Introduction to challenges while building, deploying, and scaling microservices

Deployment  
With monolithic application, we have only a single application or a single jar or war or ear file, which we can deploy into your web server or application server.

Whereas with microservice architecture, we are going to have hundreds of microservice in any organization. So, how do we deploy all those hundreds of microservices?

Do we need 100 different servers and virtual machines to deploy them? Off course not.

Portability  
If you think from portability perspective, how do we move our hundreds of microservice across environments with less effort configurations and cost?

From there, we need to deploy into the development environment.

Once the build is stable, we need to move to the UAT environment or sit environment or QA environment.

Once the testing is completed, we need to deploy the same into the production replica environment.

Once the pre-production testing is completed, eventually the deployment must be done inside the production.

If you see, the code must move across many environments. With monolithic, there is only single application, single server. We can move the code by making enough configurations with very less efforts, but with hundreds of microservice who is going to take care of all the portability issues.

If your application needs a specific JDK version or a specific web server or a specific folder structure

or specific DB configurations who are going to do all those manual works when we have hundreds of microservice.

So, these are another problem or another challenge that we may have.

Scalability  
Scaling a monolithic application is going to be super, super easy because you have only a single server.

You can try to onboard one more server, which is replica of the existing monolithic server.

With microservice, the story is different. We may have hundreds of microservice.

How do we scale a specific microservice whenever we want based upon the traffic needs?

So, we need to scale the microservice on the fly based upon the demands.

And at the same time, whenever we don't need, we need to scale down.

So, this all has to happen with the minimum effort and cost, off course, without any manual intervention.

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

So, we need to change our mindset.

We need to change how we package our web applications.

In the traditional approach, you package your application as an war or ear or jar, and you will

eventually deploy into a web server or app server.

But that is not going to work.

We need Some new technique which we can adopt inside our microservice applications.

So, do you have any clue what is this new technique?

Let me reveal the Solution that we can use to overcome this challenge.

***The Solution is containerization of our applications to overcome all these challenges, first, we should containerize all our microservice.***

**Why should we containerize our applications?**

Because when we containerize our applications, we are converting a normal maven project into a container.

These containers are very small in nature, which offer a self-isolated environment for applications,

including all the necessary dependencies.

If your microservice require a Java version or a Tomcat or DB configurations or any folder structure,

you can club all those dependencies inside a container and the same container you can deploy across

any environment or across any cloud environment without making any changes.

**So, the very next question you may have is how do we containerize our Maven applications.**

For the same we have a Solution which is Docker.

So, Docker is an open-Source platform that provides an ability to package and run an application in a

loosely isolated environment called a **container**.

You have a maven application. With the help of this docker, you can convert that into a Docker image and eventually you can run that image as a container inside any virtual machine or inside any cloud environment.

What are containers and how they are different from VMs



New jargons like containerization, containers and dockers.

What are these containers and how they are different from that Traditional virtual machine.

Before cloud computing days, every organization they used to buy their own hardware and using this hardware they will install the operating system.

So, once a hardware and operating system are installed, we can call it as a server and this server will be connected to a public network.

Inside this server, the code will be deployed with the help of web server or app server, and there will be a public IP address assigned to your server.

So, the platform team or operations team, they will map your public IP address to your domain name and using the same domain name, my end user can access my web application from his local browser.

**So, this is the basic and old story that you are aware.**

But with the cloud computing concept, we do not need to buy a server physically and maintain that inside our own data centres.

We can buy a server virtually from the cloud providers and the cloud providers behind the scenes, they will give you a virtual machine.

Why the server is called as virtual machine because you cannot physically see it, you can only feel it virtually and you can access it through the web internet.

That is why they called as **virtual machines.**

**So, let us try to understand how these virtual machines are created.**

Like you can see here at the bottom of this table, we have server. So, this is a physical hardware.

So, think like a cloud provider, like AWS, Azure or GCP.

They will have a physical hardware or a physical server available in their data centres and they cant assign a server to a single organization. Because the server capacity might be 64 GB RAM, and it also, holds four terabytes of hard disk.

So, they cannot give these entire jumbo servers to a single organization because there will be very rare such instances where an organization will request for such a jumbo server.

So, that's why on top of this server they will use a technology or they will use a concept called **hypervisor.**

Using these hypervisor Software from the same physical hardware, they can create multiple virtual machines.

You can see here from the same physical hardware, there are three virtual machines created like VM1, VM2 and VM3.

Inside these, each virtual machines, we can install our own operating system like inside VM1,

I can have a windows operating system and inside VM2 I can have Linux operating system and inside VM3three, I can have Mac operating system.

So, it is up to us what operating system we want to install.

With the hypervisor, while we are creating a virtual machine, we can install required operating system based upon our own requirements.

So, here it is very clear the resources like ram, hard disk of the physical server is virtually distributed between multiple virtual machines with the help of hypervisor concept.

So, now this VM1 can be given to a company one and VM2 can be given to the other company and so, on.

So, now coming to the microservices environment, think like there is no containerization concept.

We must live with these virtual machines in such scenarios.

What we must do, suppose I have three micro services like AccountService, LoanService and CardService.

So, for AccountService I may need Some JDK version and similarly I may need some web server or app server or Maven installed.

So, all the required binaries and libraries I must manually install inside the VM1.

And similarly, I must do inside the VM2 and VM3for the required binaries and libraries of LoanService and CardService.

So, after creating my virtual machine with the help of guest OS and after that I will install the required binaries and libraries post that only I can deploy my Accounts Microservice or Loans Microservice or Cards Microservice into my virtual machine.

Once this deployment is completed, anyone can access my Microservice using the public IP address.

So, behind the scenes, the cloud providers, they will assign a public IP address for each of the virtual machine that they are going to create.

The same public IP will be mapped to your DNS name or domain name and using the same domain name or DNS name, we can access our microservice from outside the world.

**So, this is a story with the virtual machines.**

And here you may have a question.

What is the problem with this?

I can deploy all my microservice into separate virtual machines.

There are multiple problems here.

**First Problem**

I am trying to show you only three microservices. What if you have 100 microservices.

Please do not answer like I'll create 100 virtual machines.

That is going to be a super, super complex process and it is going to attract a lot of cloud Bill for your organization.

So, when you create a virtual machine, it is a big machine.

It is as good as your laptop, which has 16 GB Ram and one terabyte hard disk inside such a big machine, if you only deploy a small microservice, do you think it is a viable solution?

Off course not.

And similarly, you may think like why can't I deploy AccountService, LoanService and CardService inside a single virtual machine?

Even with that approach, you may face some challenges.

Like your Accounts Microservice may need a Java 8 version, whereas your Loans Microservice may need a Java 17 version.

And similarly Cards Microservice may need some other program language like Python.

So, if you try to deploy all these microservices inside a single virtual machine, you see we will face lot many problems when we have different, different requirements for different, different microservice.

**Second Problem**

Like if you try to restart your virtual machine, all your AccountService, LoanService and CardService will be down, which is again a not an acceptable risk by any organization.

**Third Problem**

And at the same time, in terms of scalability, suppose you want more number of AccountService instances deployed in some other virtual machine.

Your Accounts Microservice is receiving a lot of traffic.

All of a sudden you decided to scale up your Accounts Microservice from one instance to three instance with the virtual machine concept.

By the time you create a virtual machine, by the time you install a guest operating system, by the time you complete all the binaries and libraries, installation and scale your Accounts Microservice, it is going to take at least 15 minutes of time.

With such a long 15 minutes of time, you cannot really call it as a real scalability. Because within that 15 minutes your traffic may get reduced.

By the time you scale up your Accounts Microservice, your traffic reduced to normal traffic.

And again, you decided to scale down from 3 VMs to one virtual machine.

Again, deleting the virtual machine also, is not a simple and easy task.

It will easily take again 5 to 10 minutes.

Apart from these challenges, there are many, many other challenges.

If you try to deploy your microservices using a virtual machines concept.

**That is why whenever you are trying to deploy your microservices, please say a big no to the virtual machines.**

Instead, we should go for the containers.

**So, let us try to understand how these containers are different from virtual machines.**

If you see at the bottom the server, the physical hardware is going to be same for the virtual machine and the containers.

On top of this physical hardware as a server we will install a host operating system.

So, here it can be an windows operating system or Linux operating system or Mac operating system.

So, once you have the server and host operating system, you need to install an implementation of container engine products like Docker.

So, once you install the Docker with the help of the Docker, you can create multiple containers.

And inside these containers you can deploy AccountService, LoanService and CardService.

So, if you see the beauty here for the containers, there is no need of separate operating system.

They will share the host operating system.

So, when the guest operating system is not there for containers, they are going to become lightweight and creating a container and destroying a container or restarting a container is going to take only few seconds.

So, that is the very first great advantage that we have with containers compared to virtual machine.

So, they are lightweight and they are going to be super quick in terms of restart creation and destroying.

Now the other advantage that we have with containers is, inside the same virtual machine or inside the same server, they can have a separate isolated environment.

So, think like all these three containers are deployed inside a same virtual machine or inside a same server.

But the advantage with containers is inside the container1, I can have Java eight version and inside the container2 I can have Java 17 version and inside the container3 I can have different language which is Python installed.

So, that is a one more advantage.

Every container will have its own isolated virtual network and virtual environment.

The container 1 will not know about container 2 and its environment, and the same applies for vice versa scenario as well.

Like I said before, these containers are a packed components where all the required dependencies package.

Like if your Accounts Microservice need a JDK and Spring boot related libraries, all of them will be clubbed and be available inside the container 1 itself.

Due to that, whenever you are trying to create new container inside another server or inside another environment.

It is not going to be super, super manual setup.

You do not have to install the Java; you do not have to download the spring boot libraries.

You can just take the packaged component of the container to some other environment and you can convert that package to a container with a single command.

So, that is one advantage.

**So, this will solve the portability issue.**

Now the deployment also, is going to be super, super easy because these are lightweight components and restarting, creating a new container, destroying the existing containers is also, going to take only few seconds and every container will have its own network, own resources, own memory, own storage, which cannot be accessed by the other container until unless Someone allowed it.

So, on a high level, I just wanted to say the main difference between virtual machine and containers is, containers they do not need the guest operating system, not the hypervisor to assign the resources.

Instead, they will use the container engine, which is Docker.

Definition of Containers, Containerization and Docker

**What is a container?**

A container is a ***loosely isolated environment*** inside a server or inside a virtual machine or inside your local system as well.

Since you are getting a loosely isolated environment inside a machine, you can use that isolated environment to ***deploy your microservice*** by using ***software packages***.

So, these software packages include all the code and all the dependencies to run your applications or microservices quickly and reliably on any computing environment.

Your containers to run, they just need the required code and dependencies along with the computing environment.

And this environment can be your local system.

It can be a virtual machine inside a data centre, or it can be a virtual machine inside a cloud environment.

So, the computing environment can be anywhere your container is going to work very similarly, like it works inside your local system.

That is why, regardless of how many environments you are trying to deploy your microservice, you are going to use the same container and there will not be any manual efforts to set up your container to install the dependencies to install the required servers.

So, everything is going to be available inside the container itself.

Like I said, all the required dependencies and libraries are available inside Software packages and these packages we call them as ***container images***.

So, from our microservice application we are going to generate a Docker image or container image.

From the container image, we can create any number of containers.

In other words, your Docker container is an actual running representation of your Docker image.

**What is Software containerization?**

So, using these containerization concepts only, we need to convert our microservice applications into Docker containers.

So, what is the Software Containerization?

It is an operating system virtualization method.

Like we discussed in the previous lecture, we can deploy multiple containers inside a single machine or a virtual machine and all these containers, they are going to get a virtually isolated environment and these containers, they will feel like they are running inside their own operating system.

So, with these containerization concept, we are virtualizing an entire operating system.



