**HOMEWORK 2**

**Akshay Mishra 011476673**

Q1.

a. List technologies, software (including version), and platform (include version) for the REST client, REST server, and NoSQL.

**Answer:**

**Docker:**

Client:

 Version:      17.09.0-ce

 API version:  1.32

 Go version:   go1.8.3

 Git commit:   afdb6d4

 Built:        Tue Sep 26 22:41:23 2017

 OS/Arch:      linux/amd64

Server:

 Version:      17.09.0-ce

 API version:  1.32 (minimum version 1.12)

 Go version:   go1.8.3

 Git commit:   afdb6d4

 Built:        Tue Sep 26 22:42:49 2017

 OS/Arch:      linux/amd64

 Experimental: false

on

Linux centos7desktop 3.10.0-123.el7.x86\_64 #1 SMP Mon Jun 30 12:09:22 UTC 2014 x86\_64 x86\_64 x86\_64 GNU/Linux

**Docker-compose version (Extra Credit):**

docker-compose version 1.16.1, build 6d1ac21

docker-py version: 2.5.1

CPython version: 2.7.13

OpenSSL version: OpenSSL 1.0.1t  3 May 2016

on

Linux centos7desktop 3.10.0-123.el7.x86\_64 #1 SMP Mon Jun 30 12:09:22 UTC 2014 x86\_64 x86\_64 x86\_64 GNU/Linux

**REST Client:**

curl 7.54.0 (x86\_64-apple-darwin16.0) libcurl/7.54.0 SecureTransport zlib/1.2.8

on

MacOS Sierra 10.12.6

**And**

curl 7.29.0 (x86\_64-redhat-linux-gnu) libcurl/7.29.0 NSS/3.15.4 zlib/1.2.7 libidn/1.28 libssh2/1.4.3

on

Centos 7 Desktop v3.10

**REST Server:**

Embedded Tomcat 8.5 in Spring Boot Jar inside a docker image

on

Centos 7 Desktop v3.10 (Host 1) &

MacOS Sierra 10.12.6 (Host 2)

b. For host1 and host2, list their OS (and version), and IP address.

**Answer:**

**Host 1:**

**OS:** Centos

Linux centos7desktop 3.10.0-123.el7.x86\_64 #1 SMP Mon Jun 30 12:09:22 UTC 2014 x86\_64 x86\_64 x86\_64 GNU/Linux

**IP Address:**

**(IP Address as per dashboard SJSU VSphere)**

**130.65.159.120**

**Host 2:**

**OS:** OSX

MacOS Sierra 10.12.6

**IP Address:**

**When connected from Home Wi-fi to SJSU network via VPN**

**10.0.0.68**

c. A sample entire HTTP URL (including actual IP address of host1), URI, and request body for POST to create a new employee based on the XML format or JSON format (depending on your implementation). Also indicate if there is any additional setup (e.g., HTTP header, etc.).

**Answer:**

**entire URL:**

<http://130.65.159.120:8080/cmpe282Akshay673/rest/employee>

**URI**: POST /cmpe282Akshay673/rest/employee

**Request body:**

{

"id": 1,

"fname": "Akshay",

"lname": "Mishra"

}

no additional setup (no additional request header) is needed.

Q2. Describe detailed steps to **build** webapp and db docker containers with screenshots. (You can pull docker images directly, utilize Dockerfile to build one, or build on your own, or combination of the above.)

**Answer:**

**WebApp - My Own Docker Image:**

For the web app, I built my own image and deployed JAR file inside the container. The Jar runs an embedded tomcat hence one doesn’t need to install Tomcat or any Spring dependencies, any of these in order to run the webapp.

**Base Image:** **openjdk:8-jdk**

**Modifications:**

Added my cmpe282Akshay673.jar file to the home directory in order to use it to run the app.

et ENTRYPOINT as the command that calls the JAR file and passes appropriate parameters that override the original ones in the JAR in order to mainly locate MongoDB Host, Port and URL. I leveraged the networking features of Docker here while passing the parameter.

