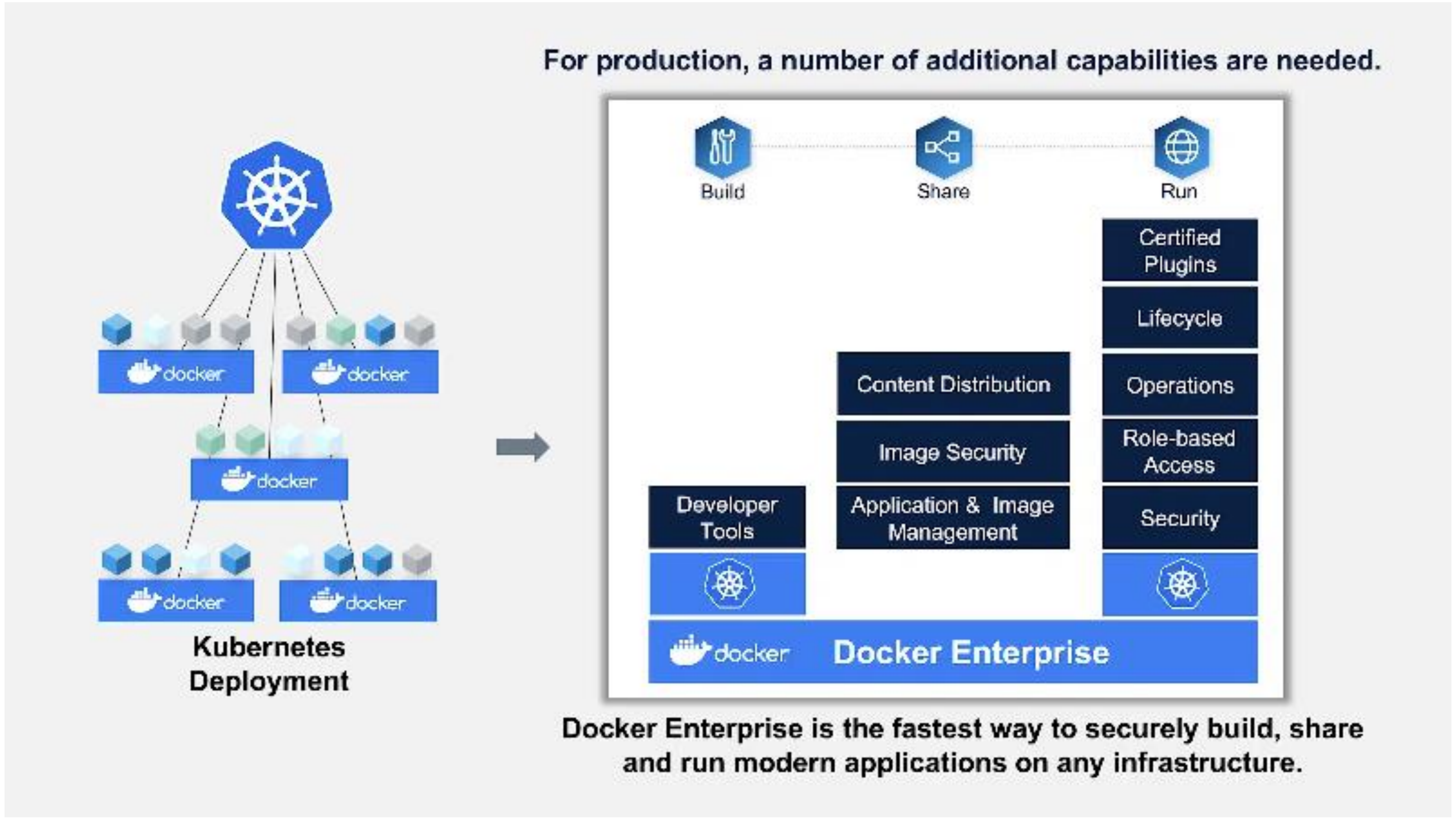
Dockers & Kubernetes



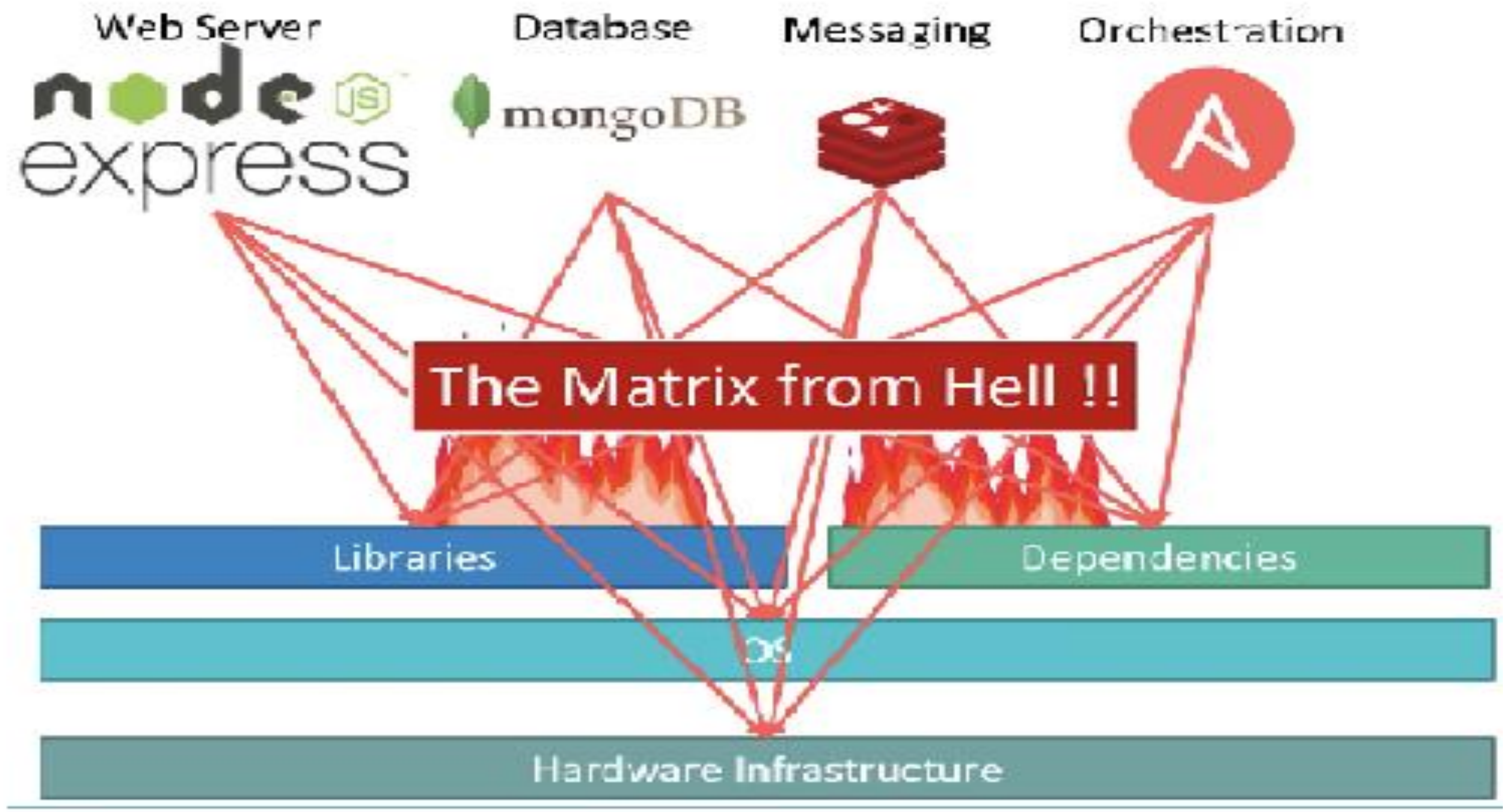
For production, a number of additional capabilities are needed.

Docker Enterprise is the fastest way to securely build, share and run modern applications on any infrastructure.

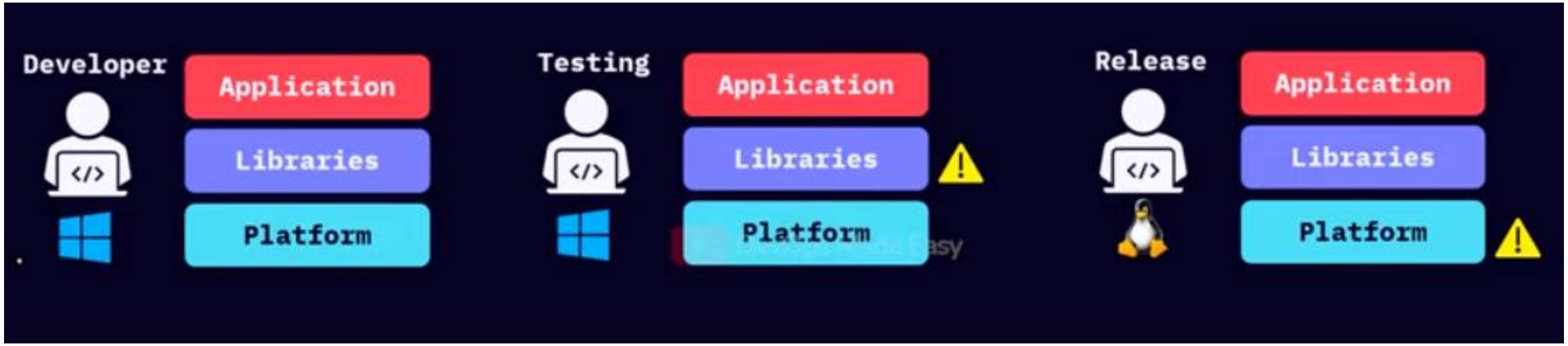
Docker Overview

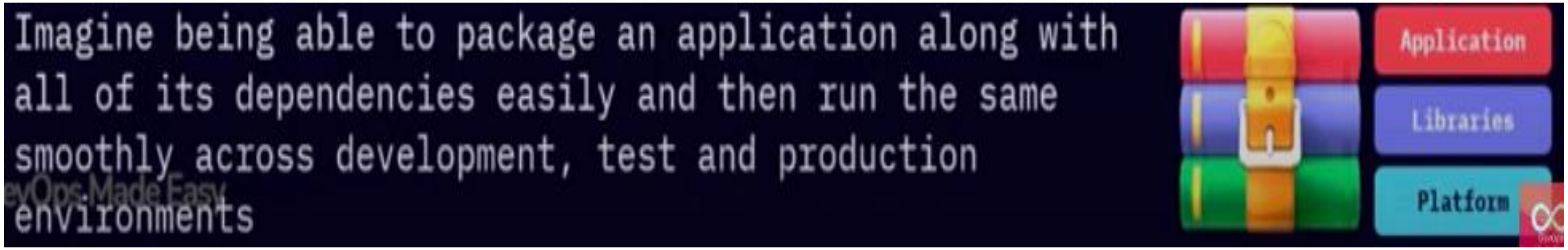
- Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.
  - ➢Similar to shipping containers: The container is always the same, regardless of the contents and thus fits on all trucks, cranes, ships..

Docker Containers



Missing Dependencies & Platform Differences

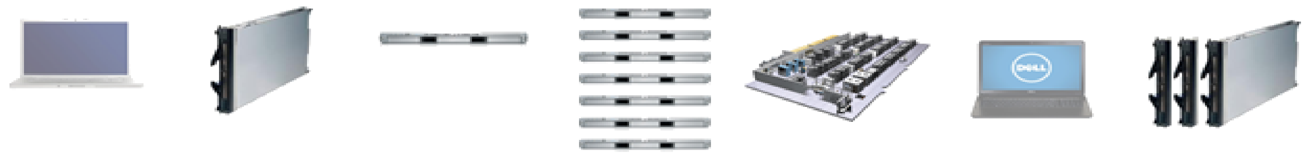




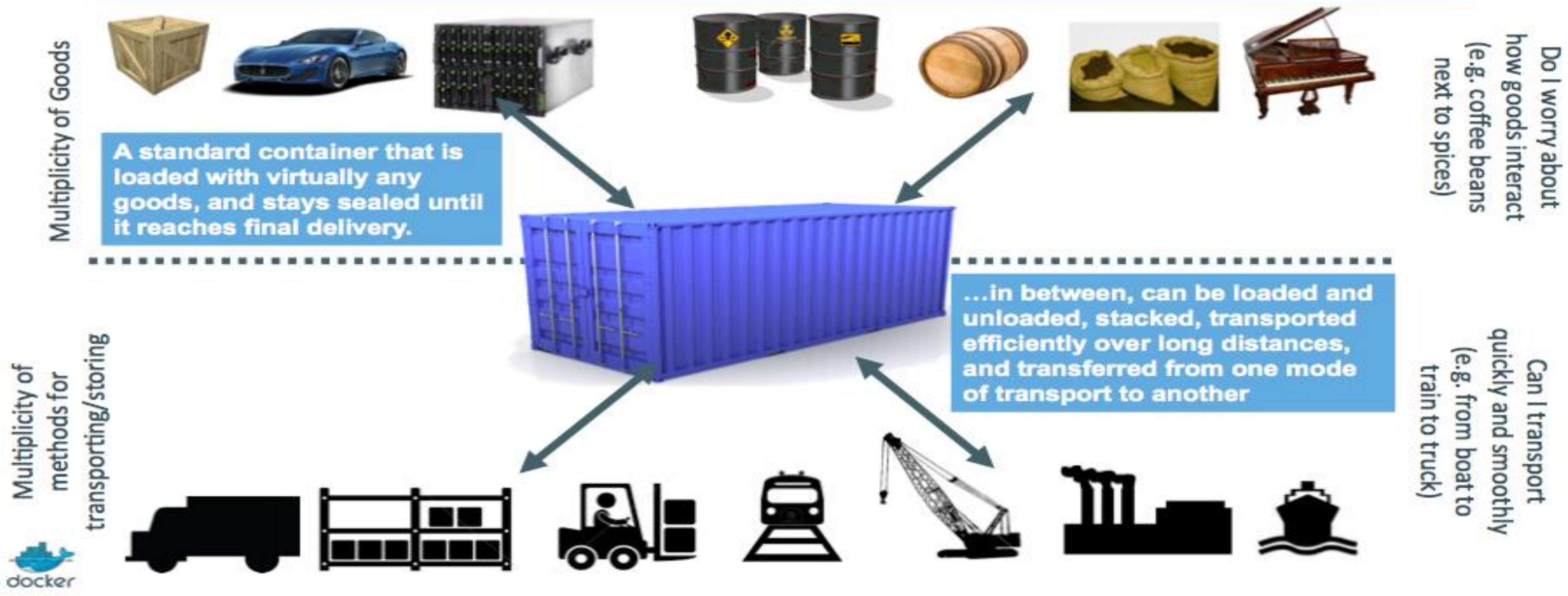
Imagine being able to package an application along with all of its dependencies easily and then run the same smoothly across development, test and production environments

The Challenge – The Matrix From Hell

	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?