Whereas with hypervisor, if you see we are virtualizing the machines, all these virtual machines like VM1, VM2, VM3, they will feel like they are all running in different, different physical hardware’s, but they are not because with the help of hypervisor we are virtualizing the machines very similarly with the help of containerization, we will virtualize the operating system.

So, all these containers 1, container 2, container 3.

They are running on a same operating system which is host operating system, but they will feel like they are running in a separate, separate operating systems.

So, since we are providing such kind of virtual isolated environment

**inside the containerization concept, the virtualization is at the operating system level,**

**whereas inside the virtual machines, the virtualization is at the hardware level.**

**What is a docker?**

Docker is an open source platform or a product that enables developers to convert their web applications or to convert their application code into a Docker image.

And with the help of this Docker, we can automate the deployment, scaling and management of the applications. with the concept of containerization.

I hope you can tie the strings here between these three components.

Software containerization is a concept.

Docker is a platform which, it implements this containerization technology.

So, using the Docker, we are going to generate the Docker images and Docker containers from our application code.

**What exactly happens in containerization?**

Now let us try to understand what exactly is going to happen inside the containerization concept.

Like I said, containers, they are going to get isolated environment, how they are going to get because containers are based on the concept of operating system virtualization, where multiple containers run on the same physical or a virtual machine that ***shares the same operating system kernel***.

And this is different from the traditional virtualization that we follow whenever we are trying to create multiple virtual machines.

And these virtual machines run on their own separate operating system instance, whereas with Dockers they are going to share the same operating system kernel.

So, inside this containerization Linux features like **namespace** and **cgroups** play a crucial role in providing the isolation and resource management.

***So, let us try to understand what are these namespace***

Namespace is a concept inside Linux, which will allow for the creation of isolated environments within the same operating system.

So, each container will get its own set of namespaces and this namespace is going to have process resource, network resource and any storage resources, communication resources and similarly user namespaces.

So, these namespaces ensures that process within a container are only aware of and can interact with the resources within their own specific namespace.

So, this provides a level of isolation from the other containers.

I hope you understand it is very simple.

Using namespace for a container, we are giving its own network process, storage everything. So, that all the components inside the container are inside the namespace.

They can interact with each other, but not other components inside other container or inside other namespaces.

***So, let us try to understand what are these cgroups***

The cgroups also, called as **control groups**.

The reason why we call them as control groups is namespaces will provide the isolation of the resources, but Someone must control the uses of the resources by a container.

So, with the help of this control groups, we can administrate and control the usage of the resources like CPU, memory, disk, network bandwidth.

That is where for the containers we can enforce resource restriction at runtime and try to avoid the scenarios like one container from monopolizing the system resources and ensuring fair allocation between multiple containers.

Since we can deploy multiple containers inside a same virtual machine or inside a same computing system, we need to make sure every container is getting its own fair allocation.

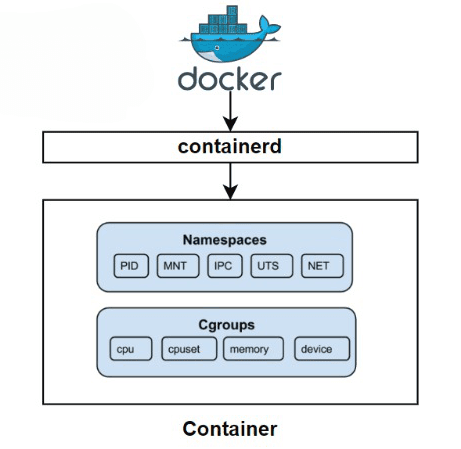
It should not be like one container is getting very less allocation, whereas other container is getting lot of memory allocation or CPU allocation.

That is not fair and it may result into issues dynamically at runtime.

That is why with these concepts of like namespaces and control groups, a separate isolated environment will be provided to the containers.

Here you may have a question like these are the Linux features.

***Blog from medium.com on Namespaces***



***INTRODUCTION***

In the realm of containerization, Docker’s use of namespaces is a powerful and foundational concept that underlies the platform’s ability to isolate and share resources efficiently.

To truly grasp the significance of Docker’s namespaces, let us embark on a journey comparing it to the general process of sharing resources among traditional processes.

***Understanding General Process Sharing Resources***

In a general computing environment, processes share the same set of resources, creating potential conflicts and challenges in resource management.

Imagine multiple processes running on a host system without any isolation. These processes may inadvertently interfere with each other, leading to issues like resource contention, security vulnerabilities, and difficulties in managing dependencies.

In this scenario, processes share a common namespace, leading to a lack of isolation and independence. Resource conflicts, such as two processes trying to access the same memory address or conflicting file access, can result in unstable and unpredictable behaviour.

***LINUX NAMESPACES***

**Namespaces** are one of a feature in the Linux Kernel and a fundamental aspect of containers on Linux.

On the other hand, namespaces provide a layer of isolation.

**Namespaces** are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.

This is a feature of Linux that allows us to create something like a virtual machine or a tool like docker, quite like the function of virtual machine tools.

Solomons Hyke, the founder of Docker, utilizes these powerful features of linux kernel and created the revolutionary tool that brings the new layer of agility and isolation in tech world.

***DOCKER***

**Docker uses namespaces to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container.**

This means that each container has its own isolated namespace for processes, network, mount points, and more.

Let us explore the key namespaces in Docker and how they enhance resource isolation:

1. PID Namespace: Process isolation (PID: Process ID)

General Process Sharing: All processes on a system share the same PID namespace.

Docker’s Approach: Each container has its own PID namespace, allowing processes within the container to have their unique set of process IDs.

2. Net Namespace: Managing network interfaces (NET: Networking)

General Process Sharing: Processes often share the same network namespace, leading to potential conflicts.

Docker’s Approach: Each container gets its isolated network namespace, enabling independent network configurations and avoiding interference between containers

3. IPC Namespace: Managing access to IPC resources (IPC: InterProcess Communication)

General Process Sharing: Processes share the same IPC namespace, leading to shared resources and potential conflicts.

Docker’s Approach: Containers have separate IPC namespaces, providing isolation for inter-process communication.

4. MNT Namespace: Managing filesystem mount points (MNT: Mount)

General Process Sharing: Processes typically share the same mount namespace, making file system management challenging.

Docker’s Approach: Containers have their own mount namespace, ensuring that file systems are isolated and can be managed independently.

5. UTS Namespace: Different host and domain names (UTS: Unix Timesharing System)

General Process Sharing: Processes usually share the same UTS namespace, making hostname conflicts possible.

Docker’s Approach: Containers have their own UTS namespace, allowing each container to have a unique hostname.

6. USER Namespace:

General Process Sharing: Processes can see the userIDs and groupIDs of other processes in same namespaces.

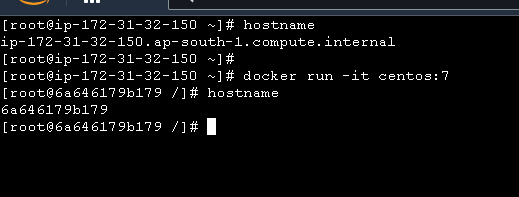
Docker’s Approach: Isolate the user IDs and group IDs of processes in a container. This means that processes in one container cannot see the user IDs and group IDs of processes in another container.

**By leveraging namespaces, Docker achieves enhanced resource isolation, making containers lightweight, portable, and secure.**

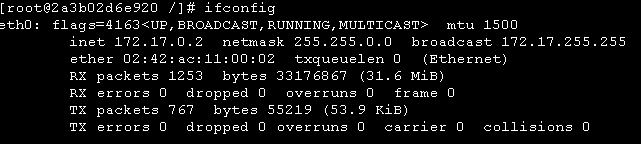
***Practical For Better Understanding***

The containers are considered as the separate OS because it has its own Namespaces and resources. Although it shares the resources from base OS with the help of c-groups but having isolated and own namespaces makes it an OS.

The docker run command initializes the new process and create the separate namespace for that process so it can use the resources in isolation.

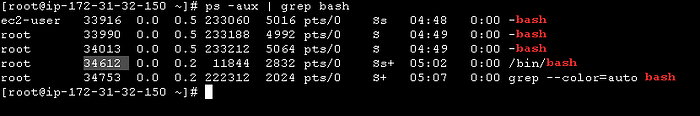


As we can see that the hostname inside the container is different from that of base hostname because of the UTS namespace. Different host and domain names (UTS: Unix Timesharing System)



The IP of the container is different from that of host IP because of net namespace.

As soon as we start the container it initializes the process of whatever we have mention in CMD of the image. Managing network interfaces (NET: Networking)

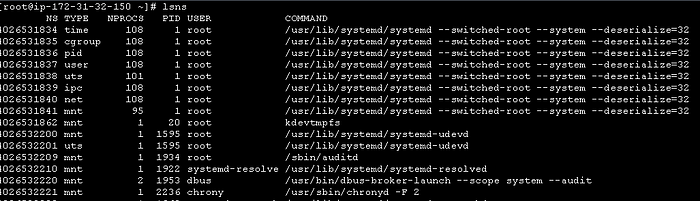


We can see that 34612 pid were in actual the pid of the container. As soon as we kill that process we can see our container automatically removed.



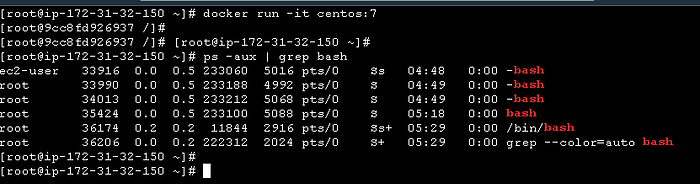
There is a command in the Linux where you can see all the namespaces associated with the particular process.

lsns



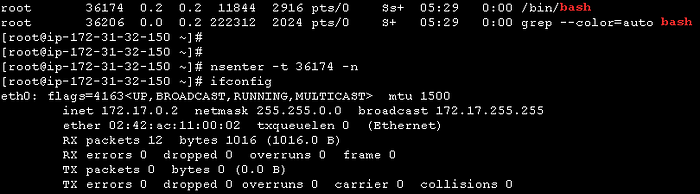
These all are the namespaces that is been associated with these processes.

On running docker run command on the terminal, we can get the new bash which can be proved by seeing in pids that a new bash process has been initiated.



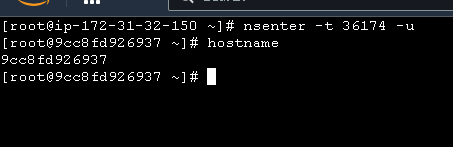
We can now give separate namespaces to this process of ID 36174 using the command nsenter.

Now without using any of the docker command we can go inside the process using namespaces command as we know that this process already has separate namespaces.



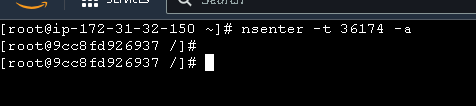
Now we can see that we get the new IP of the container without using any docker command. nsenter -t <target pid> <option for target ns>, this command takes you inside the process and inside the namespace of that process.

This is why we get the new IP of the container even without using any docker command. That means docker behind the scene using this command for getting the IP.



Similarly, we get the different hostname for this container without going inside the container.

Now with the command nsenter -t <target ip> -a , we can get access to all the namespace of that process. It is same to the command docker attach <name>



Hence it has been proved.

***Cgroups***

We could have created a process separate from the other process with Linux namespaces. But if we create multiple namespaces, then how can we limit the resources of each namespace so that it does not take up the resources of another namespace?

Cgroups (abbreviated from **control groups**) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

How the containerization concept is going to work inside my Windows operating system

If these are the Linux features, then how I'm going to containerize my applications from my local system, like inside my Mac OS or inside my windows operating system.

Even it is also, possible your production server can be a Windows server.

So, in such scenarios you may have a question like these are the Linux concepts.