**Dockerfile for WebApp:**

FROM openjdk:8-jdk

MAINTAINER akshay.mishra@sjsu.edu

ADD cmpe282Akshay673.jar cmpe282Akshay673.jar

RUN sh -c 'touch /cmpe282Akshay673.jar'

ENTRYPOINT ["java", "-Dmongodb.host=dbAkshay673","-Dmongodb.db=cmpe282Akshay673","-Dmongodb.port=27017","-Djava.security.egd=file:/dev/./urandom","-jar","/cmpe282Akshay673.jar"]

**Explanation:**

FROM openjdk:8-jdk

This basically is just saying that we are inheriting the openjdk v8 as our base image on top of which we are building our own image.

MAINTAINER akshay.mishra@sjsu.edu

This is an optional tag and used to identify the owner of the docker image. People may use this in order to contact the owner in case they have any suggestions or issues.

ADD cmpe282Akshay673.jar cmpe282Akshay673.jar

This file basically adds the JAR to the container in its home directory. This used JAR will be called via java –jar command in order to run the web app.

RUN sh -c 'touch /cmpe282Akshay673.jar'

This command just runs ‘touch’ on the jar which only modifies the last accessed date of the JAR.

EXPOSE 8080

This command is basically exposing the port 8080. Then we can use it to map later with a port on the host. By default, 8080 is open and hence we won’t have any issues even if we omit this sentence. For a port that is not open by default, it must be exposed explicitly in the dockerfile in order to be able to map it with a host port.

ENTRYPOINT ["java", "-Dmongodb.host=dbAkshay673","-Dmongodb.db=cmpe282Akshay673","-Dmongodb.port=27017","-Djava.security.egd=file:/dev/./urandom","-jar","/cmpe282Akshay673.jar"]

This line is the most important line in the dockerfile. This basically sets the entrypoint of the container. It means that the container will run this command when it is up. This command in essence runs the JAR file using java –jar command and spins up the app.

**Docker image for the DB**

**Image:** official MongoDB Docker Image (v3.15) from hub.docker.com

**Modifications:** No modifications made to the original image, only performed volume mapping on the container.

**Steps to build WebApp and MongoDB Containers:**

1. **Spin up MongoDB Container using below command:**

docker run -d -v data:/data/db --name dbAkshay673 mongo

This will pull/use existing (if previously pulled) official image of mongodb and create a container with the name as dbAkshay673

Here, we are also mapping /data/db of the container to data directory on the host. This is called as volume mapping.

1. **Build WebApp Image using below command:**

docker build --no-cache=true -t app\_akshay\_673 .

Run this from the directory where Dockerfile for the WebApp Image is stored.

This should build an image with the name as app\_akshay\_673.

Once image is built, then the webapp container can be created using below command:

docker run -it --name appAkshay673 --link dbAkshay673:dbAkshay673 -p 8080:8080 app\_akshay\_673

This should create a container named appAkshay673 and automatically map port 8080 of the container to the port 8080 of the host.

