Docker Essentials

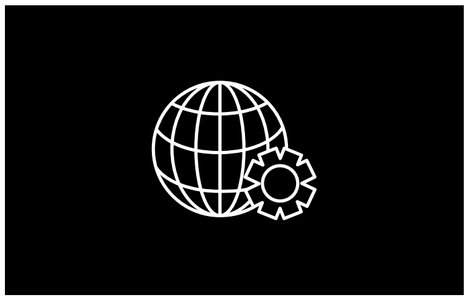
01

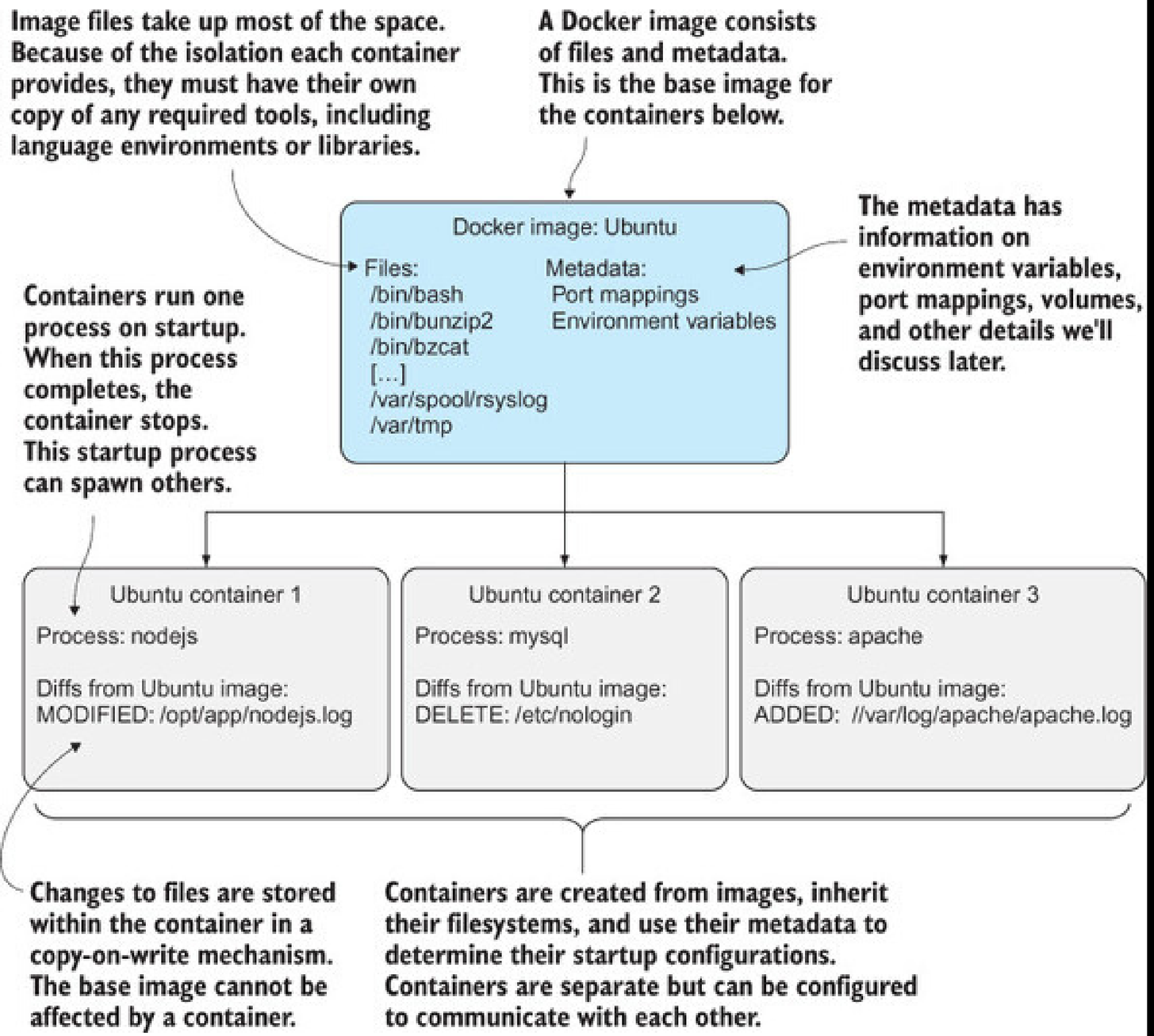
Docker Images Containers & more...