Then how the Docker containerization or containerization concept is going to work inside my Windows operating system or Mac operating system?

So, let us try to understand the same.

Whenever you try to install and Docker on a Linux operating system, it is going to be pretty simple.

You will receive the entire Docker engine inside your Linux operating system.

However, if you try to install the same docker inside a mac or Windows operating system, you will get two separate components.

One is Docker client, which is installed on your own operating system like Mac or Window Os.

And behind the scenes there will be also, a separate component installed which is a lightweight linux based virtual machine and the Docker server component is installed inside this Linux virtual machine.

This way, regardless of what is the host operating system, Docker is going to work in any kind of operating systems.

As a developer or as an end user, you will get very similar look and feel when you try to use a docker on a Linux machine or in any other operating systems.

I can alSo, confirm you the same when we install the docker inside our local system.

So, when you install the docker inside your local system, just run the command Docker version and you will get an output like you can see on the right-hand side.

So, here you can see there are two details are printed.

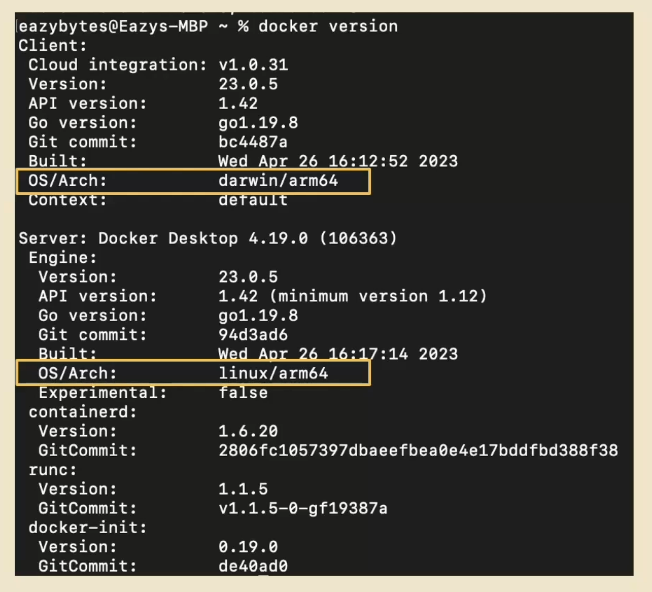
The very first one is what is the client details, which is Docker CLI for DockerCLI or Docker client.

You can see the OS darwin/arm64, which is inside my Mac operating system.

Inside Windows you may see a different output, but coming to the server, the OS is Linux because behind

the scenes, like I said, a lightweight Linux virtual machine will be installed.

That's why you are able to see clear difference between Docker client and Docker server inside the same machine.



Introduction to Docker components and its architecture

Let me try to explain you what are the important components available inside the Docker and what is the internal architecture of Docker?

Like I said before, whenever we try to install Docker inside any system, we get Docker client and

Docker server.

The very first important component that we have is **Docker client**.

So, using this Docker client only, we need to give instructions to the Docker server on how to containerize our applications.

So, inside this Docker client there are two different components that we can use to issue the commands to the Docker server.

The most used component is Docker CLI.

Using this command line interface, we can issue commands to the Docker server directly from your terminal or from your command line.

Apart from CLI, you can also issue commands to the Docker server using APIs using this Docker remote API, we can issue commands like how to run a Docker container from a Docker image.

So, all such commands we can give with this remote API approach as well.

But inside this course we are going to use the Docker CLI approach because that is the most used approach.

Apart from this Docker client, the next important component that we have is **Docker server.**

In other words, we can also call this Docker server as Docker host.

So, when I say Docker server, do not think like this is going to be installed in some other remote machine.

So, both this Docker client and Docker server or Docker hosts, both are going to be installed inside your system.

So, inside this Docker server there will be a **Docker daemon process** that is going to run continuously, which is going to accept the commands from the client, the client like CLI, we can issue commands to this server.

Like please generate a Docker image for my application.

So for my application, these are the dependencies.

This is the Java version that I am using.

These are the configuration that I need.

So, once we provide all those details, our instructions to the Docker server, it is going to convert

your spring boot application or Maven application or your microservice into a packaged software which commonly called as **Docker image**.

So, image is a representation of your packaged application.

So, it has all the required dependencies and configurations packaged into a Docker image.

Very similarly, we can also give commands to generate a container from a Docker image.

Like you can see here, there will be images and very similarly we also have containers, so we cannot

generate containers without a Docker image.

First, we need a Docker image.

When we give an instruction to the Docker server to generate a container from a Docker image, it is

going to leverage the Docker image and it is going to create a running instance of your Docker image,

which is your web application or your microservice in a running state.

So once the container is in running state, you can obviously try to access all your REST APIs or microservice business logic through the endpoint URL by using the correct port number.

Correct API path.

I hope this is clear.

So, inside your Docker server only all your images and containers are going to be stored.

Now this setup is going to work perfectly if you are working in your own machine.

Once we generate a Docker image for our microservice, once we test that with the help of Docker container and once we feel that everything is working, we need to push the Docker images to a repository from where we can continue to the steps like deployment and running the containers inside the dev environment, environment, or production environment.

Just like how we store our Java code inside the GitHub repo.

Very similarly, we need to store these Docker images available inside our local system into a remote

repository.

That is where we have the third component available.

This third component we call it as **Docker Registry**, Docker Company itself they provide a Docker registry with the name Docker Hub.

Inside this Docker hub, you can store all your Docker images and you can make them available for public use, or you can also protect them so that only authenticated users are applications will try to download your Docker images. Just like how Docker providing Docker hub.

Very similarly, there are private registries provided by famous companies like GitHub, AWS, GCP,

Azure, so most of the cloud providers and most of the version repository products like GitHub, so

they have their own private registry where you can push the Docker images from your local system to this remote repository.

For example, if you are using largely inside your organization, then it makes sense to push

your Docker images into your private registry provided by the AWS from this private registry.

All your deployment of the microservices will happen.

So, these are the three important components available inside the Docker architecture.

Now let me give a simple flow that will happen whenever I try to issue an instruction to the Docker

server.

So, in the very first step, I am going to give the instructions to the Docker server with the help of

my Docker client.

So, the instruction may be like to run a container from a Docker image.

So as soon as I give this instruction, my Docker server will validate if the Docker image is available

inside my local system.

If it is not there, it is going to fetch the required Docker image from the remote repository like

Docker Hub.

Once the Docker image is pulled into my local system using the same Docker image, a container will

be created by the Docker server.

So once the container is available, it means my application is ready and I can try to use the application.

These days, many of the products, they are providing everything as part of a Docker image.

For example, if you want to use a MySQL inside your local system, the traditional or older approach

is, you will go to the MySQL website, you will download the MySQL installation you will install inside

your system.

So that is very cumbersome process and a lengthy process.

Instead, with the help of Docker, you can pull the image of MySQL.

Once the image of MySQL is pulled from the Docker hub, you can try to run a container from the image. That means you are going to have MySQL server running inside your system.

Introduction to three approaches for Docker Image generation

To generate a Docker image from a web application or a spring boot application, there are

three mostly commonly used approaches inside the industry.

Inside this section, I am going to discuss all these three common approaches, and by the end of this

section we are going to choose one of this approach and we can continue with that option for the remaining

course.

Let me try to give a quick introduction about these three approaches.

The very first approach is by following the **Docker file**.

This is the most basic and the most traditional approach to generate a Docker image.

Inside this approach, we need to write a set of instructions inside a Docker file and based upon these

instructions, my Docker server is going to generate a Docker image.

So, this is the most basic approach, and this approach involves some learning curve of the Docker syntax and the Docker best practices.

The second approach that we have is **Buildpacks**. With the help of Buildpacks,

we do not have to write any Docker file and we do not have to provide any instructions manually to the Docker server to generate a Docker image.

Instead, with the help of a single Maven command, we can generate a Docker image and behind the scenes this maven is going to use the concept of Buildpacks.

So Buildpacks is a project initiated and developed by Heroku and Pivotal based upon the best practices that they have learned over the years.

So, this Buildpack simplifies the containerization of our web applications.

How it is going to simplify is we do not have to write any low-level Docker file like in the first approach and using this approach only we are going to generate a Docker image of Loans Microservice.

Now the last approach that we have is Google Jib. Google Jib is a product developed by the Google

itself.

Later, they open source this Java tool using this Java tool, and with the Maven plugin command,

we can generate a Docker image of any Java application very easily without writing any low level Dockerfile like in the first approach.

So, all these approaches, they have their advantages and disadvantages.

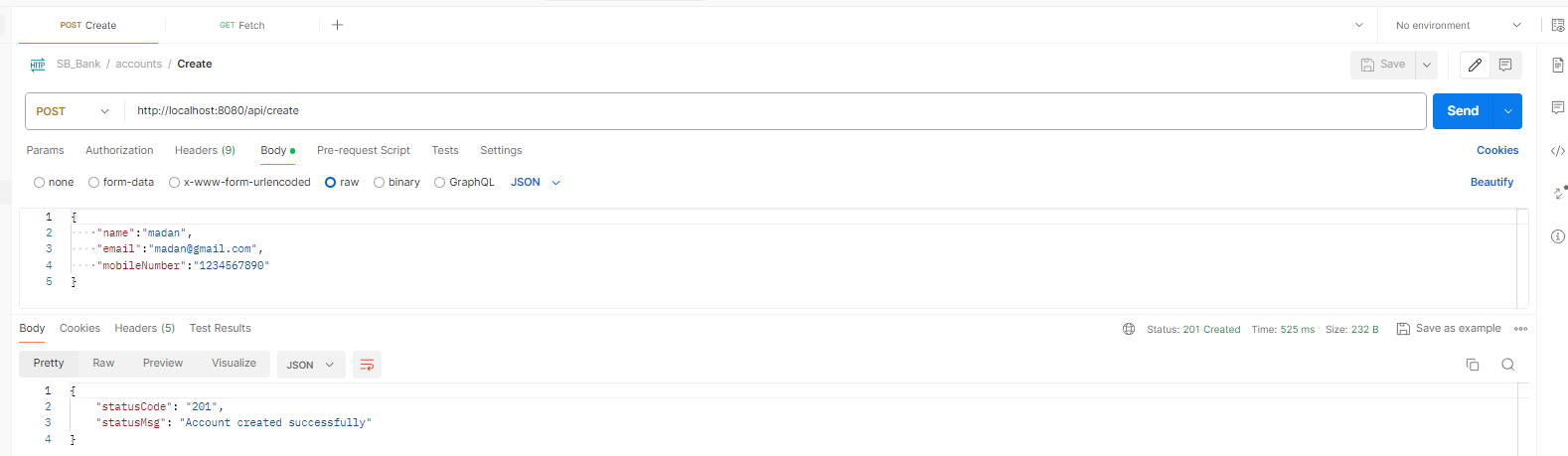
I am going to discuss in detail while we are discussing each of these approaches.