**Known Issues:** None

**Additional features You are proud of:**

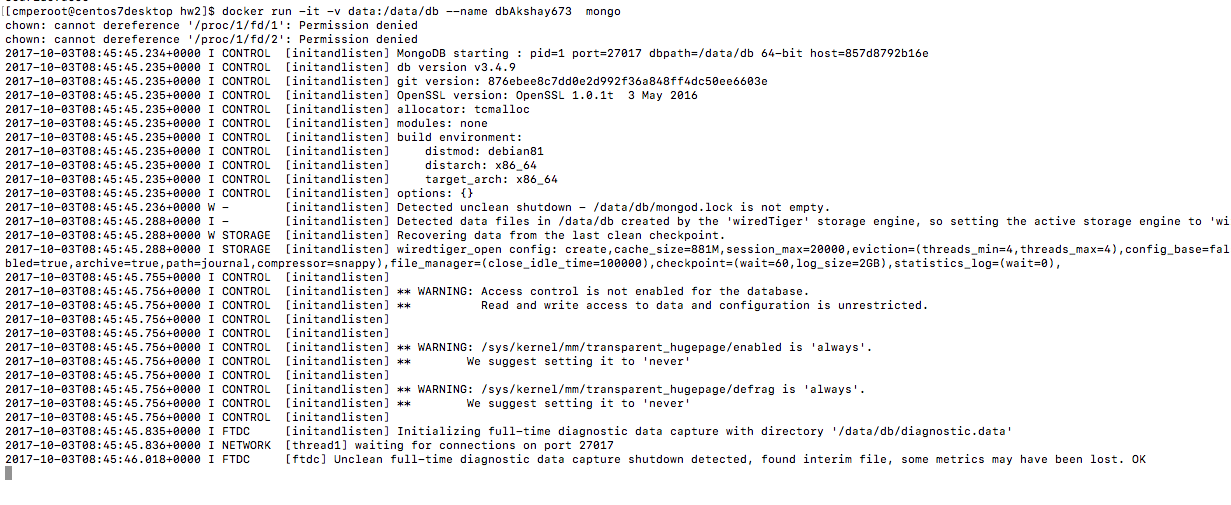
1. I am using link feature of Docker, which helps to naturally induce environment variables in the target container from the source container. These env variables are exposed from the source container. These can be used to make these two containers communicate with each other using the default bridge network.
2. I am using volume mapping on the MongoDB container to persist the data on the local host (outside the container). Even in case of calamity, if the MongoDB container stops, it can be restarted from the state where it had stopped using this persisted data on the host. Hence the system is more resilient now.
3. I am overriding the application properties in the JAR command and simply passing some environment variables that were induced with the help of ‘link’ tag. No configuration is hardcoded and hence system is more easy to configure and use.

Q3. Describe detailed steps to **deploy** webapp and db docker containers with screenshots.

**Answer:**

Docker command to deploy MongoDB container:

docker run -d -v data:/data/db --name dbAkshay673 mongo

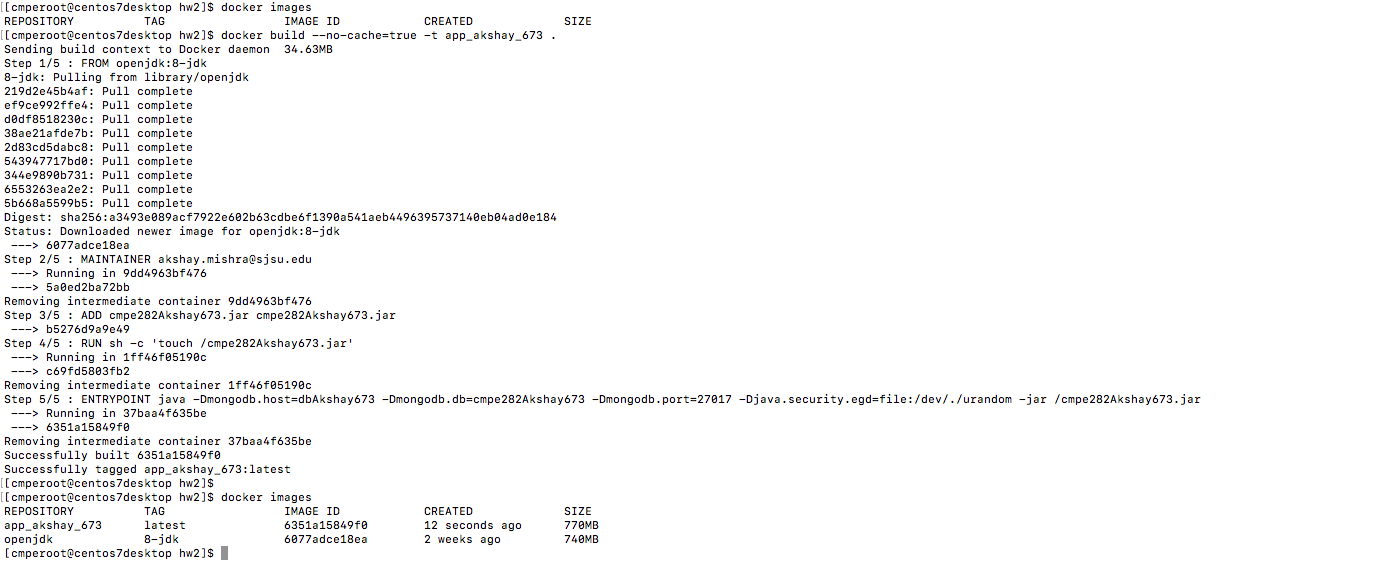
****

docker ps



**Docker command to build Web App:**

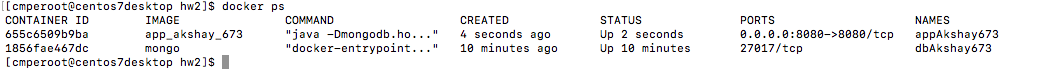
docker build --no-cache=true -t app\_akshay\_673 .

****

docker run -it --name appAkshay673 --link dbAkshay673:dbAkshay673 -p 8080:8080 app\_akshay\_673



docker ps



**Steps to deploy Web Server to Containers:**

It is not required to deploy web server separately to containers and then deploy war file, web app is completely packaged as a JAR file and it has an embedded tomcat web server already. When we build docker image, it happens automatically through the jar in which tomcat is embedded.

**Modifications in the REST Server Backend Code**

Not needed, because we have multiple externally configurable properties in the backend code already.

**Known issues**: None

**Additional features you are proud of:**

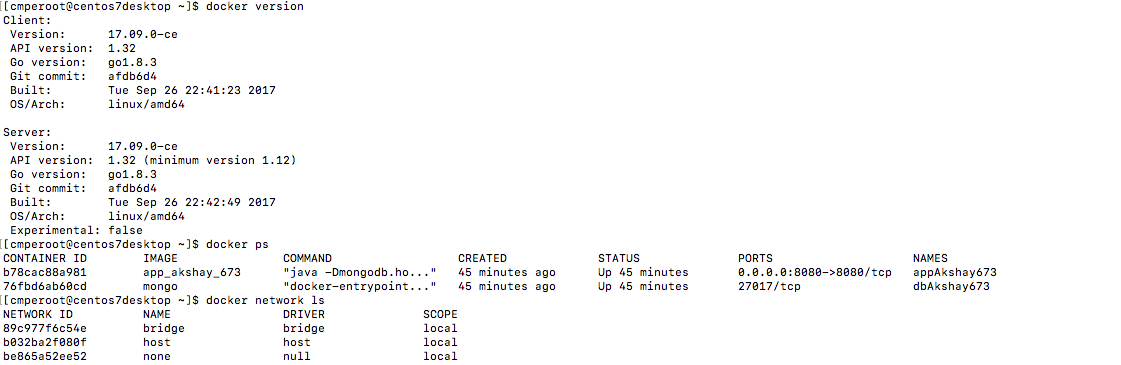
We can use default application properties or simply the ones we obtain by linking containers easily. The backend code has been designed in a generic way to achieve this flexibility and application of Docker leverages this to its best use here.

Q4. While both containers are running on host1, include the screenshots of the followings on host1

