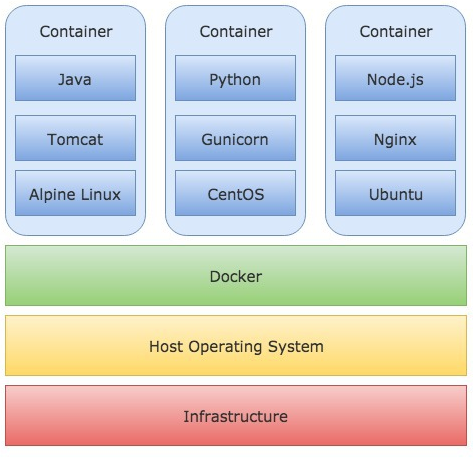
**Docker**

**Container:** A container image is a lightweight, stand-alone, executable package of pieces of software that includes everything needed to run it, code, runtime system tools, system libraries, settings.

Docker solves these problems by creating a lightweight, standalone, executable package of your application that includes everything needed to run it including the code, the runtime, the libraries, tools, environments, and configurations.

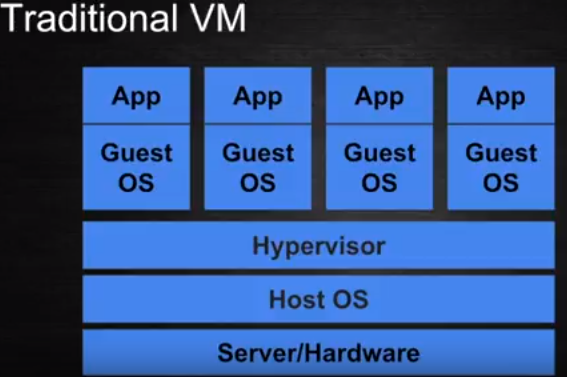
These standalone executable packages are called docker images. And a running instance of a docker image is called a docker container.

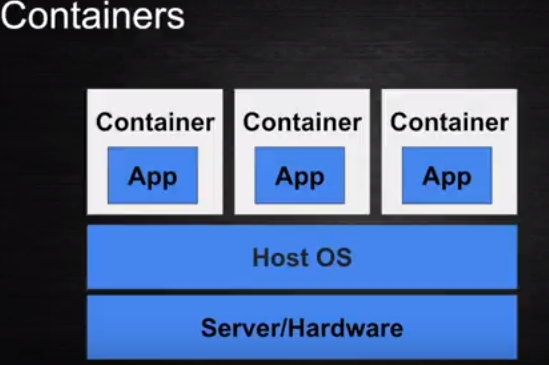


Moreover, you can run multiple containers of completely different configurations on the same infrastructure. All containers are completely isolated and run independently from each other.

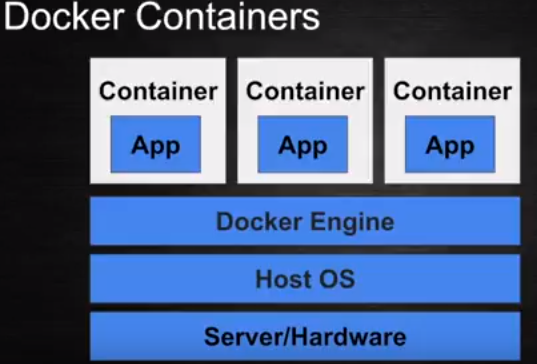
**Why we do need container now:**

* Unnecessary waste of space in creating multiple OS in VM/Hypervisor.
* License of OS for each VM/hypervisor.
* Complex configuration of VM/hypervisor.





**In the case of container hypervisor/VM, guest OSS specifics to each application gone, container is image whichs take care of required for the application.**

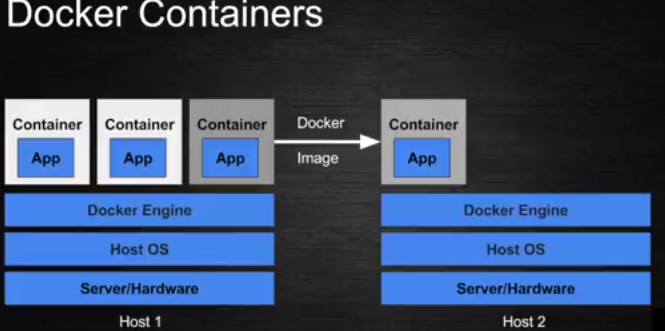
****

Different Containers with application

Docker engine abstraction of all containers to speck with the host OS.

**Advantages of Docker:**

The major advantage is image can be swift with one host to other host easily deployment up and run within minuet



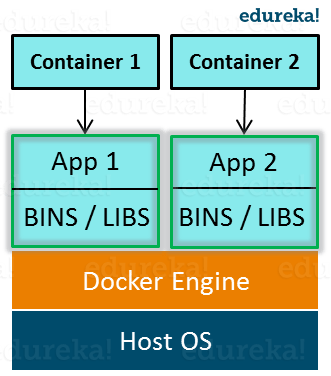
In simple words, Docker is a software containerization platform, meaning you can build your application, package them along with their dependencies into a container and then these containers can be easily shipped to run on other machines.

For example: Lets consider a linux based application which has been written both in Ruby and Python. This application requires a specific version of linux, Ruby and Python. In order to avoid any version conflicts on user’s end, a linux docker container can be  created with the required versions of Ruby and Python installed along with the application. Now the end users can use the application easily by running this container without worrying about the dependencies or any version conflicts.

**Docker Tutorial – Introduction To Docker**

Docker is a containerization platform that packages your application and all its dependencies together in the form of Containers to ensure that your application works seamlessly in any environment.

As you can see in the diagram on the below, each application will run on a separate container and will have its own set of libraries and dependencies. This also ensures that there is process level isolation, meaning each application is independent of other applications, giving developers surety that they can build applications that will not interfere with one another.



As a developer, I can build a container which has different applications installed on it and give it to my QA team who will only need to run the container to replicate the developer environment.

## ****Benefits of Docker****

Now, the QA team need not install all the dependent software and applications to test the code and this helps them save lots of time and energy. This also ensures that the working environment is consistent across all the individuals involved in the process, starting from development to deployment. The number of systems can be scaled up easily and the code can be deployed on them effortlessly.

## ****Virtualization vs Containerization****

What is Virtualization?

Virtualization is the technique of importing a Guest operating system on top of a Host operating system, it allowed developers to run multiple operating systems in different virtual machines all running on the same host.

 Virtualization also has some shortcomings. Running multiple Virtual Machines in the same host operating system leads to performance degradation. This is because of the guest OS running on top of the host OS, which will have its own kernel and set of libraries and dependencies. This takes up a large chunk of system resources, i.e. hard disk, processor and especially RAM.

Another problem with Virtual Machines which uses virtualization is that it takes almost a minute to boot-up. This is very critical in case of real-time applications.

Following are the disadvantages of Virtualization:

* Running multiple Virtual Machines leads to unstable performance
* Hypervisors are not as efficient as the host operating system
* Boot up process is long and takes time

Containerization:

Containerization is however more efficient because there is no guest OS here and utilizes a host’s operating system, share relevant libraries & resources as and when needed unlike virtual machines. Application specific binaries and libraries of containers run on the host kernel, which makes processing and execution very fast. Even booting-up a container takes only a fraction of a second. Because all the containers share, host operating system and holds only the application related binaries & libraries. They are lightweight and faster than Virtual Machines.

Advantages of Containerization over Virtualization:

* Containers on the same OS kernel are lighter and smaller
* Better resource utilization compared to VMs
* Boot-up process is short and takes few seconds

 Containers only contain application specific libraries which are separate for each container and they are faster and do not waste any resources.

Virtualization and Containerization both let you run multiple operating systems inside a host machine.

Virtualization deals with creating many operating systems in a single host machine. Containerization on the other hand will create multiple containers for every type of application as required.

**Figure:** What Is Big Data Analytics – Virtualization versus Containerization

As we can see from the image, the major difference is that there are multiple Guest Operating Systems in Virtualization which are absent in Containerization. The best part of Containerization is that it is very light weight as compared to the heavy virtualization.

The main difference between them is that Docker is an isolated process that runs in your native OS while the virtual machine is a complete isolated guest OS that runs on top of your host OS which takes more time to load. So Docker has benefits over virtual machines such as:

* Loading speed
* Small hardware resources required, unlike virtual machines.
* Running multiple Docker containers at the same time on the same OS.
* You can modify your container and deploy it or give the Docker file definition to a friend to start working on the same environment.

Actually, Docker is not a replacement for virtual machines, it comes to solve specific problems.

