Docker is a software which provides centralized platform to execute your application. It wraps software components into a complete standardized unit which contains everything require to run. Whether it is code, runtime environment, tools or libraries. It guarantees that the software will always run as expected.

Docker Introduction

Docker is a software which provides centralized platform to execute your application. It wraps software components into a complete standardized unit which contains everything require to run. Whether it is code, runtime environment, tools or libraries. It guarantees that the software will always run as expected.

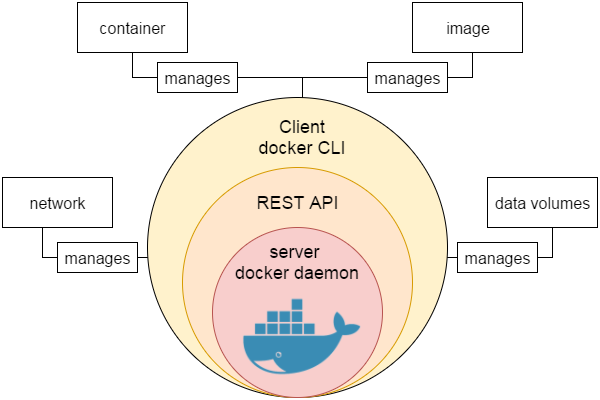
Docker provides the facility to run an application in a isolated environment which is called container. You can run many containers simultaneously on a given host. It is lightweight, so starts instantly and **uses less RAM**. It is **secure** by default because each container is isolated from one another.

Note - Docker container is not any specific platform. It can run on any computer, on any infrastructure and in any cloud.

Docker Engine

It is a client server application that contains the following major components.

* A server which is a type of long-running program called a daemon process.
* The REST API is used to specify interfaces that programs can use to talk to the daemon and instruct it what to do.
* A command line interface client.



**Fig**: Engine-component-flow

# Docker Features

Although Docker provides lots of features, we are listing some major features which are given below.

* Easy and Faster Configuration
* Increase productivity
* Application Isolation
* Swarm
* Routing Mesh
* Services
* Security Management

### **Easy and Faster Configuration**

This is a key feature of docker that helps us to configure the system easily and faster.

We can deploy our code in less time and effort. As Docker can be used in a wide variety of environments, the requirements of the infrastructure are no longer linked with the environment of the application.

### **Increase productivity**

By easing technical configuration and rapid deployment of application. No doubt it has increase productivity. Docker not only helps to execute the application in isolated environment but also it has reduced the resources.

### **Application Isolation**

It provides containers that are used to run applications in isolation environment. Each container is independent to another and allows us to execute any kind of application.

### **Swarm**

It is a clustering and scheduling tool for Docker containers. Swarm uses the Docker API as its front end, which helps us to use various tools to control it. It also helps us to control a cluster of Docker hosts as a single virtual host. It's a self-organizing group of engines that is used to enable pluggable backends.

### **Routing Mesh**

It routes the incoming requests for published ports on available nodes to an active container. This feature enables the connection even if there is no task is running on the node.

### **Services**

Services is a list of tasks that lets us specify the state of the container inside a cluster. Each task represents one instance of a container that should be running and Swarm schedules them across nodes.

### **Security Management**

It allows us to save secrets into the swarm itself and then choose to give services access to certain secrets.

It includes some important commands to the engine like secret inspect, secret create etc.

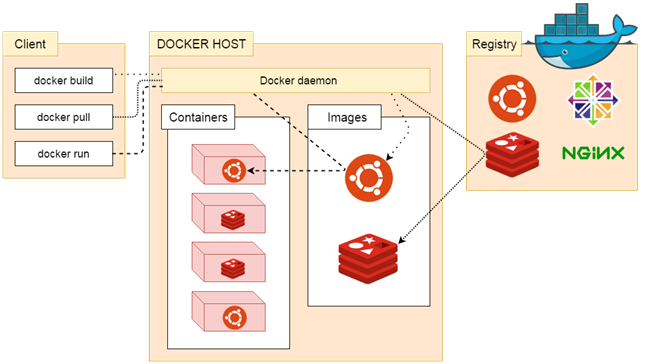
# Docker Architecture

Docker follows client-server architecture. Its architecture consists mainly three parts.

1) **Client:** Docker provides Command Line Interface (CLI) tools to client to interact with Docker daemon. Client can build, run and stop application. Client can also interact to Docker\_Host remotely.

2) **Docker\_Host:** It contains Containers, Images, and Docker daemon. It provides complete environment to execute and run your application.