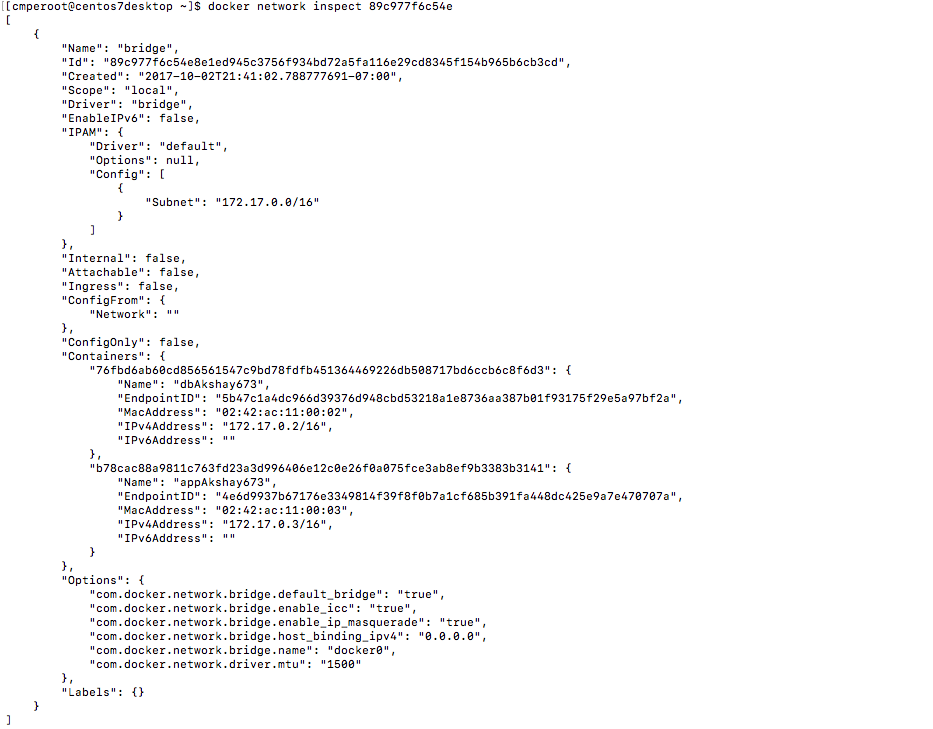
**Answer:**

**Host 1:**

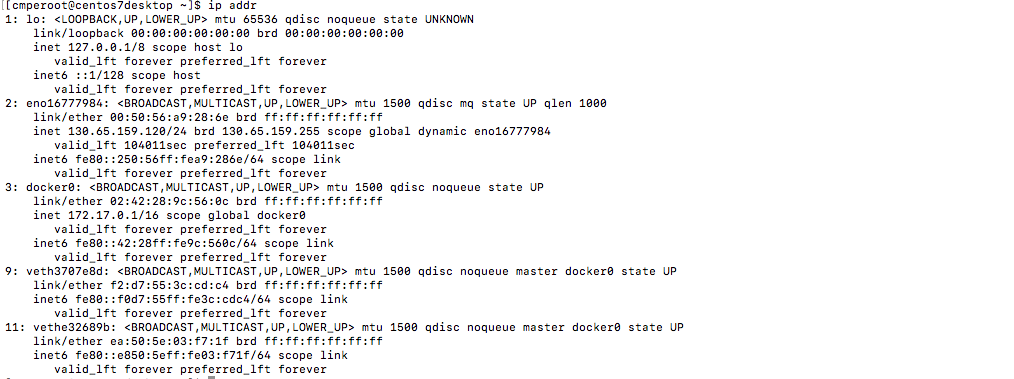
docker version & docker ps



docker network inspect



ip addr

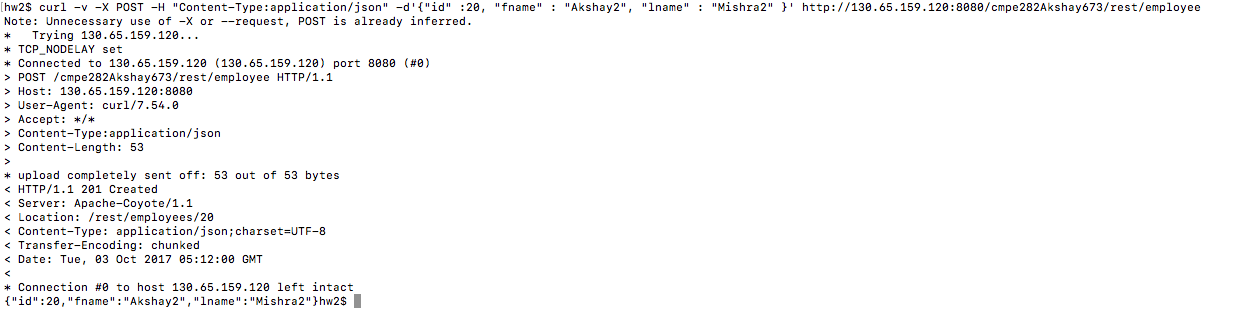


Q5. On host2, use REST client to issue the following requests and include screenshots of REST request and response (method, URL, HTTP headers) - success cases only:

**Answer:**

POST for Employee 10, 20

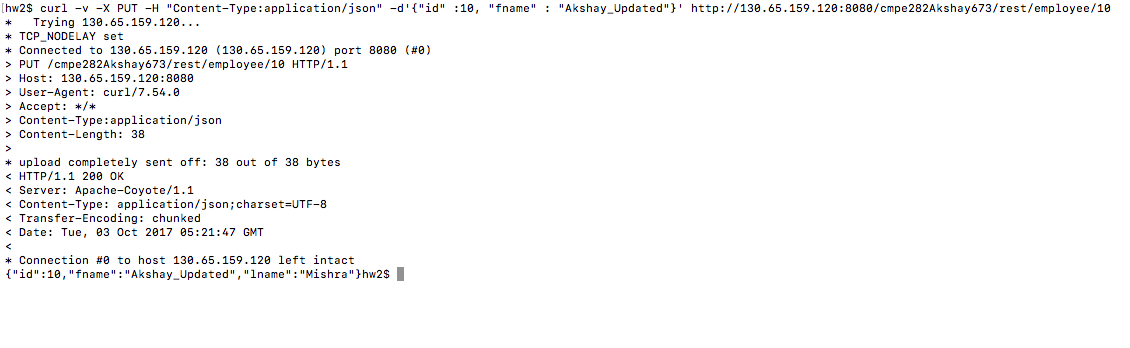




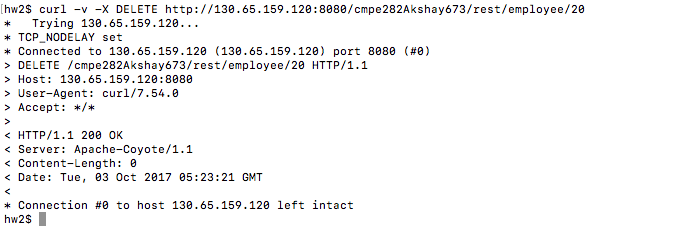
GET All Employees



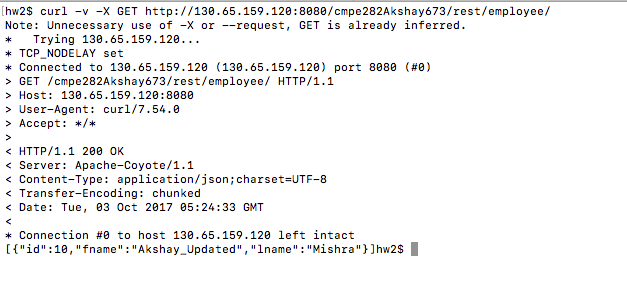
PUT Employee 10 only fname



DELETE Employee 20



GET All Employees



**EXTRA CREDIT**

Q6. In addition to the original homework, use docker compose to build and deploy containers in Q2 and Q3.

**Answer:**

**Docker-compose File:**

version: '2'

services:

app:

build: .

container\_name: appAkshay673

ports:

- 8080:8080

depends\_on:

- dbAkshay673

links:

- dbAkshay673:dbAkshay673

restart: always

dbAkshay673:

image: mongo

container\_name: dbAkshay673

volumes:

  data:

    driver: local

**Steps to use docker-compose:**

This docker-compose file defines two services namely ‘app’ and ‘dbAkshay673’ with their right configurations. Here we are linking two containers using a default bridge network.

We are specifying dockerfile loication using build option for Web App and image name for MongoDB container.

In order to run the setup using docker-compose, simply run below command from the directory where both the Web App Dockerfile as well as above docker-compose file is stored.