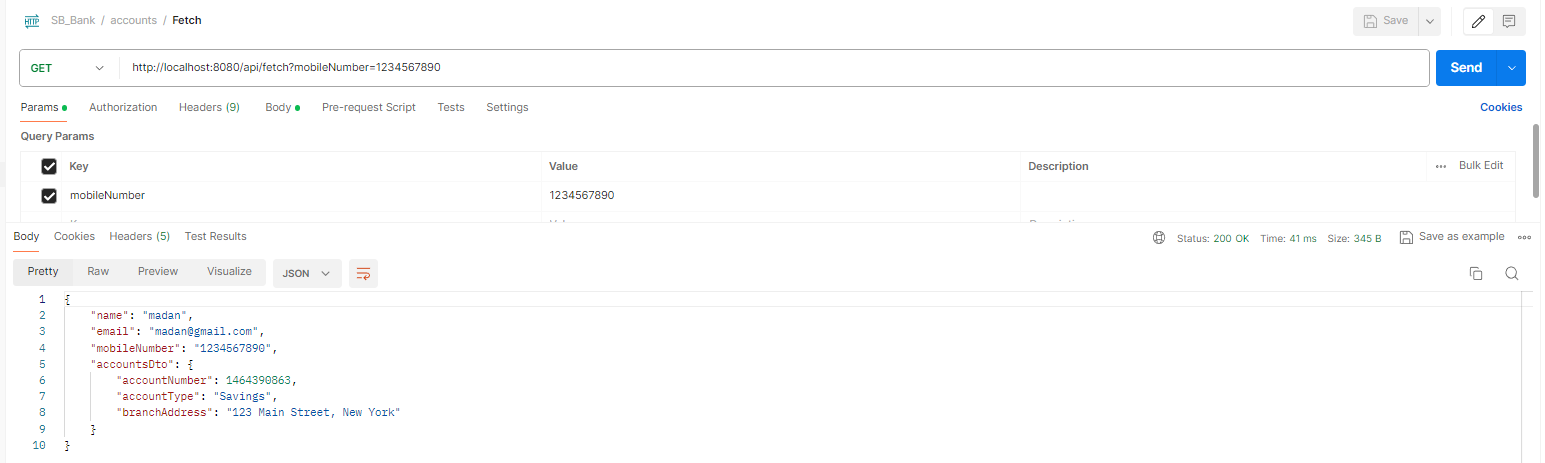
So, using this Google jib approach only, we are going to generate a Docker image of Cards Microservice.

Generate Docker Image of Accounts microservice with Dockerfile – Part 1

PS D:\Experiments\Microservices\sb-bank-application\accounts> mvn clean package

PS D:\Experiments\Microservices\sb-bank-application\accounts> mvn spring-boot:run





PS D:\Experiments\Microservices\sb-bank-application\accounts> java -jar .\target\accounts-0.0.1-SNAPSHOT.jar

Generate Docker Image of Accounts microservice with Dockerfile – Part 2

Docker Service should be running and installed   
  
Create Dockerfile  
D:\Experiments\Microservices\sb-bank-application\accounts\Dockerfile

FROM openjdk:17-jdk-slim

COPY target/accounts-0.0.1-SNAPSHOT.jar /accounts-0.0.1-SNAPSHOT.jar

ENTRYPOINT [ "java", "-jar", "accounts-0.0.1-SNAPSHOT.jar" ]

Generate Docker Image of Accounts microservice with Dockerfile – Part 3

PS D:\Experiments\Microservices\sb-bank-application\accounts> docker build . -t nileshzarkar/account:s4

[+] Building 8.3s (8/8) FINISHED docker:default

=> [internal] load build definition from Dockerfile 0.1s

=> => transferring dockerfile: 199B 0.0s

=> [internal] load .dockerignore 0.2s

=> => transferring context: 2B 0.0s

=> [internal] load metadata for docker.io/library/openjdk:17-jdk-slim 2.7s

=> [auth] library/openjdk:pull token for registry-1.docker.io 0.0s

=> [internal] load build context 4.5s

=> => transferring context: 48.41MB 4.4s

=> CACHED [1/2] FROM docker.io/library/openjdk:17-jdk-slim@sha256:aaa3b3cb27e3e520b8f116863d0580c438ed55ecfa0bc126b41f68c3f62f9774 0.0s

=> [2/2] COPY target/accounts-0.0.1-SNAPSHOT.jar /accounts-0.0.1-SNAPSHOT.jar 0.3s

=> exporting to image 0.4s

=> => exporting layers 0.4s

=> => writing image sha256:5d89ee36d87558cd5041f7be276fd6548beb1a753ba78cb3c44ad27a0279924d 0.0s

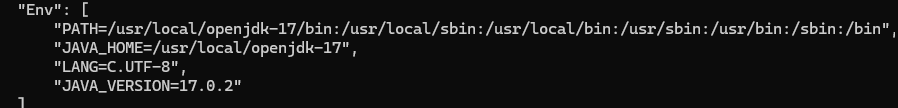
=> => naming to docker.io/nileshzarkar/account:s4 0.0s

PS D:\Experiments\Microservices\sb-bank-application\accounts> docker images

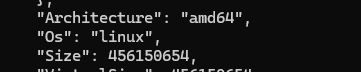
REPOSITORY TAG IMAGE ID CREATED SIZE

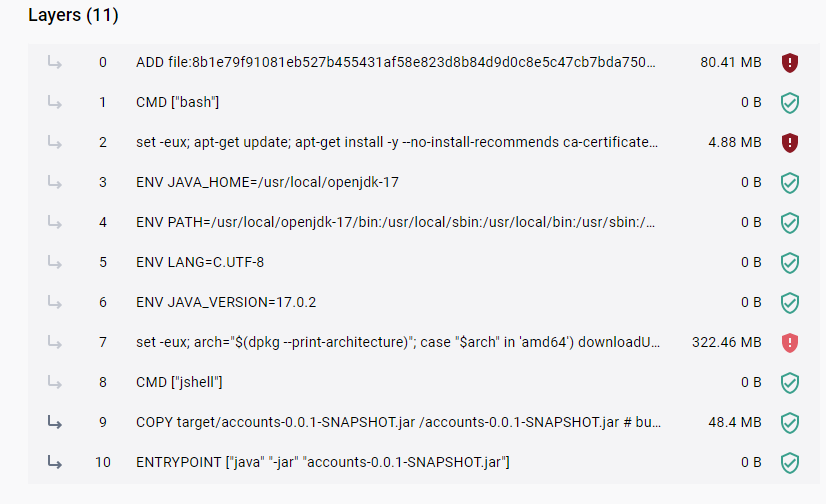
nileshzarkar/account s4 5d89ee36d875 About a minute ago 456MB

PS D:\Experiments\Microservices\sb-bank-application\accounts> docker inspect 5d8

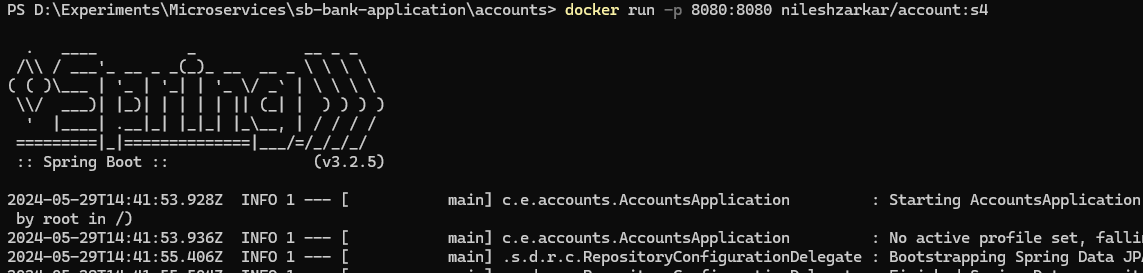


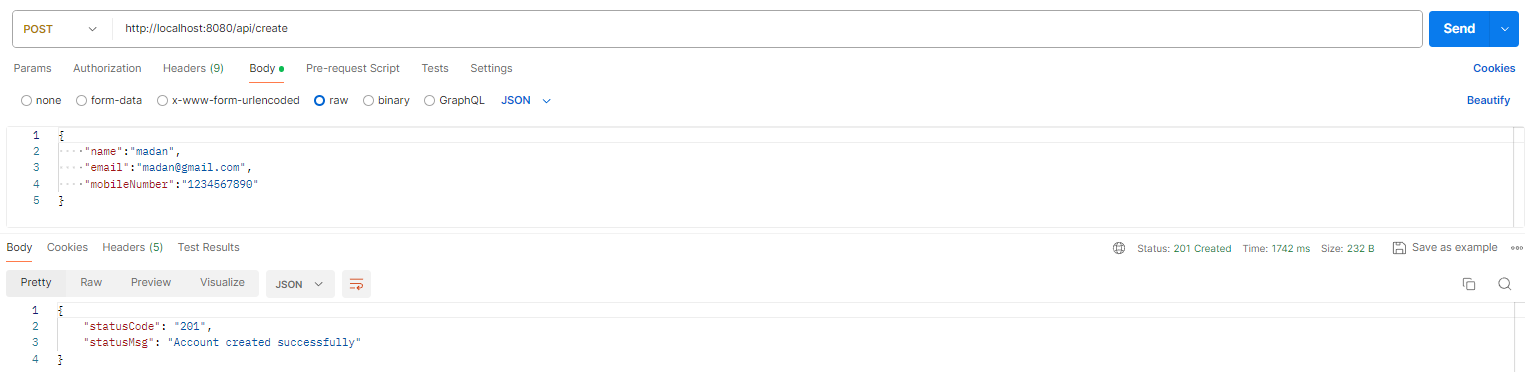


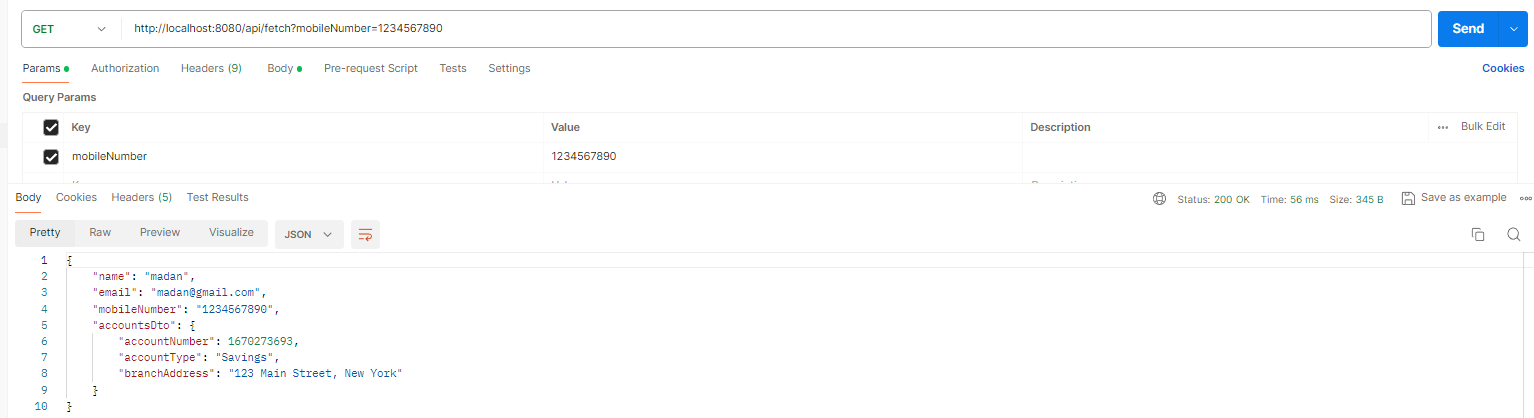


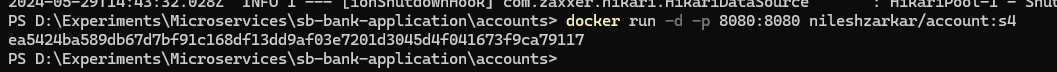


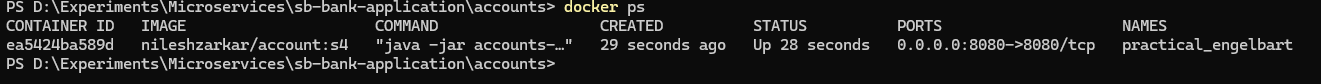
Running accounts microservice as a Docker container  
  

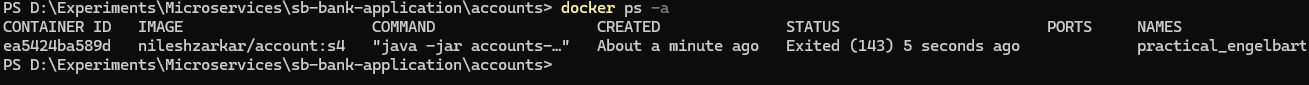





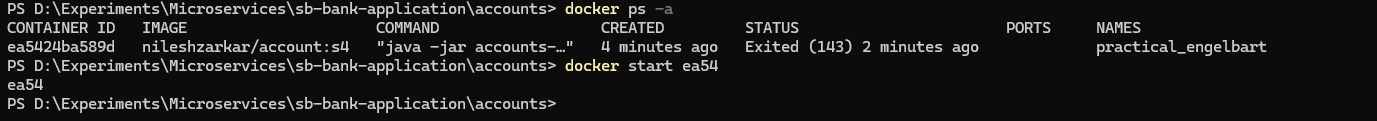






Stop the container from Docker Desktop  
  
  
  
This will start a fresh new instance of the docker container.