02 DISCOVERING DOCKER

Docker is good for

- Replacing virtual machines (VMs)
- Prototyping software
- Packaging software
- Enabling a micro services architecture
- Modeling networks
- Enabling full-stack productivity when offline
- Enabling continuous delivery
- Reducing Debugging overhead



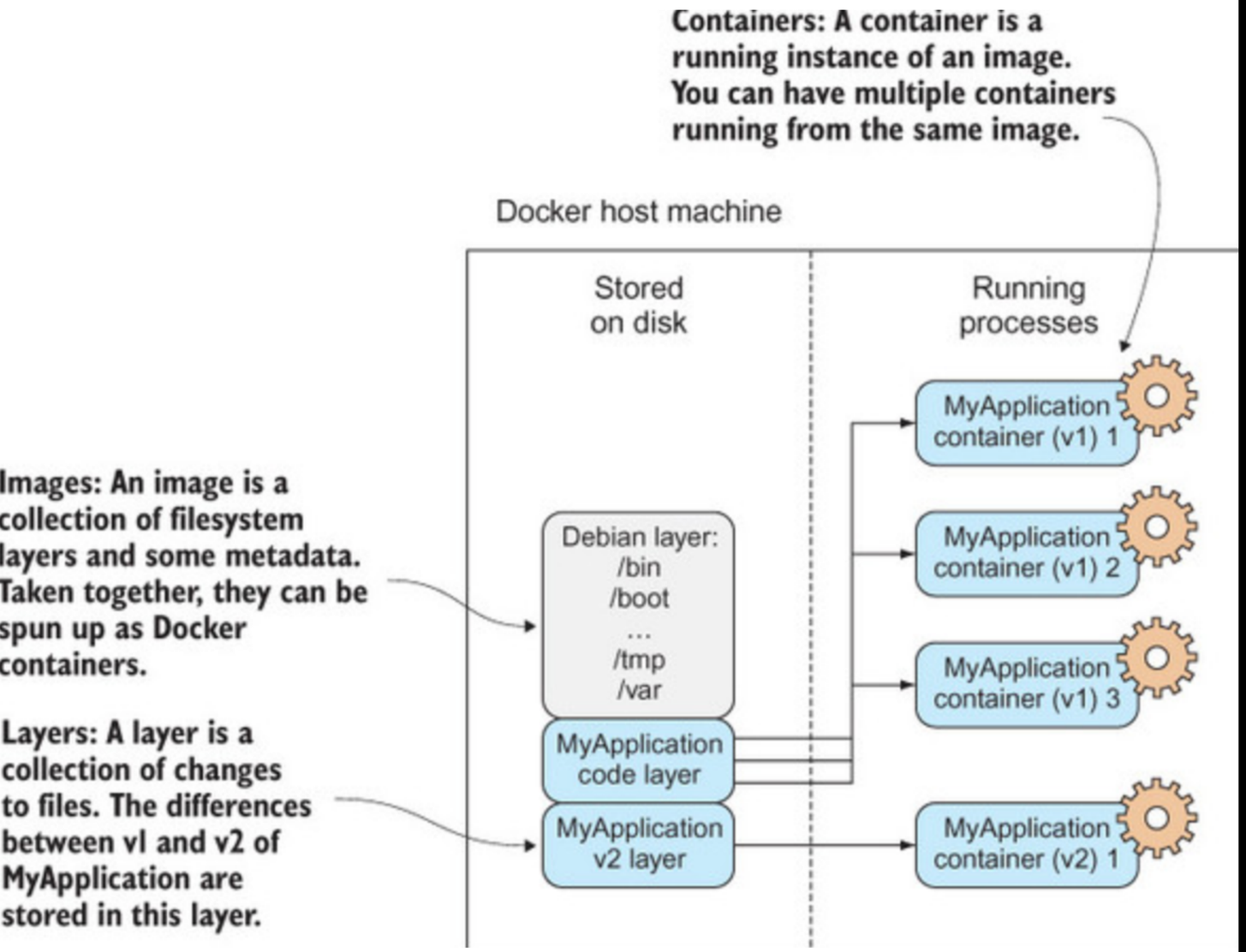


DOCKER IMAGE

A Docker image is a file, comprised of multiple layers, that is used to execute code in a Docker container.