Help From Elsewhere - Shipping Containers



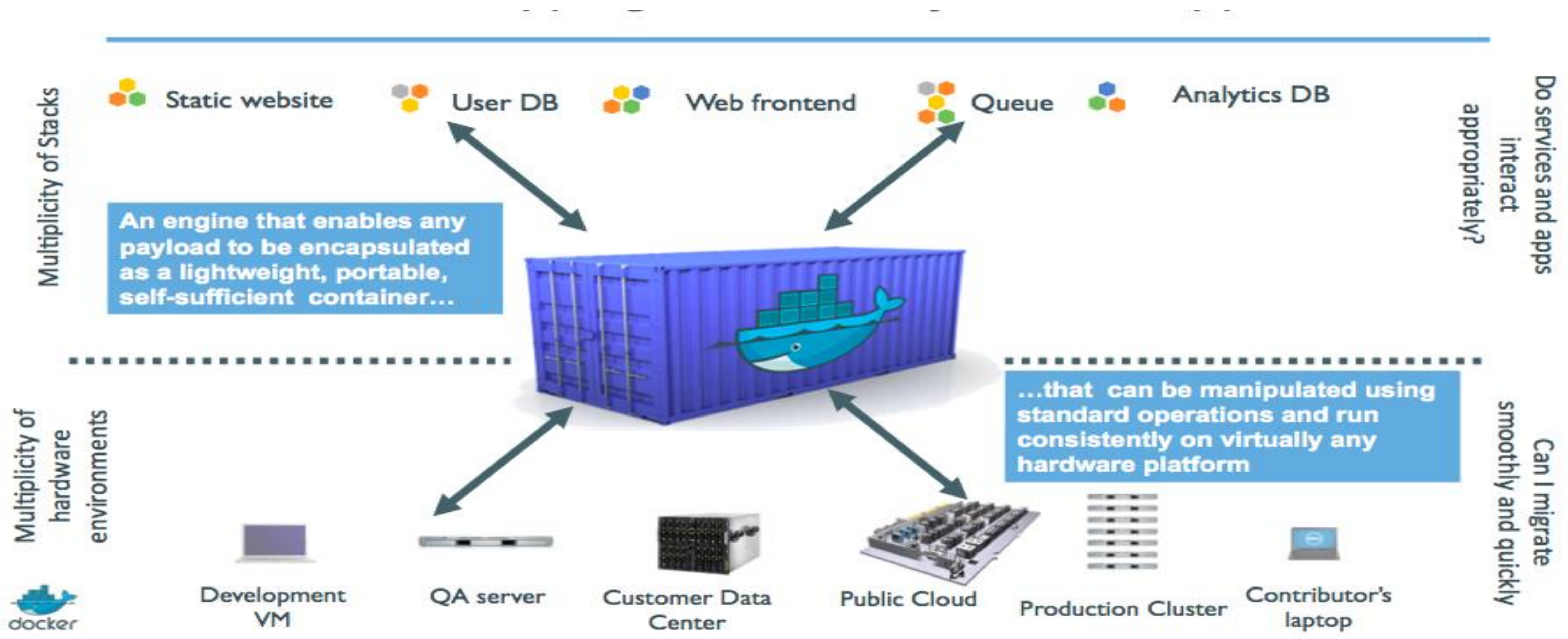
A standard container that is loaded with virtually any goods, and stays sealed until it reaches final delivery.

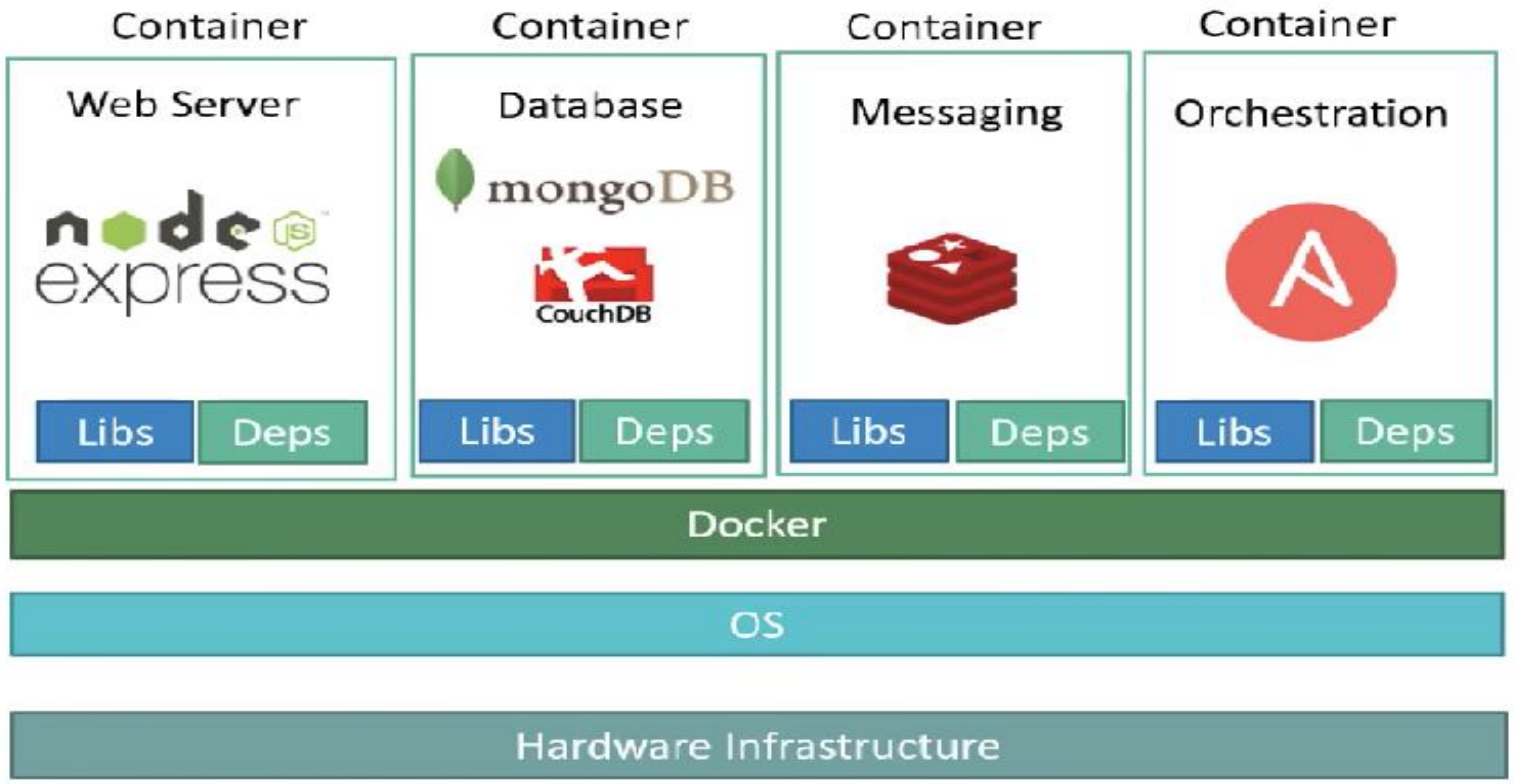
…in between, can be loaded and unloaded, stacked, transported efficiently over long distances, and transferred from one mode of transport to another

Do I worry about how goods interact (e.g. coffee beans next to spices)

Can I transport quickly and smoothly (e.g. from boat to train to truck)

Applied to IT World - Application Containers


Multiplicity of Stacks
Static website
User DB
Web frontend
Queue
Analytics DB
Do services and apps interact appropriately?
An engine that enables any payload to be encapsulated as a lightweight, portable, self-sufficient container…
…that can be manipulated using standard operations and run consistently on virtually any hardware platform
Multiplicity of hardware environments
Can I migrate smoothly and quickly
docker
Development VM
QA server
Customer Data Center
Public Cloud
Production Cluster
Contributor's laptop


Container
Container
Container
Container
Web Server
Database
Messaging
Orchestration
node js
express
mongoDB
CouchDB
Libs
Deps
Libs
Deps
Libs
Deps
Libs
Deps
Docker
OS
Hardware Infrastructure

Docker Architecture

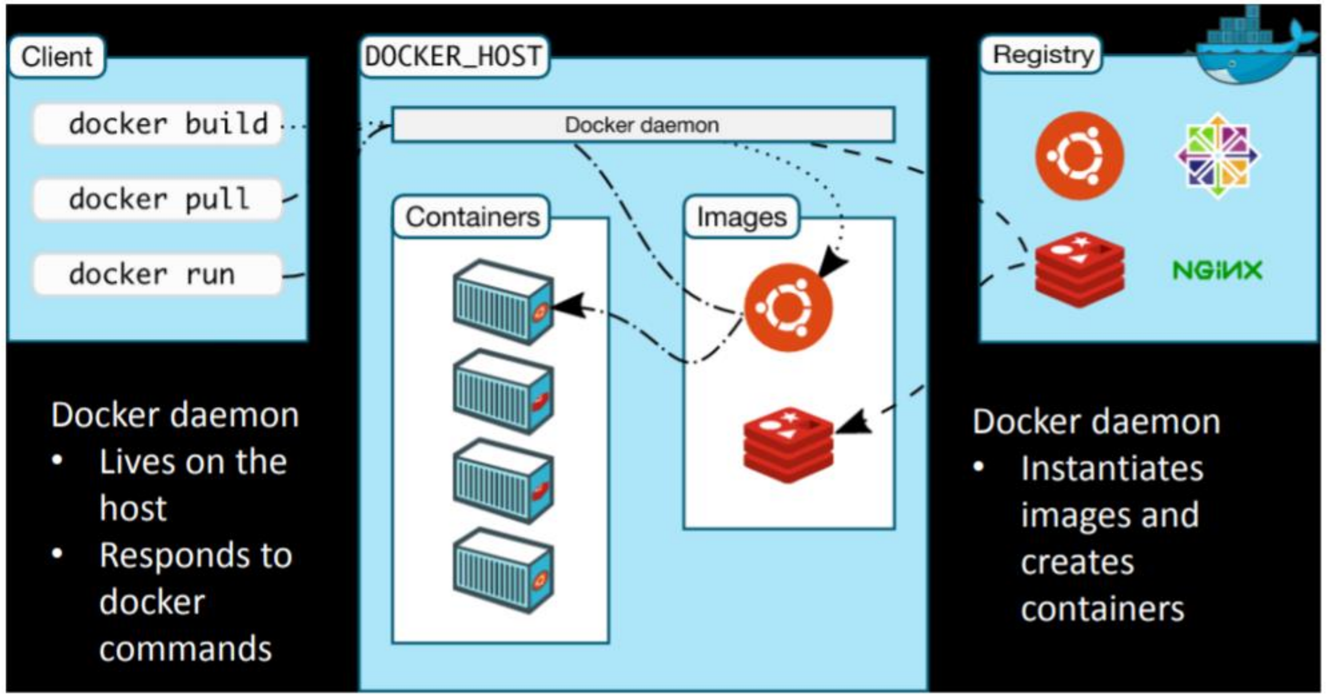


Image is instantiated to form container

Understanding Docker

- **Docker** uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, which lets you work with applications consisting of a set of containers.
- **The Docker daemon** (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.
- **The Docker client** (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