Suppose that your application needs 3 or more services which run on different operating systems so instead of running 3 virtual machines on the same host, you can run 3 containers smoothly on the same host. Sounds great!!

|  |  |  |  |
| --- | --- | --- | --- |
| **Virtual Machines Vs Docker Containers** | | | |
| **Virtual Machines** | **Docker Containers** |  |  |
| Need more resources | Less resources are used |  |  |
| Process isolation is done at hardware level | Process Isolation is done at Operating System level |  |  |
| Separate Operating System for each VM | Operating System resources can be shared within Docker |  |  |
| VMs can be customized. | Custom container setup is easy |  |  |
| Takes time to create Virtual Machine | Creation of docker is very quick |  |  |
| Booting takes minutes | Booting is done within seconds |  |  |

 Well, Let’s understand that by asking a few questions. Does any of the following sound familiar to you?

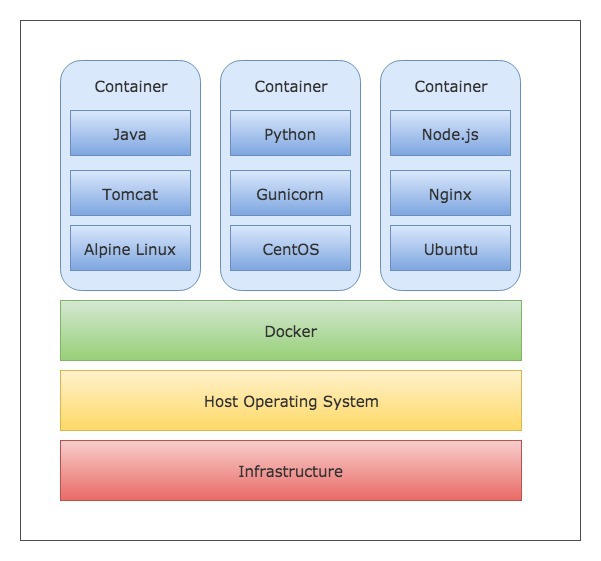
* “It runs on my machine!!”
* “I think your Tomcat version is outdated!
* “I don’t want to install 10 different libraries and tools before being able to run your application. Can’t it come in a packaged form with all the libraries and tools it needs?”
* “We have applications written in different languages, tools, system libraries, and environments. Is there a way we can run them independently on the same infrastructure?”

I bet some of them does.

Docker solves these problems by creating a lightweight, standalone, executable package of your application that includes everything needed to run it including the code, the runtime, the libraries, tools, environments, and configurations.

These standalone executable packages are called docker images. And a running instance of a docker image is called a docker container.

Now, these container images can be shared, shipped and run anywhere in any environment. They will behave exactly the same regardless of the environment they run in.



Moreover, you can run multiple containers of completely different configurations on the same infrastructure. All containers are completely isolated and run independently from each other.

Difference between Docker Image and container?

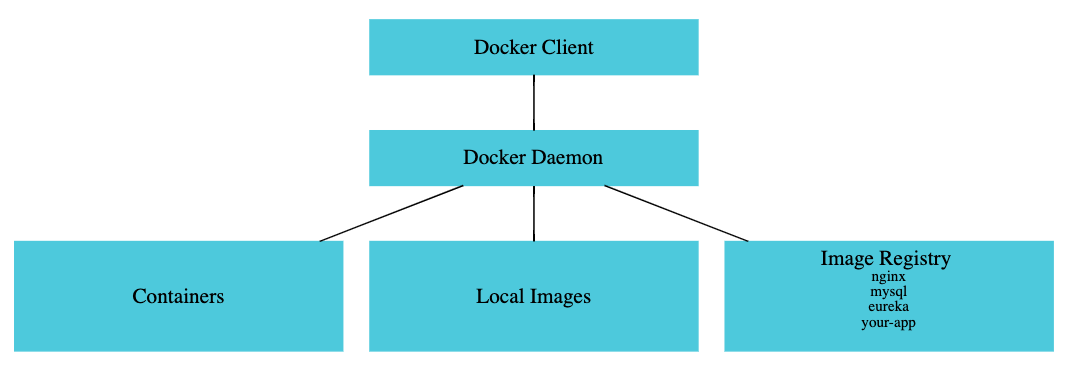
Docker container is the runtime instance of docker image.

Docker Image does not have a state, and its state never changes as it is just set of files whereas docker container has its execution state.

Image is nothing but static file, container is a running the version of specific image.

A Docker Image is the template (application plus required binaries and libraries) needed to build a running Docker Container (the running instance of that image).

As templates, images are what can be used to share a containerized applications. Collections of images are stored/shared in registries like [Docker Hub](https://hub.docker.com/). When you download an image, it can then be used (as a template) to spin up multiple running Containers in your own environment.



**Deployment of Spring Boot Application:**

**Step 1: Create spring boot application with sample restController:**

|  |
| --- |
| **SpringBoot Application:**  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  public class SpringBootDockerApplication {  public static void main(String[] args) {  SpringApplication.run(SpringBootDockerApplication.class, args);  }  }  **Rest Controller:**  **import** org.springframework.web.bind.annotation.GetMapping;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  **publicclass** DemoController {    @GetMapping("/hello")  **public** String hello() {  **return**"Hello Docker Welcome to my world";    }  } |

### Step 2: Defining a docker image with Dockerfile

|  |
| --- |
| **Properties file under root directory of spring boot application:**  **DockerFile**  # Start with a base image containing Java runtime  FROM openjdk:8  # Add a volume pointing to /tmp  #VOLUME /tmp  # The application's jar file  ARG JAR\_FILE=/target/docker-spring-boot.jar  ADD ${JAR\_FILE} docker-spring-boot.jar  # Make port 8080 available to the world outside this container  EXPOSE 8585  # Run the jar file  ENTRYPOINT ["java","-jar","docker-spring-boot.jar"]  Or  # Start with a base image containing Java runtime  FROM alpine-jdk:base  MAINTAINER nagendra.kldm@gmail.com  RUN apk --no-cache add netcat-openbsd  # The application's jar file  COPY springboot-elk-logger-0.0.1-SNAPSHOT.jar /opt/lib/  # Make port 9411 available to the world outside this container  EXPOSE 7777  # Run the jar file  ENTRYPOINT ["java","-jar","/opt/lib/springboot-elk-logger-0.0.1-SNAPSHOT.jar"]  # Add a volume pointing to /var/lib/micro-meter-volume  VOLUME /var/lib/springboot-elk-volume |

The Dockerfile is very simple and declarative. Let’s go through each line of the Dockerfile and understand the details.

* **FROM**: A docker image can use another image available in the docker registry as its base or parent image. In the above example, we use the [openjdk:8-jdk-alpine](https://hub.docker.com/_/openjdk/) image as our base image. It is a very lightweight OpenJDK 8 runtime image that uses [Alpine Linux](https://alpinelinux.org/). (It’s perfect for small Java microservices.)
* **LABEL**: The LABEL instruction is used to add metadata to the image. In the above Dockerfile, we have added some info about the maintainer of the image through LABEL instruction.
* **VOLUME**: Volumes are a mechanism to persist data generated by the container on the Host OS, and share directories from the Host OS with the container.

The VOLUME instruction creates a mount point on the container with the specified path. When you run the container, you can specify the directory on the Hot OS to which the given mount point will be mapped to. After that, anything that the container writes to the mounted path is written to the mapped directory on the Host OS.

One of the most common use cases of volumes is to store the log files generated by the container on the Host OS. For example, Let’s say that your application writes log files to a location /var/log/app.log.

You can mount a VOLUME with path /var/log in the Dockerfile, and then specify the directory on the Host OS to which this mount point will be mapped to while running the container. After that, you’ll be able to access the logs from the mapped directory on the Host OS.

In the above Dockerfile, we created a mount point with path /tmp because this is where the spring boot application creates working directories for Tomcat by default. Although it’s not required for this spring boot application because who cares about tomcat directories. But if you want to store stuff like tomcat access logs, then VOLUMES are very useful.

You can learn more about Volumes from the [official documentation](https://docs.docker.com/storage/volumes/).

* **EXPOSE**: As the name suggests, this instruction allows you to expose a certain port to the outside world.
* **ARG**: The ARG instruction defines a variable with a default value. You can override the default value of the variable by passing it at build time.
* ARG <name>[=<default value>]

Once defined, the variable can be used by the instructions following it.

* **ADD**: The ADD instruction is used to copy new files and directories to the docker image.
* **ENTRYPOINT**: This is where you configure how

**what is the difference between copy and add in dockerfile**

COPY copies a file/directory from your host to your image.

ADD copies a file/directory from your host to your image, but can also fetch remote URLs, extract TAR files, etc...

Use COPY for simply copying files and/or directories into the build context.

Use ADD for downloading remote resources, extracting TAR files, etc..

COPY and ADD are both Dockerfile instructions that serve similar purposes. They let you copy files from a specific location into a Docker image.

