

# Docker

GW Tech Talks

# Outline

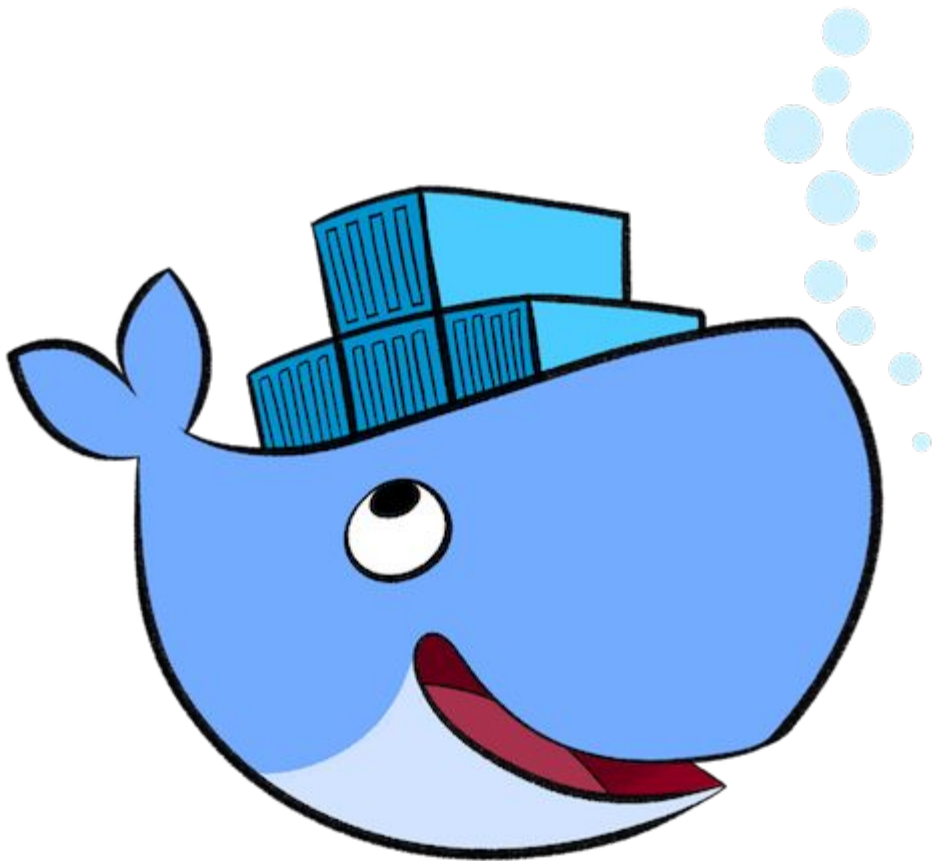
- Introduction to Docker
- Docker Engine
- Docker Architecture
- Docker CLI - Commands
- Docker Swarm
- Demo
- Conclusion

# Introduction

Why is Docker?

&

What is Docker?