- An image is essentially built from the instructions for a complete and executable version of an application, which relies on the host OS kernel.
- A Docker image is roughly equivalent to a "snapshot" in other virtual machine environments.
- It is a record of a Docker virtual machine, or Docker container, at a point in time.

CONTAINER

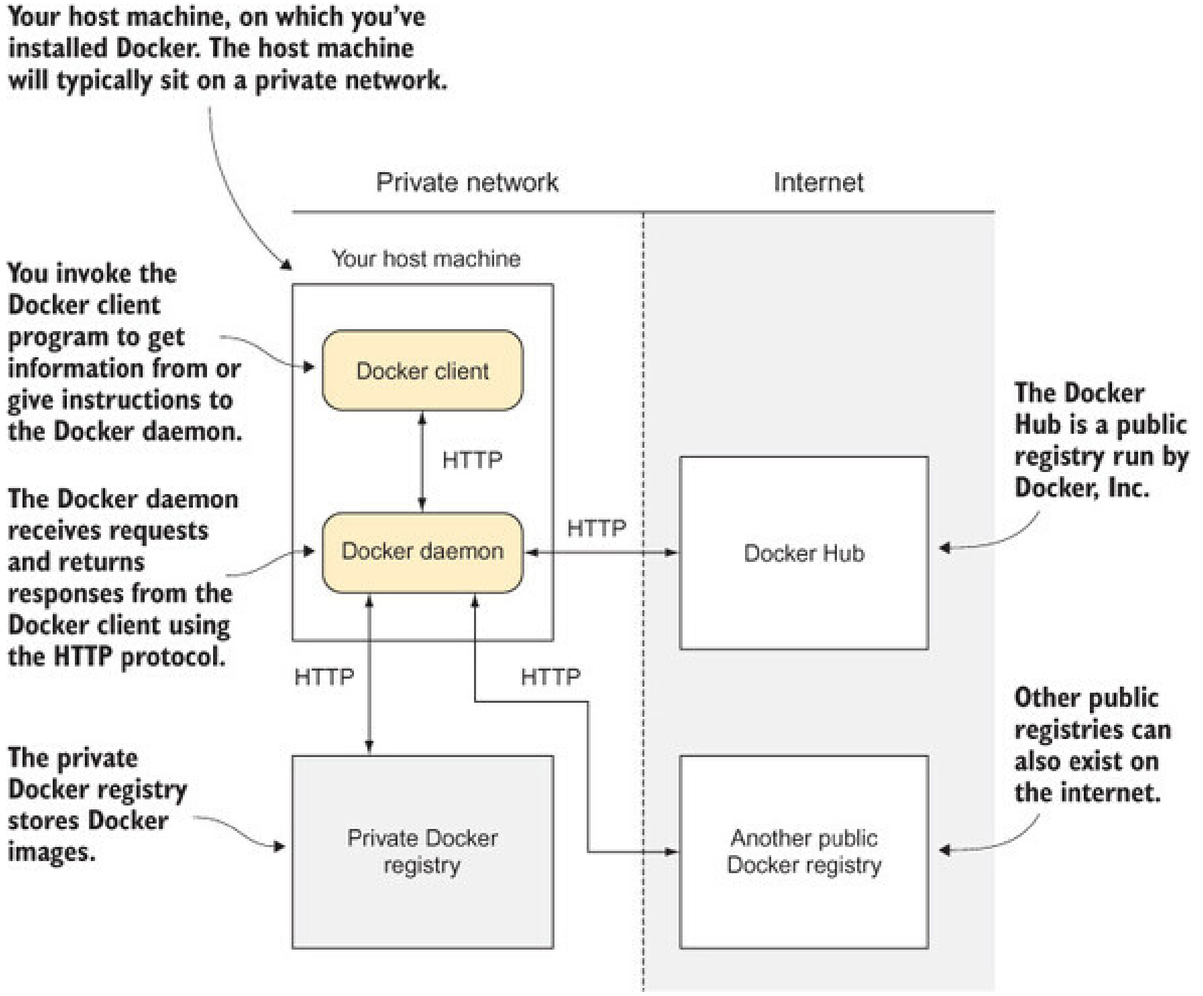


Containers are running systems defined by images. These images are made up of one or more layers (or sets of diffs) plus some meta data for Docker.
"Docker container can be viewed as a Docker image in execution"

- A container that moves from one Docker environment to another with the same OS will work without any changes, since the image includes all of the dependencies needed to execute the code.
- Docker will use resource isolation features in the OS kernel, such as cgroups in Linux, to run multiple independent containers on the same OS.