3) **Registry:** It is global repository of images. You can access and use these images to run your application in Docker environment.



### **The Docker daemon**

It is a process which is used to listen for Docker API requests. It also manages Docker objects like: images, container, network etc. A daemon can also communicate with other daemons to manage Docker services.

### **The Docker client**

The Docker client is the primary way that many Docker users interact with Docker. When we use commands such as docker run, the client sends these commands to docker d, which carries them out. The docker command uses the Docker API.

### **Docker Registries**

Docker registry is used to store Docker images. Docker provides the Docker Hub and the Docker Cloud which are public registries that anyone can use. Docker is configured to look for images on Docker Hub by default.

When we use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

# Docker Dockerfile

A Dockerfile is a text document that contains commands that are used to assemble an image. We can use any command that call on the command line. Docker builds images automatically by reading the instructions from the Dockerfile.

The docker build command is used to build an image from the Dockerfile. You can use the -f flag with docker build to point to a Dockerfile anywhere in your file system.

1. $ docker build -f /path/to/a/Dockerfile .

## Dockerfile Instructions

The instructions are not case-sensitive but you must follow conventions which recommend to use uppercase.

Docker runs instructions of Dockerfile in top to bottom order. The first instruction must be **FROM** in order to specify the Base Image.

A statement begin with # treated as a comment. You can use RUN, CMD, FROM, EXPOSE, ENV etc instructions in your Dockerfile.

Here, we are listing some commonly used instructions.

### **FROM**

This instruction is used to set the Base Image for the subsequent instructions. A valid Dockerfile must have FROM as its first instruction.

Ex.

1. FROM ubuntu

### **LABEL**

We can add labels to an image to organize images of our project. We need to use LABEL instruction to set label for the image.

Ex.

1. LABEL vendorl = "JavaTpoint"

### **RUN**

This instruction is used to execute any command of the current image.

Ex.

1. RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'

### **CMD**

This is used to execute application by the image. We should use CMD always in the following form

1. CMD ["executable", "param1", "param2"?]

This is preferred way to use CMD. There can be only one CMD in a Dockerfile. If we use more than one CMD, only last one will execute.

### **COPY**

This instruction is used to copy new files or directories from source to the filesystem of the container at the destination.

Ex.

1. COPY abc/ /xyz

**Rules**

* The source path must be inside the context of the build. We cannot COPY ../something /something because the first step of a docker build is to send the context directory (and subdirectories) to the docker daemon.
* If source is a directory, the entire contents of the directory are copied including filesystem metadata.

### **WORKDIR**

The WORKDIR is used to set the working directory for any RUN, CMD and COPY instruction that follows it in the Dockerfile. If work directory does not exist, it will be created by default.

We can use WORKDIR multiple times in a Dockerfile.

Ex.

1. WORKDIR /var/www/html

# Docker Useful Commands

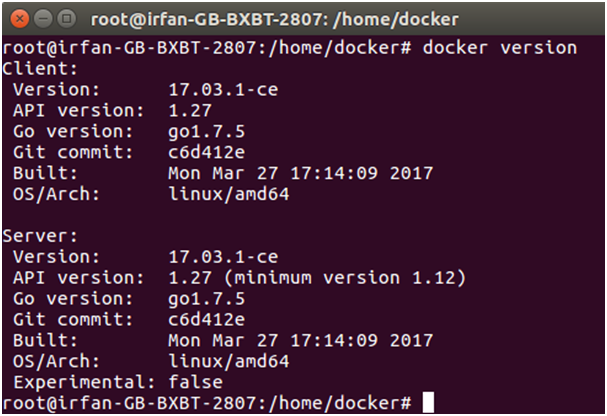
Docker is natively Linux based software so that it provides commands to interact and work in the client-server environment.

Here, we have listed some important and useful Docker commands.

### **Check Docker version**

1. $ docker version

It shows docker version for both client and server. As given in the following image.