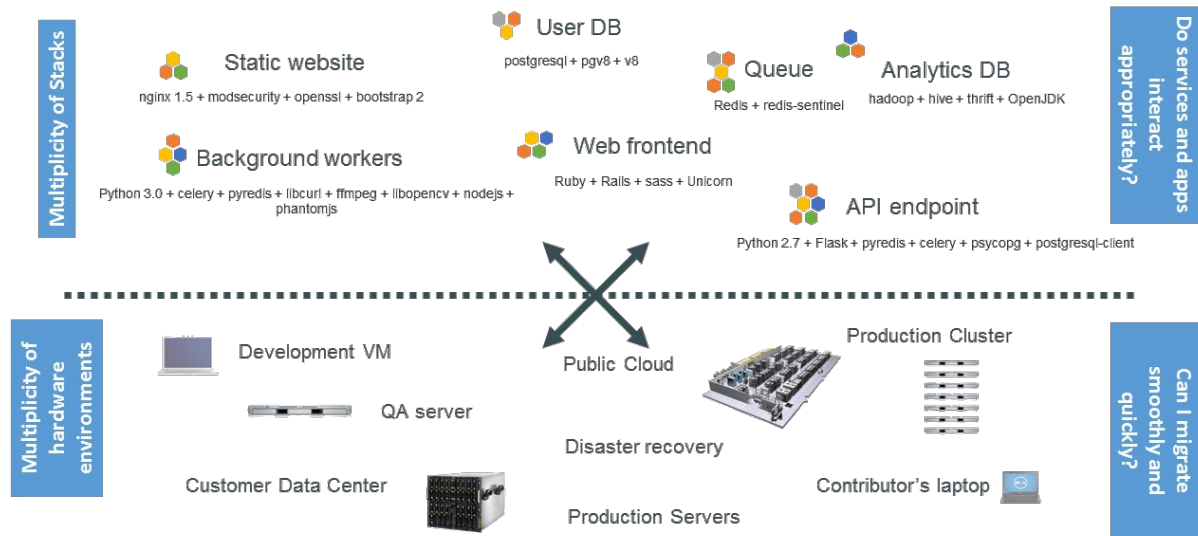


# Why is Docker?

## Deployment issues

- Different environments
- Different OS
- Different packaging
- Conflict between tools or versions

## Impact on usability and reproducibility









# Why is Docker?

## Deployment issues

- Different environments
- Different OS
- Different packaging
- Conflict between tools or versions

## Impact on usability and reproducibility

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

# Transport Pre-1960

Multiplicity of Goods
















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

Multiplicity of  
methods for  
transporting/storing



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

# Transport Pre-1960

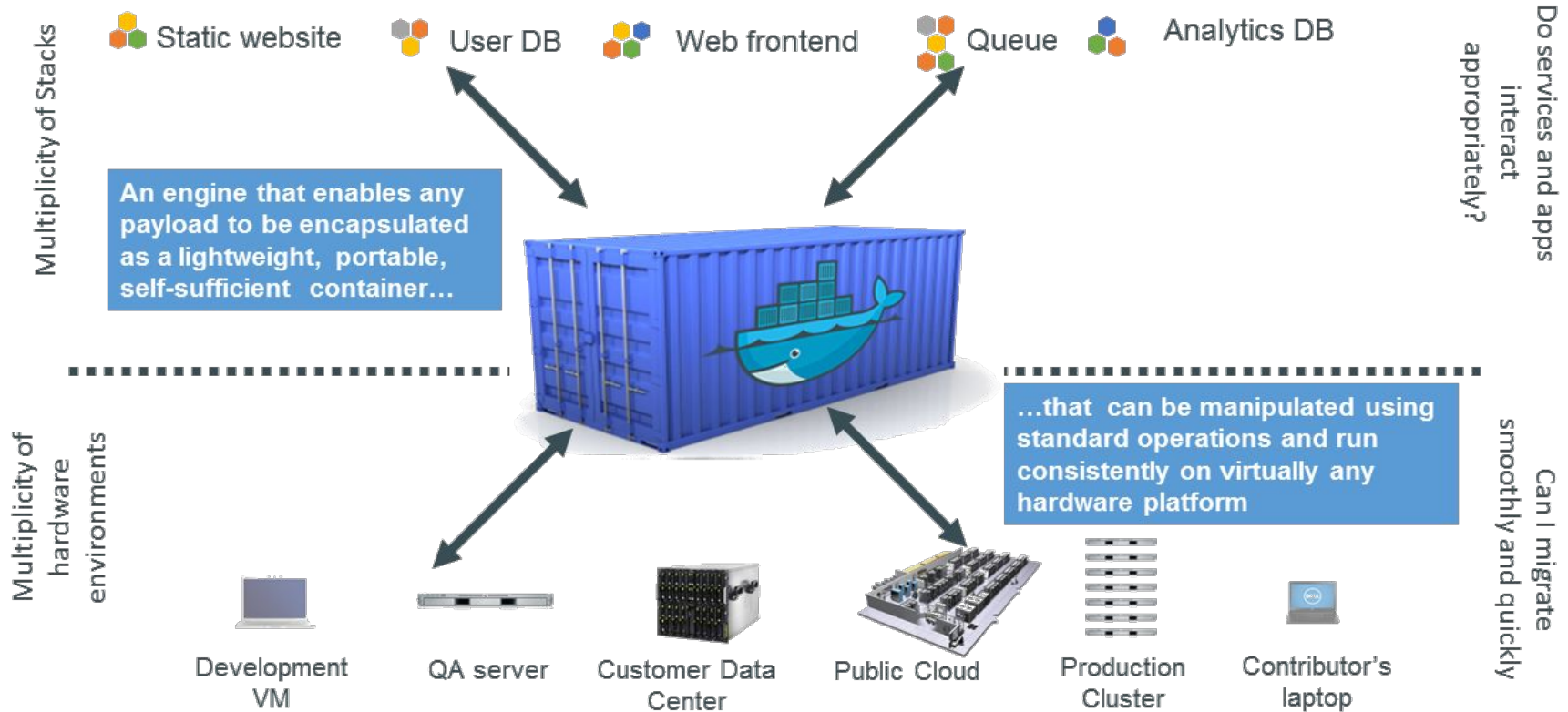
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
							