05 DOCKER ARCHITECTURE

- The docker command is a client, and the Docker daemon acts as the server doing the processing on your Docker containers and images.
- Docker containers are managed by the daemon, not your shell session.
- Docker Images are stored in a registry which can be local or remote



DOCKER IMAGES & LAYERS

- Docker images have the special characteristic of being *immutable*. They can't be modified, but they can be duplicated and shared or deleted.
- The immutability is useful when testing new software or configurations because no matter what happens, the image will still be there, as usable as ever.
- A Docker image is made up of multiple layers.
- A user composes each Docker image to include system libraries, tools and other files and dependencies for the executable code.
- Reuse saves time, since a user does not have to create everything in an image.

```
FROM node                                                                  1
 LABEL maintainer ian.miell@gmail.com                                      2
 RUN git clone -q https://github.com/docker-in-practice/todo.git           3
 WORKDIR todo                                                              4
 RUN npm install > /dev/null                                               5
 EXPOSE 8000                                                               6
 CMD ["npm","start"]                                                       7
```

```
Sending build context to Docker daemon  2.048kB                            1
 Step 1/7 : FROM node                                                      2
  ---> 2ca756a6578b                                                        3
 Step 2/7 : LABEL maintainer ian.miell@gmail.com
 ---> Running in bf73f87c88d6
 ---> 5383857304fc
Removing intermediate container bf73f87c88d6                               4
 Step 3/7 : RUN git clone -q https://github.com/docker-in-practice/todo.git
 ---> Running in 761baf524cc1
 ---> 4350cb1c977c
Removing intermediate container 761baf524cc1
Step 4/7 : WORKDIR todo
 ---> a1b24710f458
Removing intermediate container 0f8cd22fbe83
Step 5/7 : RUN npm install > /dev/null
 ---> Running in 92a8f9ba530a
npm info it worked if it ends with ok                                      5
 [...]
npm info ok
 ---> 6ee4d7bba544
Removing intermediate container 92a8f9ba530a
Step 6/7 : EXPOSE 8000
 ---> Running in 8e33c1ded161
 ---> 3ea44544f13c
Removing intermediate container 8e33c1ded161
Step 7/7 : CMD npm start
 ---> Running in ccc076ee38fe
 ---> 66c76cea05bb
Removing intermediate container ccc076ee38fe
Successfully built 66c76cea05bb                                            6
```

DEBUGGING WITH *SOCAT*

07

'socat' is a powerful command that allows you to relay data between two data channels of almost any type.