Start the stopped docker container



You can create as many numbers of containers as far as the mapped port is different  


**How port mapping works inside Docker**We can also call the port mapping as port forwarding and port publishing.

So, what is this concept is by default, the containers are connected to an isolated network within the

Docker host.   
To access a container from your local network or from your local system, you need to configure this port mapping explicitly.

For instance, where my accounts microservice is running inside a Docker network

as a Docker container at the port 8080.

If I want to expose that to the outside world with the Port 8081, I need to use this -p 8081:8080.

Always remember the very first port number indicates the value where you want to expose to the external local system or external network.

And the second value represents the container port where your container started.

First, as soon as the container started, it is going to start inside a Docker network like you can

see the accounts microservice container started at a port 8080.

This I cannot access from my outside Docker network.

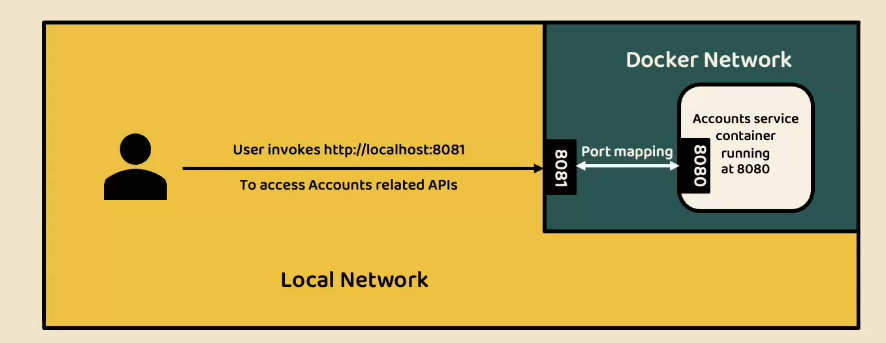
That is why we need to expose this Docker container to the outside world with the help of port mapping.

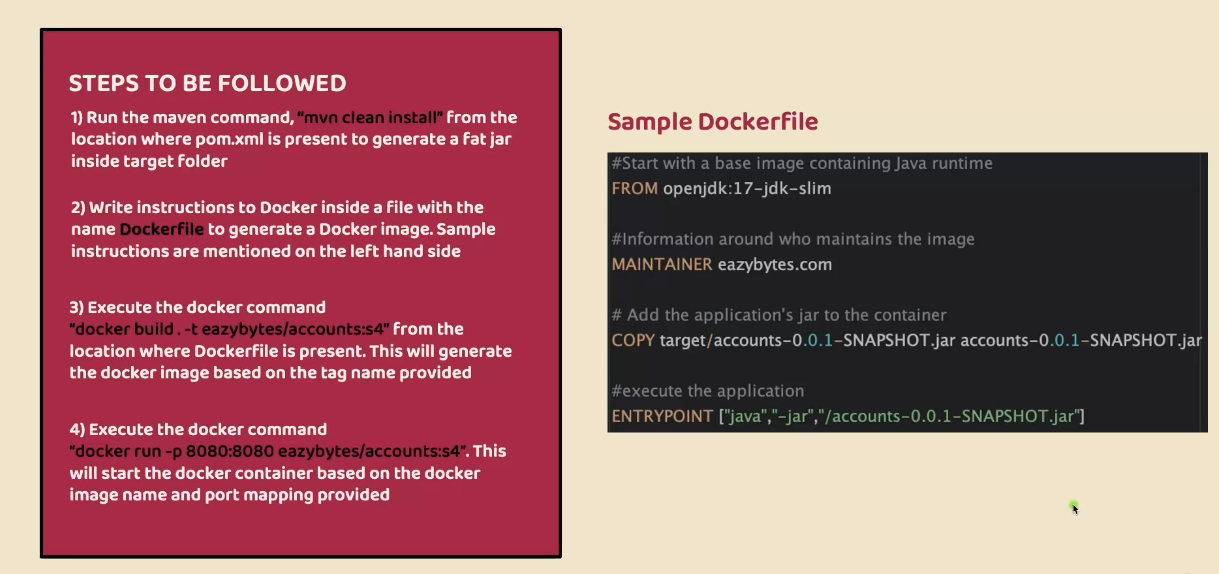
And with that I should be able to access my account microservice, which is deployed inside a Docker

network from my local network.

So, anyone who wants to communicate with my accounts, microservice container, they need to send a traffic request at the port 8081.

So, this is how port mapping or port forwarding or port publishing will work.

****

**Steps to follow for running a spring boot application as a container using Dockerfile  
**

Challenges with Dockerfile approach to generate a Docker image

Make sure there are no running containers inside my system.

Now coming to the disadvantages of Docker file,

the very first disadvantage is to write a Docker file, you need to be an expert of Docker concepts.

So right now, we have very basic Docker file where I have given very basic Docker instructions and I

told you like you need to use the so and so keyword; you need to mention so and so docker image, you need to give so and so copy commands or entry point commands.

But in real projects, if your application is large, this simple Docker file may not work for you.

So, to convert your microservices a Docker image, you need to learn a lot of Docker concepts and you need to learn a lot of instructions on how to provide them with the help of Docker file.

So, there is a lot of learning curve involved for that developer.

Why should developers should learn everything about Docker?

Because he is not a DevOps person, because he is not a platform team member.

So that is why this option is not good for developers because it involves a lot of learning curve.

And at the same time, apart from learning the concepts of Docker, you should also follow the best

practices when you try to generate a Docker image.

The best practices like your Docker image should be as small as possible.

You should try to use lot of caching compression while generating the docker image so that the process is going to be super quick and the Docker image is going to be super small, which will improve your overall microservices setup.

And apart from these standards, we should also not ignore any security related concept.

We should make sure our Docker image is perfectly secured and there are no security vulnerabilities

inside our Docker image.

So, there is a lot of standards that you need to follow.

And to implement all these standards and to follow all these best practices, you

need to put a lot of efforts, you need to learn Docker concept, you need to learn Docker best practices, post that only you can implement all of them inside a Docker file. And the same Docker file you need to maintain for all kind of microservices like accounts, loans, cards.

If you have 100 different microservices, you need to maintain 100 different Docker files for them.

And maintaining all these Docker files is another nightmare and there will be other challenges like

versioning.

So, there are many challenges that you are going to face with this approach.

That is why developers are always looking for an approach where a Docker image is going to be generated automatically without writing any low-level instructions inside a Docker file.

That is where we have solutions like **Build packs** and **Google Jib** came into picture.

Since developers are facing all these challenges, these two products are built over the years and the

same are being used extensively by the developers and the platforms teams these days.

So that is why let us try to learn these two different approaches.

One is Buildpacks and the other one is Google Jib.

Generate Docker image of Loans microservice with BuildPacks

<https://buildpacks.io/>

So, what is a **Buildpacks**.

Using Buildpacks, we can transform our application source code into a Docker image that can run on any cloud.

There is no need of writing low level instructions with 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.

So initially they started this concept of Buildpacks.

Later, both Pivotal and Heroku.

They work together and they build a cloud native build packs and with these cloud native build packs, we can generate Docker images very easily because behind the scenes build a pack is going to scan all your source code, it is going to scan your dependencies and accordingly it is going to generate a Docker image.

So, during the generation of this Docker image, this build packs is going to follow all the standards

of Docker in terms of security, in terms of compressing, in terms of caching.

Update the pom.xml for the docker image name

<build>

        <plugins>

            <plugin>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-maven-plugin</artifactId>

                <configuration>

                    <image>

                        <name>nileshzarkar/${project.artifactId}:s4</name>

                    </image>

                </configuration>

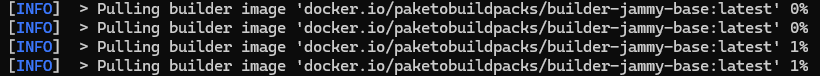
            </plugin>

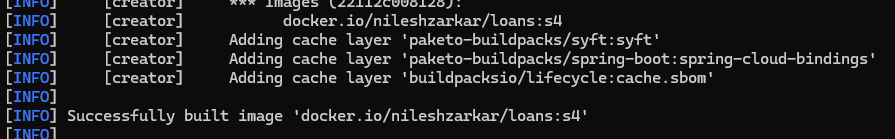
        </plugins>

</build>

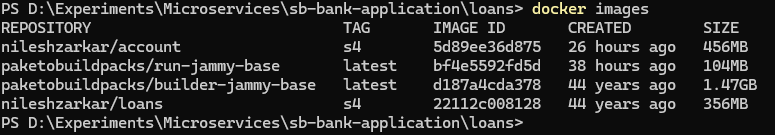
**Note: Docker service should be running behind the scene**

PS D:\Experiments\Microservices\sb-bank-application\loans> mvn spring-boot:build-image

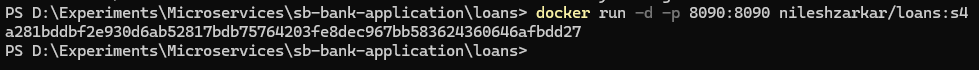


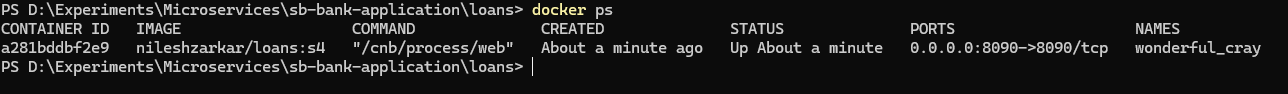


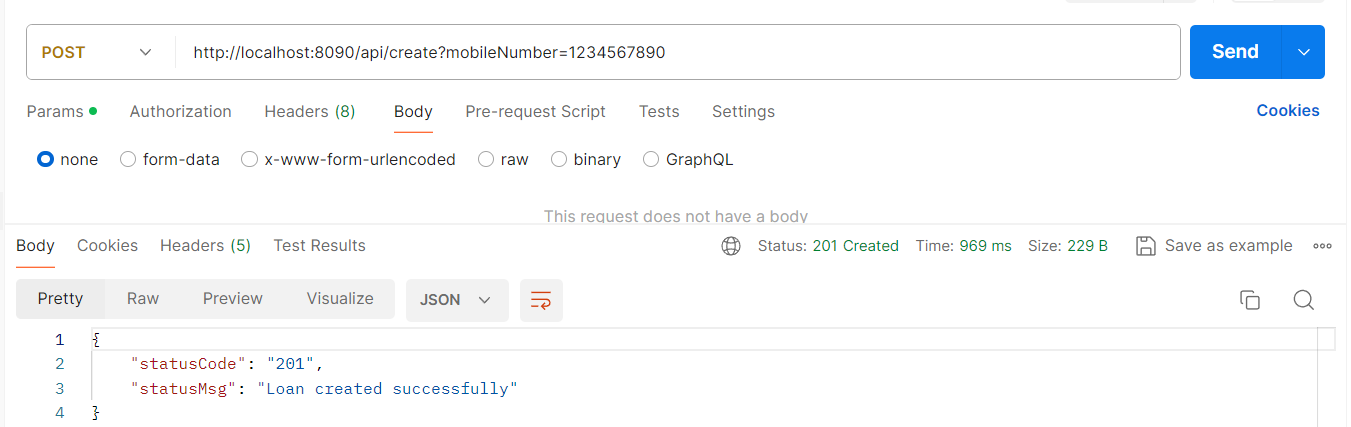
Check the images created by the build packs