Docker-compose up -d --build

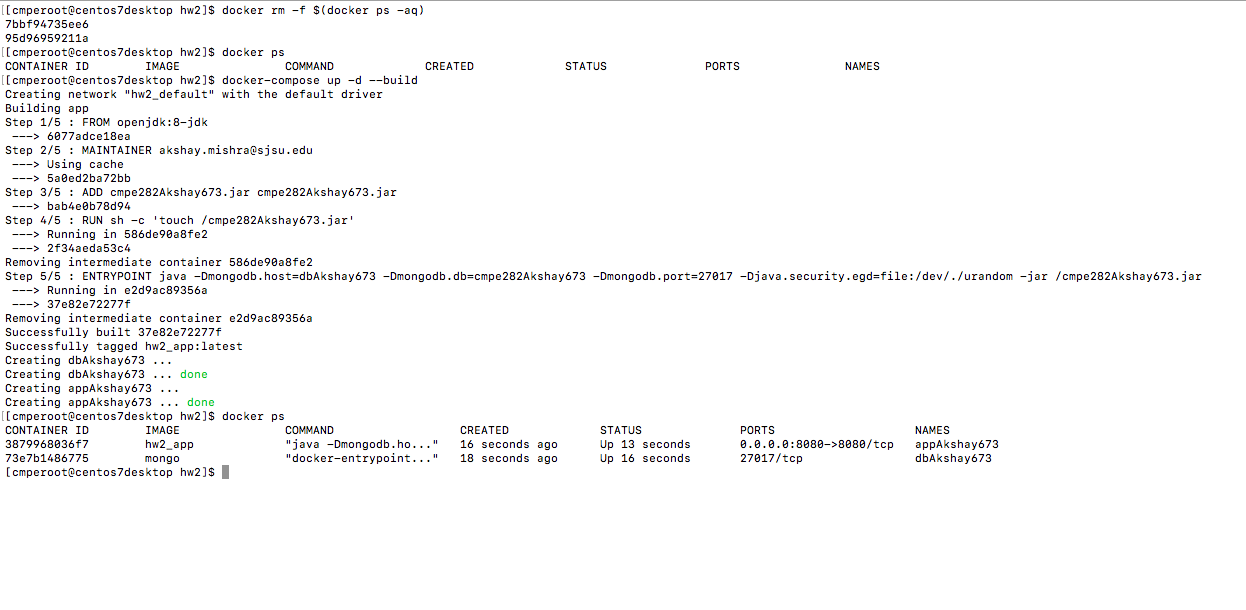
This will basically spin up both the containers with given configurations such as container names, links, port mapping and volume mapping.

**Scope for** **more automation**

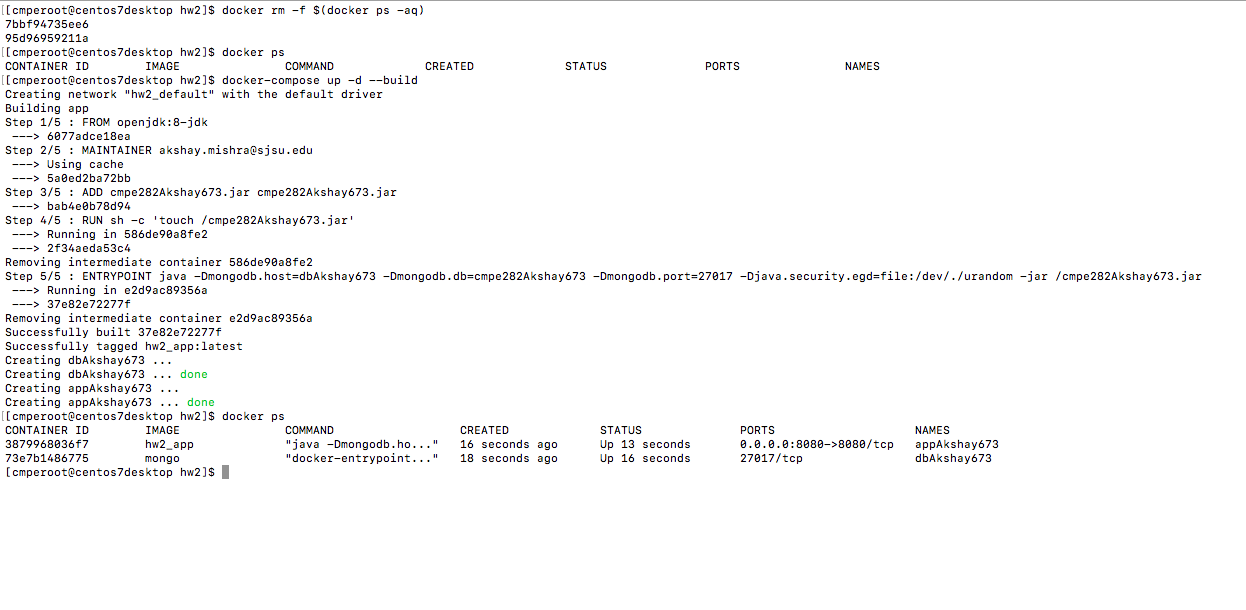
There is no need to copy any war file manually, the process is already fully automated. At most what one can do is map the volume in which JAR file is kept to a directory inside the container. That way the JAR won’t be needed to copy in the docker-file, and it will be mapped directly from the host file. The latest jar can be updated and container can pick it up from there.

