DevOps Tools Day18

Prakash Ramamurthy

prakash.ramamurthy@wipro.com

- Dockerfile
- Building Dockerized Application
- Handson-Demonstration

Dockerfile

Dockerfile

Concept and Features

- Text file filled with commands used to build Docker image
- Used to build a Docker image automatically
- Allows execution of several command-line instructions in succession
- Helps documentation of how Docker image is built
- Builds a good practice of maintaining Docker images
- Dockerfile can be stored in source code repositories
- Helps avoid storage for bulky binary images
- Easy to trust as the image is built spontaneously with its content being known
- Avoids the need to pull bulky binary images from the binary repositories
- Takes time to build the image first time

Dockerfile

Usage and Format

- Docker command “docker build” is used to build the images
- Dockerfile may be located anywhere and using flag “–f” the path need to be specified.
- Dockerfile may use the current folder for external files “.” or folder path may be specified
- Example:

```
$ docker build -f ./newdockbuild/dockbuilder -t prakaram/curlubuntu
./newdockbuild/
```

- Dockerfile uses the format of “Instruction” followed by “Arguments”
- Instructions are few reserved keywords defined in the reference file
- Arguments are mostly Linux commands used in relation with Instructions

```
# Comment

INSTRUCTION arguments
```

Dockerfile

Instructions: FROM, RUN, CMD

- Instruction FROM initializes a new build stage and sets the “Base Image” for subsequent instructions
- Valid Dockerfile must start with a FROM instruction.

```
Example:      FROM java:8
```

- Instruction RUN execute any command in a new layer on top of the current image and commit the results.
- Resulting committed image will be used for the next step in the Dockerfile

```
RUN javac HelloWorld.java
```

- There can be only one CMD instruction in a Dockerfile. If there are more than one, then only the last CMD will take effect.
- Instruction CMD provide defaults for an executing container
- Defaults can include an executable or command

```
CMD java -jar HelloWorld.jar
```

Dockerfile

Instructions: ENV, WORKDIR

- Instruction ENV sets the environment variable <key> to the value <value>

```
Example:      ENV <key> <value>
ENV PATH $JAVA_HOME/bin:$PATH
```

- Instruction WORKDIR sets the working directory for any of the following instructions and for their operations
- If the WORKDIR doesn’t exist, it will be created, even if its not used subsequently

```
Example: WORKDIR /opt/
```

Dockerfile

Instructions: ADD, COPY

- Instruction ADD allows copying of files, directories or even URLs of remote files with path specified as <src> and would add them to filesystem of image to be build at the path specified by <dest>

```
ADD <src> <dest>
```

```
ADD jdk-8u162-linux-x64.tar.gz /opt/
```

- Instruction COPY also allows copying of files and directories with path specified at <src> and and would add them to filesystem of image to be build at the path specified by <dest>

```
COPY <src> <dest>
```

```
COPY . /appjava
```

- ADD allow use of URL in <src>, and if <src> is specifying a tar file, then ADD will also unarchive the tar file.

Building Dockerized Application

Dockerized Application

Dockerizing

- Choose a base docker image and use this with FROM keyword
- Create Working directory using WORKDIR keyword
- Create Environment variables if required using ENV keyworld
- Add required files from host system work folder using ADD or COPY keyword
- Use the Linux command to be used for software installation with RUN keyword
- Check whether the software installed expose any service on particular port. EXPOSE such ports.
- Finally set a command to be executed whenever the container is built using this image with CMD

Hands-On Demonstration

Sensitivity: Internal & Restricted

Hands-On Demonstration

Working on Docker Whale

- Launch Docker Whale command displaying a message

```
$ docker container run docker/whalesay cowsay "An Important Message"
```

- Modify the above activity adding command within the image

```
$ cat Dockerfile
FROM docker/whalesay:latest
CMD echo "An Important Message" | cowsay
```

- Build the new image

```
$ docker build -t whale001 .
```

- Create the container from new image

```
$ docker container run whale001
```

Hands-On Demonstration

Working on Java

- Hello World Java application:

```
$ cat HelloWorld.java
public class HelloWorld {

  public static void main(String[] args){

    System.out.println("Hello World :) ");

  }
}
```

- Compiling Java Application

```
$ javac HelloWorld.java
```

- Running Java Application

```
$ java HelloWorld
Hello World :)
```

Hands-On Demonstration

Working on Java

- Check content for files created

```
$ ls
HelloWorld.class  HelloWorld.java
```