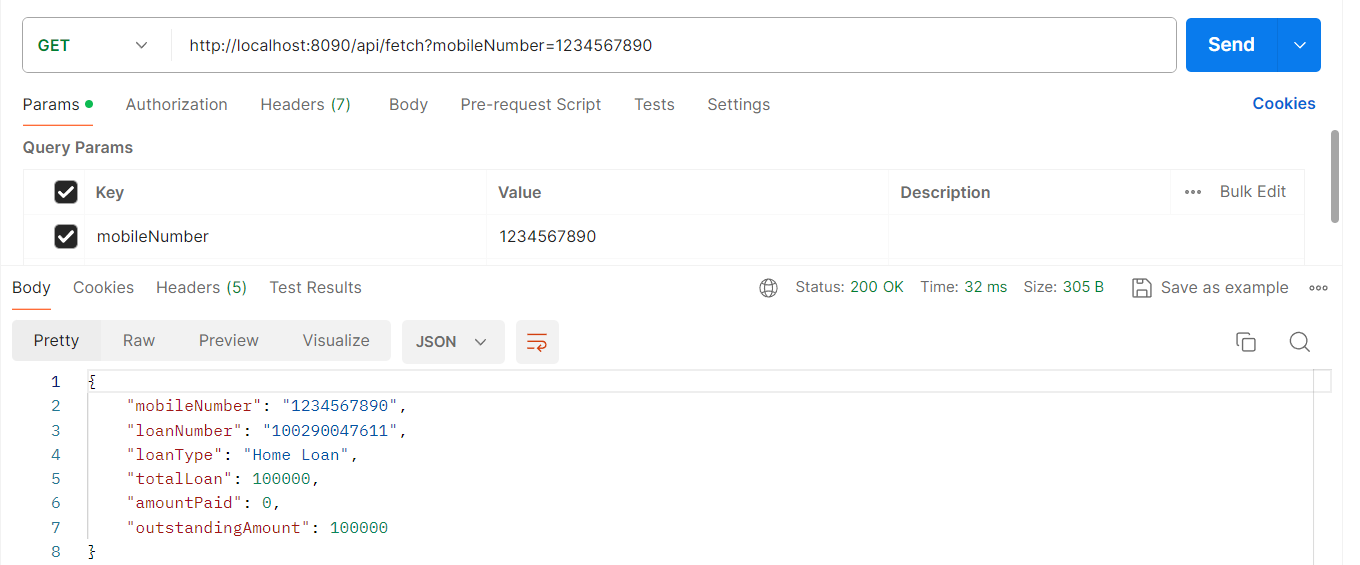


So, you see the image size of account using Dockerfile is **456 MB** and the image size of loans using buildpacks is **356 MB**. Since it followed best standards, cache multiple layers, compress multiple components inside docker image etc..









Steps to be followed:  
Step 1: We need to add the image name details inside your pom.xml like

<build>

        <plugins>

            <plugin>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-maven-plugin</artifactId>

                <configuration>

                    <image>

**<name>nileshzarkar/${project.artifactId}:s4</name>**

                    </image>

                </configuration>

            </plugin>

        </plugins>

    </build>

Note: Please make sure you also have the Spring boot Maven plugin configured inside your pom.xml.

Step 2: We need to run a mvn command which is

PS D:\Experiments\Microservices\sb-bank-application\loans> **mvn spring-boot:build-image**

So when you try to run this command behind the scenes, your spring boot Maven plugin is going to utilize Buildpacks to generate a Docker image without the need of Docker file.

Step 3: Once you generate a Docker image, you can run this Docker image as a Docker container by using the command which is

PS D:\Experiments\Microservices\sb-bank-application\loans>

**docker run -d -p 8090:8090 nileshzarkar/loans:s4**

Generate Docker image of Cards microservice with Google Jib

<https://github.com/GoogleContainerTools/jib>

<https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin>

<plugin>

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

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

<version>3.4.1</version>

<configuration>

<to>

<image>myimage</image>

</to>

</configuration>

</plugin>

Update pom.xml

….  
<groupId>com.eazybytes</groupId>

<artifactId>cards</artifactId>

<version>0.0.1-SNAPSHOT</version>

<**packaging>jar</packaging>**

….

Update pom.xml with the jib plugin

….

<build>

        <plugins>

            …

            <plugin>

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

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

                <version>3.4.1</version>

                <configuration>

                  <to>

                    <image>nileshzarkar/${project.artifactId}:s4</image>

                  </to>

                </configuration>

              </plugin>

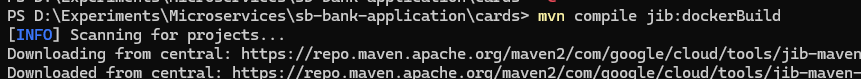
        </plugins>

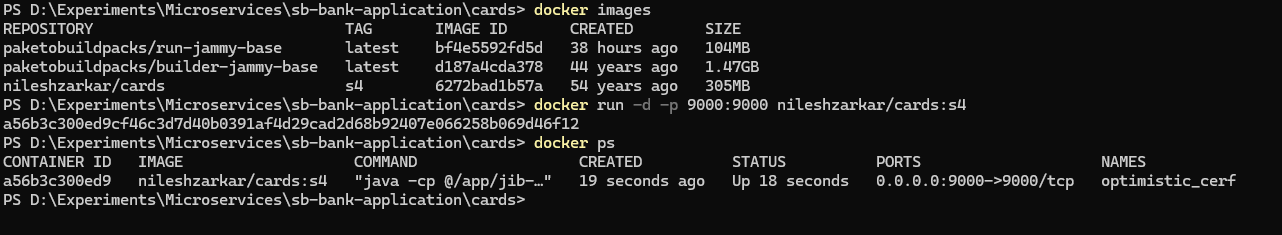
    </build>

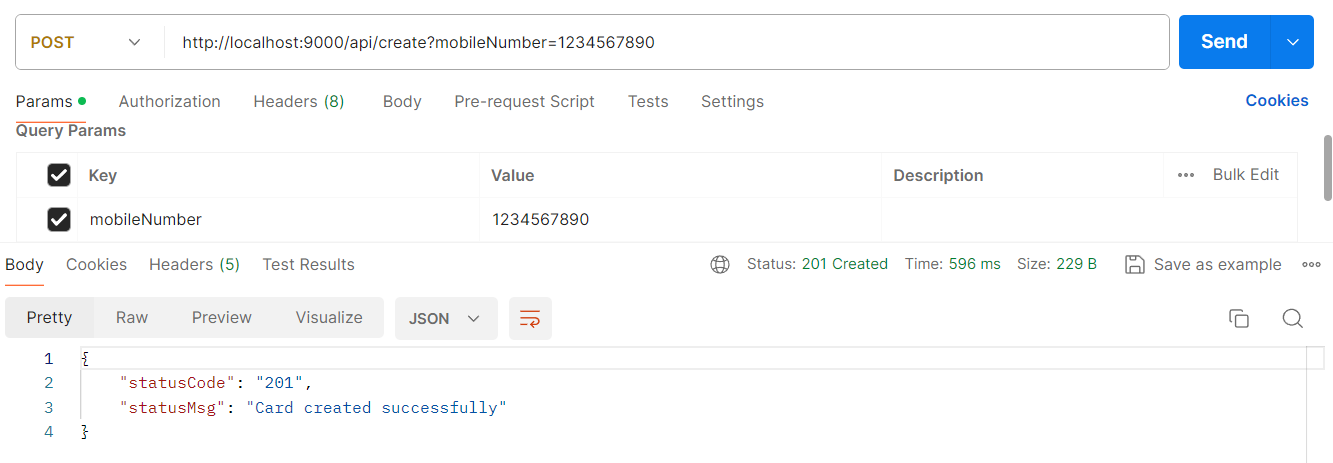
….

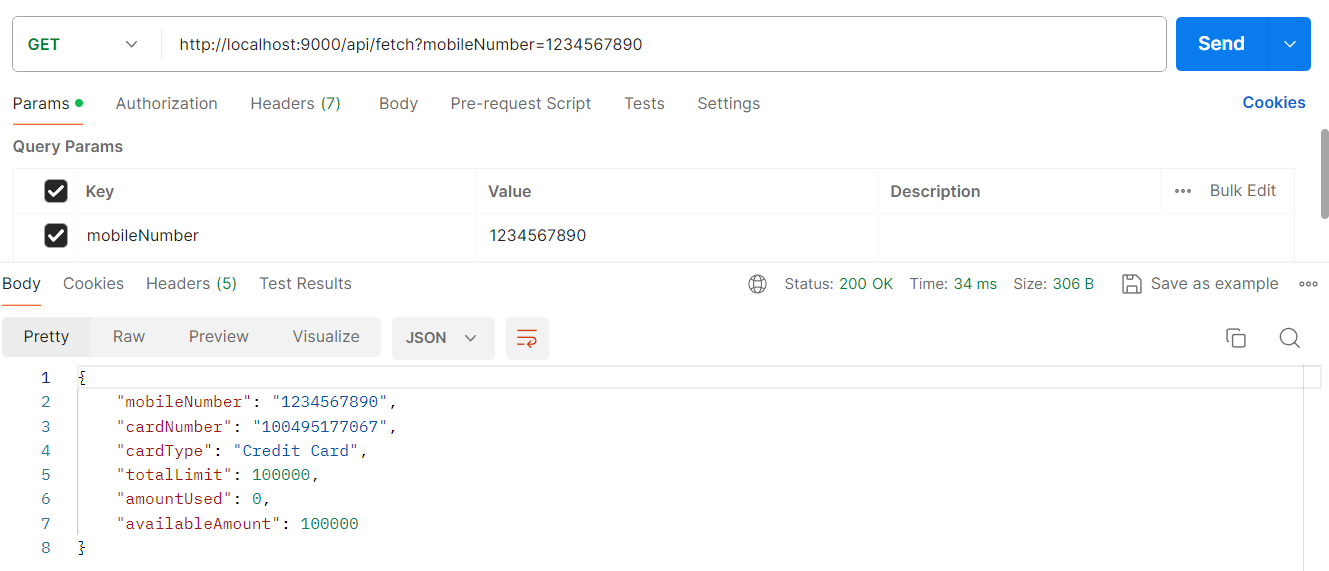
To create the image

mvn compile jib:dockerBuild









mvn compile jib:build

Subsequent images are pushed to the docker repository.

<configuration>

<to>

<image>docker.io/my-docker-id/my-app</image>

</to>

</configuration>

Similarly, you can configure for GCP, AWS, JFrog etc.

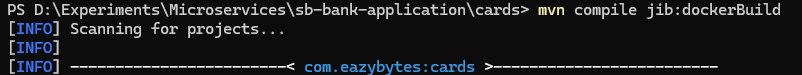
Pushing Docker images from your local to remote Docker hub repository  
  
Using Dockerfile

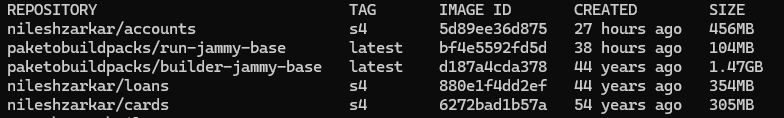


Using buildpacks



Using Google jib











Here we are pushing the images on windows machine where Docker Desktop is running, so it took the docker hub credentials form there.

