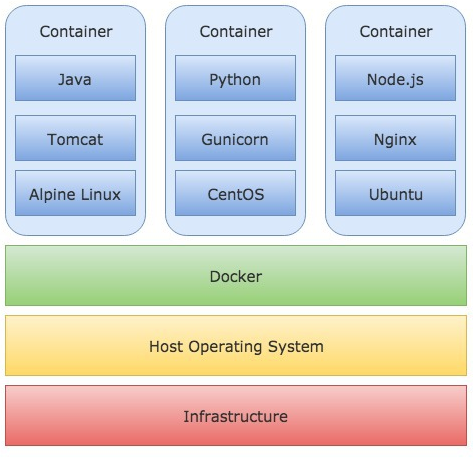
**Docker**

**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.

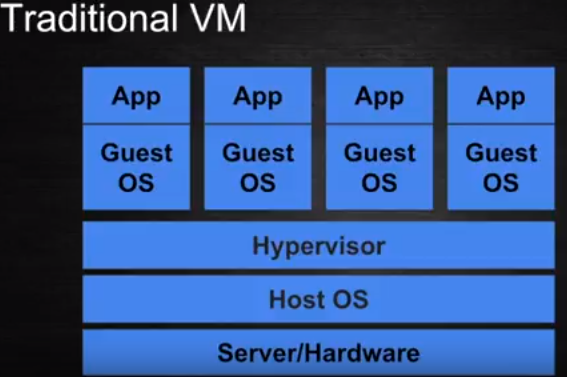
These standalone executable packages are called docker images. And a running instance of a docker image is called a docker container.

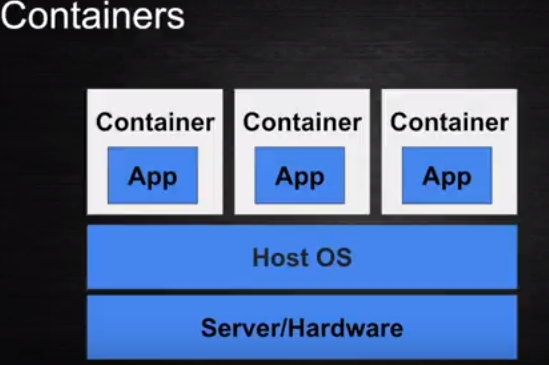


Moreover, you can run multiple containers of completely different configurations on the same infrastructure. All containers are completely isolated and run independently from each other.

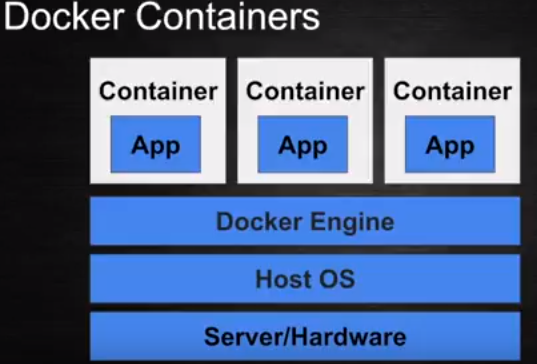
**Why we do need container now:**

* 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.**

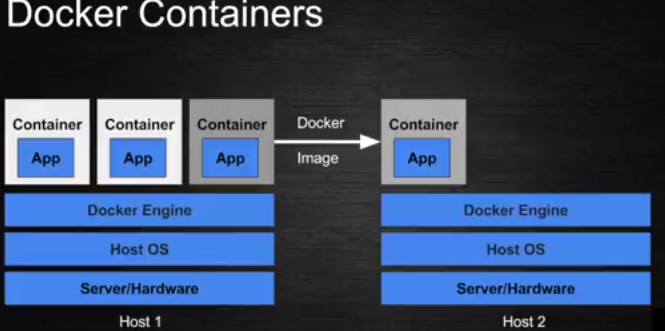
****

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



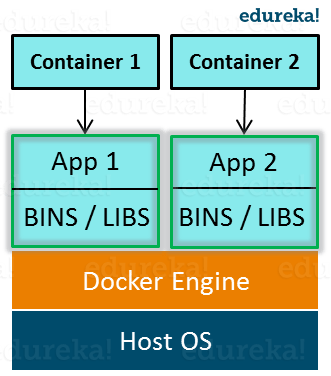
In simple words, Docker is a software containerization platform, meaning you can build your application, package them along with their dependencies into a container and then these containers can be easily shipped to run on other machines.

