Docker

Hos Es

Table of Contents

1. Intro

Yeah I don’t like long intros either, but this one needs a little more digging. Let’s begin.

1.1. What is Docker?

Docker is a container management service. You may also hear it referred as *container engine*. All it can do is **create**, **run** and **manage** containers. Let’s go a little bit deeper.

1.2. What is a container

Let me put it this way, what you need to run a program? A computer (servers are also computers) and what a computer (beside hardware) has? Software. Now what software do you use to work with the computer’s hardware? **Operating System**. Containers are just a small isolated things that emulates a Operating System. In a different approach we can say: **Using OS-level virtualization to deliver software in packages called *containers***.

1.3. Containers VS. Virtual-Machines

Every VM requires its own dedicated operating system (OS) is a major flaw. Every OS consumes CPU, RAM and other resources that could otherwise be used to power more applications. Every OS needs patching and monitoring. And in some cases, every OS requires a license. All of this results in wasted time and resources.

A major difference is that containers do not require their own full-blown OS. In fact, all containers on a single host share the host’s OS. This frees up huge amounts of system resources such as CPU, RAM, and storage. It also reduces potential licensing costs and reduces the overhead of OS patching and other maintenance.

[vm vs container] | */images/notes/docker/vm-vs-container.png*

1.4. Linux containers

Modern containers started in the Linux world and are the product of an immense amount of work from a wide variety of people over a long period of time. Just as one example, Google LLC has contributed many container-related technologies to the Linux kernel. Without these, and other contributions, we wouldn’t have modern containers today.

There are many operating system virtualization technologies similar to containers that pre-date Docker and modern containers. Some even date back to System/360 on the Mainframe. BSD Jails and Solaris Zones are some other well-known examples of Unix-type container technologies.

2. Basics

Pulling an image

```
docker image pull ubuntu:latest
```

2.1. Dockerfile

You can create your own Docker image from a file called `Dockerfile`. This file includes instructions to build an application. Instructions like installing dependencies, copying files, running compile / build commands and so on. *Things that needs to be done for preparing the environment that you app will run in it*.

Dockerfile — Simple webapp

```
FROM alpine
LABEL maintainer="example@mail.com"
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

Example 1. Keywords

- `FROM`: The **base** image to use
- `LABEL`: Image's metadata
- `RUN`: Command to run inside the image at build time
- `COPY`: Copy files from `<local>` to `<image>`
- `WORKDIR`: Change the working directory
  - Note that if the directory does not exists, it will be created
- `EXPOSE`: The port that will be exposed
  - available to the host/other images
- `ENTRYPOINT`: The command that will be run at the end
  - The actual app; all the previous commands are just a preparation for this command/section

Dockerfile — Build image

```
docker buildx build -f Dockerfile -t test-name .
```

```
[+] Building 35.5s (10/10) FINISHED
docker:default
```

```
 => [internal] load build definition from Dockerfile
0.0s
 => => transferring dockerfile: 363B
0.0s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/alpine:latest
0.0s
 => [1/5] FROM docker.io/library/alpine
0.0s
 => [internal] load build context
0.0s
 => => transferring context: 36.77kB
0.0s
 => [2/5] RUN apk add --update nodejs npm curl
9.2s
 => [3/5] COPY . /src
0.0s
 => [4/5] WORKDIR /src
0.0s
 => [5/5] RUN npm install
26.0s
 => exporting to image
0.2s
 => => exporting layers
0.2s
 => => writing image
sha256:1825da98372a937a095e523f71f59dcd6a4c41f6f0f573ed1bfe118fae8d50ed   0.0s
 => => naming to docker.io/library/test
0.0s
```

Docker CLI — Get list of images

```
docker image ls
```

```
REPOSITORY            TAG                 IMAGE ID       CREATED         SIZE
test                  latest              1825da98372a   6 minutes ago   95.4MB
```

Docker CLI — Run an image/container

```
docker container run -d --name web1 --publish 8080:8080 test:latest
```

```
63e334d59c7d937c99f6cbe6e5628c3b180ab26cb4b3f7e3d2cdd007cfd78f37
```

Docker CLI — Get list of running images/containers

```
docker ps
```

```
CONTAINER ID   IMAGE         COMMAND           CREATED          STATUS
PORTS                                       NAMES
63e334d59c7d   test:latest   "node ./app.js"   14 seconds ago   Up 13
seconds   0.0.0.0:8080->8080/tcp, :::8080->8080/tcp   web1
```

You can also use multiple images inside one container:

Multi-stage Dockerfile

```
# pull golang image to build the app
FROM golang:alpine as builder
# add a label
LABEL maintainer="Hossein Esmailzadeh <hosteam01@gmail.com>"

## inside golang:alpine image
# change the working directory
WORKDIR /app

# copy golang's related files (checksum and module files)
COPY go.mod go.sum ./
# get the needed modules
RUN go mod download
# copy the source code
COPY . .

# compile the app
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .

## inside our image
# use a alpine (very small OS) image as base
FROM alpine:latest

# install dependencies
RUN apk --no-cache add ca-certificates
# change the working directory
WORKDIR /root/
# copy the built binary from the golang:alpine image inside our container
COPY --from=builder /app/main .

# let port 9090 to be accessible for others
EXPOSE 9090

# run the binary at the end
CMD ["./main"]
```