- Create manifest file

```
$ cat manifest.txt
Manifest-Version: 1.0
Created-By: training
Main-Class: HelloWorld
```

- Create jar file from “hello world” application

```
$ jar cfm HelloWorld.jar manifest.txt HelloWorld.class
$ ls

HelloWorld.class  HelloWorld.jar  HelloWorld.java  manifest.txt
```

- Run the jar file

```
$ java -jar HelloWorld.jar
Hello World :)
```

Hands-On Demonstration

Containerizing Java application

- Create Dockerfile to containerize this JAVA “Hello World” application

```
$ cat Dockerfile
FROM java:8
CMD java -jar HelloWorld.jar
```

- Build the image and run the container

```
$ docker build -t hellojava .
```

- Run the container

```
$ docker container run -it hellojava
ERROR
```

Note: Though the image is built it fails ro create container as the HelloWorld.jar file was not available in the image, as it is not being added.

Hands-On Demonstration

Containerizing Java application

- Rectify the Dockerfile by adding jar file

```
$ cat Dockerfile
FROM java:8

ADD HelloWorld.jar HelloWorld.jar

CMD java -jar HelloWorld.jar
```

- Build the docker image. Now the ready jar file is added to image.

```
$ docker build -t hellojava1 .
```

- Run the container to get the output

```
$ docker container run hellojava1
Hello World :)
```

- Works successfully

Hands-On Demonstration

Containerizing Java application

- Create another with interactive access to check the content

```
$ docker container run -it hellojava1 /bin/bash
# pwd
/
# ls
HelloWorld.jar   boot  etc   lib    media  opt     root  sbin  sys  usr
Bin   dev   home  lib64  mnt        proc           run   srv   tmp  var

Note: Check the availability of HelloWorld.jar file in '/' folder
```

Hands-On Demonstration

Containerizing Java application

- Modify the Dockerfile to specify a working directory for application instead of '/' directory and adding HelloWorld.jar file to Work Directory.

```
$ cat Dockerfile
FROM java:8
WORKDIR /appjava
ADD HelloWorld.jar HelloWorld.jar
CMD java -jar HelloWorld.jar
```

- Build the new docker image hellojava2

```
$ docker build -t hellojava2 .
```

- Create new container and get output

```
$ docker container run hellojava2
Hello World :)
```

Hands-On Demonstration

Containerizing Java application

- Create container with interactive access to check the content

```
$ docker container run -it hellojava2 /bin/bash
# pwd
/appjava
# ls
HelloWorld.jar
```

- Interactive access show availability of HelloWorld.jar file under /appjava work directory

Hands-On Demonstration

Containerizing Java application

- Rewrite the Dockerfile to add the HelloWorld.java and manifest.txt

```
$ cat Dockerfile
FROM java:8
WORKDIR /appjava
COPY . /appjava
```

- Build the image

```
$ docker build -t hellojava3 .
```

- Create container and check content for the required files for java application

```
$ docker container run -it hellojava3 /bin/bash
# ls
Dockerfile HelloWorld.java manifest.txt
```

Hands-On Demonstration

Containerizing Java application

- Add “javac” java compiler command to Dockerfile to generate “HelloWorld.class” file.

```
$ cat Dockerfile
FROM java:8
WORKDIR /appjava
COPY . /appjava
RUN javac HelloWorld.java
```

- Build the image and create container to check for the content.

```
$ docker build -t hellojava4 .

$ docker container run -it hellojava4 /bin/bash
# ls
Dockerfile  HelloWorld.class  HelloWorld.java  df  manifest.txt
```

Hands-On Demonstration

Containerizing Java application

- Rewrite the Dockerfile to add the command to generate HelloWorld.jar file.

```
$ cat Dockerfile
FROM java:8
WORKDIR /appjava
COPY . /appjava
RUN javac HelloWorld.java
RUN jar cfm HelloWorld.jar manifest.txt HelloWorld.class
```

- Build the image
- `$ docker build -t hellojava5 .`

- Create container with interactive access the check creation of jar file

```
$ docker container run -it hellojava5 /bin/bash
# ls
Dockerfile  HelloWorld.class  HelloWorld.jar  HelloWorld.java  df  manifest.txt
```

Hands-On Demonstration

Containerizing Java application

- Add CMD to execute the jar file to provide the output

```
$ cat Dockerfile
FROM java:8
WORKDIR /appjava
COPY . /appjava
RUN javac HelloWorld.java
RUN jar cfm HelloWorld.jar manifest.txt HelloWorld.class
CMD java -jar HelloWorld.jar
```