- **Docker objects**

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

- **Images**

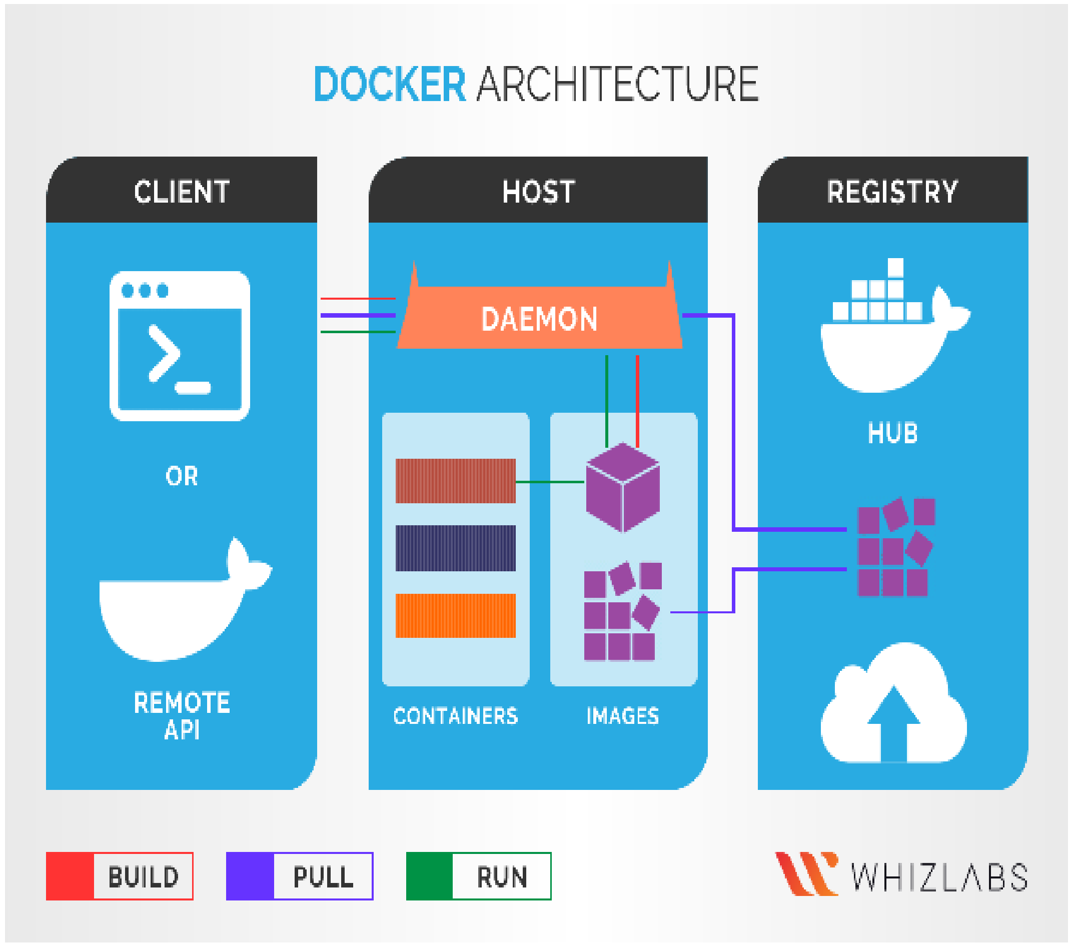
An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run. You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

- **Containers**

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state. By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine. A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

How Does Docker Works?

- The Docker architecture consists of four main components along with Docker containers which we've covered earlier.
- **Docker client** – the main component to create, manage, and run containerized applications. The Docker client is the primary method of controlling the Docker server via a CLI like Command Prompt (Windows) or Terminal (macOS, Linux).
- **Docker server** – also known as the Docker daemon. It waits for REST API requests made by the Docker client and manages images and containers.
- **Docker images** – instruct the Docker server with the requirements on how to create a Docker container. Images can be downloaded from websites like Docker Hub. Creating a custom image is also possible – to do it, users need to create a Dockerfile and pass it to the server. It's worth noting that Docker doesn't clear any unused images, so users need to delete image data themselves before there's too much of it.
- **Docker registry** – an open-source server-side application used to host and distribute Docker images. The registry is extra useful to store images locally and maintain complete control over them. Alternatively, users can access the aforementioned Docker Hub – the world's largest repository of Docker images.



Difference between Physical Server, Virtual Machine and Docker

Physical Server:

- Early to Mid-2000s
- One Software App per Physical Server

Drawbacks:

- CAPEX Costs
- OPEX Costs
  - Power and Cooling
  - Administration Costs
- Overpowered massively overpriced

Virtual Machines

Benefits:

- Multiple VMs on Single Machine.
- Consolidate apps into single physical machine
  - Cost savings
  - Faster server provisioning

Drawbacks:

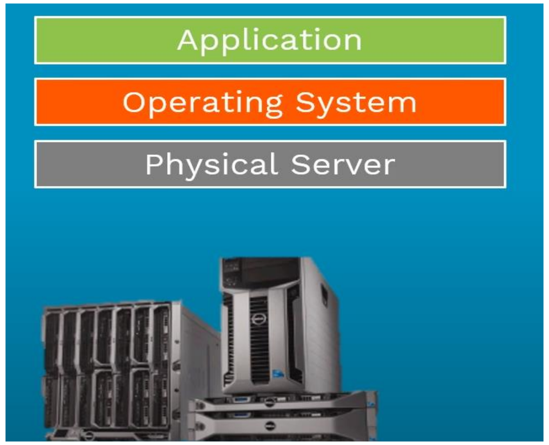
- Requires compute and storage
- os Licenses
- CAPEx costs
- OPEX costs

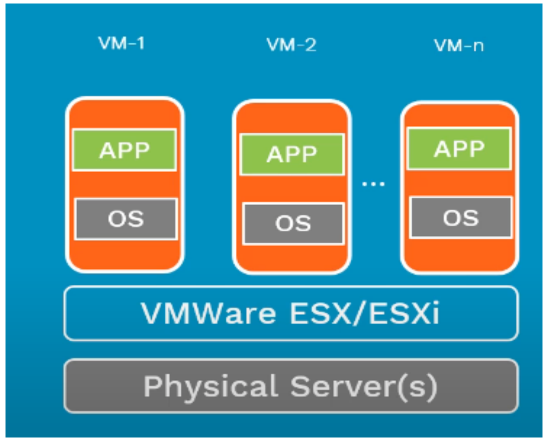
Containers

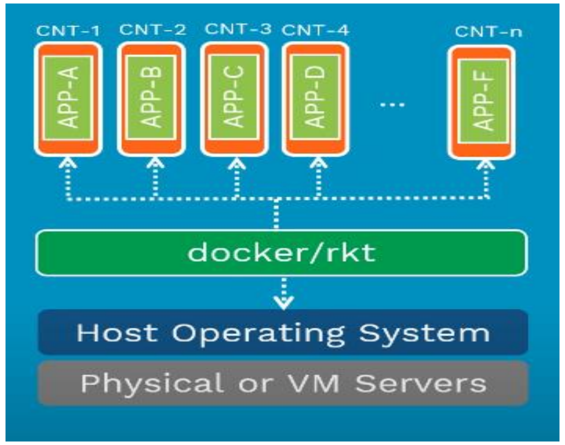
- Containers virtualize at the OS level
- Multiple containers will run on a single OS.

Benefits:

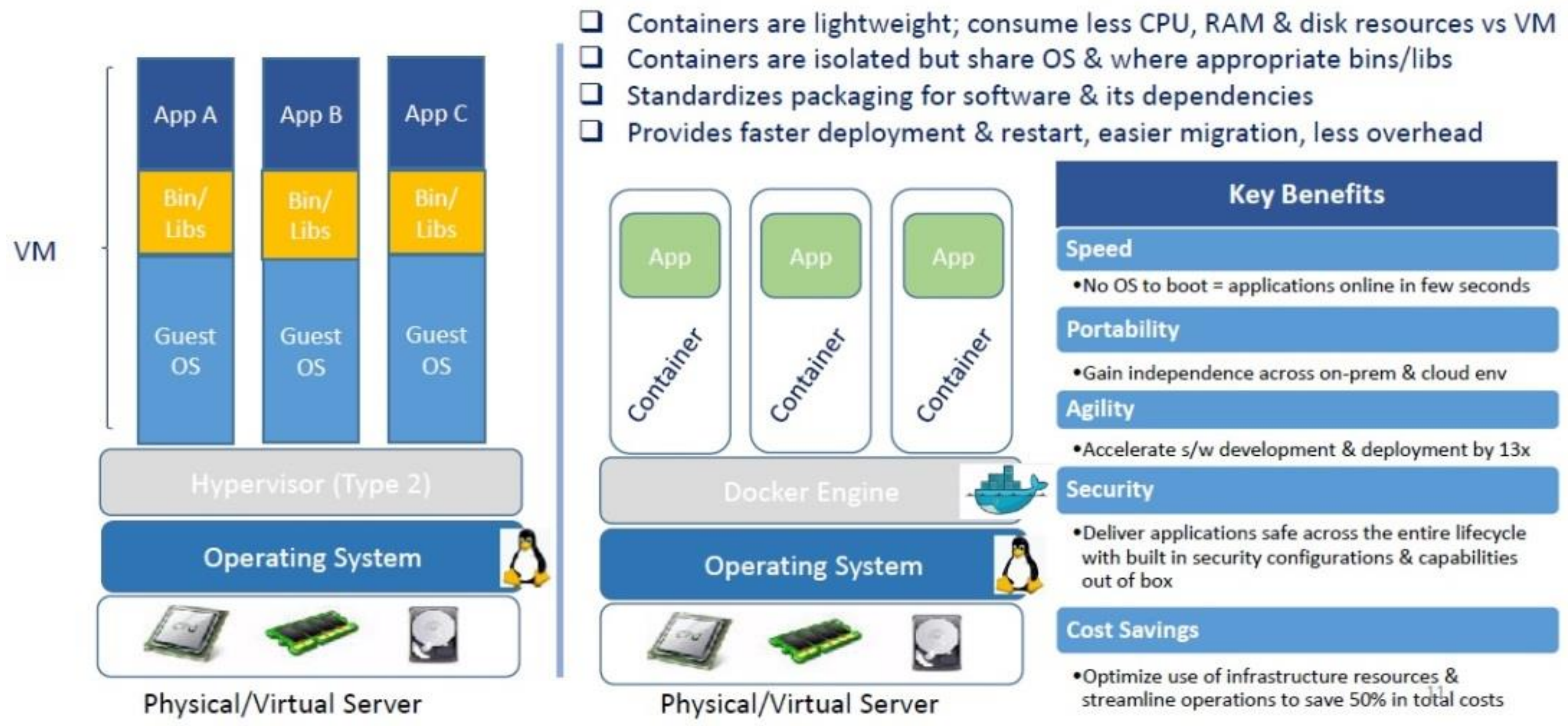
- Containers become very light weight.
- It boots up in a matter of seconds.
- Takes a fraction of disk and memory space.







Docker vs. Virtual Machine



- ❑ Containers are lightweight; consume less CPU, RAM & disk resources vs VM
- ❑ Containers are isolated but share OS & where appropriate bins/libs
- ❑ Standardizes packaging for software & its dependencies
- ❑ Provides faster deployment & restart, easier migration, less overhead

Key Benefits

Speed
- No OS to boot = applications online in few seconds

Portability
- Gain independence across on-prem & cloud env

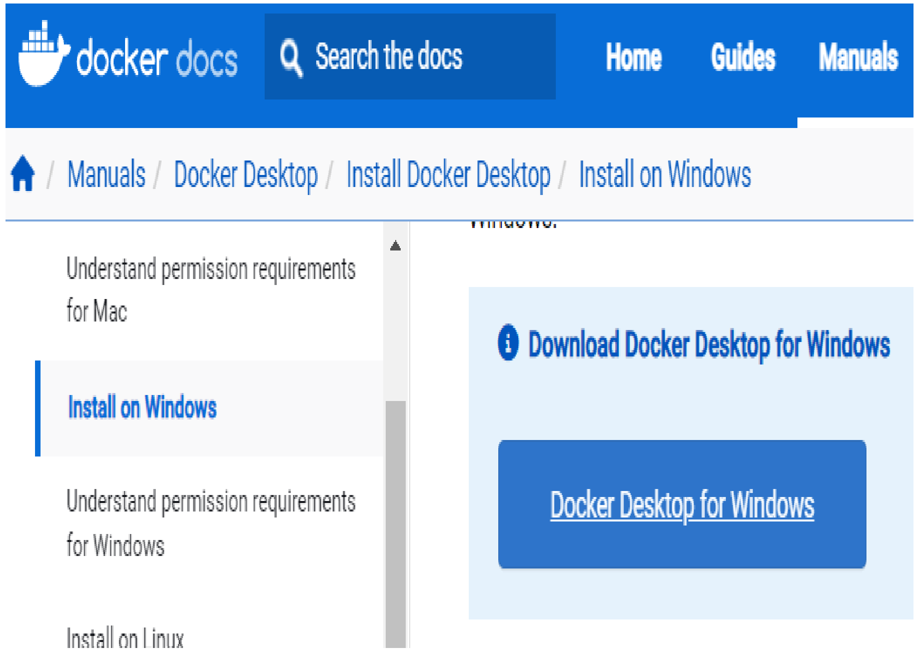
Agility
- Accelerate s/w development & deployment by 13x

Security
- Deliver applications safe across the entire lifecycle with built in security configurations & capabilities out of box

Cost Savings
- Optimize use of infrastructure resources & streamline operations to save 50% in total costs

Source: https://www.docker.com/whatisdocker/

Docker Installation steps on windows


docker docs
Search the docs
Home
Guides
Manuals
Manuals / Docker Desktop / Install Docker Desktop / Install on Windows
Understand permission requirements for Mac
Install on Windows
Understand permission requirements for Windows
Install on Linux
Download Docker Desktop for Windows
Docker Desktop for Windows

1. Double-click Docker Desktop Installer.exe to run the installer.
2. When prompted, ensure the Use WSL 2 instead of Hyper-V option on the Configuration page is selected or not depending on your choice of backend. If your system only supports one of the two options, you will not be able to select which backend to use.
3. Follow the instructions on the installation wizard to authorize the installer and proceed with the install.
4. When the installation is successful, click Close to complete the installation process.