COPY takes in a src and destination. It only lets you copy in a local file or directory from your host (the machine building the Docker image) into the Docker image itself.

ADD lets you do that too, but it also supports 2 other sources. First, you can use a URL instead of a local file / directory. Secondly, you can extract a tar file from the source directly into the destination.

# Docker difference between run, cmd, entrypoint commands

* RUN executes the command(s) that you give in a new layer and creates a new image. This is mainly used for installing a new package.
* CMD sets default command and/or parameters, however, we can overwrite those commands or pass in and bypass the default parameters from the command line when docker runs
* ENTRYPOINT is used when you want to run a container as an executable

### COPY vs ADD

Both ADD and COPY are designed to add directories and files to your Docker image in the form of ADD <src>... <dest>or COPY <src>... <dest>. Most resources, including myself, suggest to use COPY.

The reason behind this is that ADD has extra features compared to COPY that make ADD more unpredictable and a bit over-designed. ADD can pull files from url sources, which COPY cannot. ADD can also extract compressed files assuming it can recognize and handle the format. You cannot extract archives with COPY.

The ADD instruction was added to Docker first, and COPY was added later to provide a straightforward, rock solid solution for copying files and directories into your container’s file system.

If you want to pull files from the web into your image I would suggest to use RUN and curl and uncompress your files with RUN and commands you would use on the command line.

### ENV

ENV is used to define environment variables. The interesting thing about ENV is that it does two things:

1. You can use it to define environment variables that will be available in your container. So when you build an image and start up a container with that image you’ll find that the environment variable is available and is set to the value you specified in the Dockerfile.
2. You can use the variables that you specify by ENV in the Dockerfile itself. So in subsequent instructions the environment variable will be available.

### RUN

RUN will execute commands, so it’s one of the most-used instructions. I would like to highlight two points:

1. You’ll use a lot of apt-get type of commands to add new packages to your image. It’s always advisable to put apt-get update and apt-get install commands on the same line. This is important because of layer caching. Having these on two separate lines would mean that if you add a new package to your install list, the layer with apt-get update will not be invalidated in the layer cache and you might end up in a mess. [Read more here](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#run).
2. RUN has two forms; RUN <command> (called shell form) and RUN ["executable", "param1", "param2"] called exec form. Please note that RUN <command> will invoke a shell automatically (/bin/sh -c by default), while the exec form will not invoke a command shell

### Step 3 : Building the Docker image:

### Now that we have defined the Dockerfile, let’s build a docker image for our application. Type the following command from the root directory of the project to build the docker image

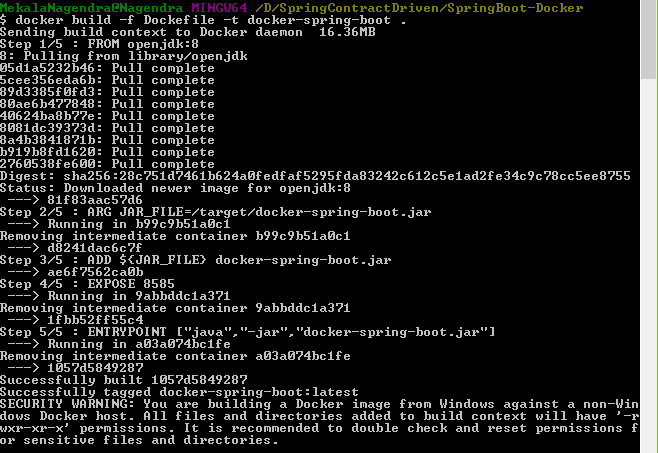
**Command:**

docker build -f Dockefile -t docker-spring-boot .

or

docker build -f Dockefile -t docker-spring-boot:v1 .

Executed all tasks defined in docker file properties file:

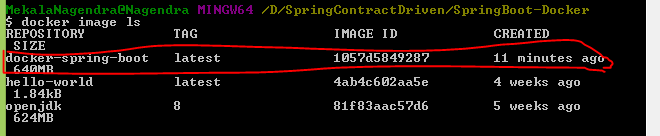


You can see the list of all the docker images on your system using the following command

**Command:**

docker image ls

This should display our newly built docker image.



### Running the docker image

Once you have a docker image, you can run it using docker run command like so -

$ docker run -p 8585:8085 docker-spring-boot

<http://localhost:8585/hello>

In the run command, we have specified that the port 8080 on the container should be mapped to the port 5000 on the Host OS.

Once the application is started, you should be able to access it at http://localhost:8585

The container runs in the foreground, and pressing CTRL + C will stop it. Let’s now see how to run the container in the background.

In the run command, we have specified that the port 8080 on the container should be mapped to the port 5000 on the Host OS.

Once the application is started, you should be able to access it at [http://localhost:5000](http://localhost:5000/).

The container runs in the foreground, and pressing CTRL + C will stop it. Let’s now see how to run the container in the background.

1. **how to search a docker image in hub.docker.com**

docker search httpd

1. **Download a docker image from hub.docker.com**

docker image pull <image\_name>:<image\_version/tag>

1. **List out docker images from your local system**

docker image ls

1. **Create/run/start a docker container from image**

docker run -d --name <container\_Name><image\_name>:<image\_version/tag>

d - run your container in back ground (detached)

1. **Expose your application to host server**

docker run -d -p <host\_port>:<container\_port> --name <container\_Name><image\_name>:<Image\_version/tag>

docker run -d --name httpd\_server -p 8080:80 httpd:2.2

1. **List out running containers**

docker ps

docker container ls

1. **List out all docker container (running, stpooed, terminated, etc...)**

docker ps –a

docker container ls -a

1. **run a OS based container which interactive mode (nothing but login to container after it is up and running)**