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****

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 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?

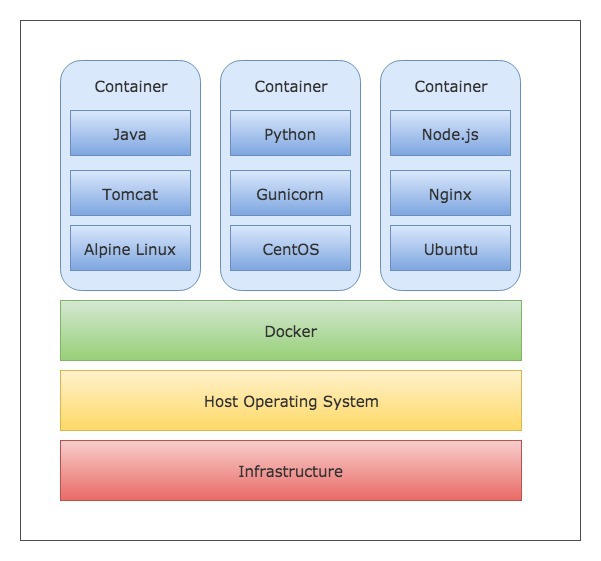
* *“It runs on****my machine****!!”*
* *“I think your****Tomcat version****is outdated!*
* *“I don’t want to install 10 different libraries and tools before being able to run your application. Can’t it come in a****packaged form****with all the libraries and tools it needs?”*
* *“We have applications 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.

**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  **public** **class** 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"] |

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

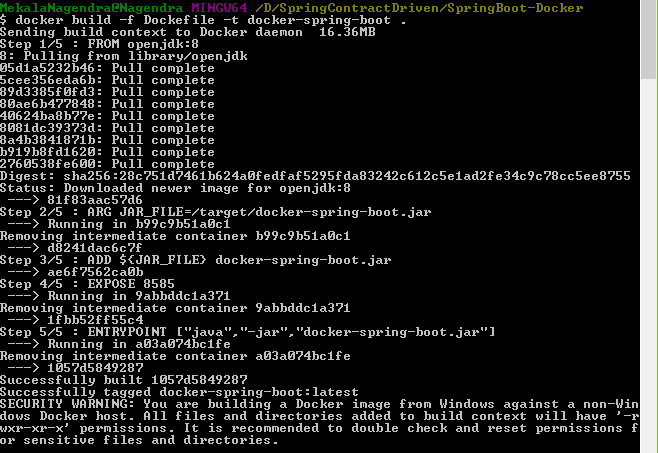
### Step 3 : Building the Docker image:

### 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

Executed all tasks defined in docker file properties file:

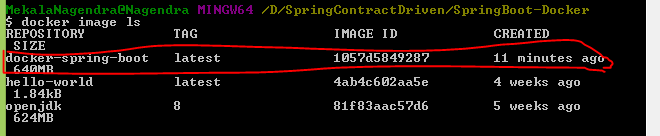


You can see the list of all the docker images on your system using the following command

**Command:**

docker image ls

This should display our newly built docker image.