- To check the docker installation
  - docker --version
  - docker run hello-world

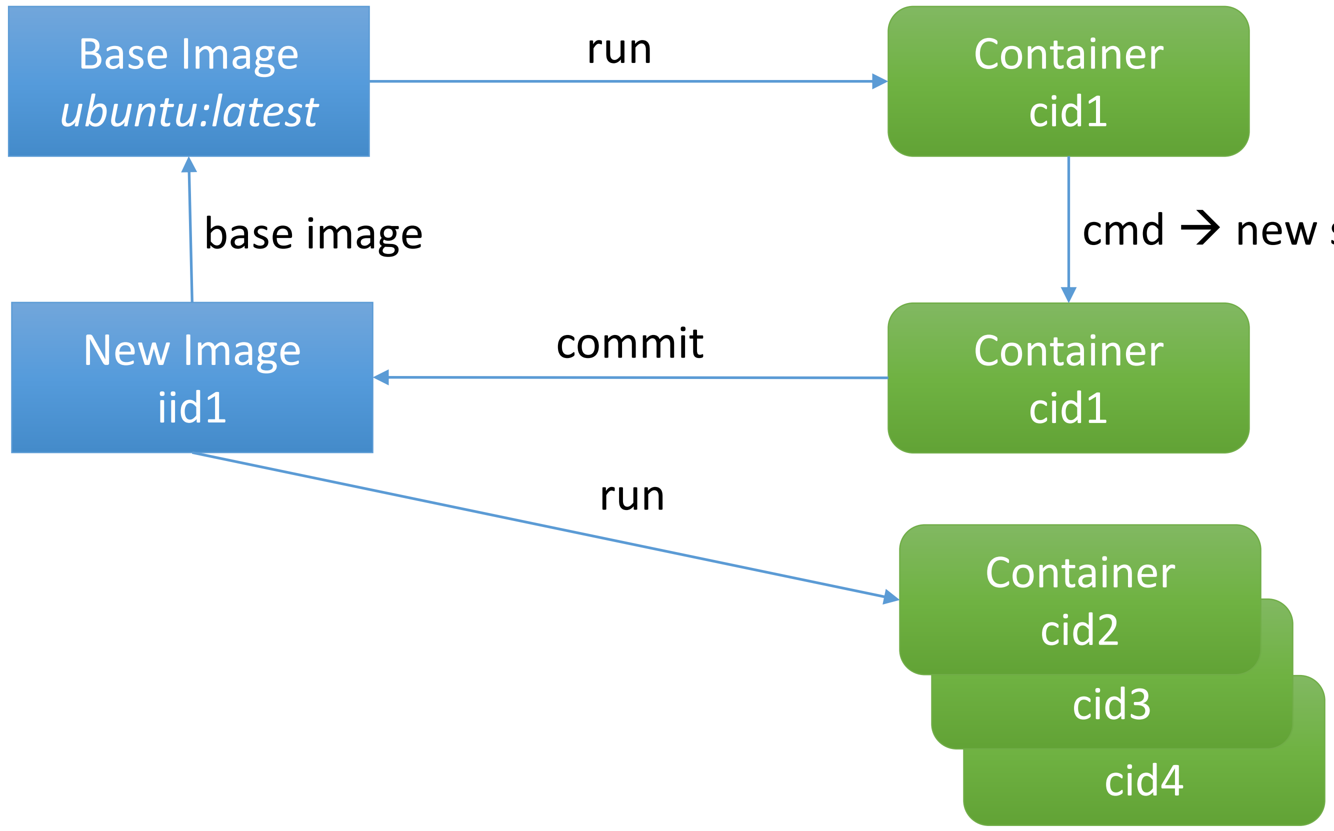
Terminology - Image

- Persisted snapshot that can be run
  - *images:* List all local images
  - *run*: Create a container from an image and execute a command in it
  - *tag*: Tag an image
  - *pull*: Download image from repository
  - *rmi*: Delete a local image
    - This will also remove intermediate images if no longer used

Terminology - Container

- Runnable instance of an image
  - *ps:* List all running containers
  - *ps –a*: List all containers (incl. stopped)
  - *top*: Display processes of a container
  - *start*: Start a stopped container
  - *stop*: Stop a running container
  - *pause*: Pause all processes within a container
  - *rm*: Delete a container
  - *commit*: Create an image from a container

Image vs. Container



Hands-on

- docker pull ubuntu
- docker pull ubuntu:20.04
- docker images
    - ✓ This generates a list of images known to Docker on your machine
- docker run -ti --rm ubuntu /bin/bash
- docker run -ti ubuntu /bin/bash
    - ✓ This executes an image.
    - ✓ An executing image is called a “container”.
    - ✓ You are now inside the container.
    - ✓ Execute “ls”.
    - ✓ A directory structure is set up but only a bare bones OS has been loaded
    - ✓ automatically removes the container when it exits

Running Linux Commands on Ubuntu Docker

- cat /etc/os-**release**
- apt **update** && apt **install** lsb-core

List the containers

- docker ps

Summary of docker commands

- docker build : build docker image from Dockerfile
- docker run : run docker image
- docker logs : show log data for a running or stopped container
- docker ps : list running docker containers (analogous to ps)
- docker ps –a : list all containers including not running
- docker images : list all images on the local volume
- docker rm : remove/delete a container | docker rmi : remove/delete an image
- docker tag : name a docker image
- docker login : login to registry

Docker and container: Hands-on

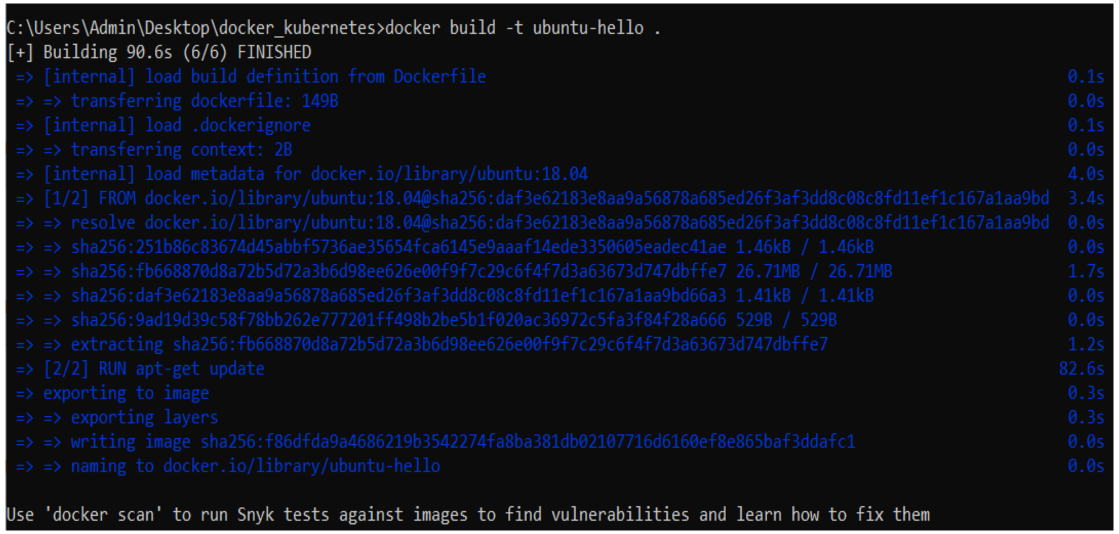
- Install software on container
  - apt-get update
  - apt-get install vim
  - apt-get install nodejs
  - apt-get install npm
  - This installs the software you will use during this session and exits the container
- docker ps –a
  - This generates a list of all of the containers that have been run.
- docker commit <old_container> <new_name>
  - Creates an image with the name <new_name>
- docker images
- run –i –t <new_name>
- Ctrl+d
  - to exit container
- Docker restart <container_id> to restart the container

Building Container Images

- What is a Docker File?
- A Dockerfile is a text document that contains all the commands you would normally execute manually in order to build a Docker image. Docker can build images automatically by reading the instructions from a Dockerfile.
- Can be versioned in a version control system like Git or SVN, along with all dependencies
- Docker Hub can automatically build images based on dockerfiles on Github
- We all know the importance of dockerfile in creating an efficient and flexible Docker Image. A dockerfile contains a set of instructions that are executed step by step when you use the docker build command to build the docker image. It contains certain instructions and commands that decides the structure of your image, the amount of time taken to build the image, contains instructions related to docker build context, contains information related to the packages and libraries to be installed in the container and many more. Hence, it becomes very important to create an efficient, reusable, clean dockerfile as it contains the blueprint of the image that you will build.

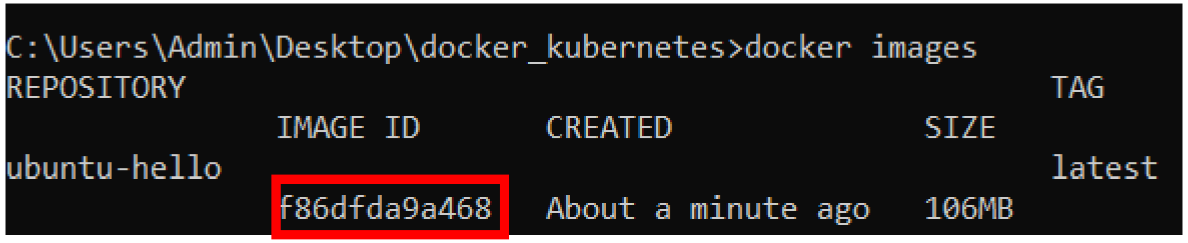
Dockerfile Example

```
# getting base image from ubuntu

FROM ubuntu:18.04

RUN apt-get update

CMD ["echo", "HEY from docker image"]
```



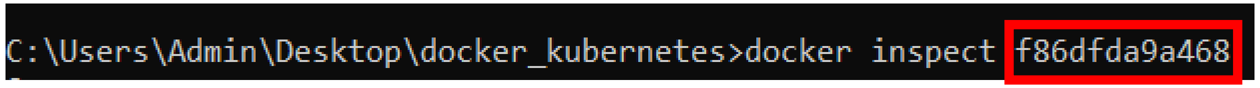
```
C:\Users\Admin\Desktop\docker_kubernetes>docker build -t ubuntu-hello .
[+] Building 90.6s (6/6) FINISHED
 => [internal] load build definition from Dockerfile                                                        0.1s
 => => transferring dockerfile: 149B                                                                        0.0s
 => [internal] load .dockerignore                                                                           0.1s
 => => transferring context: 2B                                                                             0.0s
 => [internal] load metadata for docker.io/library/ubuntu:18.04                                             4.0s
 => [1/2] FROM docker.io/library/ubuntu:18.04@sha256:daf3e62183e8aa9a56878a685ed26f3af3dd8c08c8fd11ef1c167a1aa9bd  3.4s
 => => resolve docker.io/library/ubuntu:18.04@sha256:daf3e62183e8aa9a56878a685ed26f3af3dd8c08c8fd11ef1c167a1aa9bd  0.0s
 => => sha256:251b86c83674d45abbf5736ae35654fca6145e9aaaf14ede3350605eadec41ae 1.46kB / 1.46kB              0.0s
 => => sha256:fb668870d8a72b5d72a3b6d98ee626e00f9f7c29c6f4f7d3a63673d747dbffe7 26.71MB / 26.71MB            1.7s
 => => sha256:daf3e62183e8aa9a56878a685ed26f3af3dd8c08c8fd11ef1c167a1aa9bd66a3 1.41kB / 1.41kB              0.0s
 => => sha256:9ad19d39c58f78bb262e777201ff498b2be5b1f020ac36972c5fa3f84f28a666 529B / 529B                  0.0s
 => => extracting sha256:fb668870d8a72b5d72a3b6d98ee626e00f9f7c29c6f4f7d3a63673d747dbffe7                   1.2s
 => [2/2] RUN apt-get update                                                                               82.6s
 => exporting to image                                                                                      0.3s
 => => exporting layers                                                                                     0.3s
 => => writing image sha256:f86dfda9a4686219b3542274fa8ba381db02107716d6160ef8e865baf3ddafc1                 0.0s
 => => naming to docker.io/library/ubuntu-hello                                                             0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

docker build -t ubuntu-hello .



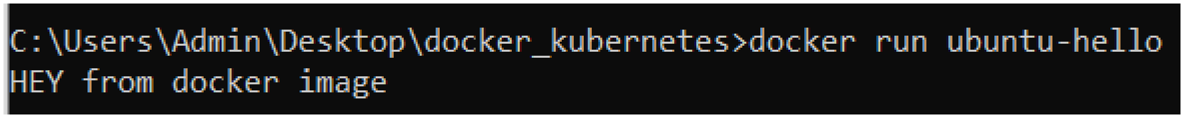
```
C:\Users\Admin\Desktop\docker_kubernetes>docker images
REPOSITORY                                                              TAG
              IMAGE ID       CREATED              SIZE
ubuntu-hello                                                            latest
              f86dfda9a468   About a minute ago   106MB
```

docker images



```
C:\Users\Admin\Desktop\docker_kubernetes>docker inspect f86dfda9a468
```

docker inspect [imageId]



```
C:\Users\Admin\Desktop\docker_kubernetes>docker run ubuntu-hello
HEY from docker image
```

docker run ubuntu-hello .

Important keywords used in Dockerfile

- **FROM**
- You must have noticed that almost all the dockerfile starts with the FROM command. The FROM command is of the form −

```
FROM <image name>:<tag name>
```

- A FROM command allows you to create a base image such as an operating system, a programming language, etc. All the instructions executed after this command take place on this base image. It contains an image name and an optional tag name. If you already have the base image pulled previously in your local machine, it doesn't pull a new one. There are several pre published docker base images available in the docker registry. You can also push your own customized base image inside the docker registry.
- Examples of FROM instruction along with different base images are −