docker run -i -t --name centos\_server centos:latest

i - interactive

t - Terminal

1. **Stop a container**

docker stop <container\_id>

docker container stop 151a77679241

1. **Start a container which is in stopped or exit state**

docker start <container\_id>

docker container start c165f459e7d7

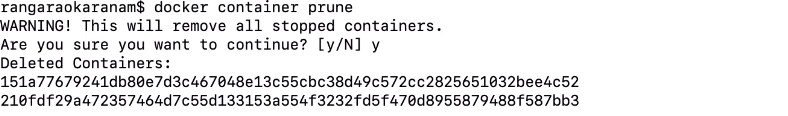
1. **Remove a container**

docker rm <container\_id>

docker container rm fed549e69e9d

remove all containers usinf single command

docker container prune

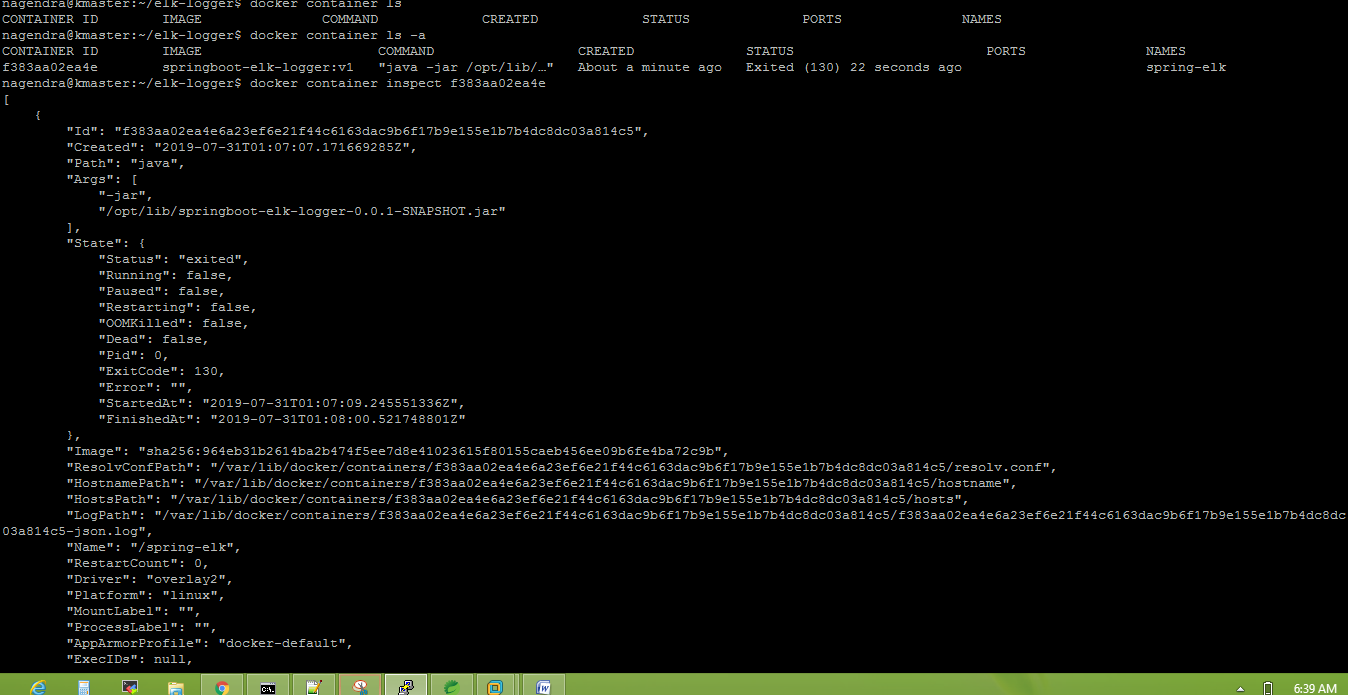


1. **login to a docker container**

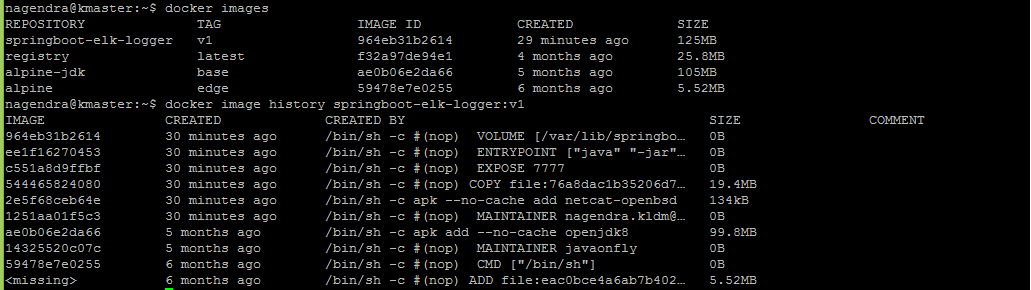
docker exec -it <container\_Name> /bin/bash

1. docker info
2. docker search <imagename>
3. docker container logs c165f459e7d7
4. docker container inspect command display the container details information liker crated date, java vm arguments , state(staus,running,paused,restarted), image name, logs details location path,

docker container inspect f383aa02ea4e



1. docker image history springboot-elk-logger:v1



**Important commands:**

docker ps –aq // it will display only container ids

docker ps –l // displayed only latest container

docker ps –lq //display only latest container ids

docker logs container-id //display container logs using container id

docker container logs c165f459e7d7

ex: docker logs 122d5dddd5

docker logs -f 7fc8916283da // it will display the live logs like tail command

docker start container-id // start the container using id

docker stop container-id // stop the container using id

docker kill container-id // it will take litlebit less time than stop command.

docker ps -a | wc –l // running docker container count

docker rmi –f container-id // forcefully removed container

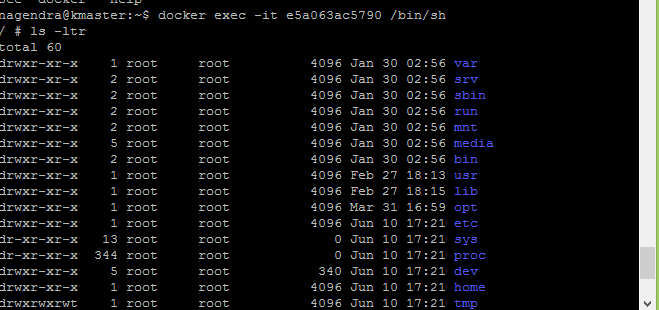
docker rm $(docker ps -aq)

docker ps | grep –I webapp2 // it will disply containder ruunig info

**docker exec command: login into container for troubleshooting application like..**

Docker engine has a command line tool docker which is used to interact with containers. The command option exec is used to run a command in a running container. By passing some parameters, we should be able to get bash session. The command syntax used is:docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

Example : docker exec -it e5a063ac5790 /bin/sh



Below is a list of options which can be used with this.

Note: ctrl + pq for exist container.

-d, --detach Detached mode: run command in the background

-i, --interactive Keep STDIN open even if not attached

-t, --tty Allocate a pseudo-TTY

-u, --user string Username or UID (format: <name|uid>[:<group|gid>])

-w, --workdir string Working directory inside the container

If Container not running check the logs:

docker ps -a | grep -i <containe-id>

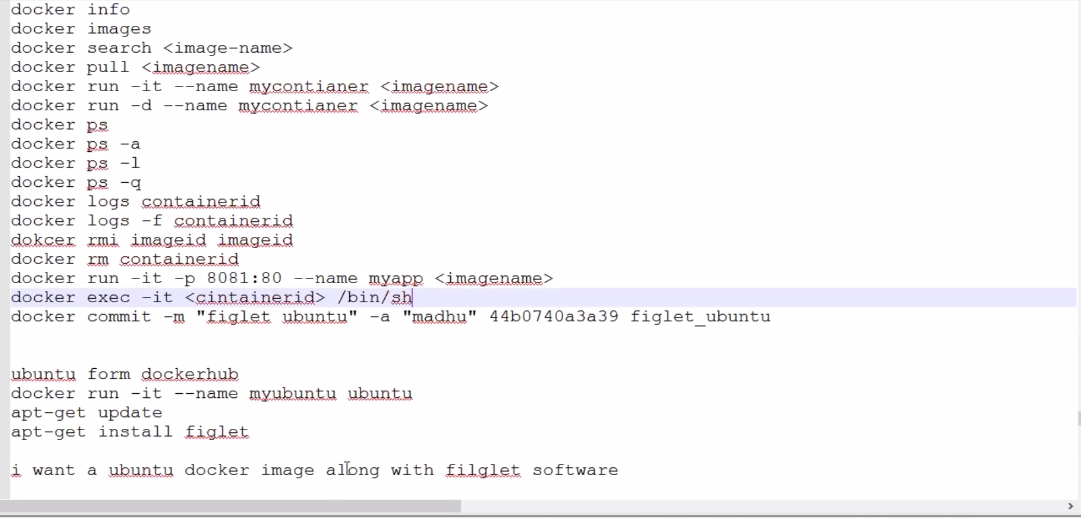
Ex: docker ps -a | grep -i e5a063ac5790

docker logs e5a063ac5790

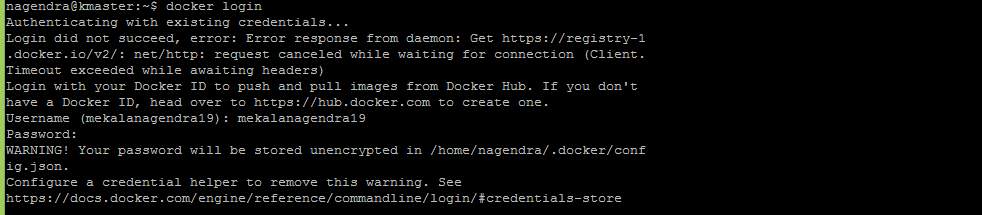
**docker container commit:**

Create a new image from a container’s changes

modify its internal state, and then save the container as an image

****

Docker login command

****

## Tag Description:

## Docker tags convey useful information about a specific image version/variant. They are aliases to the ID of your image

Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE

docker tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG]

**Example:**

docker tag spring-elk mekalanagendra19/springmvcexample:v1

docker push mekalanagendra19/springmvcexample:v1

**Without creating repository:**

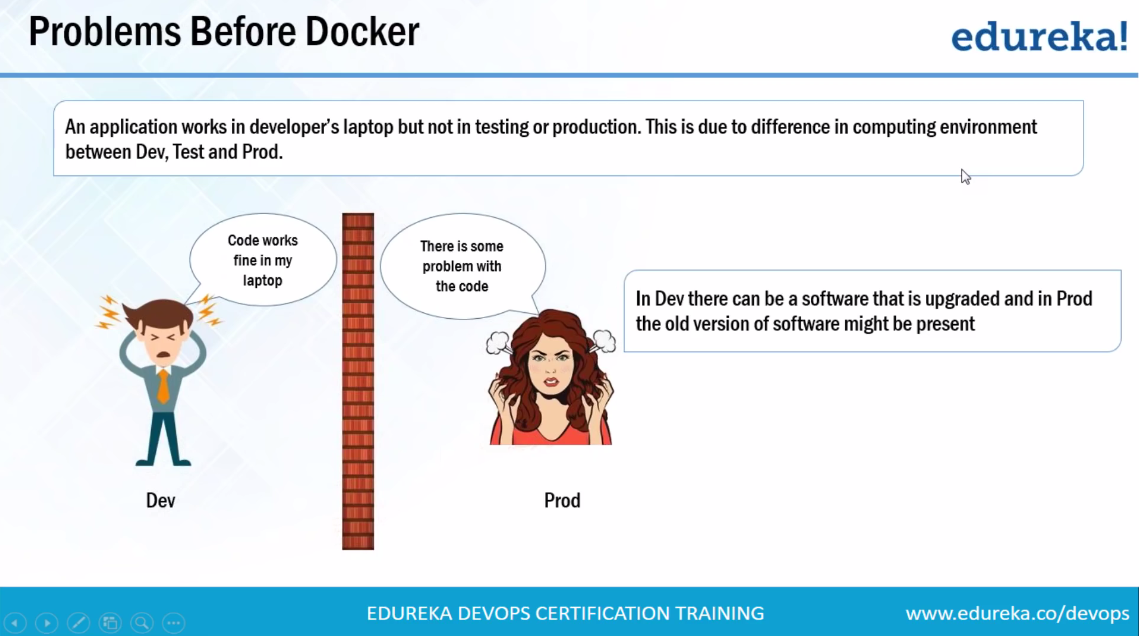
docker tag demo-app mekalanagendra19/demo-app:v1

