Docker and Challenges While Building and Deploying Microservices

# Challenges: Deploying, Scaling and Portability

## Portability

With possible tens of microservices that need to be moved from environment to environment, from dev to test to QA to Production Replica to Production, we need to make it less costly in terms of configurations, etc.

## Scalability

In monolithic application you have one code base and you can make replicas out of the main monolithic server and scale out the application.

In microservices though we need to scale up and down based on the traffic on each microservice with minimum cost and effort.

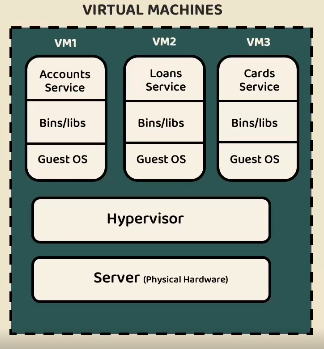
## The Solution:

In the monolithic apps we make a jar or war file out the project and deploy it on the server. In microservice ecosystem to solve the challenges above we **Containerize** our microservices.

* Containers are self-contained, isolated environment that carry all the necessary dependencies to run the application.
* By containerizing our projects, they become portable and we can run them on any cloud environments with a unified management, regardless of the technology and languages that are used inside these containers.
* **Docker** is an open-source platform that allows us to **package** and run our applications in a loosely isolated environment called a **container**.

# Containers vs Virtual Machines

* Before cloud computing you would have had to buy a hardware, install an OS, make the server public through an IP address and deploy your application on it.
* With cloud computing though you don’t have a physical data center. Instead, the cloud provider give you access to a virtual machine that runs on their hardware. They do this by using a hypervisor: a software that you can use to run multiple virtual machines on a single physical machine. The hypervisor allocates the underlying physical computing resources such as CPU and memory to individual virtual machines as required.

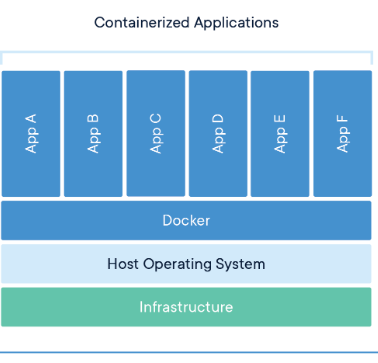


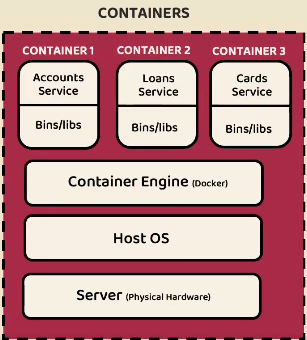
* **Without Containerization**, you would have to either buy one of these virtual machines for each of your microservices to serve their different requirements. And just imagine you had like 100 microservices.

Or you could run all of them inside one VM and let’s say you had to restart your VM, all the services will be down if you run them on a single VM. The thing about different dependencies of each microservice, I’m yet to know more about their requirements.

In terms of scalability, if all of a sudden, the traffic on one microservice goes up, by the time you set up a new VM for that microservice your traffic might reduce again and you would have to scale down again.

* With Containerization you can use the kernel of the host operating system from the underlying infrastructure (that can be a physical or virtual machine) inside each container. As a result, these containers are lightweight and cheap to create and destroy. As opposed to VMs where they have their own operation systems.





* So, using containers will eliminate the portability issue as these containers are self-contained and you can use them wherever you want.
* In terms of deployment, they are lightweight and easy to restart, create and delete. And this will help with the scalability issue as well.
* So, in a sense, the container engine acts like a Hypervisor allowing all containers use the host OS and have their resources allocated on demand.

# Docker Introduction

## Container vs Image

Docker images are essentially snapshots of a filesystem that contain everything needed to run a piece of software, including the operating system environment (though typically stripped down to minimal components), libraries, and dependencies.

A container is a runnable application or service that can be turned on and off and restarted. An image on the other hand is a reusable file made of dependencies, codes, libraries and docker files that are used to create containers.

A container is basically a run-time instance of an image. And you can have multiple containers using one single image. Think of images as java classes/templates and containers as instances of those classes.

## Docker Engine vs Docker Desktop

* Docker Engine is an open-source containerization technology that has 3 parts:

A long-running, daemon process, dockerd. An API to interact with this process and a CLI client that uses this API, docker. If you’re running on Linux, both the daemon process and the client can be run on your Linux OS.