```
FROM ubuntu
FROM centos
FROM python:3
```

Important keywords used in Dockerfile

- **RUN**
- A RUN instruction is used to run specified commands. You can use several RUN instructions to run different commands. But it is an efficient approach to combine all the RUN instructions into a single one.
- Each RUN command creates a new cache layer or an intermediate image layer and hence chaining all of them into a single line, becomes efficient. However, chaining multiple RUN instructions could lead to cache bursts as well.
- Some example of RUN commands are:

```
RUN apt−get −y install vim

RUN apt−get −y update
```

You can chain multiple RUN instructions in the following way:

```
RUN apt−get −y update \

&& apt−get −y install firefox \

&& apt−get −y install vim
```

Important keywords used in Dockerfile

- **CMD**
- If you want to run a docker container by specifying a default command that gets executed for all the containers of that image by default, you can use a CMD command. In case you specify a command during the docker run command, it overrides the default one. Specifying more than one CMD instructions, will allow only the last one to get executed.
- Example of a CMD command –

```
CMD echo "Welcome to MSRIT"
```

- If you specify the above line in the dockerfile and run the container using the following command without specifying any arguments, the output will be “Welcome to MSRIT”

```
docker run –it <image_name>
```

Output – “Welcome to MSRIT”

- In case you try to specify any other arguments such as /bin/bash, etc, the default CMD command will be overridden.

Important keywords used in Dockerfile

- **ENTRYPOINT**
- The difference between ENTRYPOINT and CMD is that, if you try to specify default arguments in the docker run command, it will not ignore the ENTRYPOINT arguments. The exec form of an ENTRYPOINT command is −
- ENTRYPOINT [“<executable-command>”, “<parameter 1>”, “<parameter 2>”, ….]
- If you have used the exec form of the ENTRYPOINT instruction, you can also set additional parameters with the help of CMD command. For example −

```
ENTRYPOINT ["/bin/echo", "Welcome to MSRIT"]
CMD ["Hello World!"]
```

- Running docker run command without any argument would output −

Welcome to MSRIT Hello World!

- If you specify any other CLI arguments, “Hello World!” will get overridden.

Important keywords used in Dockerfile

- **WORKDIR :**You can specify your working directory inside the container using the WORKDIR instruction. Any other instruction after that in the dockerfile, will be executed on that particular working directory only.

  For example,
  ```
  WORKDIR /usr/src/app
  ```
  Sets the working directory to /usr/src/app inside the container.
- **COPY:** This instruction allows you to copy a directory from your local machine to the docker container. This would copy all the files inside the directory ∽/Desktop/myapp in your local machine to your current working directory inside the docker container. For example,
  ```
  FROM ubuntu
  WORKDIR /usr/src/app
  COPY ∽/Desktop/myapp .
  ```
- **LABEL** You can use a LABEL instruction to add description or meta data for a docker image. Its a key−value pair.
  ```
  LABEL description="This is a sample image"
  ```

Important keywords used in Dockerfile

- **ADD**
- Similar to COPY instruction, you can use ADD to copy files and folders from your local machine to docker containers. However, ADD also allows you to copy files from a URL as well as a tar file.
- For example, `ADD ~/Desktop/myapp/practice.tar.gz /usr/src/app` Would copy all the contents inside the tar file to /usr/src/app inside the container.
- `ADD <URL such as a github url> <Destination path inside the container>` This command would copy all the files inside the github url to the destination.
- **EXPOSE**
- The EXPOSE instruction inside the dockerfile informs that the container is listening to the specified port in the network. The default protocol is TCP. Example `EXPOSE 8080`
- Will map the 8080 port to the container. You can use the −p flag with the docker run command to make the container listen to another container or the host machine.

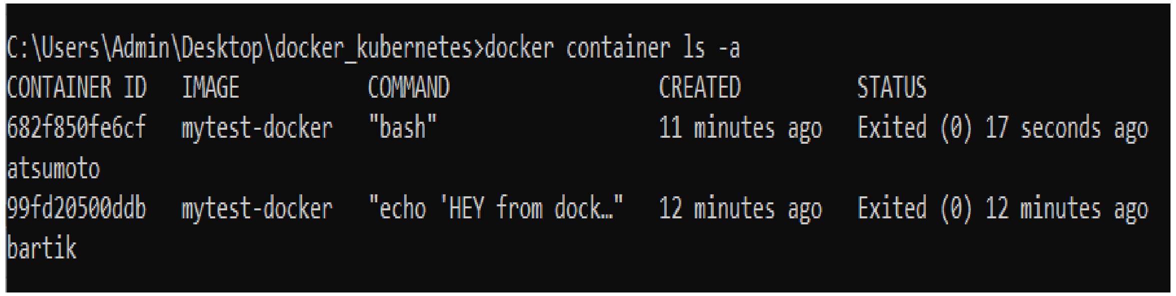
Running commands together in a Docker file.

Dockerfile

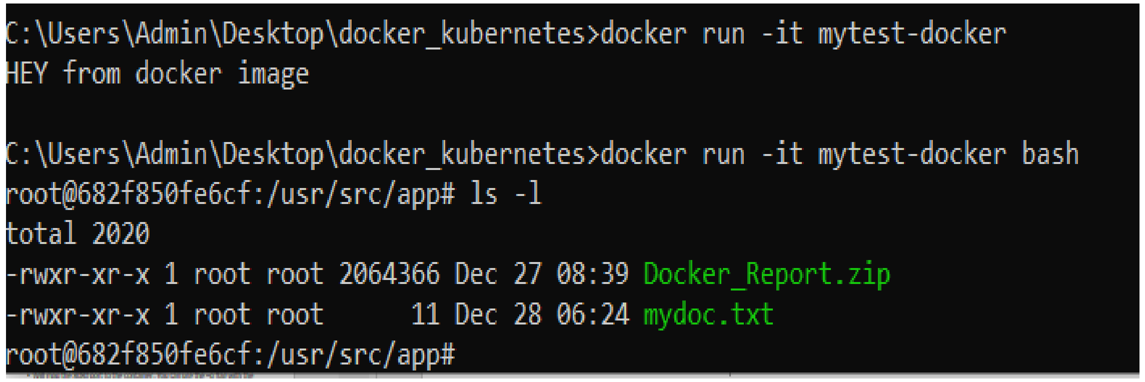
```
# getting base image from ubuntu
FROM ubuntu:18.04
RUN apt-get update
RUN ["apt-get", "install", "-y", "vim"]
LABEL description="This is a sample image"
CMD ["echo", "HEY from docker image"]

WORKDIR /usr/src/app

#copy mydoc.txt  inside the in your local machine to your
#current working directory inside the docker container.
COPY mydoc.txt .

ADD Docker_Report.zip  /usr/src/app
```

Commands to Execute

```
docker build -t mytest-docker .

docker run -it mytest-docker

docker run -it mytest-docker bash

docker images

docker container ls -a
```



```
C:\Users\Admin\Desktop\docker_kubernetes>docker container ls -a
CONTAINER ID   IMAGE           COMMAND                  CREATED          STATUS
682f850fe6cf   mytest-docker   "bash"                   11 minutes ago   Exited (0) 17 seconds ago
atsumoto
99fd20500ddb   mytest-docker   "echo 'HEY from dock…"   12 minutes ago   Exited (0) 12 minutes ago
bartik
```



```
C:\Users\Admin\Desktop\docker_kubernetes>docker run -it mytest-docker
HEY from docker image

C:\Users\Admin\Desktop\docker_kubernetes>docker run -it mytest-docker bash
root@682f850fe6cf:/usr/src/app# ls -l
total 2020
-rwxr-xr-x 1 root root 2064366 Dec 27 08:39 Docker_Report.zip
-rwxr-xr-x 1 root root      11 Dec 28 06:24 mydoc.txt
root@682f850fe6cf:/usr/src/app#
```

Try vim command inside shell

Multi-stage builds

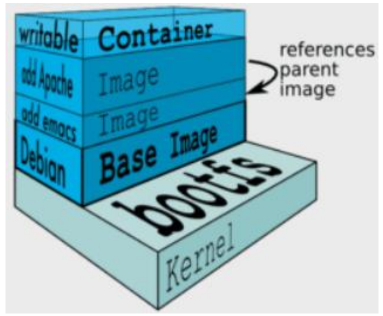
- One of the most anticipated features of Docker’s recent 17.05 release is multi-stage builds.
- Multi-stage builds are a method of organizing a Dockerfile to minimize the size of the final container, improve run time performance, allow for better organization of Docker commands and files, and provide a standardized method of running build actions.
- A multi-stage build is done by creating different sections of a Dockerfile, each referencing a different base image. This allows a multi-stage build to fulfill a function previously filled by using multiple docker files, copying files between containers, or running different pipelines.
- With a docker multistage build, you can create better container images.
- The real reasons behind a docker multistage build are **security** and **efficiency**.
    - Security because in the end you copy your binary files, and only your binary files, into the final environment, reducing the surface of attacks (that is, the potential features that a hacker might exploit).
    - Efficiency because in this way you can ship a more lightweight version of the system, which will save a significant amount of RAM, especially when operating at scale with many containers. If you save 100MB per container, as soon as you spin 10 containers you would have saved 1GB of RAM.

Before multi-stage builds

- One of the most challenging things about building images is keeping the image size down. Each RUN, COPY, and ADD instruction in the Dockerfile adds a layer to the image, and you need to remember to clean up any artifacts you don’t need before moving on to the next layer. To write a really efficient Dockerfile, you have traditionally needed to employ shell tricks and other logic to keep the layers as small as possible and to ensure that each layer has the artifacts it needs from the previous layer and nothing else.
- It was actually very common to have one Dockerfile to use for development (which contained everything needed to build your application), and a slimmed-down one to use for production, which only contained your application and exactly what was needed to run it. This has been referred to as the “builder pattern”. Maintaining two Dockerfiles is not ideal.

Image Construction

- Docker images are made up of a series of union file system layers, the technical storage underpinnings of which is determined by the storage driver. Each executed instruction within a Dockerfile resides in its own layer of the final Docker image. The first layer will always be the inherited base image, as defined by the FROM instruction. Each subsequent layer will store your application, files, etc.
- How layers are constructed affects the build time, storage, and overall performance of your application. The more instructions we have (for example RUN, COPY, ADD) in our Dockerfile, the greater the final size of the image.
- There are many other things that are responsible for increasing the size of the image, like the context, base image, unnecessary dependencies, packages, and a number of instructions.
- With the new multi-stage build feature, we can concisely define multiple build steps and arrive at a final Docker image that meets best practices for production.
- Use alpine images wherever possible
    - Alpine image is a minimal Docker image based on Alpine Linux with a complete package index and only 5 MB in size!



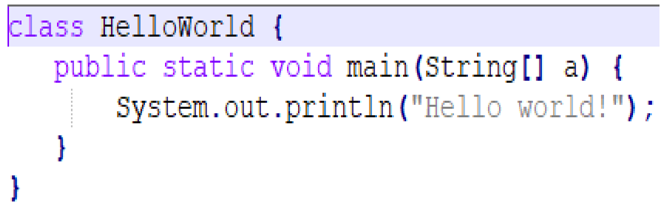
How to do a Docker Multistage Build

- To do a docker multistage build, you simply have to use one or more FROM statement in your dockerfile. Each new FROM statement indicates that a new step is starting. You should also pair the FROM statement with AS so that you can give a name to each step, you will need it later.
- Then, you can run all the normal commands in each step. Additionally, for the COPY command you have the –from which allows you to copy data from a previous stage, rather than from the directory where you are building the container.
- In this way, you can build on a stage, have the files there, and then copy them from a stage to the next. That is the whole purpose of docker multistage build.

Multi-Stage Docker Build Examples

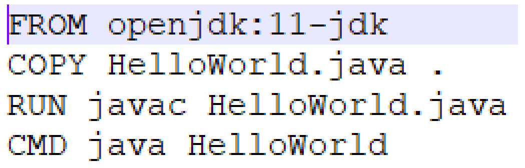
Java Example: To understand the concept of Multi-stage Docker builds better, let us consider a simple Java Hello World application.

HelloWorld.java



```
class HelloWorld {
   public static void main(String[] a) {
      System.out.println("Hello world!");
   }
}
```

Dockerfile



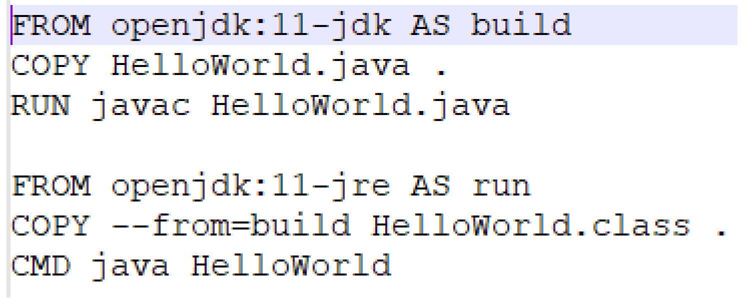
```
FROM openjdk:11-jdk
COPY HelloWorld.java .
RUN javac HelloWorld.java
CMD java HelloWorld
```

to_execute

Build the image with the following command

```
docker build -t helloworld:huge .
```

Let's modify our Dockerfile with the following content to show how **multi-stage Docker** build works.



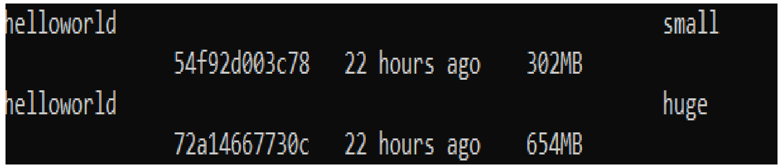
```
FROM openjdk:11-jdk AS build
COPY HelloWorld.java .
RUN javac HelloWorld.java