### **Build Docker Image from a Dockerfile**

1. $ docker build -t image-name docker-file-location

**-t**: it is used to tag Docker image with the provided name.

### **Run Docker Image**

1. $ docker run -d image-name

**-d** : It is used to create a daemon process.

### **Check available Docker images**

1. $ docker images

### **Check for latest running container**

1. $ docker ps -l

**-l** : it is used to show latest available container.

### **Check all running containers**

1. $ docker ps -a

**-a** : It is used to show all available containers.

### Stop running container

1. $ docker stop container\_id

**container\_id** : It is an Id assigned by the Docker to the container.

### **Delete an image**

1. $ docker rmi image-name

### **Delete all images**

1. $ docker rmi $(docker images -q)

### **Delete all images forcefully**

1. $ docker rmi -r $(docker images -q)

**-r** : It is used to delete image forcefully.

### **Delete all containers**

1. $ docker rm $(docker ps -a -q)

### **Enter into Docker container**

1. $ docker exec -it container-id bash

**Step1 : Docker with console application:**

1. Create a folder docker-java-app

* Cretae Hello.java file

class Hello{

public static void main(String[] args){

System.out.println("This is java app \n by using Docker");

}

}

1. ->create Dockerfile file name should be only Dockerfile

FROM java:8

COPY . /var/www/java

WORKDIR /var/www/java

RUN javac Hello.java

CMD ["java", "Hello"]

1. Change folder to docker-java-app and run the following command

C:\docker-java-app> docker build -t java-app -f ./Dockerfile.txt .

1. To run the application

C:\docker-java-app>docker run java-app

**Step2 : Docker with SpringBoot application:**

HelloController:

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

package com.cts;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class HelloController {

@GetMapping("/hello")

public String hello(){

return "Hello from docker";

}

}

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

package com.cts;

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);

}

}

POM.xml

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

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.4.1.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.cts</groupId>

<artifactId>Spring-boot-docker</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>Spring-boot-docker</name>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

</project>

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

Dockerfile:

FROM openjdk:8

ADD target/Spring-boot-docker-0.0.1-SNAPSHOT.jar spring-boot-docker.jar

EXPOSE 8090

ENTRYPOINT ["java", "-jar", "spring-boot-docker.jar"]

Step1: =>Create Dockerfile in Root folder of the project

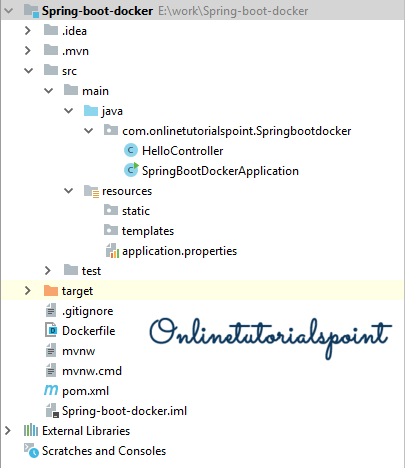
Step2:=> mvn clean install

ProjectFolder:\>docker build -f Dockerfile -t spring-boot-docker .

ProjectFolder:\>docker images

ProjectFolder:\> docker run -p 8090 spring-boot-docker

## Application Structure:



## Dependencies:

No separate dependencies required for docker.

pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

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

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.2.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.onlinetutorialspoint</groupId>

<artifactId>Spring-boot-docker</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>Spring-boot-docker</name>

<description>Demo project for Spring Boot Docker Deployment</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

<finalName>spring-boot-docker</finalName>

</build>

</project>

Creating a Simple Spring Boot Rest Controller.

HelloController.java

package com.onlinetutorialspoint.Springbootdocker;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class HelloController {

@GetMapping("/hello")

public String hello(){

return "Hello from docker";

}

}

Create Docker Configuration file.

Dockerfile

FROM openjdk:8

ADD target/spring-boot-docker.jar spring-boot-docker.jar

EXPOSE 8090

ENTRYPOINT ["java", "-jar", "spring-boot-docker.jar"]

**FROM:** is used to get the image from docker hub. Here we are getting the openjdk:8 version 8 from docker hub as our application is java based.

**ADD:** is used to adding the application jar (/target/spring-boot-docker.jar) file to Docker container.

**EXPOSE:** is used to define the port on which the container should expose.

**ENTRYPOINT:** is used by the docker to start the application.

Spring Boot Main class.

SpringBootDockerApplication.java

package com.onlinetutorialspoint.Springbootdocker;

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);

}

}

Build the Application using maven: It will generate spring-boot-docker.jar file and placed on /target folder.

Terminal

E:\work\Spring-boot-docker>mvn clean install

[INFO] Scanning for projects...

[INFO]

[INFO] ------------< com.onlinetutorialspoint:Spring-boot-docker >-------------

[INFO] Building Spring-boot-docker 0.0.1-SNAPSHOT

[INFO] --------------------------------[ jar ]---------------------------------

[INFO]

[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ Spring-boot-docker ---

[INFO] Deleting E:\work\Spring-boot-docker\target

[INFO]

[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ Spring-boot-docker ---

[INFO] Using 'UTF-8' encoding to copy filtered resources.