**Step 4:**

### 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

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

docker ps -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>

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

docker start <container\_id>

1. Remove a container

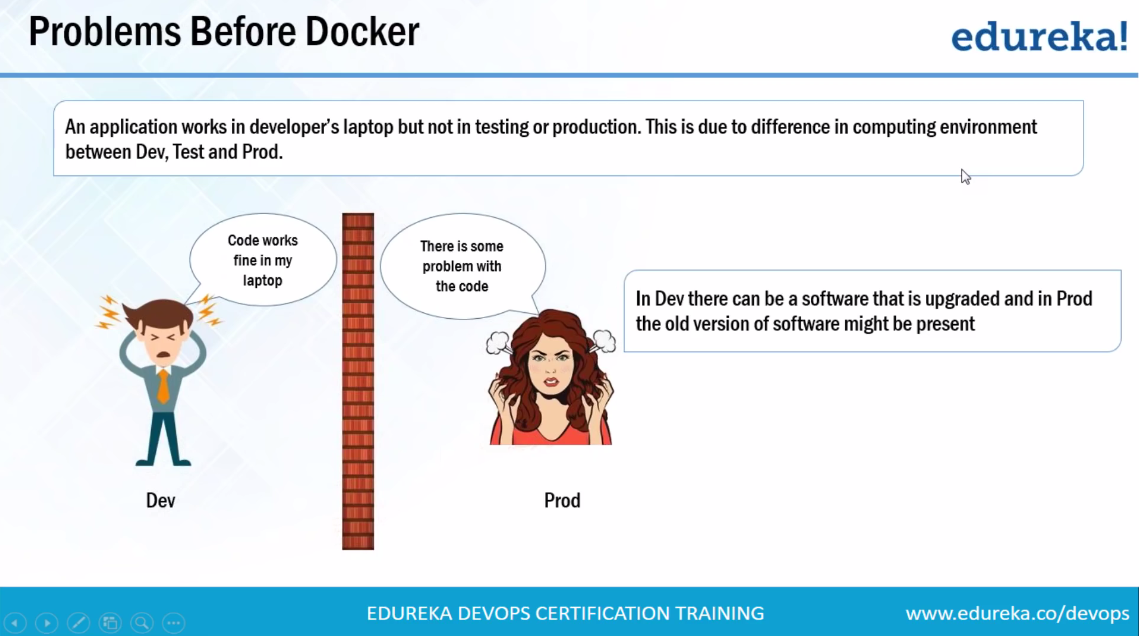
docker rm <container\_id>

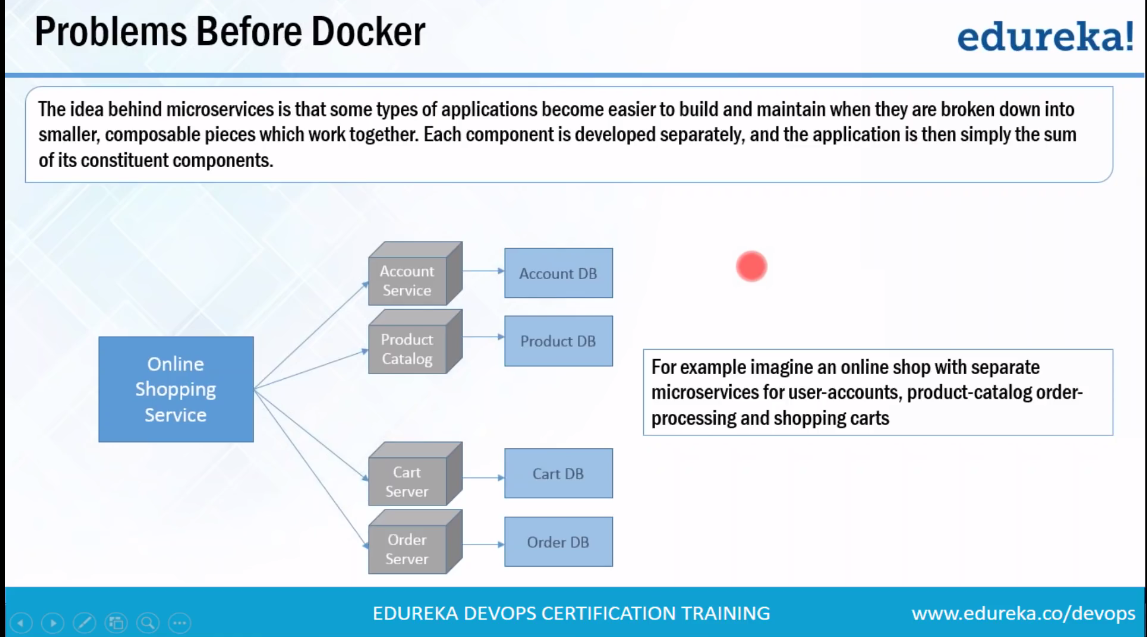
1. login to a docker container

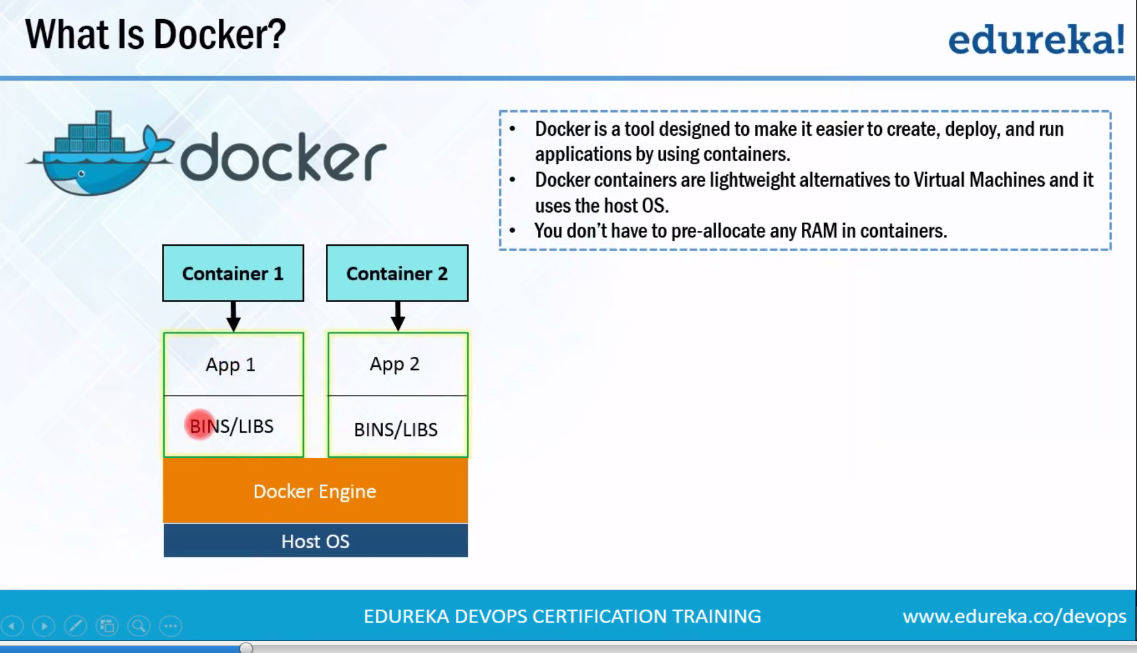
docker exec -it <container\_Name> /bin/bash

1. docker info
2. docker search <imagename>

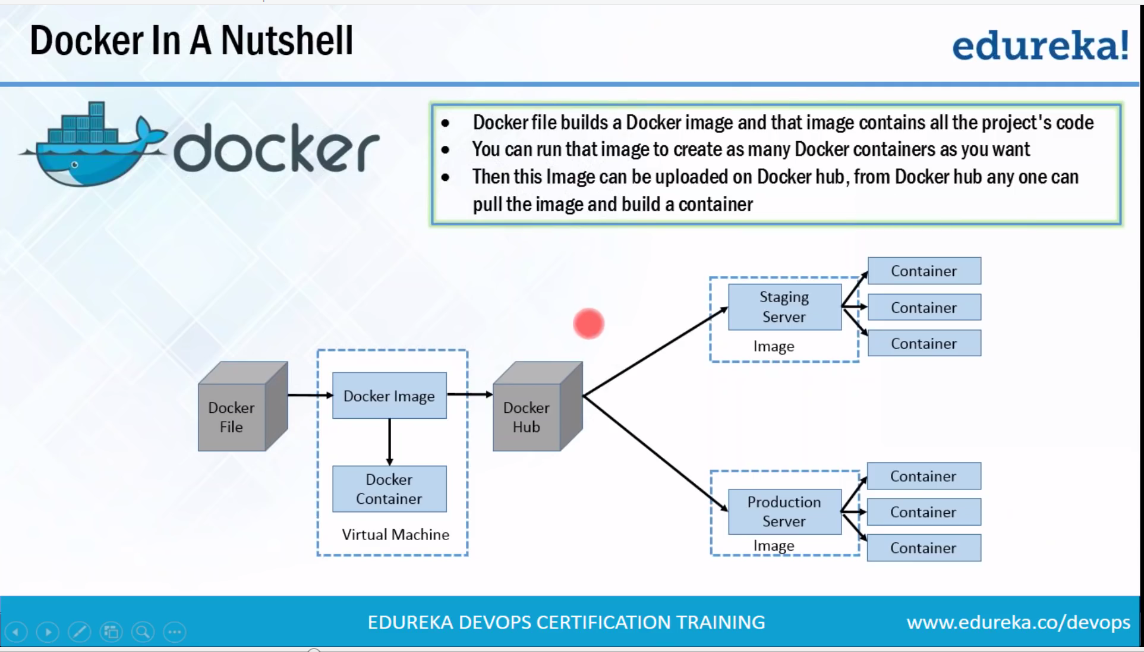
**Problems Before docker:**

****





**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

**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

**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.

**TIPS**

**How to scale services**

—scale

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

**example:**

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

**Docker Data Volumes:**

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

**Use of Volumes**

Decoupling container from storage

Share volume (storage/data) among different containers

Attach volume to container

On deleting container volume does not delete

NOTES

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

Non-Docker processes on the Docker host or a Docker container can modify them at any time

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 are managed by Docker and are isolated from the core functionality of the host machine

A given volume can be mounted into multiple containers simultaneously.