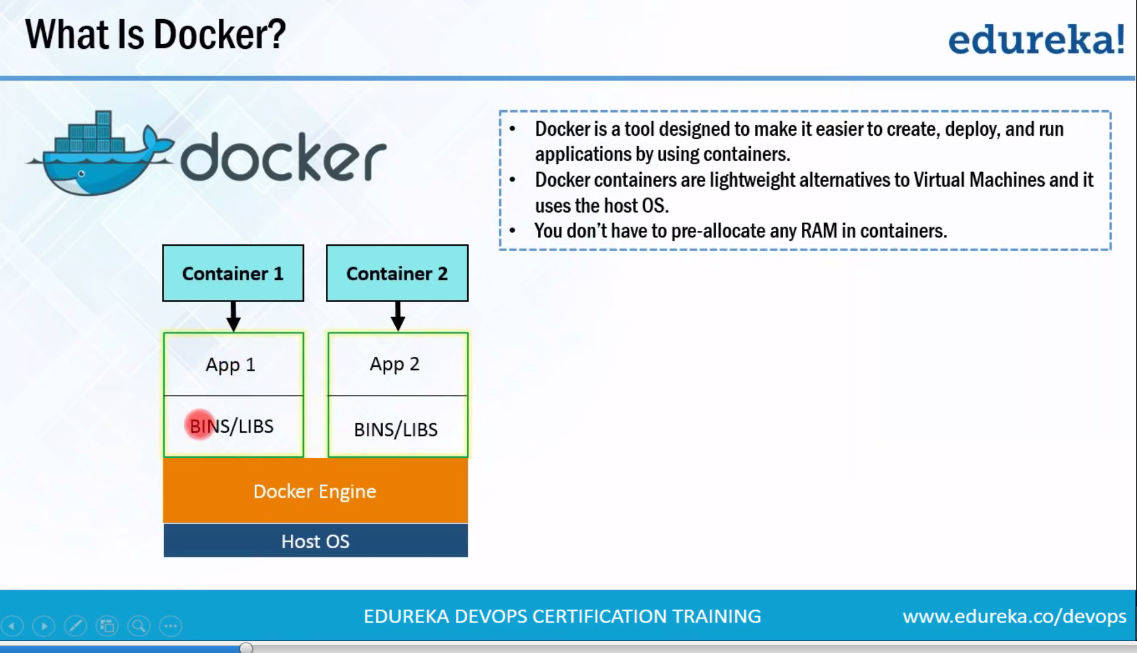
docker push mekalanagendra19/demo-app:v1

docker run --name demo-app –container mekalanagendra19/demo-app:v1

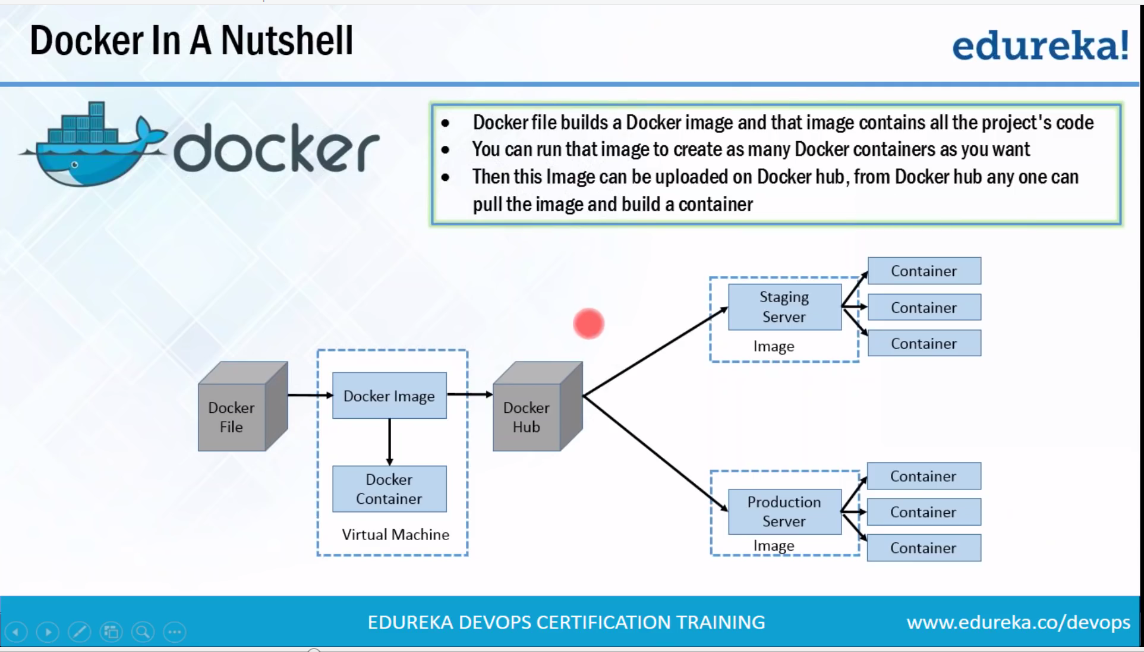
what is different b/w cmd and entrypoint

Problems Before docker:

****



**Docker work flow:**



**Docker Compose file:**

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration. To learn more about all the features of Compose, see [the list of features](https://docs.docker.com/compose/overview/#features).

Compose works in all environments: production, staging, development, testing, as well as CI workflows.

It is a tool which is used to create and start Docker application by using a single command. We can use it to file to configure our application's services.

You define your set of containers in a YAML configuration file, and it manages the runtime configuration of the containers

It is a great tool for development, testing, and staging environments.

It provides the following 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

**Docker file:**

Docker can build images automatically by reading the instructions from a Dockerfile.

Docker builds images automatically by reading the instructions from a text file called Dockerfile. It contains all commands, in order, needed to build a given image. A Dockerfile adheres to a specific format and set of instructions as shown below:

**DockerCompose file:**

**Docker Compose** is used to run multiple containers as a single service. For example, suppose you had an application which required NGNIX and MySQL, you could create one file which would start both the containers as a service without the need to start each one separately.

Docker Compose is a tool to run multiple containers, define how they are connected, how many instances should be deployed, etc.

|  |
| --- |
| **version: '2.2'**  **services:**  config-server:  container\_name: config-server  build:  context: .  dockerfile: Dockerfile-configserver  image: config-server:latest  expose:  - 9090  ports:  - 9090:9090  networks:  - emp-network  volumes:  - config-repo:/var/lib/config-repo  eureka-server:  container\_name: eureka-server  build:  context: .  dockerfile: Dockerfile-EurekaServer  image: eureka-server:latest  expose:  - 9091  ports:  - 9091:9091  networks:  - emp-network  EmployeeSearchService:  container\_name: EmployeeSearch  build:  context: .  dockerfile: Dockerfile-EmployeeSearch  image: employeesearch:latest  environment:  SPRING\_APPLICATION\_JSON: '{"spring": {"cloud": {"config": {"uri": "http://config-server:9090"}}}}'  entrypoint: /opt/bin/EmployeeSearch-entrypoint.sh  expose:  - 8080  ports:  - 8080:8080  networks:  - emp-network  links:  - config-server:config-server  - eureka-server:eureka-server  depends\_on:  - config-server  - eureka-server  logging:  driver: json-file  EmployeeDashboardService:  container\_name: EmployeeDashboard  build:  context: .  dockerfile: Dockerfile-EmployeeDashboard  image: employeedashboard:latest  environment:  SPRING\_APPLICATION\_JSON: '{"spring": {"cloud": {"config": {"uri": "http://config-server:9090"}}}}'  entrypoint: /opt/bin/EmployeeDashBoard-entrypoint.sh  expose:  - 8081  ports:  - 8081:8081  networks:  - emp-network  links:  - config-server:config-server  - eureka-server:eureka-server  depends\_on:  - config-server  - eureka-server  logging:  driver: json-file  ZuulServer:  container\_name: ZuulServer  build:  context: .  dockerfile: Dockerfile-ZuulServer  image: zuulserver:latest  expose:  - 8084  ports:  - 8084:8084  networks:  - emp-network  links:  - eureka-server:eureka-server  depends\_on:  - eureka-server  logging:  driver: json-file  networks:  emp-network:  driver: bridge  volumes:  config-repo:  external: true |