* With Docker Desktop that you’re most likely to use on mac and windows (there is one for Linux but it might be slightly different), the daemon is inside a virtual machine and the client is on your host machine. That virtual machine also comes with a GUI and some extensions, Kubernetes, etc.
* So, on windows you have a virtual machine with a GUI that your containers will run in, and a client that is on the host OS. *docker version* command will show this.
* That virtual machine is probably going to be the WSL. So, the Docker Desktop itself is not a virtual machine. (How?)
* With docker desktop you can also switch between windows and Linux containers. (How?)

## Other Terms

* Docker host probably refers to the OS or the VM that dockerd that’s called Docker Server I guess, runs on.
* Docker Registry is a place that images can be stored in and pulled from. It can be a public registry like ducker hub or a private one on AWS for example.

# Docker Image Generation

There are three ways to generate docker images. We go through each one to generate a docker image out of spring boot projects.

## Dockerfile

Docker builds images by reading the instructions from a Dockerfile. A Dockerfile is a text file containing instructions for building your source code.

* As a side note, this guy uses the JAR packaging to make a docker image out of his spring boot project.
* The JAR file contains all the dependencies needed to run the app except for Java Itself and I think that the JDK and Java SE library is not present inside the JAR file and the host OS will provide it. This type of JAR files containing all the dependencies and things you need to run the app on its own called Fat/Uber JAR.
* The spring-boot-mvn-plugin that is added to the build tag of the POM is used this way:

**mvn spring-boot:run**

I’m almost sure that this command runs the application without creating a jar file and the running that jar.

* If you want to run the JAR file with java:

**java -jar target/artifact-id-version.jar**

* The same out put will show up on the terminal running both commands.
* Back to docker: to make a docker image out of the spring boot project, you create a file with the exact name of **Dockerfile** without any extension in the root of your project.
* For basic instructions in writing Dockerfiles go to the link <https://docs.docker.com/build/concepts/dockerfile/>
* To build the image, if the Dockerfile is in the current directory(.) run:

**docker build -t test:latest .**

* To inspect the image run:

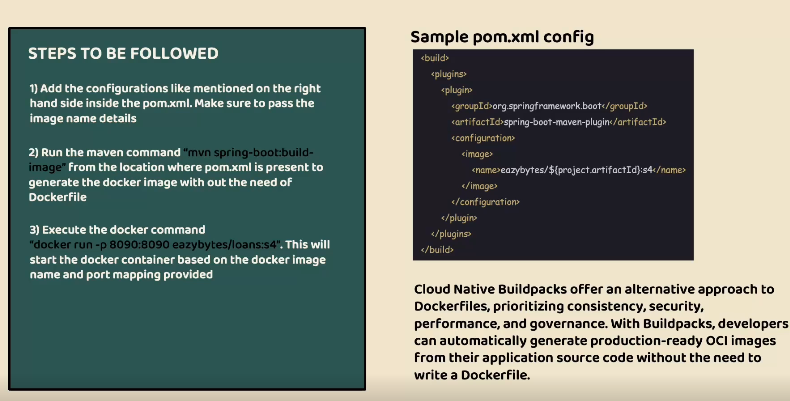
**docker inspect image \_id**

* To run the image as a container:

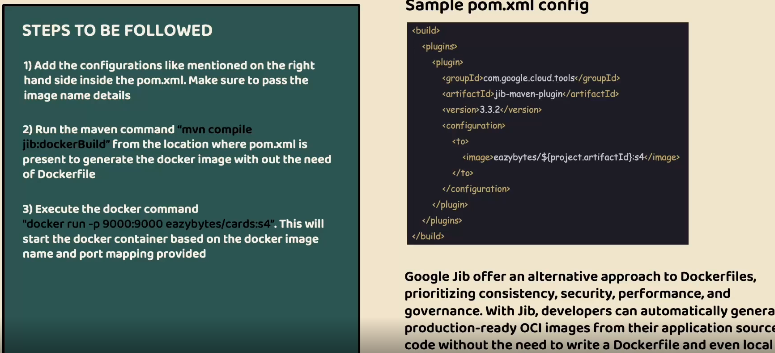
**docker run -p 127.0.0.1:8000:8000 username/repository:tag**

* To stop and start containers run: docker start/stop \_id
* Pay attention to the difference between these two:
* ENTRYPOINT: executes when a container is being made out of an image.
* CMD: it can be a default command that will run if no parameters are passed to the run command or it can provide default arguments for ENTRYPOINTS.

## Buildpacks



## Google Jib



## Push to Docker Registry

docker image push registry-host:5000/erfanscott/repo:tag

# To Be Read Later

* Docker compose
* Ram allocation