FROM openjdk:11-jre AS run
COPY --from=build HelloWorld.class .
CMD java HelloWorld
```

Build the image with the following command

```
docker build -t helloworld:small .
```

Now, let's compare both images. Check the images created with the following command,

```
docker images
```



```
helloworld                                          small
                54f92d003c78   22 hours ago    302MB
helloworld                                          huge
                72a14667730c   22 hours ago    654MB
```

You can see the difference in size between the two images. This way, you can separate the build and runtime environments in the same Dockerfile. Use build environment as a dependency [COPY --from=build HelloWorld.class .] while creating the Dockerfile with the approach of multi-stage docker build. This will help minimize the size of Docker images.

Node.Js Example

get_ex1.js

```
var express = require('express');
var app = express();

app.get('/index.html', function (req, res) {
  res.sendFile( __dirname + "/" + "index.html" );

})
app.get('/process_get', function (req, res) {
response = {
      first_name:req.query.first_name,
      last_name:req.query.last_name
  };
  console.log(response);
                    console.log("Sent data are (GET): first name
:"+req.query.first_name+" and last name :"+req.query.last_name);
  //res.end(JSON.stringify(response));
  res.end("Sent data are (GET): first name :"+req.query.first_name+" and last
name :"+req.query.last_name);
})
var server = app.listen(8080, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)
})
```

index.html

```
<html>
<body>
<form action="process_get" method="get">
First Name: <input type="text" name="first_name">  <br>
Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
</form>
<a href="form.html">Google</a>
<a href="/">welcome</a>
</body>
</html>
```

Node.Js Example

Dockerfile (normal)

```
FROM node:14-alpine
ADD . /app
WORKDIR /app
COPY package.json .
RUN npm install --production
COPY . .
EXPOSE 8080
CMD [ "node", "get_ex1.js" ]
```

```
docker build -t helloworld:nodebig .
docker run -p 3002:8080 -d helloworld:nodebig
```

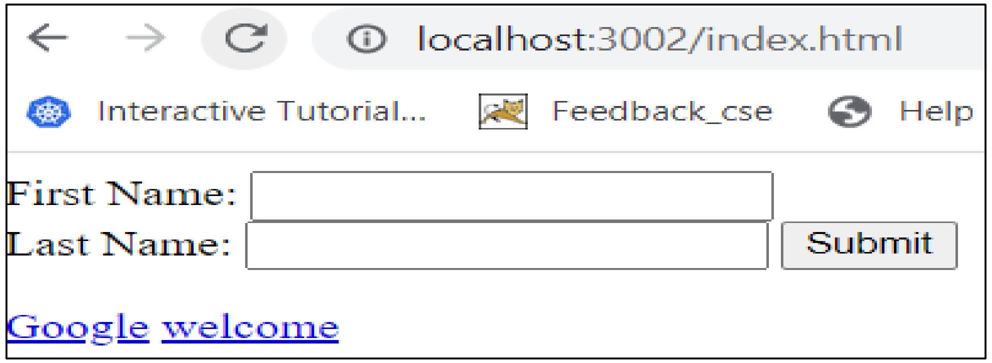
Docker file (multi-stage build)

```
FROM node:14-alpine as base
ADD . /app
WORKDIR /app
COPY package.json .
RUN npm install --production

FROM node:10-alpine
WORKDIR /app
COPY --from=base /app /app
EXPOSE 8080
CMD [ "node", "get_ex1.js" ]
```

```
docker build -t helloworld:nodesmall .
docker run -p 3003:8080 -d helloworld:nodesmall
```

- Now, compare the image sizes. One with the usual Dockerfile is 48.81 MB, and the other created with a multi-stage Docker build. The image created by the multi-stage Docker build approach is more optimized.

To check output on the browser:
http://localhost: 3002 (3003)

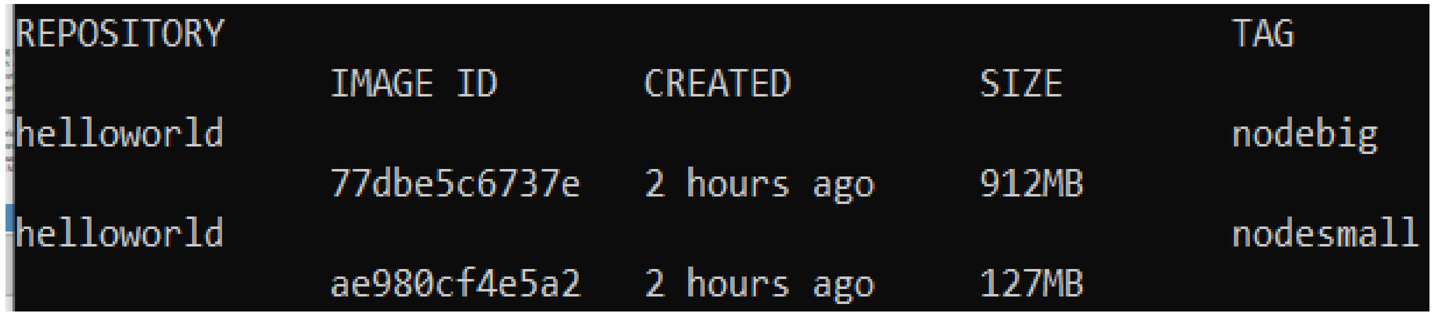


localhost:3002/index.html

Interactive Tutorial... Feedback_cse Help

First Name:
Last Name:
Submit

Google welcome



```
REPOSITORY                                                   TAG
             IMAGE ID       CREATED           SIZE
helloworld                                                   nodebig
             77dbe5c6737e   2 hours ago       912MB
helloworld                                                   nodesmall
             ae980cf4e5a2   2 hours ago       127MB
```

Use multi-stage builds

- With multi-stage builds, you use multiple FROM statements in your Dockerfile. Each FROM instruction can use a different base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, leaving behind everything you don’t want in the final image. To show how this works, let’s adapt the Dockerfile from the previous section to use multi-stage builds.

Storing and Distributing Images

- **Docker Hub**

- Docker Hub is a service provided by Docker for finding and sharing container images with your team. It’s the world’s largest repository of container images with an array of content sources including container community developers, open source projects and independent software vendors (ISV) building and distributing their code in containers.
- Users get access to free public repositories for storing and sharing images or can choose a subscription plan for private repositories.
- Docker Hub provides the following major features:
  - Repositories: Push and pull container images.
  - Teams & Organizations: Manage access to private repositories of container images.
  - Docker Official Images: Pull and use high-quality container images provided by Docker.
  - Docker Verified Publisher Images: Pull and use high- quality container images provided by external vendors.
  - Builds: Automatically build container images from GitHub and Bitbucket and push them to Docker Hub.
  - Webhooks: Trigger actions after a successful push to a repository to integrate Docker Hub with other services.
  - Users looking for a zero maintenance, ready-to-go solution are encouraged to head-over to the Docker Hub, which provides a free-to-use, hosted Registry, plus additional features (organization accounts, automated builds, and more).

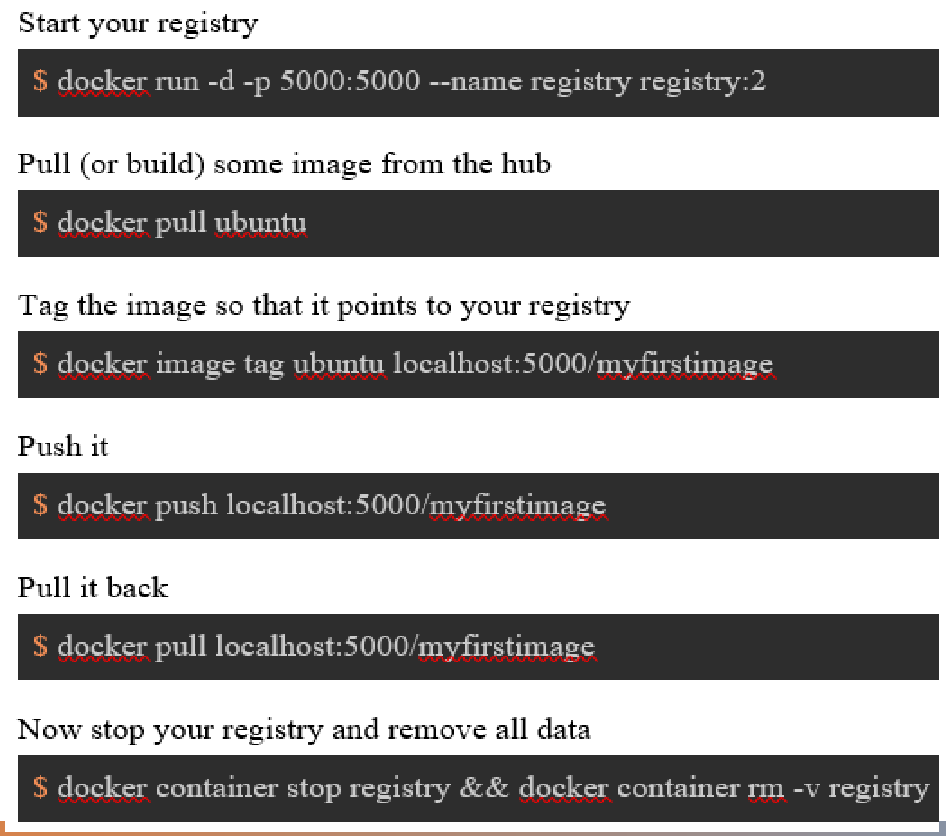
Storing and Distributing Images

- **Docker Store** :Docker Store and Docker Cloud are now part of Docker Hub, providing a single experience for finding, storing and sharing container images.
- **Docker Registry**
- The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images. The Registry is open-source, under the permissive Apache license. You can find the source code on GitHub.
- You should use the Registry if you want to:
  - tightly control where your images are being stored
  - fully own your images distribution pipeline
  - integrate image storage and distribution tightly into your in-house development workflow

Docker Registry

- The Registry is compatible with Docker engine version 1.6.0 or higher.

Basic commands



Start your registry

```
$ docker run -d -p 5000:5000 --name registry registry:2
```

Pull (or build) some image from the hub

```
$ docker pull ubuntu
```

Tag the image so that it points to your registry

```
$ docker image tag ubuntu localhost:5000/myfirstimage
```

Push it

```
$ docker push localhost:5000/myfirstimage
```

Pull it back

```
$ docker pull localhost:5000/myfirstimage
```

Now stop your registry and remove all data