**Screenshots:**

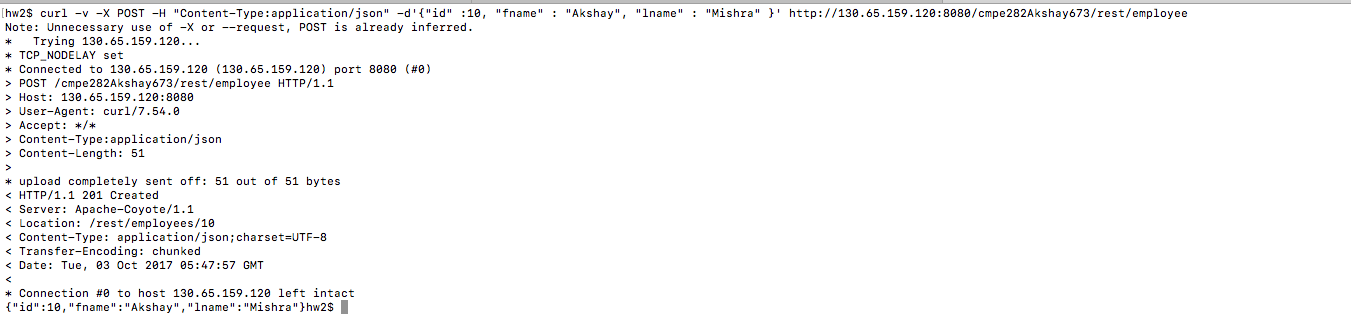
Docker compose command to spin up both the containers:

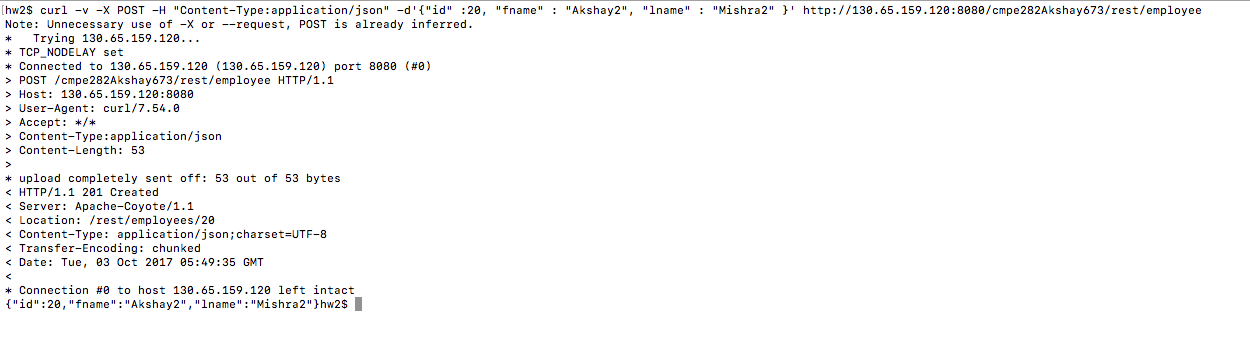


docker ps



First two cases with docker-compose:





Get All Employees