Introduction to Docker Compose

As of now, you can see we have three different images like cards, loans, and accounts.

These are the Docker images of three microservices that we have built so far.

To run our microservices we need to convert these Docker images into containers with the help of docker run command.

So, if I want to start my three microservice, I need to issue docker run command three different times

along with the image name and the port mapping.

What if I want to start multiple instances of my microservices in such scenarios, giving this docker

run command manually for each microservice and for each instance of it is super, super cumbersome process and it is time consuming process.

If I have 100 microservice, I need to issue these Docker run command 100 different times, which is

not quite viable option for any human.

To overcome this challenge, we have another component inside Docker ecosystem with the name Docker Compose.

You can see Docker Compose is a tool for defining and running multiple container docker applications.

So, with the help of with Compose, we can use a Yaml file to configure all our application service.

Then with a single command we can create or start all our services.

So instead of running multiple Docker run commands, I can define all my Docker image details and how I want to start them inside a single Yaml file.

Once I define all these details inside a single Yaml file, I can issue a single command and with a

single command all my microservice containers are going to start and this docker compose.

We can make it work inside any environment like production, staging, development testing as well as

continuous integration workflows.

So that is why it is highly recommended to use Docker Compose.

If you have multiple microservices inside your system and these Docker compose will help you to start, stop and rebuild services.

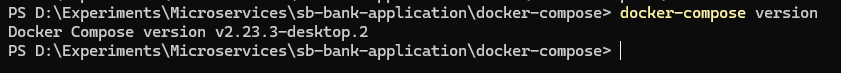
🡺View the status of running services.

🡺Stream the log output.

🡺Run a one-off command.

This way it has many other advantages.

So, whenever you install Docker desktop inside your system by default, Docker compose related components also will be installed.



Now to get started with the Docker compose.

Like I said, 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.

services:

  accounts:

    image: "nileshzarkar/accounts:s4"

    container\_name: accounts-ms

    ports:

      - "8080:8080"

    deploy:

      resources:

        limits:

          memory: 700m

    networks:

     - eazybank

  loans:

    image: "nileshzarkar/loans:s4"

    container\_name: loans-ms

    ports:

      - "8090:8090"

    deploy:

      resources:

        limits:

          memory: 700m

    networks:

     - eazybank

  cards:

    image: "nileshzarkar/cards:s4"

    container\_name: cards-ms

    ports:

      - "9000:9000"

    deploy:

      resources:

        limits:

          memory: 700m

    networks:

     - eazybank

networks:

  eazybank:

    driver: “bridge”

services – Define any number of services (accounts, loans, postgresql etc)

accounts – service name  
image – which image you want to use to create the container

container\_name – give your specific name else it will get random name and it will become difficult to identify the service

ports – used for defining multiple port mapping

deploy/resource/limits/memory – limit the maximum memory allocation to this specific container

When we start these services, they all are going to start in different isolated networks.

What if there is dependency between these microservices or these microservices need to talk to each other.

In default scenario it is not going to work since all will get created inside their own network.

To make sure the inter-communication is working we need to tag all these service inside the same network.   
networks – we can define any number of networks.  
To create the network with the name eazybank  
networks/eazybank/driver  
  
networks:

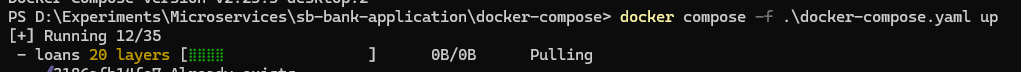
  eazybank:

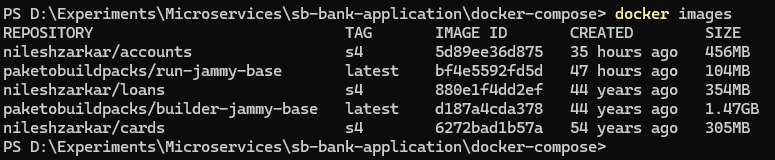
    driver: “bridge”

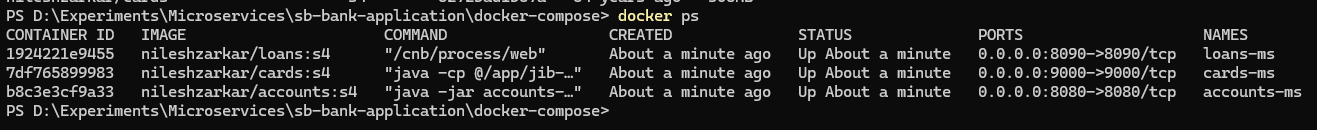
With this we are telling to our docker server, create a network with name eazybank with a driver **bridge**   
Since we are using same network in accounts, loans and cards microservice. The docker server will establish a bridge so that the microservices can communicate with each other.

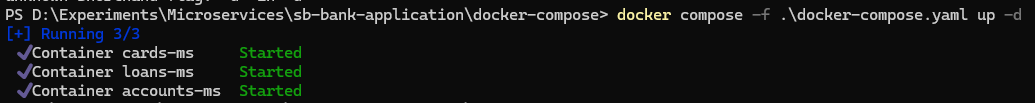
Running all microservice containers using Docker Compose command

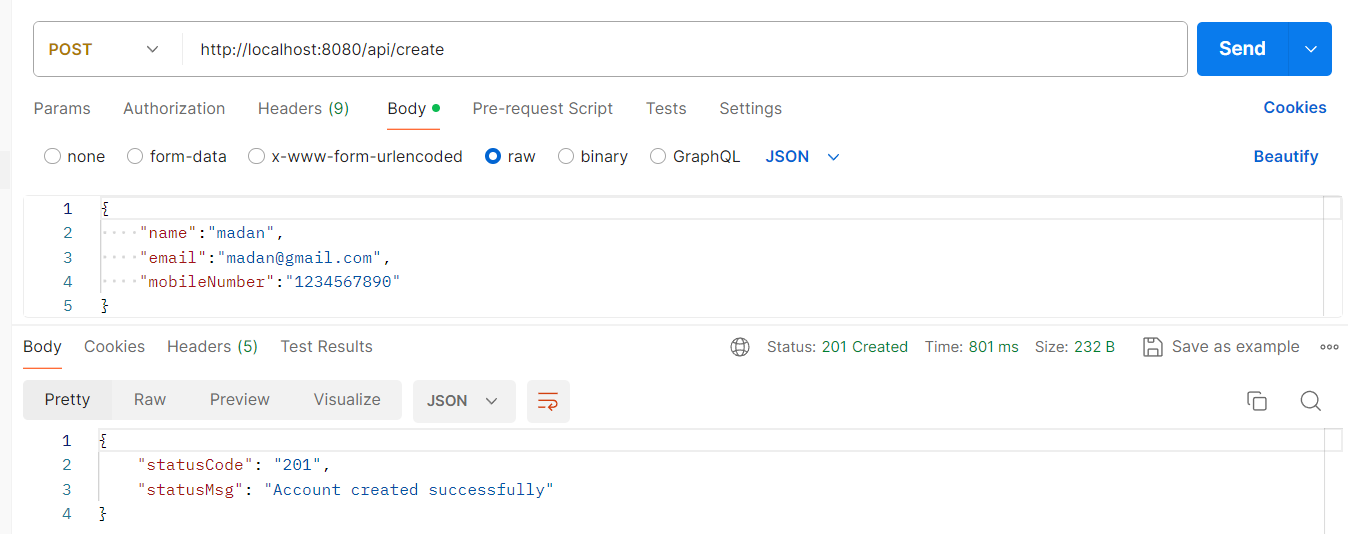
The docker compose file should be in a separate project or folder.