When no running container is using a volume, the volume is still available to Docker and is not removed automatically. You can remove unused volumes using docker volume prune.

When you mount a volume, it may be named or anonymous.

Anonymous volumes are not given an explicit name when they are first mounted into a container

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**

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

**Fiegn Client**

**What Is a Feign Client?**

Netflix provides Feign as an abstraction over REST-based calls, by which microservices can communicate with each other, but developers don't have to bother about REST internal details.

Feign, as a client, is an important tool for microservice developers to communicate with other microservices via Rest API.

Feign uses interfaces annotated with @FeignClient to generate API requests and map responses to Java classes. Let's first compile a list of API calls:

Action HTTP Method URL

(C) Create a user POST https://jsonplaceholder.typicode.com/users

(R) Get a list of users GET https://jsonplaceholder.typicode.com/users

(R) Get one user GET https://jsonplaceholder.typicode.com/users/{id}

(U) Update user details PUT or PATCH https://jsonplaceholder.typicode.com/users/{id}

(D) Delete a user DELETE [https://jsonplaceholder.typicode.com/users/{id}](https://jsonplaceholder.typicode.com/users/%7bid%7d)

|  |
| --- |
| @FeignClient(name="UserService" )//Service Id of EmployeeSerach service  public interface UserClient {  @RequestMapping(method = RequestMethod.POST, value = "/users")  User createUser(User user);    @RequestMapping(method = RequestMethod.GET, value = "/users")  List<user> getUsers();    @RequestMapping(method = RequestMethod.GET, value = "/users/{userId}")  User getUser(@PathVariable("userId") Long userId);    @RequestMapping(method = RequestMethod.PUT, value = "/users/{userId}")  User updateUser(@PathVariable("userId") Long userId, User user);    @RequestMapping(method = RequestMethod.DELETE, value = "/users/{userId}")  void deleteUser(@PathVariable("userId") Long userId);  } |

Feign dynamically generates the implementation of the interface we created, so Feign has to know which service to call beforehand. That's why we need to give a name for the interface, which is the {Service-Id} of UserService

**N.B.:** When using @PathVariable for Feign Client, always use value property or it will give you the errorjava.lang.IllegalStateException: PathVariable annotation was empty on param 0

Config server Advantages:

* Version controlled configuration
* Dynamic loading of config data

Docker Basic Commands Step by Step for Beginners Basic : docker version : docker -v : docker infod : docker --help : docker login ———————————— Images : docker images : docker pull : docker rmi ———————————— Containers : docker ps : docker run : docker start : docker stop ———————————— System : docker stats : docker system df : docker system prune

Now you can control this container using its name:

Get Logs: docker logs mynginx

Start Container: docker start mynginx

Stop Container: docker stop mynginx

Delete Container: docker rm mynginx

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:**

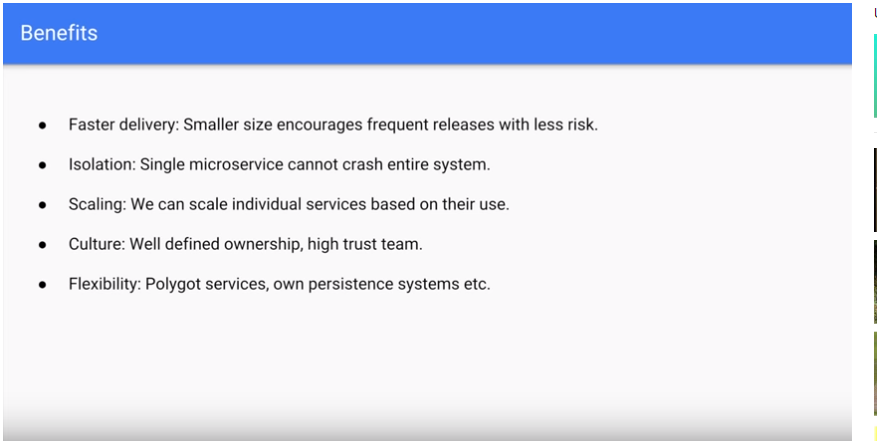
<https://www.callicoder.com/spring-boot-docker-example/>

<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.*