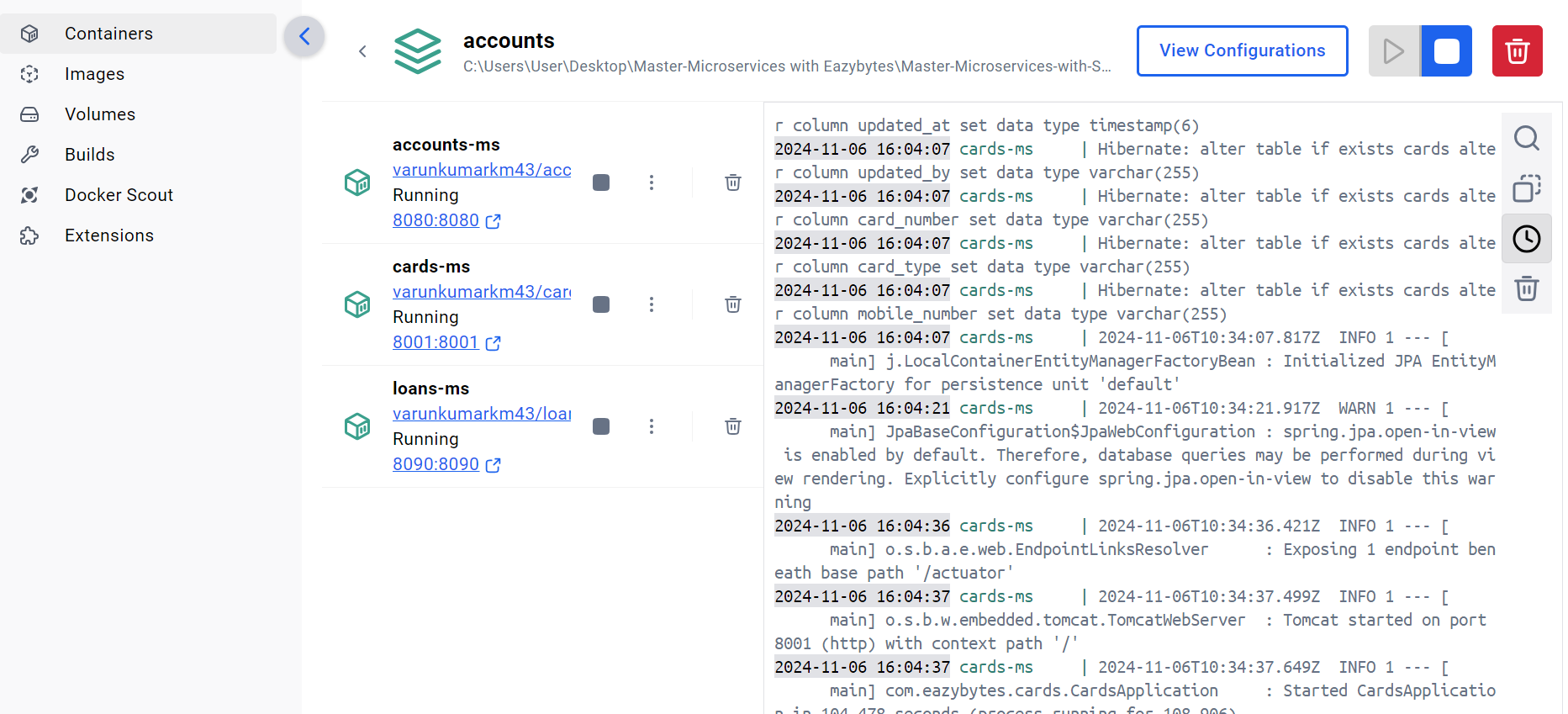
Now we are running **docker compose up -d** command in the docker compose file present directory, if we try to run this command all our containers is going to start and the same log is going to be display inside our terminal, we don’t want to keep our terminal busy, so that’s why start in a detached mode.