```
$ docker container stop registry && docker container rm -v registry
```

```
get a list of images on docker registry v2
      $http://localhost:5000/v2/_catalog
To login to Registry
      $docker login localhost:5000
```

Docker Trusted Registry

- Docker Trusted Registry (DTR) is the enterprise-grade image storage solution from Docker. You install it behind your firewall so that you can securely store and manage the Docker images you use in your applications.
- Docker trusted registry or simply Docker registry is an enterprise offering from Docker. The most common terminology that you will hear with Docker Enterprise Edition is DTR and UCP (universal control plane).
- In order for DTR to work UCP has to be installed and for UCP to be installed you would need Docker Enterprise Edition. Once you install Docker EE you can get a free license from DockerHub.

DTR Features

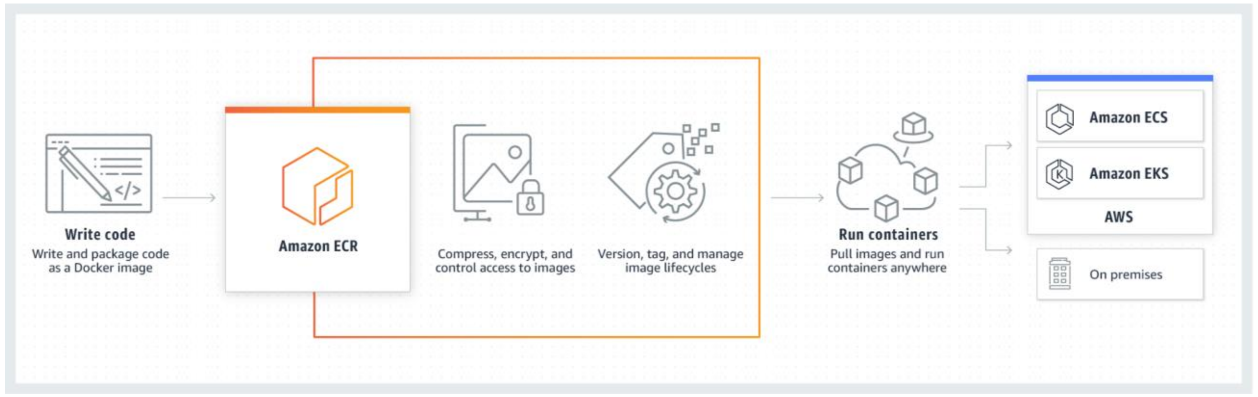
- **Image and job management:** DTR can be installed on any platform where you can store your Docker images securely, behind your firewall. DTR has a user interface that allows authorized users in your organization to browse Docker images and review repository events. It even allows you to see what Dockerfile lines were used to produce the image and, if security scanning is enabled, to see a list of all of the software installed in your images.
- **Availability:** DTR is highly available as it has multiple replicas of containers in case anything fails.
- **Efficiency:** DTR has this ability to clean the unreferenced manifests and cache the images as well for faster pulling of images.
- **Built-in access control :** STR has great authentication mechanisms like RBAC , LDAP sync. It uses the same authentication as of UCP.
- **Security scanning:** Image Scanning is built in feature provided out of the box by DTR.
- **Image signing:** DTR has built in Notary, you can use Docker Content Trust to sign snd verify images.

Azure container Registry

- Azure Container Registry allows you to build, store, and manage container images and artifacts in a private registry for all types of container deployments. Use Azure container registries with your existing container development and deployment pipelines. Use Azure Container Registry Tasks to build container images in Azure on-demand, or automate builds triggered by source code updates, updates to a container's base image, or timers.

Amazon ECR

- Push container images to Amazon ECR without installing or scaling infrastructure, and pull images using any management tool.
- Share and download images securely over Hypertext Transfer Protocol Secure (HTTPS) with automatic encryption and access controls.
- Access and distribute your images faster, reduce download times, and improve availability using a scalable, durable architecture.
- **How it works :**Amazon ECR is a fully managed container registry offering high-performance hosting, so you can reliably deploy application images and artifacts anywhere.


Write code
Write and package code as a Docker image
Amazon ECR
Compress, encrypt, and control access to images
Version, tag, and manage image lifecycles
Run containers
Pull images and run containers anywhere
Amazon ECS
Amazon EKS
AWS
On premises

Managing Containers

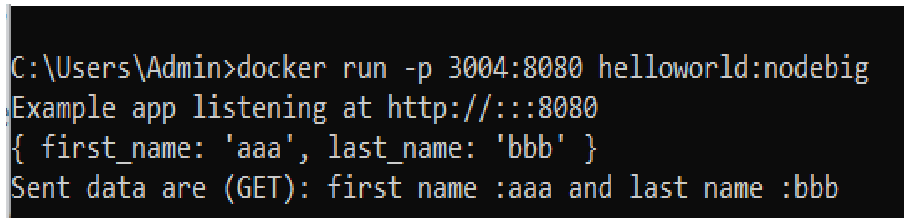
- A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
- By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container’s network, storage, or other underlying subsystems are from other containers or from the host machine.
- A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Run a Container in the attached/Detached Mode

- There are two ways of running a container – in **attached mode** (in the foreground) or in **detached mode** (in the background).
- In the attached mode, Docker can start the process in the container and attach the console to the process's standard input, standard output, and standard error.
- In a detached mode container runs in the background and returns a terminal to perform other tasks. However, sometimes we need to connect to a container to perform a few tasks.
- If you want to keep the container and current terminal session separate, you can run the container in the background using the -d attribute. Using detached mode also allows you to close the opened terminal session without stopping the container.
- The command for running a container in the background is: docker container run -d [docker_image]

Run node application in attached mode

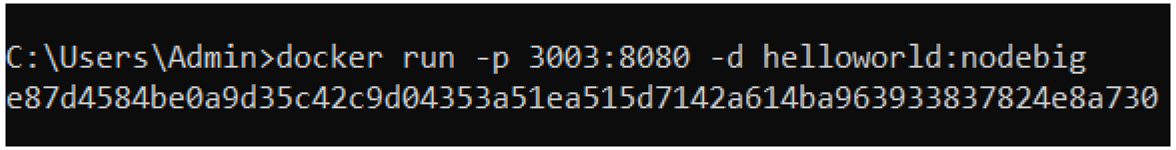
docker run -p 3004:8080 helloworld:nodebig



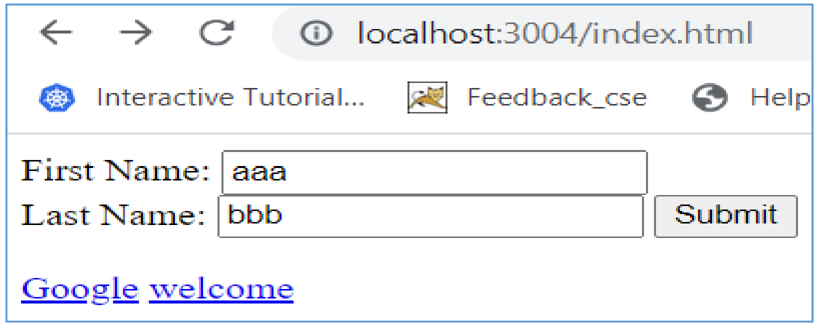
```
C:\Users\Admin>docker run -p 3004:8080 helloworld:nodebig
Example app listening at http://:::8080
{ first_name: 'aaa', last_name: 'bbb' }
Sent data are (GET): first name :aaa and last name :bbb
```

Run node application in detached mode

docker run -p 3003:8080 –d helloworld:nodebig



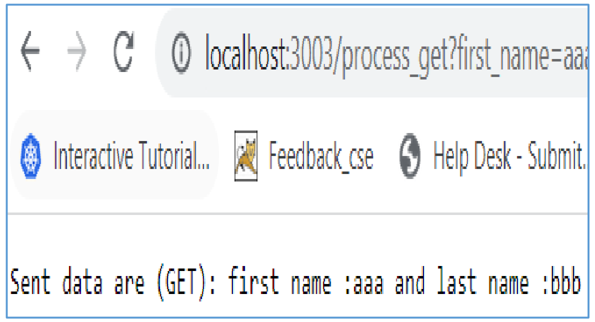
```
C:\Users\Admin>docker run -p 3003:8080 -d helloworld:nodebig
e87d4584be0a9d35c42c9d04353a51ea515d7142a614ba963933837824e8a730
```



localhost:3004/index.html

First Name: aaa
Last Name: bbb

Google welcome

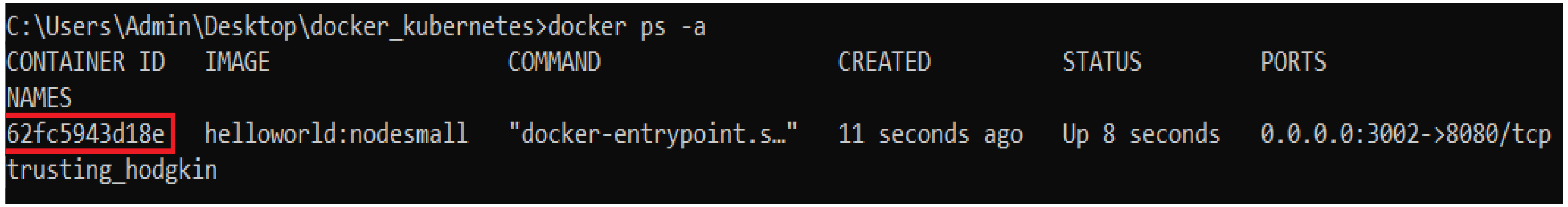


localhost:3003/process_get?first_name=aa

Sent data are (GET): first name :aaa and last name :bbb

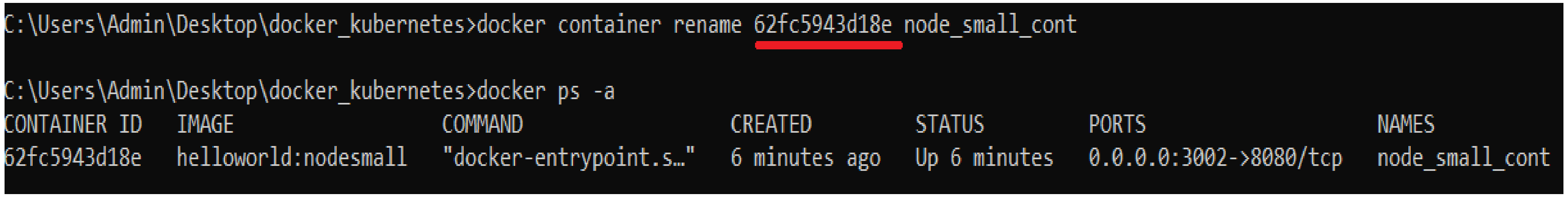
docker container rename

- Get controller ID



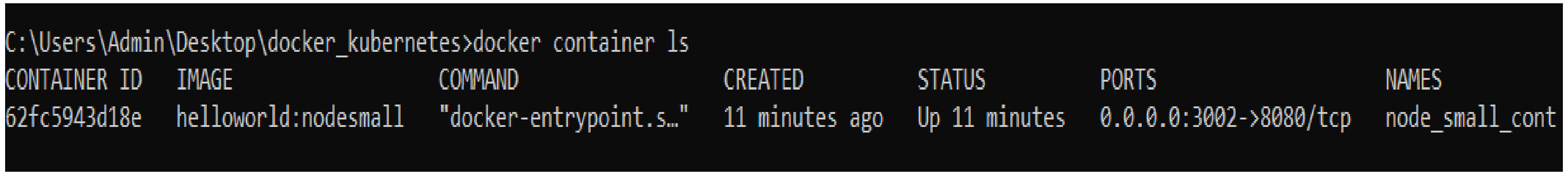
```
C:\Users\Admin\Desktop\docker_kubernetes>docker ps -a
CONTAINER ID   IMAGE                 COMMAND                  CREATED          STATUS         PORTS
NAMES
62fc5943d18e   helloworld:nodesmall   "docker-entrypoint.s…"   11 seconds ago   Up 8 seconds   0.0.0.0:3002->8080/tcp
trusting_hodgkin
```

- Rename the controller ID with name node_small_cong



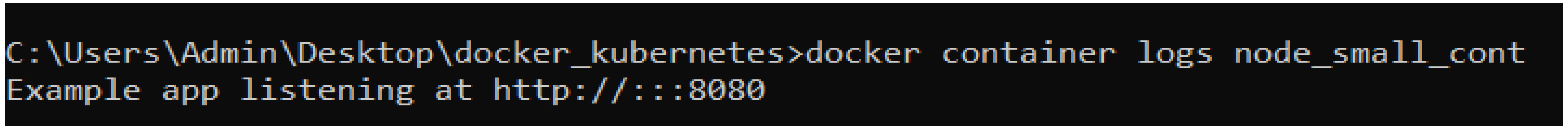
```
C:\Users\Admin\Desktop\docker_kubernetes>docker container rename 62fc5943d18e node_small_cont

C:\Users\Admin\Desktop\docker_kubernetes>docker ps -a
CONTAINER ID   IMAGE                 COMMAND                  CREATED         STATUS         PORTS                    NAMES
62fc5943d18e   helloworld:nodesmall   "docker-entrypoint.s…"   6 minutes ago   Up 6 minutes   0.0.0.0:3002->8080/tcp   node_small_cont
```

List Containers



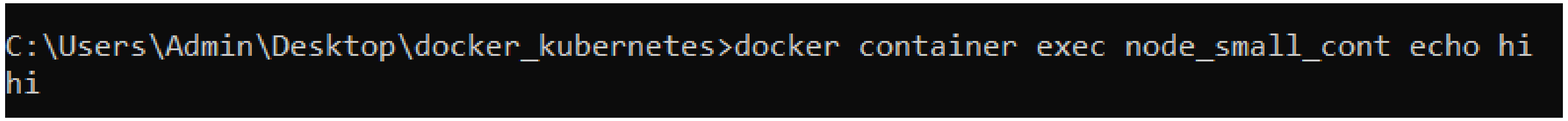
```
C:\Users\Admin\Desktop\docker_kubernetes>docker container ls
CONTAINER ID   IMAGE               COMMAND                  CREATED          STATUS          PORTS                    NAMES
62fc5943d18e   helloworld:nodesmall   "docker-entrypoint.s…"   11 minutes ago   Up 11 minutes   0.0.0.0:3002->8080/tcp   node_small_cont
```

Fetch the logs of a container



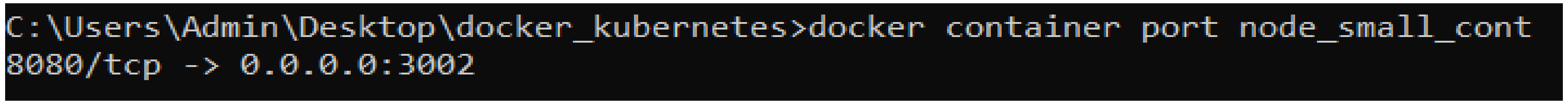
```
C:\Users\Admin\Desktop\docker_kubernetes>docker container logs node_small_cont
Example app listening at http://:::8080
```

Run a command in a running container



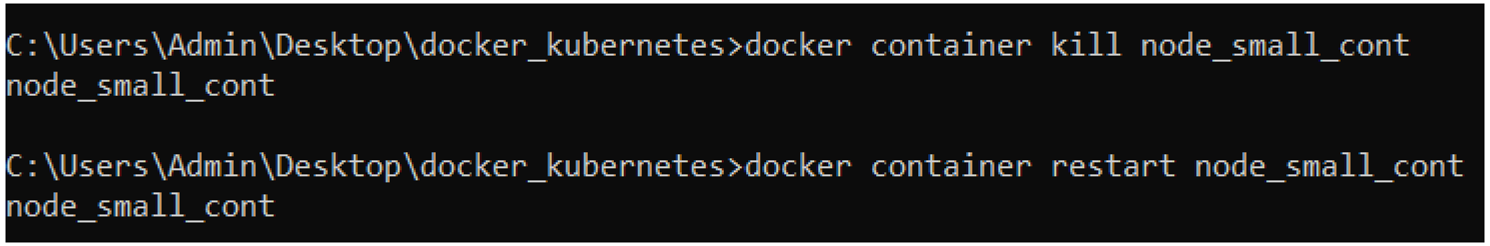
```
C:\Users\Admin\Desktop\docker_kubernetes>docker container exec node_small_cont echo hi
hi
```

List port mappings or a specific mapping for the container



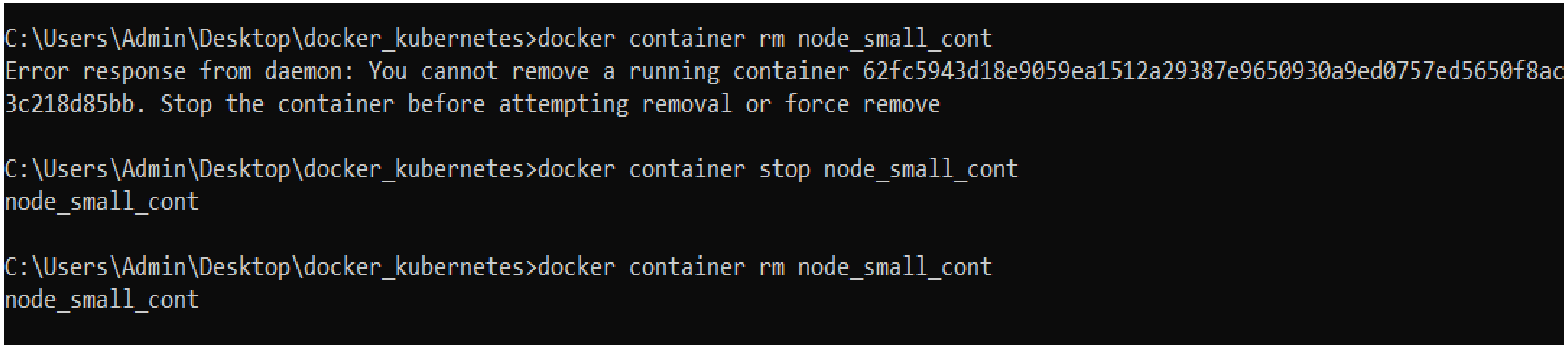
```
C:\Users\Admin\Desktop\docker_kubernetes>docker container port node_small_cont
8080/tcp -> 0.0.0.0:3002
```

Kill and restart container