In the Docker compose file below are a few important entries:

* **version**: a mandatory field where we need to maintain the version of the Docker Compose format.
* **services**: each entry defines the container we need to spawn.
* **build**: if mentioned, then Docker Compose should build an image from the given Dockerfile.
* **image**: the name of the image which will be created.
* **networks**: the name of the network to be used. This name should be present in the networks section.
* **links**: this will create an internal link between the service and the mentioned service. Here, the EmployeeSearch service needs to access the config and Eureka server.
* **depends**: this is needed to maintain the order. The EmployeeSearch container depends on the Eureka and Config Server. Hence, Docker makes sure that the Eureka and Config Server containers are spawned before the EmployeeSearch Container is spawned. 3

**What’s the difference between**up**,**run**, and**start**?**

Typically, you want **docker-compose up**. Use up to start or restart all the services defined in a **docker-compose.yml.** In the default “attached” mode, you’ll see all the logs from all the containers. In “detached” mode (-d), Compose exits after starting the containers, but the containers continue to run in the background.

The **docker-compose run** command is for running “one-off” or “ad-hoc” tasks. It requires the service name you want to run and only starts containers for services that the running service depends on. Use run to run tests or perform an administrative task such as removing or adding data to a data volume container. The [run command](http://tekslate.com/docker-commands/) acts like docker run -ti in that it opens an interactive terminal to the container and returns an exit status matching the exit status of the process in the container.

The **docker-compose start** command is useful only to restart containers that were previously created but were stopped. It never creates new containers.

Example:

**Docker start deamon:**

sudo service docker start

**Docker stop deamons**

sudo service docker stop

**Step 1 : Create docker compose file at any location on your system**

docker-compose.yml

**Step 2 : Check the validity of file by command**

docker-compose config

**Step 3 : Run docker-compose.yml file by command**

docker-compose up -d

**Steps 4 : Bring down application by command**

docker-compose down

docker-compose down --volumes

**Display current containers running.**

docker-compose ps

Subsequent executions of docker-compose up won’t require building the image.

However, you can rebuild the image whenever you want, by using:

docker-compose build, which will build all the services but not run them.

docker-compose up --build, which will build all the services and then run them.

How to scale services

—scale

**command scales the “**database**” service to 4 tasks.**

docker-compose up -d --scale database=4

**command scales the “frontend” service to 50 tasks.**

docker service scale frontend=50

**multiple service scale at time**

docker service scale backend=3 frontend=5

### Command line

*# Use the up command with the --scale flag instead. - Scale SERVICE to NUM instances.*

*# docker-compose up --scale SERVICE=NUM*

docker-compose up --scaleweb=2worker=3

*# deprecated*

docker-compose scale web=2worker=3

It will overrides the scale setting in the [Compose file](https://gerardnico.com/vm/docker/service/docker-compose.yml) if present.

Advertising

### docker-compose.yml

Added in version 2.2 file format. [Version 2](https://docs.docker.com/compose/compose-file/compose-file-v2/#scale)

*web*:

image: busybox:latest

command: echo 'scaled'

scale: 3

**example:**

[**https://thepracticaldeveloper.com/2017/12/11/dockerize-spring-boot/**](https://thepracticaldeveloper.com/2017/12/11/dockerize-spring-boot/)

**Docker Data Volumes**

Volume is for if deleted container, container generated data we can persistent in host machine, it can be used for anther container

Whenever container is deleted along with container data also removed, because normally container store data inside container.

Docker volumes are used to store container data like logs or data on host machine, whenever container removed.

If we wants to persistent container data we are going to data volumes.

Sometimes however application need to share access to data or persistent data after container is removed.

data volumes are useful for managing your data with Docker containers and host machines. Here you will find two way of managing data volumes on Docker containers.

* 1. Using the data volume container.
* 2. Using share data between the host and the Docker container

In a real-life scenario, we should be pushing the jar file to an Artifact Repository Manager system such as Nexus or Artifactory and the container should download the file from the repo manager.

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers

: docker volume //get information

: docker volume create

: docker volume ls

: docker volume inspect// display the one or more volumes

: docker volume rm// remove all unused local volume

: docker volume prune // remove one or more volumes

By default all files created inside a container are stored on a writable container layer

The data doesn’t persist when that container is no longer running

A container’s writable layer is tightly coupled to the host machine where the container is running. You can’t easily move the data somewhere else.

Docker has two options for containers to store files in the host machine

so that the files are persisted even after the container stops

**VOLUMES and BIND MOUNTS**

Volumes are stored in a part of the host filesystem which is managed by Docker

Non-Docker processes should not modify this part of the filesystem

Bind mounts may be stored anywhere on the host system

In Bind Mounts, the file or directory is referenced by its full path on the host machine.

Volumes are the best way to persist data in Docker

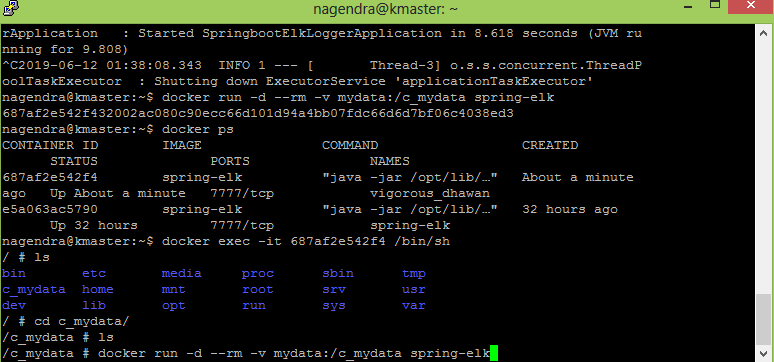
Volumes also support the use of volume drivers, which allow you to store your data on remote hosts or cloud providers, among other possibilities.

By using volume we can share the data one container to others

**Commands**

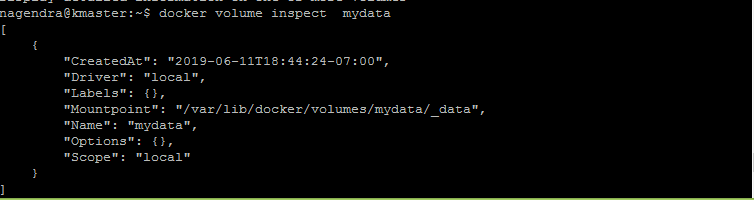
**Create docker volume:**

* docker volume create –name mydata
* docker run -d --rm -v mydata:/c\_mydata spring-elk \
* --rm means whenever container stopped/exists, container automatically deleted



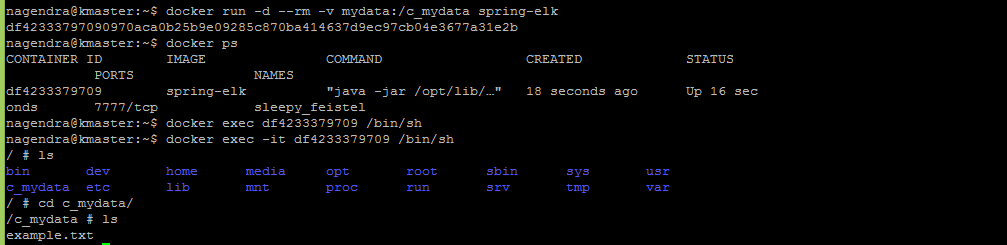
* docker volume inspect mydata

verify the volume is present or not



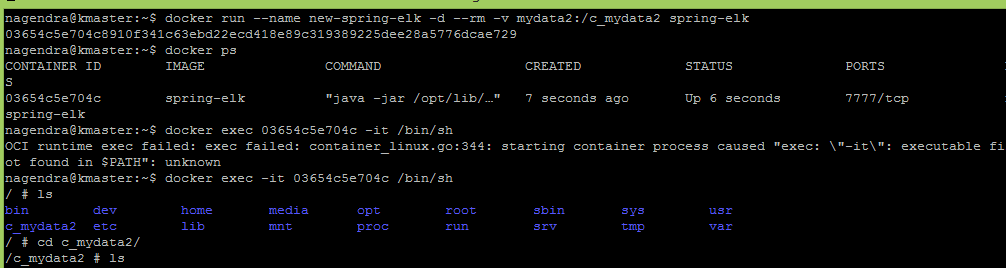
**Bind the volueme into new container**

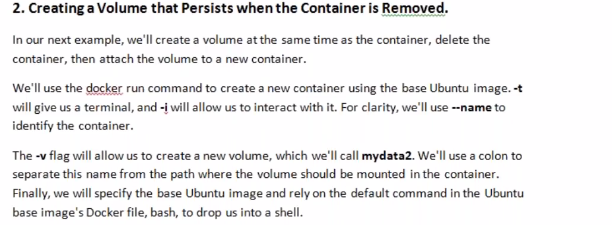
* docker run -d --rm -v mydata:/c\_mydata spring-elk

****

**Create new volume at the time of running container with bind with inside container folder, no need to create volume using docker volume create command.**

* docker run --name new-spring-elk -d --rm -v mydata2:/c\_mydata2 spring-elk

****

****

**Display volumes in host machine**

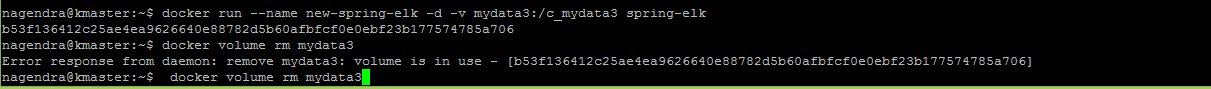
* docker volume ls



**Remve docker volume inside container**

It will volume already using container.