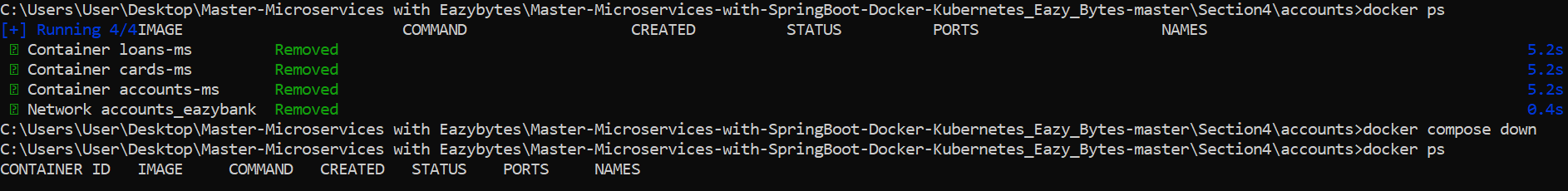
Using this single command, we can able to start three different microservices that are running with the specifies port number.



This is the parent folder under this accounts folder we have three different microservices are running. Finally we can check into the postman all our services up and running.

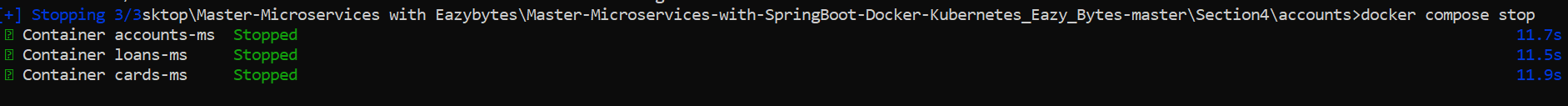


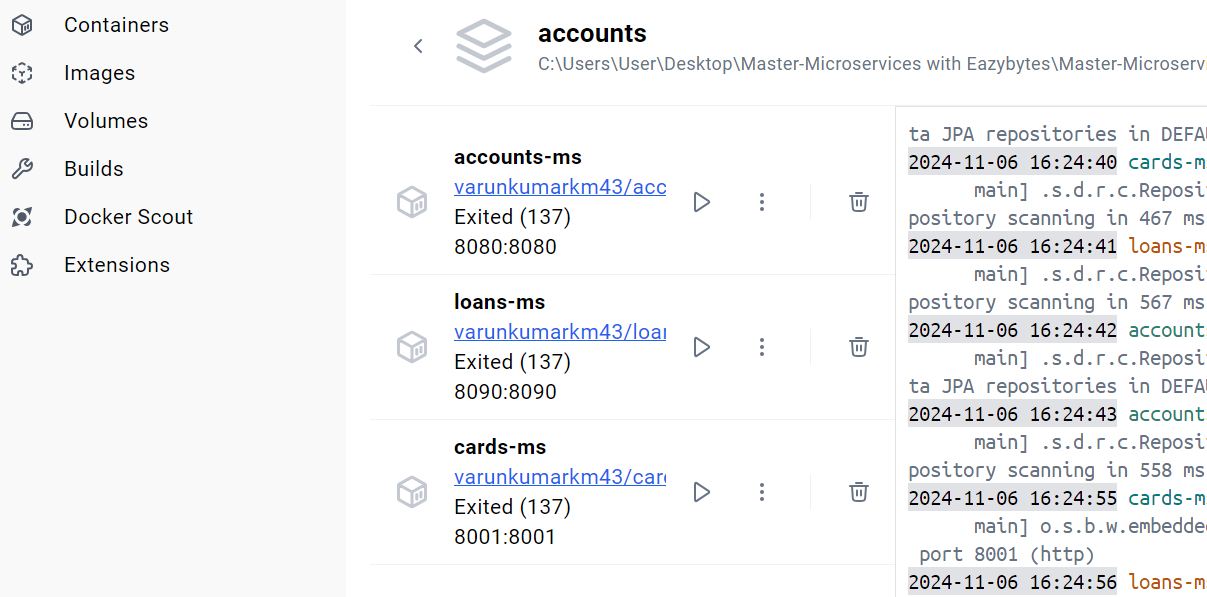
We can use single command to run all the containers and we can also use single command **docker compose down** to stop all the containers. We can see the docker desktop, our all containers are no more there and all are deleted. If we don’t want them to be deleted we can run a command which is **docker compose stop** But preferred always use docker compose down command.