- Build the image

```
$ docker build -t hellojava6 .
```

- Run the container to get the result

```
$ docker container run hellojava6
Hello World :)
```

Hands-On Demonstration

Python Application

```
$ cat app.py
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Hands-On Demonstration

Containerizing Python Application

```
$ cat requirements.txt
Flask
Redis
$ cat Dockerfile
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Hands-On Demonstration

Containerizing Python Application

- Build docker image with python application

```
$ docker build -t sayhello .
```

Note: If the above command fails
to pull the required image
execute the below command

```
$ docker image pull python:2.7-slim
```

- Create container with port mapping
- `$ docker container run -d -p 11022:80 sayhello`

- Access the application on browser

http://loalhost:11022

Hands-On Demonstration

Web Application in Container

- Get war file to be deployed

```
$ ls web
NewApp1  NewApp1.war
```

- Create the container syncing the web folder with webapps volume

```
$ docker container run -d -p 11055:8080 -v
/home/osgdev/dockerlab/web:/usr/local/tomcat/webapps tomcat:8
```

- Create a Dockerfile to deploy war file automatically

```
$ cat Dockerfile
FROM tomcat:8
ADD NewApp1.war /usr/local/tomcat/webapps/
EXPOSE 8080
CMD ["catalina.sh", "run"]
```

Hands-On Demonstration

Web Application in Container

- Remove the folder created by war file with unarchived content

```
$ sudo rm -rf NewApp1
```

- Check whether workfolder has required files

```
$ ls
Dockerfile  NewApp1.war
```

- Build the tomcat1 image which can deploy the war file

```
$ docker build -t tomcat1 .
```

- Create the container with war file automatically deployed

```
$ docker run -d -p 11055:8080 tomcat1
```

- Check for application on browser:

http://localhost:11055

Hands-On Demonstration

Creating Tomcat Container Image

- Create a Dockerfile that can create Tomcat container image

```
$ cat Dockerfile
FROM debian:stretch
WORKDIR /opt/
ADD jdk-8u162-linux-x64.tar.gz /opt/
ADD apache-tomcat-8.5.27.tar.gz /opt/
```

- Build the tomcat container image

```
$ docker image build -t tomcat2 .
```

Hands-On Demonstration

Creating Tomcat Container Image

- Create tomcat container with interactive access

```
$ docker container run -it tomcat2
# ls
apache-tomcat-8.5.27  jdk1.8.0_162
# ls apache-tomcat-8.5.27/
LICENSE  RELEASE-NOTES bin   lib   temp     work
NOTICE                  RUNNING.txt          conf  logs  webapps
# ls jdk1.8.0_162/
COPYRIGHT    THIRDPARTYLICENSEREADME-JAVAFX.txt  db   jre  release
LICENSE      THIRDPARTYLICENSEREADME.txt     include                lib  src.zip
README.html  bin      javafx-src.zip  man
```

Note: JDK and tomcat tar files are correspondingly unarchived in respective folders

Hands-On Demonstration

Creating Tomcat Container Image

- Create Dockerfile setting the path for JDK, exposing tomcat port and the command to start the tomcat server

```
$ cat Dockerfile
FROM debian:stretch
ADD jdk-8u162-linux-x64.tar.gz /opt/
ADD apache-tomcat-8.5.27.tar.gz /opt/
ENV JAVA_HOME=/opt/jdk1.8.0_162
ENV PATH $JAVA_HOME/bin:$PATH
EXPOSE 8080
CMD ["/opt/apache-tomcat-8.5.27/bin/catalina.sh" , "run"]
```

- Build the tomcat image

```
$ docker image build -t tomcat3 .
```

- Create container with port forwarding using new image

```
$ docker container run -d -p 11077:8080 tomcat3
```

Hands-On Demonstration

Creating Tomcat Container Image

- Redo the Dockerfile to deploy the war file automatically into image

```
$ cat Dockerfile
FROM debian:stretch
WORKDIR /opt/
ADD jdk-8u162-linux-x64.tar.gz /opt/
ADD apache-tomcat-8.5.27.tar.gz /opt/
ENV JAVA_HOME=/opt/jdk1.8.0_162
ENV PATH $JAVA_HOME/bin:$PATH
ADD ./web/*.war /opt/apache-tomcat-8.5.27/webapps/
EXPOSE 8080
CMD ["/opt/apache-tomcat-8.5.27/bin/catalina.sh" , "run"]
```