# Solution: Intermodal Shipping Container

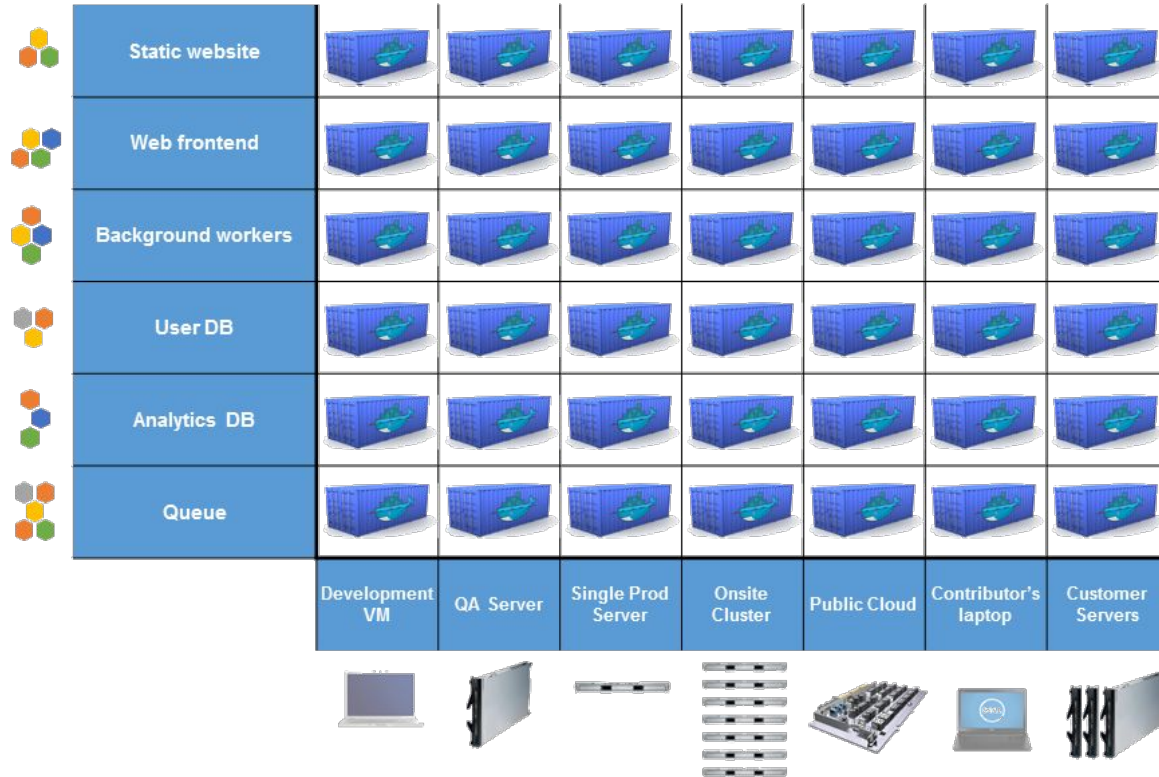




# Solution: Docker is a container system for code



# Solution: Docker eliminates the matrix from hell



# What is Docker?

Docker is an open platform for developing, shipping, and running applications.

- Run everywhere
  - Regardless of kernel version
  - Regardless of host distro
  - Physical or virtual, cloud or not
  - Container and host architecture must match...
- Run anything
  - If it can run on the host, it can run in the container
  - If it can on a Linux kernel, it can run