[INFO] Copying 1 resource

[INFO] Copying 0 resource

[INFO]

[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ Spring-boot-docker ---

[INFO] Changes detected - recompiling the module!

[INFO] Compiling 2 source files to E:\work\Spring-boot-docker\target\classes

[INFO]

[INFO] --- maven-resources-plugin:3.1.0:testResources (default-testResources) @ Spring-boot-docker ---

[INFO] Using 'UTF-8' encoding to copy filtered resources.

[INFO] skip non existing resourceDirectory E:\work\Spring-boot-docker\src\test\resources

[INFO]

[INFO] --- maven-compiler-plugin:3.8.0:testCompile (default-testCompile) @ Spring-boot-docker ---

[INFO] Nothing to compile - all classes are up to date

[INFO]

[INFO] --- maven-surefire-plugin:2.22.1:test (default-test) @ Spring-boot-docker ---

[INFO] No tests to run.

[INFO]

[INFO] --- maven-jar-plugin:3.1.1:jar (default-jar) @ Spring-boot-docker ---

[INFO] Building jar: E:\work\Spring-boot-docker\target\spring-boot-docker.jar

## Creating Docker Image :

**Step 1:** Build the application using Docker. Go to the terminal and execute the below docker commands.

docker build -f Dockerfile -t spring-boot-docker

E:\work\Spring-boot-docker>docker build -f Dockerfile -t spring-boot-docker .

Using the docker build command we are going to executing the -f Dockerfile (which we defined as part of docker file configuration) and creating tag with the name of -t spring-boot-docker, and the (.) dot tells where the Dockerfile exist, for our case the Dockerfile available in current working directory so that given as dot.

Terminal

Sending build context to Docker daemon 16.89MB

Step 1/4 : FROM openjdk:8

8: Pulling from library/openjdk

ab1fc7e4bf91: Pull complete

35fba333ff52: Pull complete

f0cb1fa13079: Pull complete

3d1dd648b5ad: Pull complete

a9f886e483d6: Pull complete

4346341d3c49: Pull complete

006f2208d67a: Pull complete

fb85cf26717d: Pull complete

Digest: sha256:6d881fb5c8dbc6ad1e9392ce35e289afb53b3148450848fb8f6aabc5d106720f

Status: Downloaded newer image for openjdk:8

---> 5f4603da3fbc

Step 2/4 : ADD target/spring-boot-docker.jar spring-boot-docker.jar

---> 3c8f17d9a553

Step 3/4 : EXPOSE 8090

---> Running in f2ac5b50dce6

Removing intermediate container f2ac5b50dce6

---> c797a30a9883

Step 4/4 : ENTRYPOINT ["java", "-jar", "spring-boot-docker.jar"]

---> Running in 2add22058b5d

Removing intermediate container 2add22058b5d

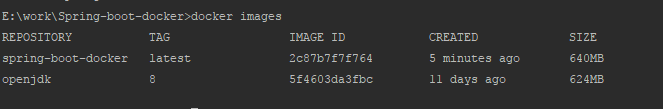
---> 2c87b7f7f764

Successfully built 2c87b7f7f764

Successfully tagged spring-boot-docker:latest

We can observe the above logs containing 4 different steps (which we defined in Dockerfile) while executing docker build command.

**Step 2:**  After successful completion of step 1 we can see the docker image by hitting the docker images command.



We can see our latest spring-boot-docker image in the list.

**Step 3:** Run the docker image on 8090 port using the below command.

Terminal: docker run -p 8090:8090 spring-boot-docker

E:\work\Spring-boot-docker>docker run -p 8090:8090 spring-boot-docker

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v2.1.2.RELEASE)

2019-02-03 05:39:53.679 INFO 1 --- [ main] c.o.S.SpringBootDockerApplication : Starting SpringBootDockerApplication v0.0.1-SNAPSHOT on d26c4095ca5a with PID 1 (/spr

ing-boot-docker.jar started by root in /)

2019-02-03 05:39:53.696 INFO 1 --- [ main] c.o.S.SpringBootDockerApplication : No active profile set, falling back to default profiles: default

2019-02-03 05:39:57.202 INFO 1 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8090 (http)

2019-02-03 05:39:57.296 INFO 1 --- [ main] o.apache.catalina.core.StandardService : Starting service [Tomcat]

2019-02-03 05:39:57.308 INFO 1 --- [ main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.14]

2019-02-03 05:39:57.338 INFO 1 --- [ main] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal performance in

2019-02-03 05:39:57.565 INFO 1 --- [ main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext

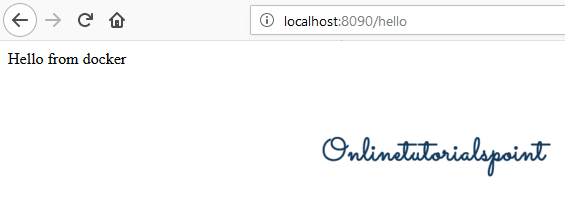
2019-02-03 05:39:57.566 INFO 1 --- [ main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 3741 ms

2019-02-03 05:39:58.271 INFO 1 --- [ main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'

2019-02-03 05:39:58.878 INFO 1 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8090 (http) with context path ''

2019-02-03 05:39:58.886 INFO 1 --- [ main] c.o.S.SpringBootDockerApplication : Started SpringBootDockerApplication in 6.606 seconds (JVM running for 7.754)

Access the application:



**Step3: To move docker images to Dockerhub**

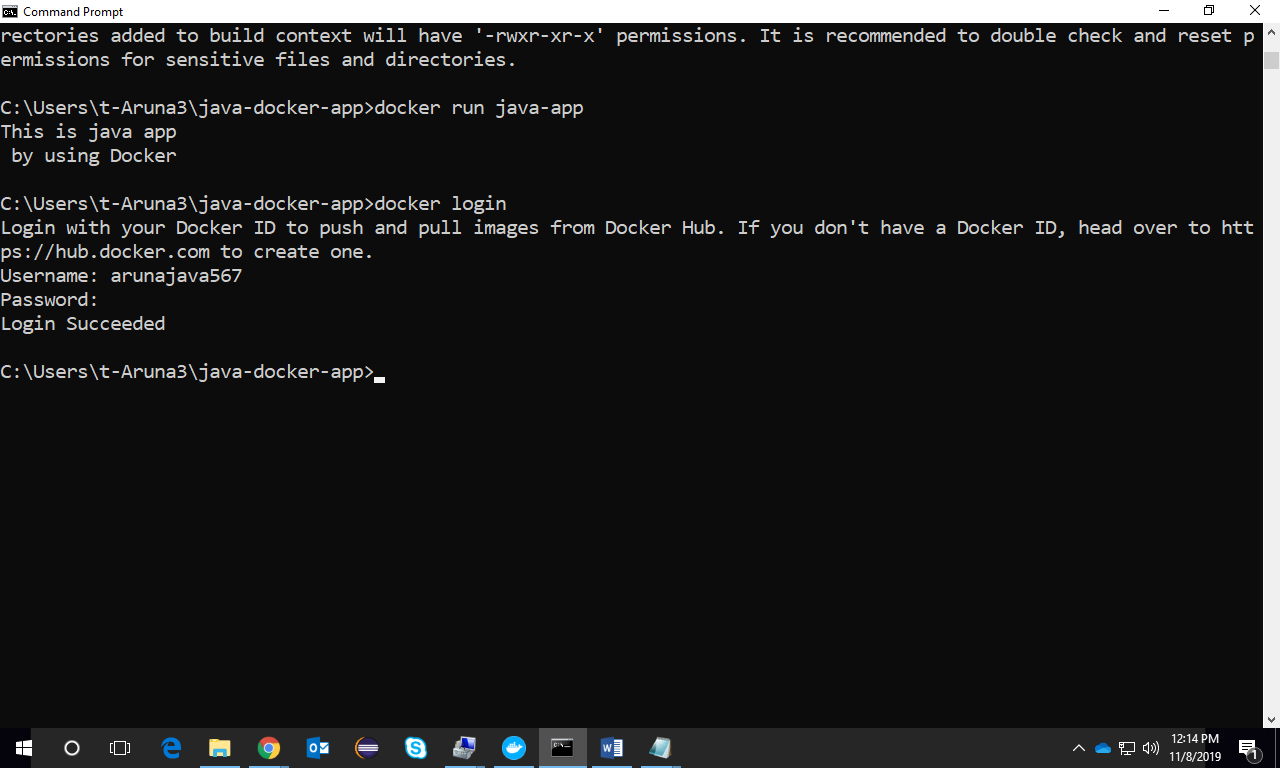
To move docker image to remote repository :

<https://hub.docker.com/>

create a user with credentials id,pwd to access global repository

c:\docker-java-app> docker login

give id and pwd details



C:\docker-java-app>docker tag java-app arunajava567/java-app

C:\docker-java-app>docker push  arunajava567/java-app

Goto <https://hub.docker.com/repositories>