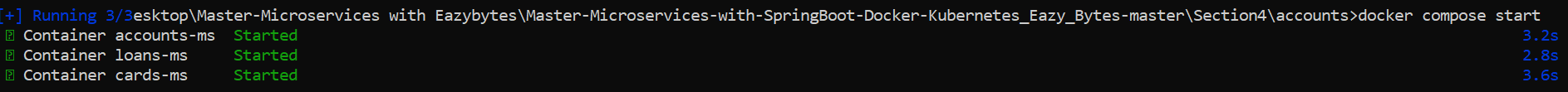
**Demo for Docker compose commands**:

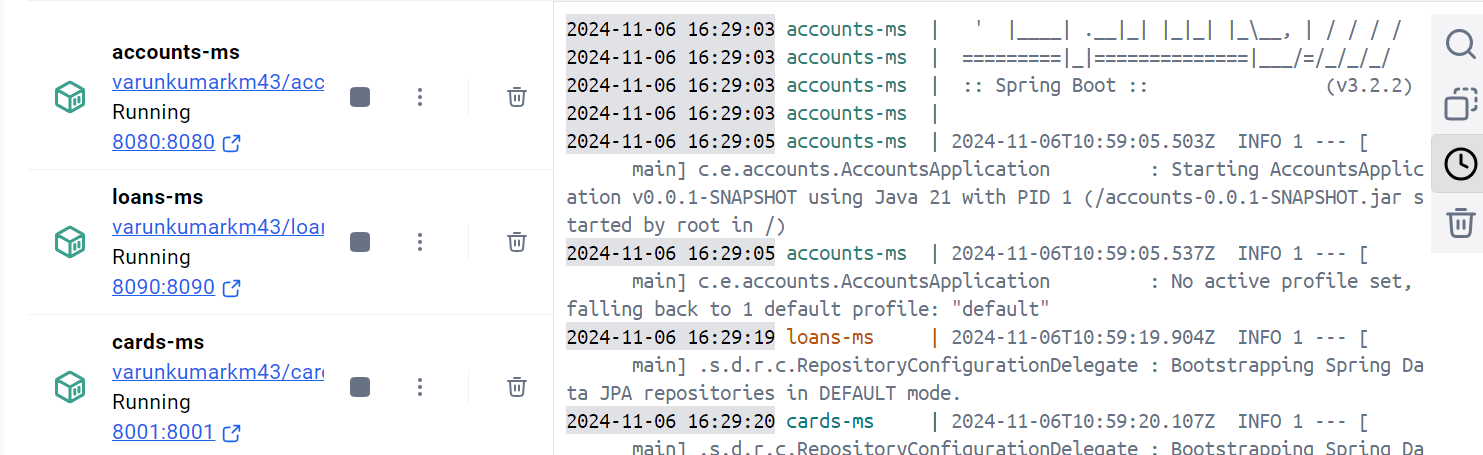
Whenever we use the **docker compose up -d** command it is going to create our containers from scratch.   
Whenever using **docker compose stop** command our all the containers are stopped but not deleted





And again, using **docker compose start** command our containers are up and started.





Since we are using **docker compose down** command after that we are using **docker compose start** command it won’t work it simply fail the reason there is no service with cards, loans and accounts it showing.