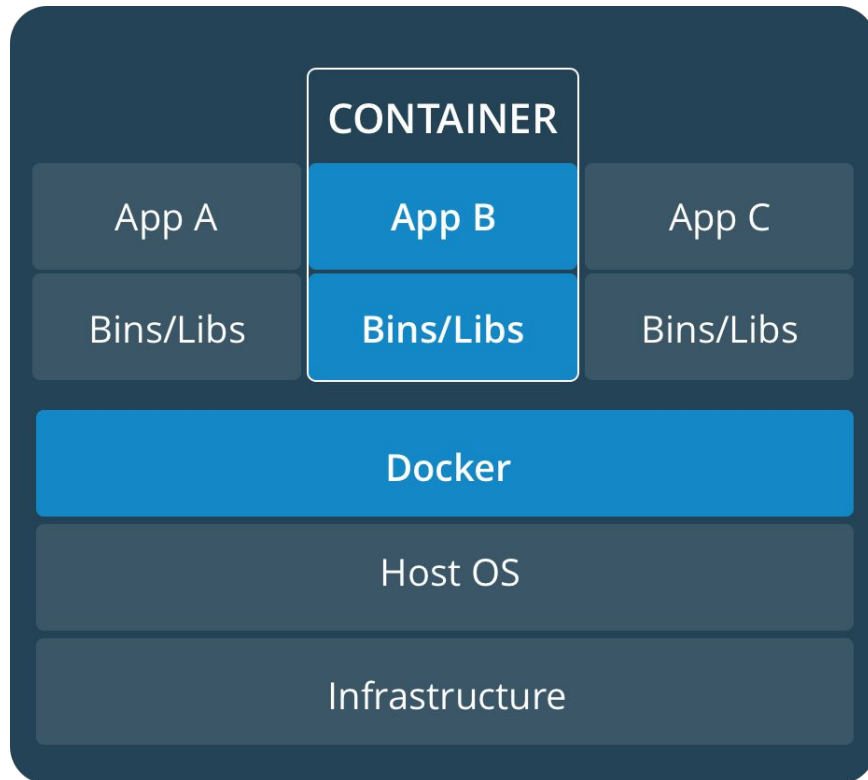


# What is Docker?

A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings

Each container has its own

- Root filesystem
- Processes
- Memory
- Network ports



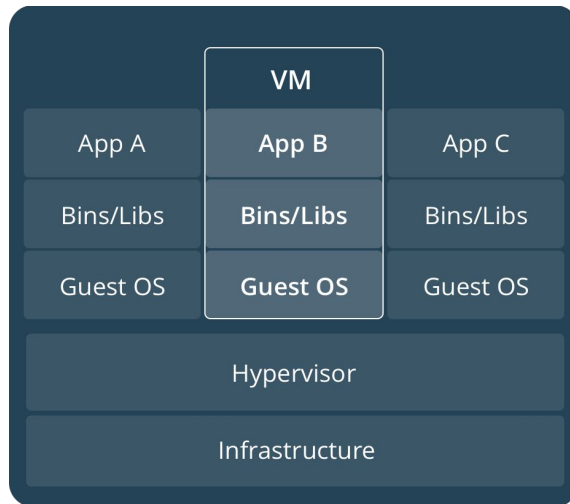
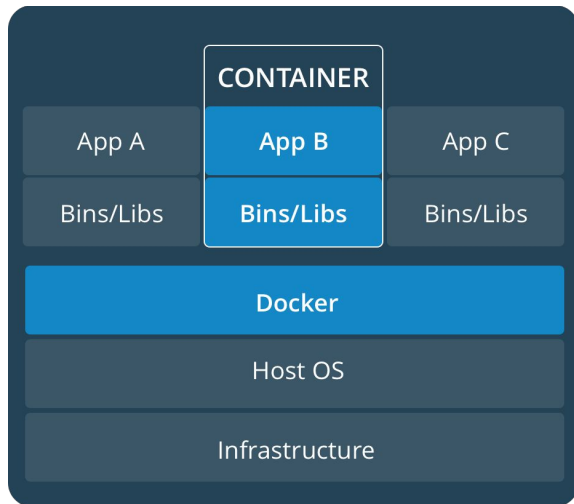
# Advantages of Docker

Distributing and using software as a Docker image gives you:

- **Bundled Dependencies** – Docker images contain all their own dependencies, which means you don't have to do any installation yourself (compared to an application that is just source code or a .deb installer)
- **Cross-platform Installation** – Docker containers contain their own operating system, so they will run on any platform (even Windows!)
- **Easy Distribution** - Can be distributed as a single .tar image file, or put on docker hub so it can be docker pull'd
- **Safety** – Files in a container can't access files on the host machine, so users can trust dockerized applications
- **Ease-of-Use** – Docker containers can always be run using one single docker run command
- **Easy Upgrades** – Docker containers can be easily swapped out for newer versions, while all persistent data can be retained in a data volume

# Docker vs VMs

- Virtual machines solve the same problem as docker, but are much less lightweight
- Virtual machines package the entire guest OS, while Docker uses the host kernel and a minimal OS that can be shared between containers
- Docker requires less CPU, RAM, storage space than VMs
- More containers per machine than VMs
- Greater portability