* docker volume rm mydata3

****

Removed with volume id

* docker rm b53f136412c25ae4ea9626640e88782d5b60afbfcf0e0ebf23b177574785a706

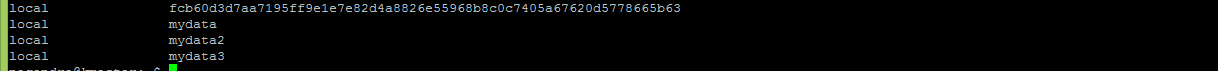


We can forcefully deleted if container running using –f command

* docker rm–f b53f136412c25ae4ea9626640e88782d5b60afbfcf0e0ebf23b177574785a706

Removed volume in host machine

* docker volume ls



* docker volume rm mydata3

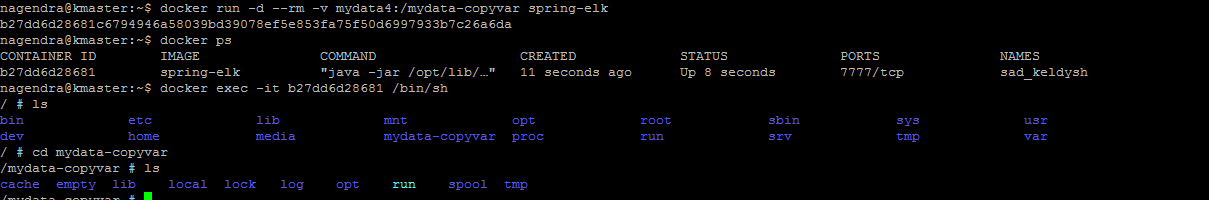


Copy of data inside container, binding to outside container

* docker run -d --rm -v mydata4:/var spring-elk

copy var file data to new container

* docker run -d --rm -v mydata4:/mydata-copyvar spring-elk



docker run --name MyJenkins1 -v myvol1:/var/jenkins\_home -p 8080:8080 -p 50000:50000 jenkins

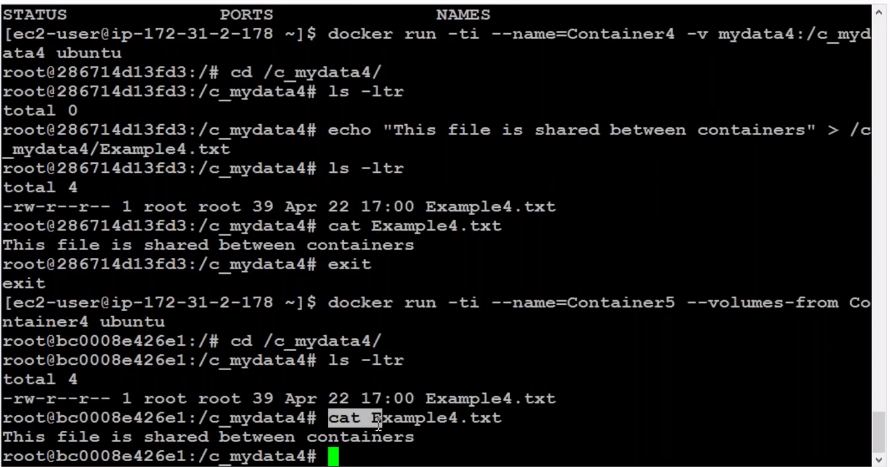
docker run --name MyJenkins2 -v myvol1:/var/jenkins\_home -p 9090:8080 -p 60000:50000 jenkins

ocker run --name MyJenkins3 -v /Users/raghav/Desktop/Jenkins\_Home:/var/jenkins\_home -p 9191:8080 -p 40000:50000 jenkins

**Backup, restore, or migrate data volumes**

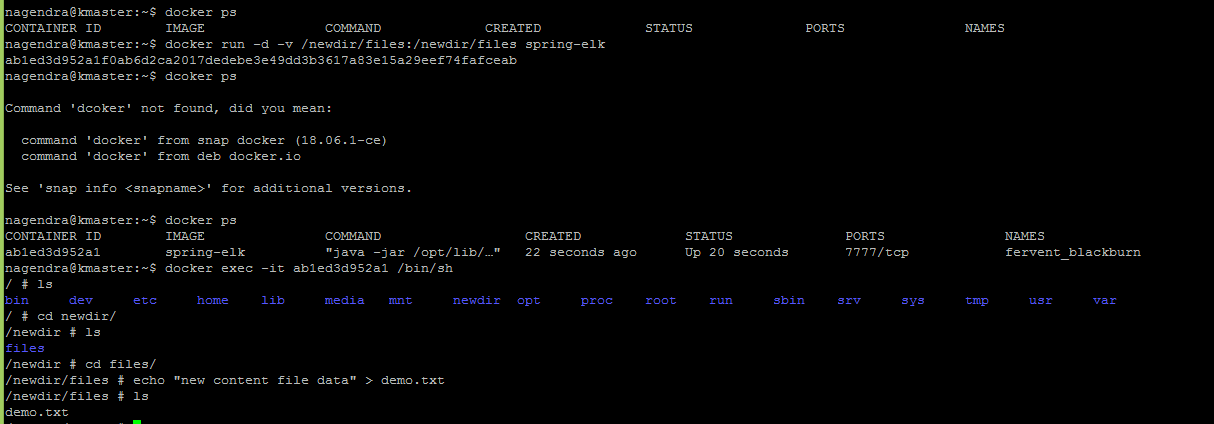
Volumes are useful for backups, restores, and migrations. Use the --volumes-from flag to create a new container that mounts that volum

Sharing the data from one container into anther container usinf –volumes-from arguments

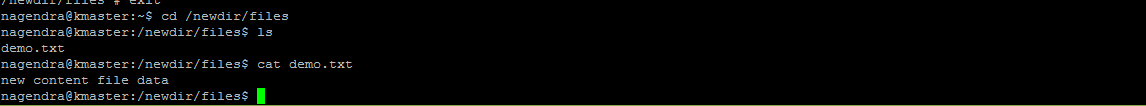


Use / we can able display volume data

* docker run -d -v /newdir/files:/newdir/files spring-elk



We can able see volume file dat



We can update file in host machine automatically reflect inside container directory

ss



**Docker private register**

Docker Private Registry is a highly scalable server-side application that can be used to store and distribute the Docker images internally within your organization. Docker also has its own public registry (Docker Hub) that allows you to store Docker images. But, the images you upload on Docker Hub becomes public. Anyone can access and use your images from Docker Hub. So it is not the best option for your organization. Docker Private Registry allows you to set up a Docker registry for your project privately so that only your organization can store and use Docker images on it. Using Docker Private Registry, you can easily control your images, fully own your images distribution pipeline, and integrate image storage and distribution tightly into your in-house development workflow. If you want to quickly deploy a new image over a large cluster of machines, then Docker Private Registry is the best solution for you.

**Step 1:** Download the docker private register

sudo docker run –d –p 5000:5000 –-name registry registry:2

or

sudo docker run –d –p 5000:5000 --restart=always –-name registry registry:2

**Step 2:**

Now let’s tag one of our existing images so that we can push it to our local repository. In our example, since we have the centos image available locally, we are going to tag it to our private repository and add a tag name of centos.

sudo docker tag 67591570dd29 localhost:5000/centos

The following points need to be noted about the above command −

67591570dd29 refers to the Image ID for the centos image.

localhost:5000 is the location of our private repository.

We are tagging the repository name as centos in our private repository.