**Important Docker commands**:

1. **docker images** – To list all the docker images present in the docker server.
2. **docker image inspect [image-id]** – To display detailed image information for a given image id. So, whenever we trying to pass an image id or container id or any other id inside docker we don’t have to mention the entire id value we can simply mention first 3-4 characters and our docker server is smart to detect the entire image id or container id based on the short value.
3. **docker image rm [image-id]** – To remove one or more images for a given image id, **rm** indicates remove.
4. **docker build .-t[image-name]** – To generate a docker image based on a Dockerfile. We should provide the complete docker image name with the help of **-t** indicates tag.
5. **docker run -p[hostport]:[containerport][image\_name]** – To start a docker container based on a given image.
6. **docker ps** – To show all running containers.
7. **docker ps -a** – To show all containers including running and stopped.
8. **docker container start [container-id]** – To start one or more stopped containers. We want to start a specific container which is previously stopped then we can run this command. If we want multiple containers to start then we can mention all those containers id separated by a space value.
9. **docker container pause [container-id]** – Containers to pause for sometime or if we want our container not accept any traffic then we can run this command along with the container id.
10. **docker container unpause [container-id]** – Once the container is paused definitely after sometime we want unpause it in such scenarios we can run this command.
11. **docker container stop [container-id]** – To stop the running container whenever we are we want to stop running container we need to use this command. Whenever use this command our docker server is going give some time for our container to close any resources which means our docker server is not going to kill our running container instantly. It is going to give some time 5secs and within this 5secs our docker container close any resources that is opened by it like database connections or file systems.
12. **docker container kill [container-id]** – We can kill one or more running containers instantly.
13. **docker container restart [container id]** – To restart one or more containers.
14. **docker container inspect [container-id]** – To inspect all the details for a given container id.
15. **docker container logs [container-id]** – For debugging if we want to see the logs of a container, we can run this command this will fetch all the logs of a given container id.
16. **docker container logs -f [container-id]** – Sometimes we want to continuously follow the logs of a container like we want to see the logs in action what is happening in such scenario we can use this command which tells to our docker server that we want to follow the log output of a given container id continuously inside our terminal.
17. **docker rm [container-id]** – To remove one or more container based on the containers is. Same time we can provide all those containers which we want to remove separated the space.
18. **docker container prune** – This will help us to remove all the stopped containers with using this single command.
19. **docker image push [container\_registry/username:tag]** – To push an image from a container registry.
20. **docker image pull [container\_registry/username:tag]** – To pull an image from a container registry.
21. **docker image prune** – This is going to remove all the unused images. So, how our docker server is going to identify all unused images are if there are no associated containers of an image is present either in start and stop mode then that image will be considered as an unused image by our docker server.
22. **docker container stats** – This will show all containers statistics like CPU, memory, I/o usage.
23. **docker system prune** – Remove stopped containers, dangling images, and unused networks, volumes, and cache.
24. **docker rmi [image-id]** – rmi indicate remove image with this to remove the one or more image which is associate with the image is that we have to provided.
25. **docker login -u [username]** – whenever we want to login into our remote docker hub from our CLI we need to use this command and followed by what is our username. Post that it will ask to the password with that from our CLI we get logged in. But we don’t have to follow this because inside docker desktop there is a signin option which will allow us to signin with the help of browser as long as we logged in with the docker desktop our CLI also is automatically connected to our docker hub account.
26. **docker logout** – To logout form docker hub container registry.
27. **docker history [image-name]** – using this command providing an image name we can see all the intermediate layers and commands that were executed while building an image. So, this is going to help debug our image if we are facing any issues behind the scenes there will be lot of work that will be done by the buildpacks, google jib and docker server while generating a docker image.
28. **docker exec -it [container-id] sh** – Sometimes we may want to run some commands inside our running container so the running container is present in its own network to run any CLI command or any terminal commands from the docker desktops we always have an option to directly go to the terminal and run the commands where as if we want the similar behaviour if the help of docker CLI we can execute this command.
29. **docker compose up** – To create and start containers based on given docker compose file. **docker compose start**.
30. **docker compose down** – To stop and remove containers for services defined in the compose file. **docker compose stop.**

So, these are all the important commands we can these commands in our day-to-day basis and these commands will be also importance for the interview.

**Introduction to docker extensions and Logs Explorer**:

We have 1000 of docker extensions available which we can used based upon our requirement into our day-to-day basis

So, whenever we have a problem while using docker containers, docker images or docker desktop always search for if there is any extension that will solve our problem, we can install these extensions we can easily work.