- Build the image and launch the container to check results on browser

```
$ docker image build -t tomcat4 .
$ docker container run -d -p 11077:8080 tomcat4
```

- Browser Link: http://localhost:11077

Hands-On Demonstration

Dockerizing Node.js App: package.json

```
{
  "name": "docker_web_app",
  "version": "1.0.0",
  "description": "Node.js on Docker",
  "author": "First Last <first.last@example.com>",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.16.1"
  }
}
```

Hands-On Demonstration

Dockerizing Node.js App: server.js

```
'use strict';

const express = require('express');

// Constants
const PORT = 8080;
const HOST = '0.0.0.0';

// App
const app = express();
app.get('/', (req, res) => {
  res.send('Hello world\n');
});

app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

Hands-On Demonstration

Dockerizing Node.js App

- Create a .dockerignore file for docker build to ignore them

```
$ cat .dockerignore
node_modules
npm-debug.log
```

- Get node:carbon image from dockerhub, as it may not be pulled automatically

```
$ docker image pull node:carbon
```

- Listing the files in the work folder

```
$ ls -a
.  ..  Dockerfile  .dockerignore  package.json  server.js
```

Note: Dockerfile given in next slide

Hands-On Demonstration

Dockerizing Node.js App

```
$ cat Dockerfile
FROM node:carbon

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm install --only=production

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "npm", "start" ]
```

Hands-On Demonstration

Creating Node.js Container Image

- Build the nodejs image

```
$ docker build -t nodewebapp .
```

- Create container with new image

```
$ docker container run -p 11088:8080 -d nodewebapp
```

Hands-On Demonstration

Creating Docker Service using image

- Initilize Docker Swarm

```
$ docker swarm init
```

- Swarm initialized: current node (ge1tu2ymefv7a8i1o8yf9mryg) is now a manager.

- Check the Overlay network created

```
$ docker network ls
bdc5b03153da        docker_gwbridge        bridge             local
xwupecx1xmzm        ingress                overlay            swarm
```

Hands-On Demonstration

Creating Docker Service using image: docker-compose.yml

```
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: hellojava6
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "80:80"
    networks:
      - webnet
networks:
  webnet:
```

Hands-On Demonstration

Creating Docker Service using image

- Create Docker Stack using docker-compose.yaml

```
$ docker stack deploy -c docker-compose.yaml hello-java
```

- Look for the networks created

```
$ docker network ls
```

- Listing the services created

```
$ docker service ls
```

- Listing the distribution of tasks

```
$ docker service ps hello-java_web
```

- Listing the containers created on machine

```
$ docker container ls -a
```

Hands-On Demonstration

Creating Docker Service using image

- Removing the service only

```
$ docker service rm hello-java_web
```

- List the service to check removal

```
$ docker service ls
```

- Remove all the stopped containers

```
$ docker container prune
```

- Remove network separately

```
$ docker network rm hello-java_webnet
```

Hands-On Demonstration

Creating Docker Service using image

```
$ cat docker-compose-sayhello.yaml
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: sayhello
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "11055:80"
    networks:
      - webnet
networks:
  webnet:
```

Hands-On Demonstration

Creating Docker Service using image

- Create the service stack for python App

```
$ docker stack deploy -c docker-compose-sayhello.yaml sayhello
```

- List the networks created

```
$ docker network ls
```

- List the services created

```
$ docker service ls
```

- List the running tasks

```
$ docker service ps sayhello_web
```

- List the containers

```
$ docker container ls
```

Hands-On Demonstration

Removing Docker service

- List the available service

```
$ docker service ls
```

- Remove the service

```
$ docker service rm sayhello_web
```

- Check for service removal

```
$ docker service ls
```

- Check whether the networks are remved

```
$ docker network ls
```

- Check for container status

```
$ docker container ls
```

- Remove all containers for clearnup

```
$ docker container prune
```

Hands-On Demonstration

Removing Docker service stack

- Create the stack again to demonstrate removal

```
$ docker stack deploy -c docker-compose-sayhello.yaml sayhello
```

- Removing Stack

```
$ docker stack rm sayhello
```

- Check for service removal

```
$ docker service ls
```

- Check for removal of network

```
$ docker network ls
```

- Leave the swarm – but this fails as manager cannot leave swarm

```
$ docker swarm leave
```

- Manager to leave the swarm by force

```
$ docker swarm leave --force
```

Thank You