- socat -v UNIX-LISTEN:/tmp/dockerapi.sock,fork \
  UNIX-CONNECT:/var/run/docker.sock &

```
$ docker -H unix:///tmp/dockerapi.sock ps -a                                    1
 > 2017/05/15 16:01:51.163427  length=83 from=0 to=82
GET /_ping HTTP/1.1\r
Host: docker\r
User-Agent: Docker-Client/17.04.0-ce (linux)\r
\r
< 2017/05/15 16:01:51.164132  length=215 from=0 to=214
HTTP/1.1 200 OK\r
Api-Version: 1.28\r
Docker-Experimental: false\r
Ostype: linux\r
Server: Docker/17.04.0-ce (linux)\r
Date: Mon, 15 May 2017 15:01:51 GMT\r
Content-Length: 2\r
Content-Type: text/plain; charset=utf-8\r
\r
OK> 2017/05/15 16:01:51.165175  length=105 from=83 to=187                       2
 GET /v1.28/containers/json?all=1 HTTP/1.1\r
Host: docker\r
User-Agent: Docker-Client/17.04.0-ce (linux)\r
\r
< 2017/05/15 16:01:51.165819  length=886 from=215 to=1100                       3
 HTTP/1.1 200 OK\r
Api-Version: 1.28\r
Content-Type: application/json\r
Docker-Experimental: false\r
Ostype: linux\r
Server: Docker/17.04.0-ce (linux)\r
Date: Mon, 15 May 2017 15:01:51 GMT\r
Content-Length: 680\r
\r
[{"Id":"1d0d5b5a7b506417949653a59deac030ccbcbb816842a63ba68401708d55383e",
➡ "Names":["/example1"],"Image":"todoapp","ImageID":
➡ "sha256:ccdda5b6b021f7d12bd2c16dbcd2f195ff20d10a660921db0ac5bff5ecd92bc2",
➡ "Command":"npm start","Created":1494857777,"Ports":[],"Labels":{},
➡ "State":"exited","Status":"Exited (0) 45 minutes ago","HostConfig":
➡ {"NetworkMode":"default"},"NetworkSettings":{"Networks":{"bridge":
➡ {"IPAMConfig":null,"Links":null,"Aliases":null,"NetworkID":
➡ "6f327d67a38b57379afa7525ea63829797fd31a948b316fdf2ae0365faeed632",
➡ "EndpointID":"","Gateway":"","IPAddress":"","IPPrefixLen":0,
➡ "IPv6Gateway":"","GlobalIPv6Address":"","GlobalIPv6PrefixLen":0,
➡ "MacAddress":""}}},"Mounts":[]}]                                             4
 CONTAINER ID        IMAGE               COMMAND             CREATED
➡      STATUS                      PORTS               NAMES               5
 1d0d5b5a7b50        todoapp             "npm start"         45 minutes ago
➡      Exited (0) 45 minutes ago                       example1
```

ALLOWING CONTAINER COMMUNICATION

If you have multiple Docker containers running on port 80 in their internal environment, they can’t all be accessible on port 80 on your host machine.

Communication between multiple containers

Employ user-defined networks to enable containers to communicate with each other

- docker network create my_network
  0c3386c9db5bb1d457c8af79a62808f78b42b3a8178e75cc8a252fac6fdc09e4
- docker network connect my_network blog1
-
```
$ docker run -it --network my_network ubuntu:16.04 bash
root@06d6282d32a5:/# apt update && apt install -y curl
[...]
root@06d6282d32a5:/# curl -sSL blog1 | head -n5
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US"
xml:lang="en-US">
<head>
        <meta name="viewport" content="width=device-width" />
        <meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
root@06d6282d32a5:/# curl -sSL blog2
curl: (6) Could not resolve host: blog2
```

This command creates a new virtual network living on your machine that you can use to manage container communication. By default, all containers that you connect to this network will be able to see each other by their names.

09 Linking containers for port isolation

- ```
  $ docker run --name wp-mysql \
    -e MYSQL_ROOT_PASSWORD=yoursecretpassword -d mysql
  $ docker run --name wordpress \
    --link wp-mysql:mysql -p 10003:80 -d  wordpress
  ```
- You also link the wp-mysql container to the WordPress container (--link wp-mysql:mysql). References to a mysql server within the WordPress container will be sent to the container named wp-mysql
- Be aware that links won't wait for services in linked containers to start; hence the instruction to pause between commands. A more precise way of doing this is to look for "mysqid: ready for connections" in the output of docker logs wp-mysql before running the WordPress container.
- In order for containers to be linked in this way, their ports must be specified as exposed when building the images. This is achieved using the EXPOSE command within the image build's Dockerfile. The ports listed in EXPOSE directives in Dockerfiles are also used when using the -P flag ("publish all ports." rather than -p, which publishes as a specific port) to the docker run command.
  ```

DOCKER VOLUMES

Volumes are a core part of Docker, and the issue of external data reference is yet another fast-changing area of the Docker ecosystem.

- Use Docker's volume flag to access host files from within the container

  docker run -v /var/db/tables:/var/data1 -it debian bash

- Beware of mapping over existing directories. The container's directory will be mapped even if it already exists in the image. This means that the directory you're mapping to within the container will effectively disappear. Fun things happen if you try to map a key directory! Try mounting an empty directory over /bin, for example.
  Also note that volumes are assumed not to persist in Dockerfiles. If you add a volume and then make changes to that folder within a Dockerfile, the changes won't be persisted to the resulting image.

- The -v argument doesn't map the volume to a host directory, so it creates the directory within the scope of this container's responsibility. This directory is populated with a single file with touch, and the container immediately exists—a data container need not be running to be used. We've used the small but functional busybox image to reduce the amount of extra baggage our data container needs.
  Then you run up another container to access the file you just created:

  *docker run -v /shared-data --name dc busybox *
  touch /shared-data/somefile

DISCUSSION

The --volumes-from flag allows you to reference the files from the data container by mounting them in the current container —you just need to pass it the ID of a container with volumes defined.

docker run -t -i --volumes-from dc busybox /bin/sh
/ # ls /shared-data
somefile

RUN AS NON-ROOT

Add yourself to the docker group

$ sudo addgroup -a username docker

Security Issues When Building Docker Images

A consequence of the design of Dockerfiles and their production of Docker images is that the final image contains the data state at each step in the Dockerfile. In the course of building your images, secrets may need to be copied in to ensure the build can work. These secrets may be SSH keys, certificates, or password files. Deleting these secrets before committing your image doesn't provide you with any real protection, as they'll be present in higher layers of the final image. A malicious user could easily extract them from the image.