```
C:\Users\Admin\Desktop\docker_kubernetes>docker container kill node_small_cont
node_small_cont

C:\Users\Admin\Desktop\docker_kubernetes>docker container restart node_small_cont
node_small_cont
```

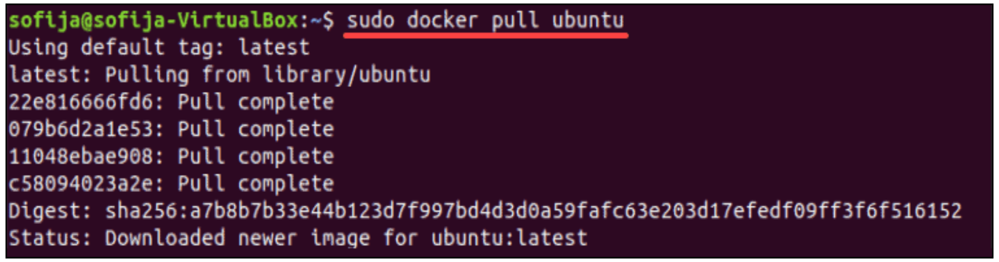
Remove container: First you have to stop the container and then remove



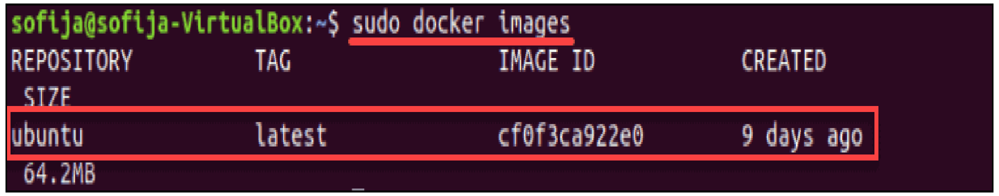
```
C:\Users\Admin\Desktop\docker_kubernetes>docker container rm node_small_cont
Error response from daemon: You cannot remove a running container 62fc5943d18e9059ea1512a29387e9650930a9ed0757ed5650f8ac
3c218d85bb. Stop the container before attempting removal or force remove

C:\Users\Admin\Desktop\docker_kubernetes>docker container stop node_small_cont
node_small_cont

C:\Users\Admin\Desktop\docker_kubernetes>docker container rm node_small_cont
node_small_cont
```

Committing Changes to Docker Image

Step 1: Pull a Docker Image

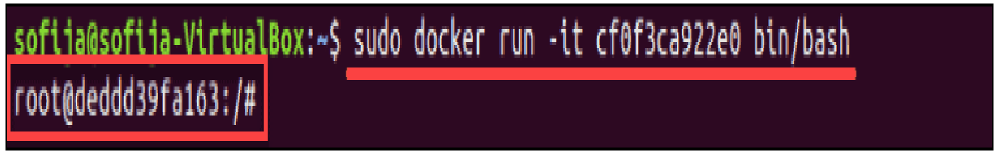


```
sofija@sofija-VirtualBox:~$ sudo docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
22e816666fd6: Pull complete
079b6d2a1e53: Pull complete
11048ebae908: Pull complete
c58094023a2e: Pull complete
Digest: sha256:a7b8b7b33e44b123d7f997bd4d3d0a59fafc63e203d17efedf09ff3f6f516152
Status: Downloaded newer image for ubuntu:latest
```



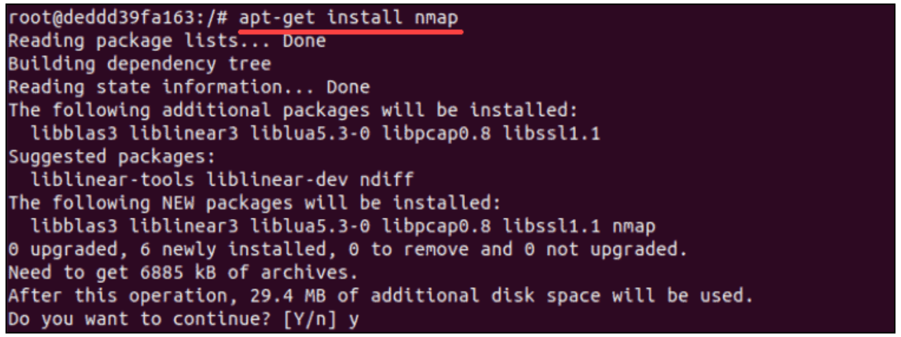
```
sofija@sofija-VirtualBox:~$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
 SIZE
ubuntu              latest              cf0f3ca922e0        9 days ago
 64.2MB
```

Step 2: Deploy the Container



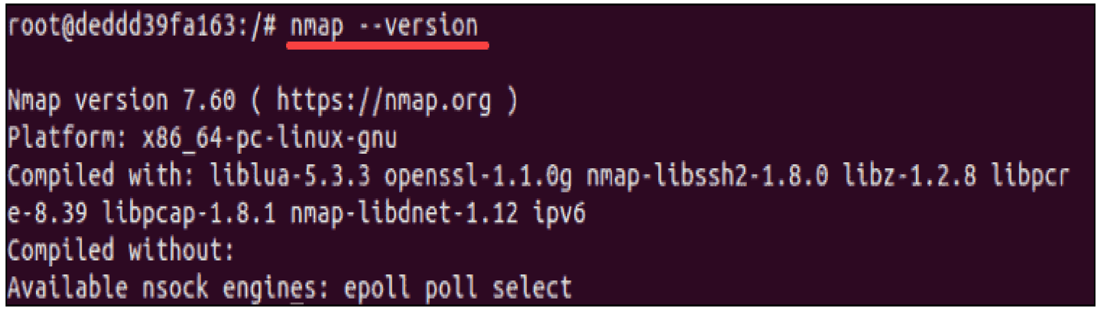
```
sofija@sofija-VirtualBox:~$ sudo docker run -it cf0f3ca922e0 bin/bash
root@deddd39fa163:/#
```

Step 3: Modify the Container



```
root@deddd39fa163:/# apt-get install nmap
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libblas3 liblinear3 liblua5.3-0 libpcap0.8 libssl1.1
Suggested packages:
  liblinear-tools liblinear-dev ndiff
The following NEW packages will be installed:
  libblas3 liblinear3 liblua5.3-0 libpcap0.8 libssl1.1 nmap
0 upgraded, 6 newly installed, 0 to remove and 0 not upgraded.
Need to get 6885 kB of archives.
After this operation, 29.4 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

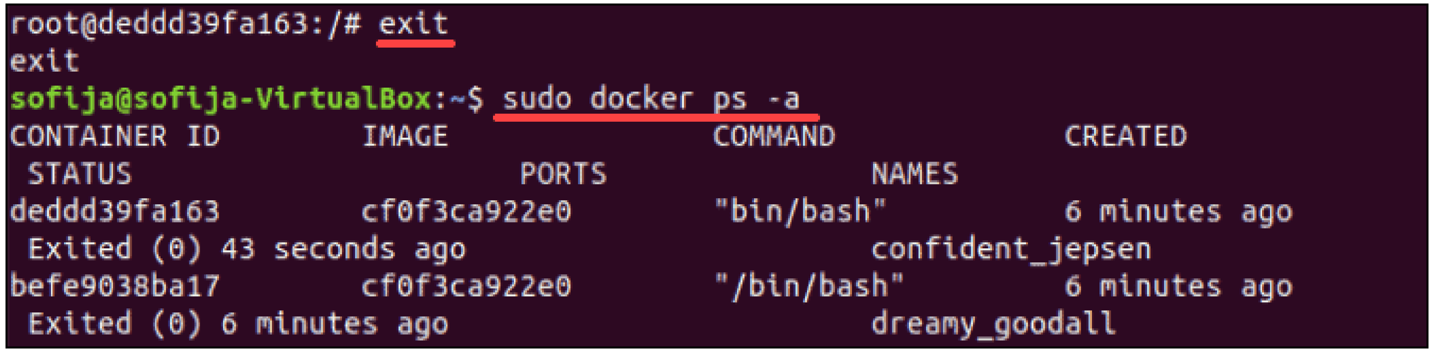
You can verify the installation by running:



```
root@deddd39fa163:/# nmap --version

Nmap version 7.60 ( https://nmap.org )
Platform: x86_64-pc-linux-gnu
Compiled with: liblua-5.3.3 openssl-1.1.0g nmap-libssh2-1.8.0 libz-1.2.8 libpcr
e-8.39 libpcap-1.8.1 nmap-libdnet-1.12 ipv6
Compiled without:
Available nsock engines: epoll poll select
```

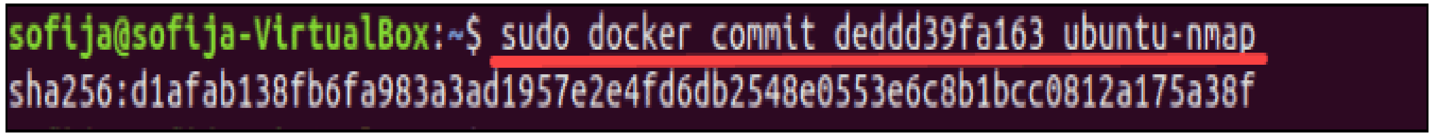
Docker commit

4. You will need the **CONTAINER ID** to save the changes you made to the existing image. Copy the ID value from the output.



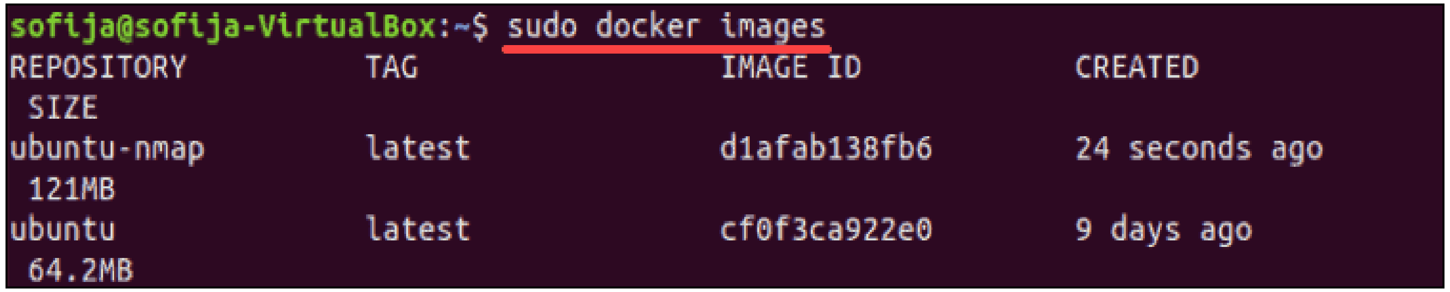
```
root@deddd39fa163:/# exit
exit
sofija@sofija-VirtualBox:~$ sudo docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
 STATUS                     PORTS               NAMES
deddd39fa163        cf0f3ca922e0        "bin/bash"          6 minutes ago
 Exited (0) 43 seconds ago                      confident_jepsen
befe9038ba17        cf0f3ca922e0        "/bin/bash"         6 minutes ago
 Exited (0) 6 minutes ago                       dreamy_goodall
```

5. Commit Changes to Image



```
sofija@sofija-VirtualBox:~$ sudo docker commit deddd39fa163 ubuntu-nmap
sha256:d1afab138fb6fa983a3ad1957e2e4fd6db2548e0553e6c8b1bcc0812a175a38f
```

6. Your newly created image should now be available on the list of local images.



```
sofija@sofija-VirtualBox:~$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
 SIZE
ubuntu-nmap         latest              d1afab138fb6        24 seconds ago
 121MB
ubuntu              latest              cf0f3ca922e0        9 days ago
 64.2MB
```