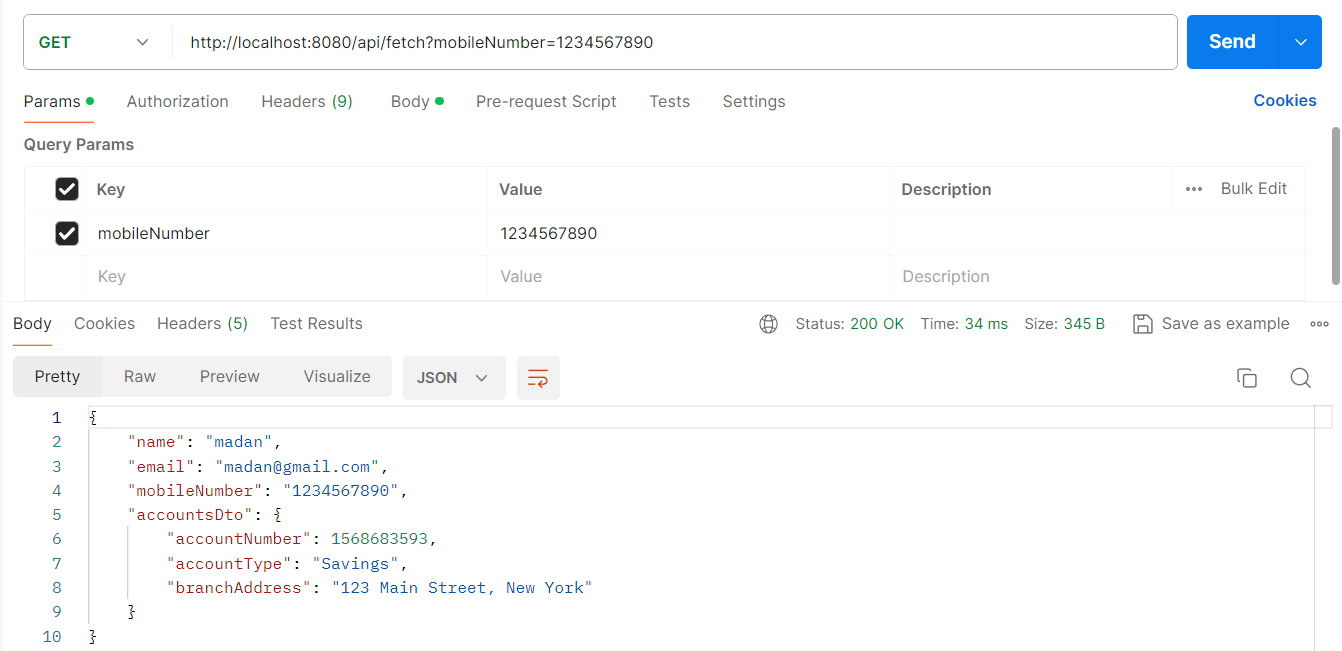


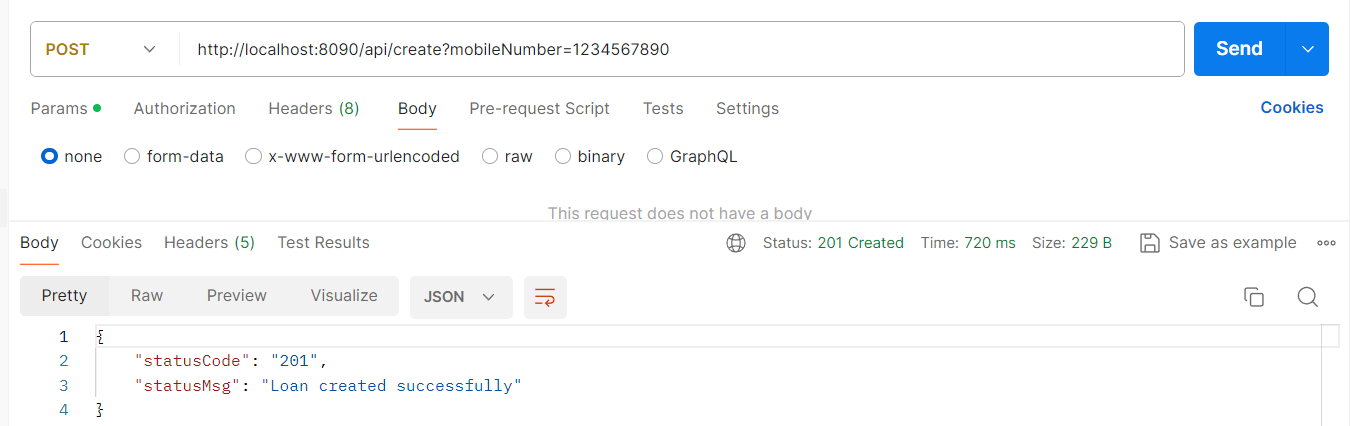


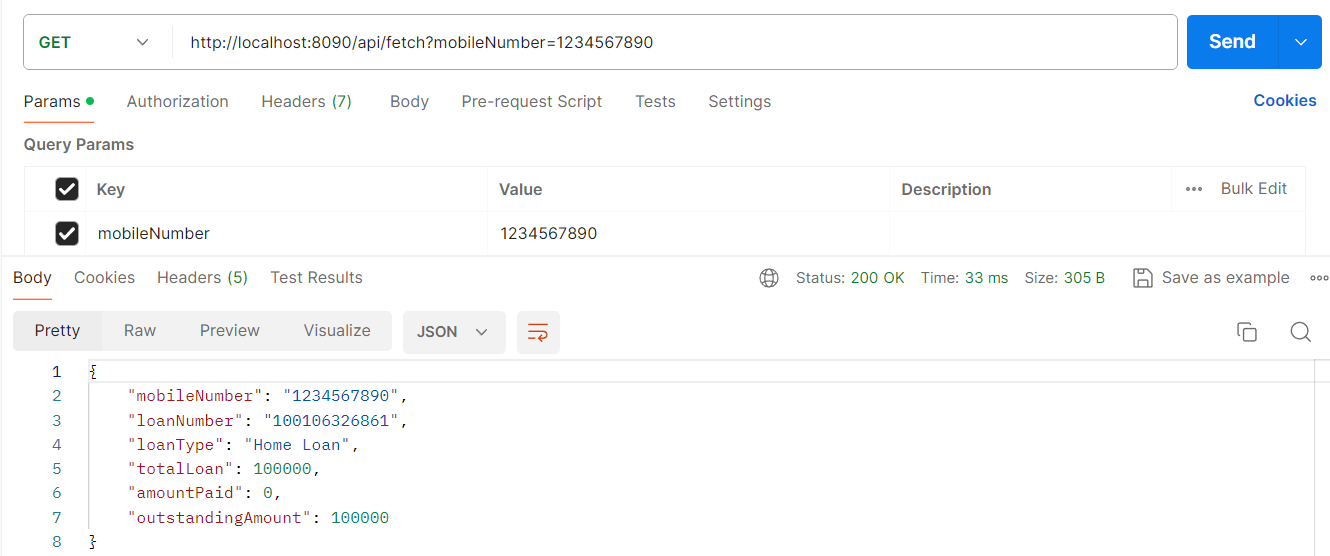


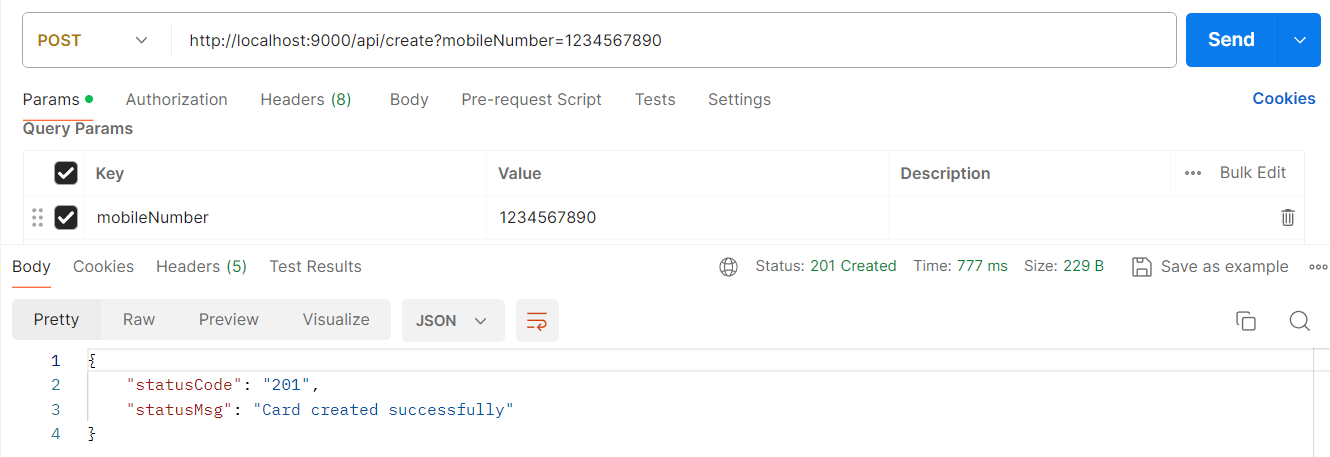
You can also start the containers in background   


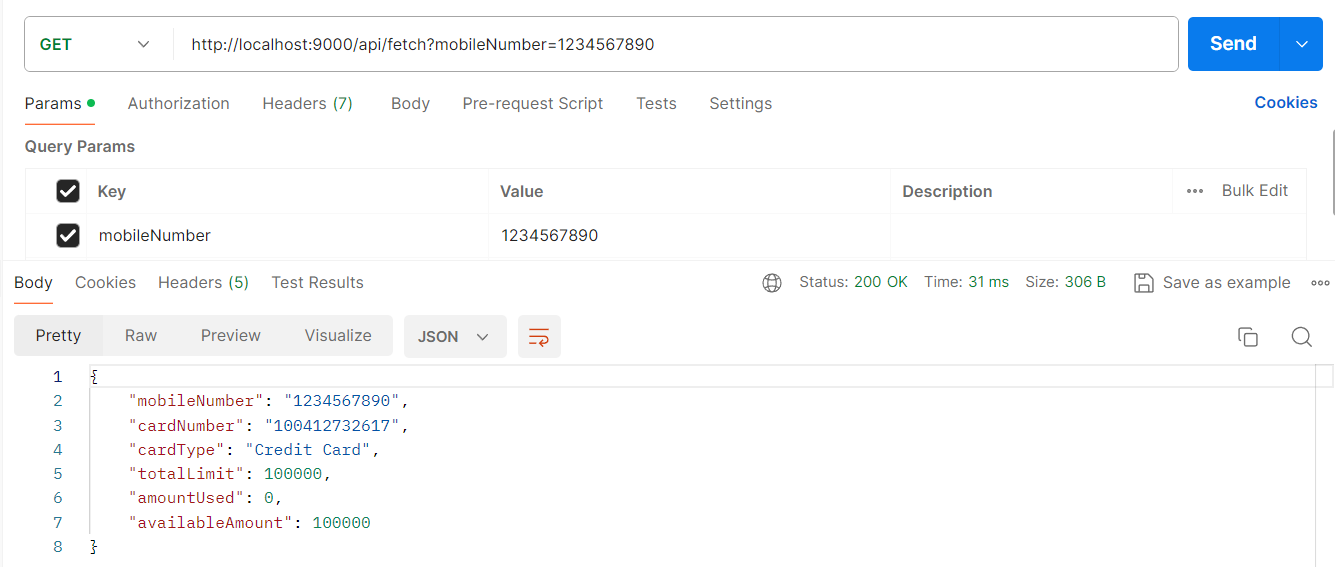




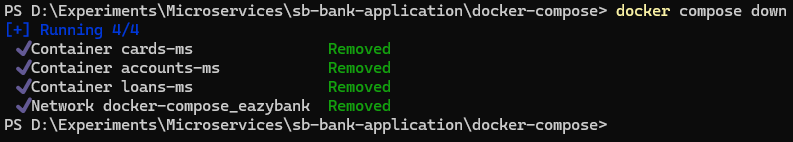




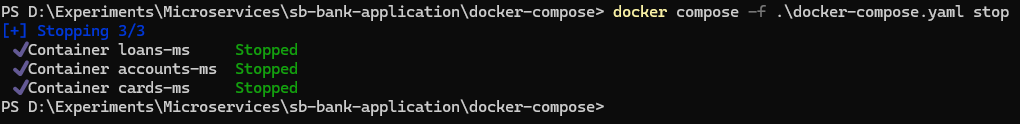




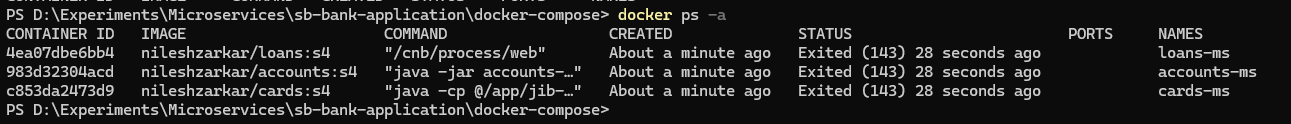
Demo of docker compose commands

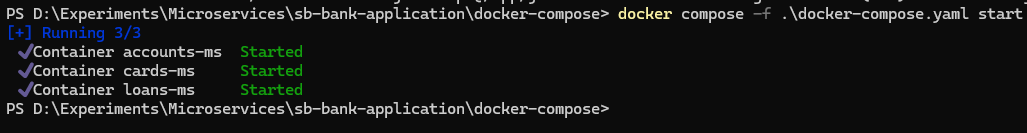


Stop and remove the containers



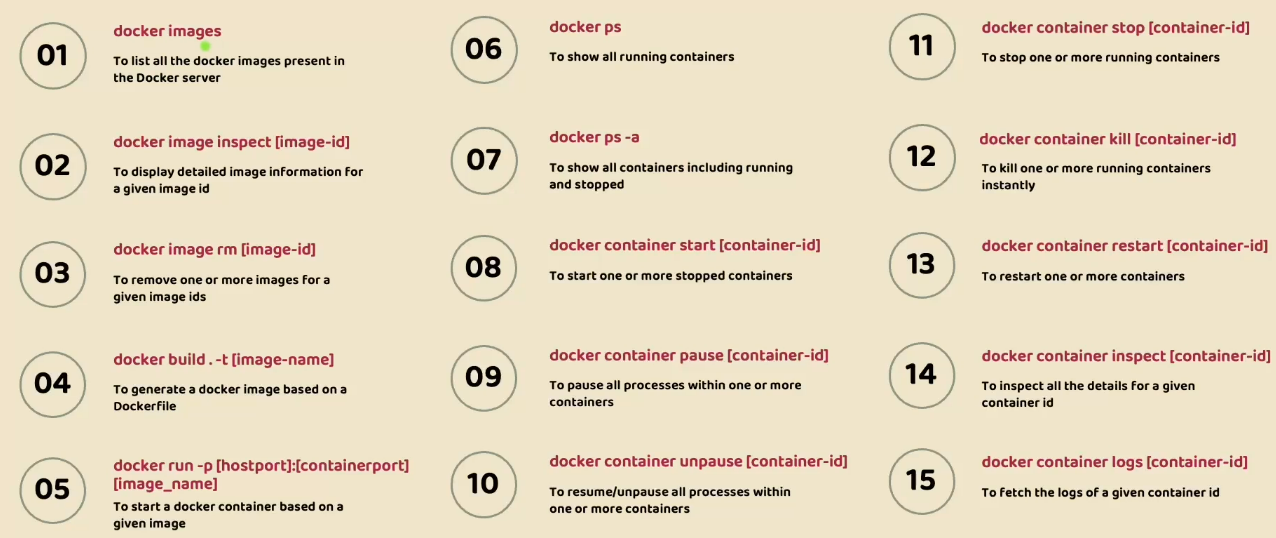
Just stop the container, does not remove it.





Start the stopped containers

Deep Dive on Docker commands



Logs Explorer Docker extension