- ```
  FROM debian
  RUN echo "My Big Secret" >> /tmp/secret_key      1
  RUN cat /tmp/secret_key                          2
  RUN rm /tmp/secret_key
  ```

- ```
  docker history mysecret                                                    1
  IMAGE          CREATED          CREATED BY                                      SIZE
  55f3c131a35d 3 days ago /bin/sh -c rm /tmp/secret_key                          2
  5b376ff3d7cd 3 days ago /bin/sh -c cat /tmp/secret_key                 0 B
  5e39caf7560f 3 days ago /bin/sh -c echo "My Big Secret" >> /tmp/se 14 B
  3
  c90d655b99b2 6 days ago /bin/sh -c #(nop) CMD [/bin/bash]              0 B
  30d39e59ffe2 6 days ago /bin/sh -c #(nop) ADD file:3f1a40df75bc567 85.01
  MB 4
  511136ea3c5a 20 months ago
  ```

- ```
  docker run 5b376ff3d7cd cat /tmp/secret_key
  My Big Secret
  ```

Now you have a "dangerous" container with a secret inside that you've seen can be hacked to reveal its secrets.
  ```

12 Fixing Security Issue With Docker Images

Solution 1

- To make this image safe, you'll need to flatten it. This means you'll keep the same data in the image but remove the intermediate layering information. To achieve this, you need to export the image as a trivially run container and then re-import and tag the resulting image:

```
$$ docker run -d mysecret /bin/true                    1
 28cde380f0195b24b33e19e132e81a4f58d2f055a42fa8406
e755b2ef283630f
$ docker export 28cde380f | docker import - mysecret          2
 $ docker history mysecret
IMAGE          CREATED        CREATED BY  SIZE
fdbeae08751b  13 seconds ago          85.01 MB            3
```

Solution 2

- Use Multi Stage Build

```
FROM centos as archives
ARG
AWS_SECRET_ACCESS_KEY=AWS_SECRET_ACCESS_KEY
ARG AWS_SECURITY_TOKEN=AWS_SECURITY_TOKEN
ARG AWS_SESSION_TOKEN=AWS_SESSION_TOKEN
RUN export
AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY} \
   && export AWS_SECURITY_TOKEN=${AWS_SECURITY_TOKEN} \
   && aws s3 cp s3://bucket/oracledb/ /downloads/ --recursive \
   && unset AWS_SECRET_ACCESS_KEY \
   && unset AWS_SECURITY_TOKEN

FROM centos as compressed
RUN dnf update -y
RUN mkdir -p /u01/software
RUN mkdir -p /u05
COPY --from=archives /downloads /u01/software/
ENTRYPOINT ["/u01/software/run.sh"]
```

Summary

- Docker Architecure
- Docker Images
  - Building Docker Images
  - Layered Architecture
  - Security Issues
- Docker Containers
  - Communication via networks
  - Mounting Volumes

Docker Images

Docker Containers

Volumes & Networks

QUESTIONS?

Let's discuss different use cases that are currently in use where docker is extensively used

- MicroService Architecture
- Applications Running on Cloud
- Kubernetes
- CICD process etc..

15

NAGA BIJESH ROY RAYA

https://www.linkedin.com/in/nagabijeshroy/