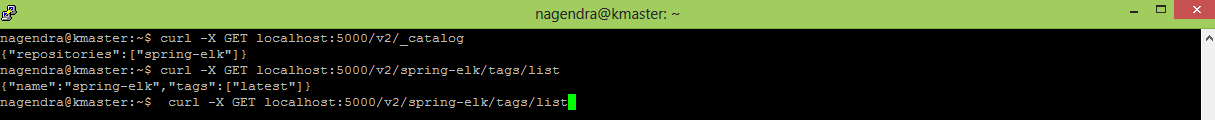
**Step 3: push image into docker private register**

sudo docker push localhost:5000/centos

**Step 4: display private repository**

curl -X GET localhost:5000/v2/\_catalog

curl -X GET localhost:5000/v2/spring-elk/tags/list

****

<https://www.tutorialspoint.com/docker/docker_private_registries.htm>

### Shell and Exec forms

All three instructions (RUN, CMD and ENTRYPOINT) can be specified in shell form or exec form. Let’s get familiar with these forms first, because the forms usually cause more confusion than instructions themselves.

##### Shell form

<instruction> <command>

Examples:

RUN apt-get install python3

CMD echo "Hello world"

ENTRYPOINT echo "Hello world"

When instruction is executed in shell form it calls /bin/sh -c <command>under the hood and normal shell processing happens. For example, the following snippet in Dockerfile

ENV name John Dow

ENTRYPOINT echo "Hello, $name"

when container runs as docker run -it <image> will produce output

Hello, John Dow

Note that variable name is replaced with its value.

##### Exec form

This is the preferred form for CMD and ENTRYPOINT instructions.

<instruction> ["executable", "param1", "param2", ...]

Examples:

RUN ["apt-get", "install", "python3"]

CMD ["/bin/echo", "Hello world"]

ENTRYPOINT ["/bin/echo", "Hello world"]

When instruction is executed in exec form it calls executable directly, and shell processing does not happen. For example, the following snippet in Dockerfile

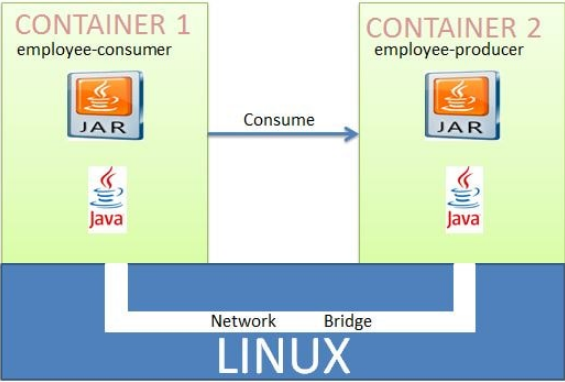
ENV name John Dow

ENTRYPOINT ["/bin/echo", "Hello, $name"]

when container runs as docker run -it <image> will produce output

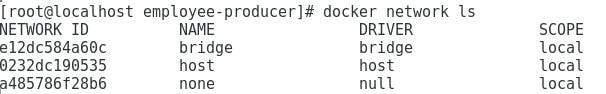
**Inter Docker Container Communication Using Docker Networking**

We will be using Docker Networking to allow multiple containers to interact with each other.



Let’s first check the available networks

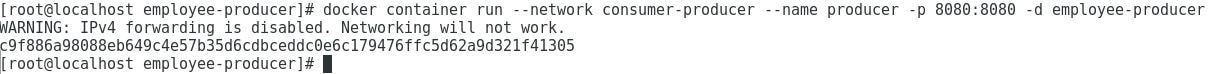
docker network ls

  
Next we will create our own network will be of type bridge

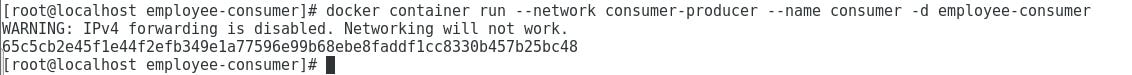
docker network create consumer-producer

docker-create-network  
Lets start the employee producer container on the newly created network.

docker container run **--network consumer-producer** --name producer -p 8080:8080 -d employee-producer

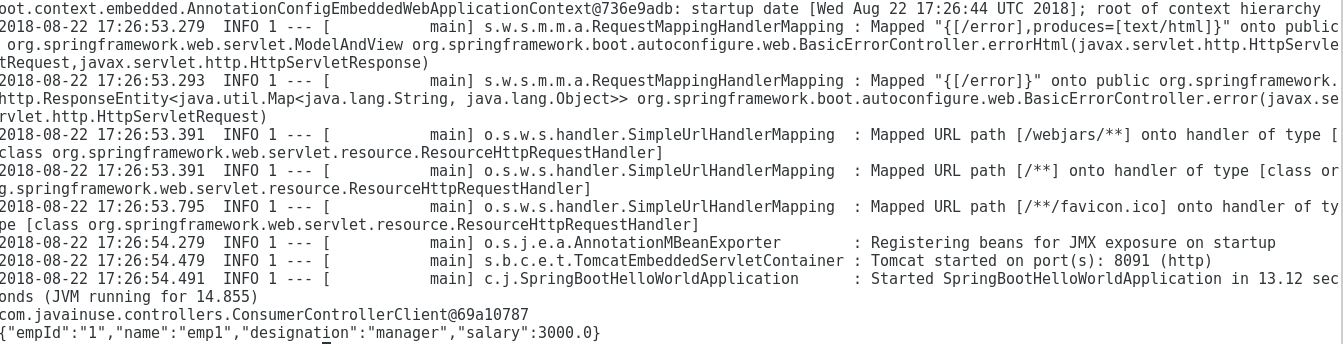
  
Lets start the employee consumer+ container on the newly created network.

docker container run **--network consumer-producer** --name consumer -d employee-consumer



Lets check the consumer container logs-

docker container logs consumer



<https://www.javainuse.com/devOps/docker/docker-networking>

Docker command:

=============

https://codefresh.io/howtos/using-docker-maven-maven-docker/

**Docker jenking with push and pull images:**

https://github.com/ValaxyTech/DevOpsDemos/blob/master/SimpeDevOpsProjects/Project-4.MD

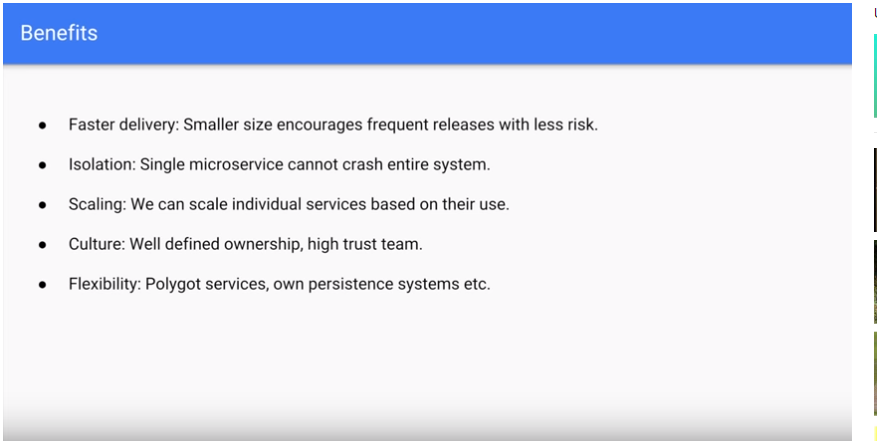
**Best tutorial:**

<https://www.callicoder.com/spring-boot-docker-example/>

<https://www.callicoder.com/spring-boot-websocket-chat-example/>

service messs:===========

https://www.youtube.com/watch?v=QiXK0B9FhO0



What we would like to do is stop failures from cascading down, and provide a way to self-heal, which improves the system’s overall resiliency

**Hystrix** is the implementation of [Circuit Breaker pattern](https://martinfowler.com/bliki/CircuitBreaker.html), which gives a control over latency and failure between distributed services.

The main idea is to stop cascading failures by failing fast and recover as soon as possible —*Important aspects of fault-tolerant systems that self-heal.*



**Pull the image from hub**

docker run -d -e MYSQL\_ROOT\_PASSWORD=dummypassword -e MYSQL\_USER=todos-user -e MYSQL\_PASSWORD=dummytodos -e MYSQL\_DATABASE=todos --p 3306:3306 mysql:5.7

docker run -d -p 8761:8761 springcloud/eureka

**shows docker containers:**

docker container ls

<https://takacsmark.com/dockerfile-tutorial-by-example-dockerfile-best-practices-2018/#env>

<https://linuxacademy.com/blog/kubernetes/getting-started-with-kubernetes-using-minikube/>

<https://takacsmark.com/dockerfile-tutorial-by-example-dockerfile-best-practices-2018/>