# Docker vs VMs

Criteria	VM	Containers
Image Size	3X	X
Boot Time	>10s	~1s
Computer Overhead	>10%	<5%
Disk I/O Overhead	>50%	Negligible
Isolation	Good	Fair
Security	Low-Medium	Medium-High
OS Flexibility	Excellent	Poor
Management	Excellent	Evolving
Impact on Legacy application	Low-Medium	High

Ref: M. K. Weldon "The Future X Network: A Bell Labs Perspective," CRC Press, 2016, 476 pp., ISBN:9781498779142

Washington University in St. Louis

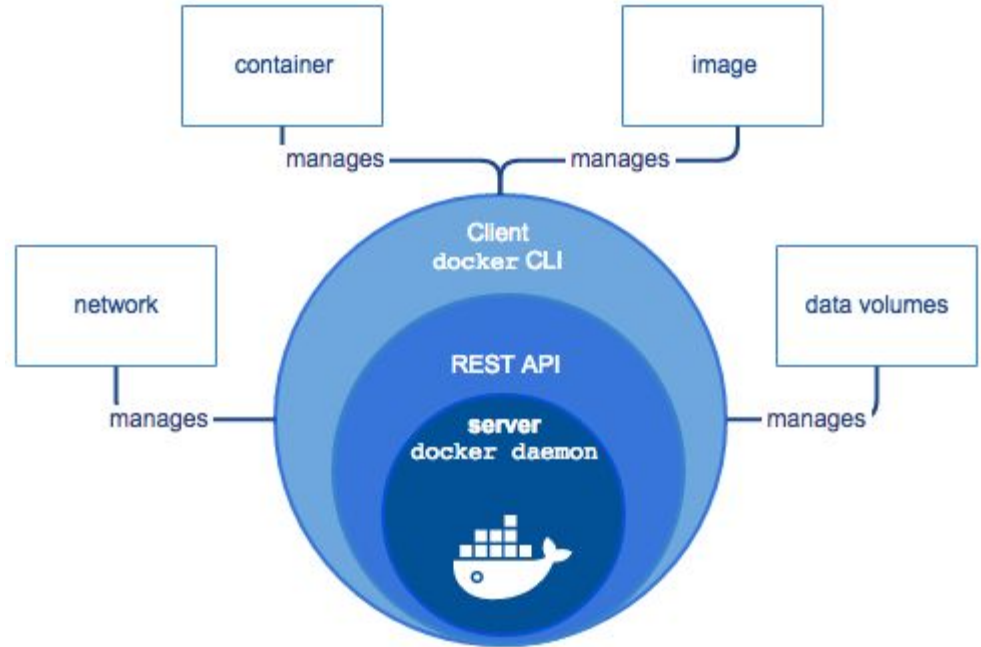
<http://www.cse.wustl.edu/~jain/cse570-18/>

©2018 Raj Jain

# Docker Engine

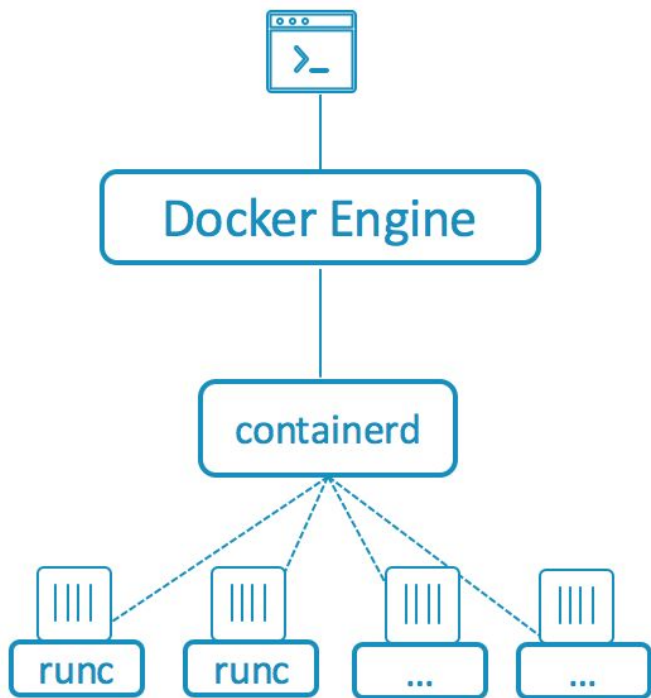
Docker Engine is a client-server application with these major components:

- A server which is a type of long-running program called a daemon process
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client





# Docker Engine



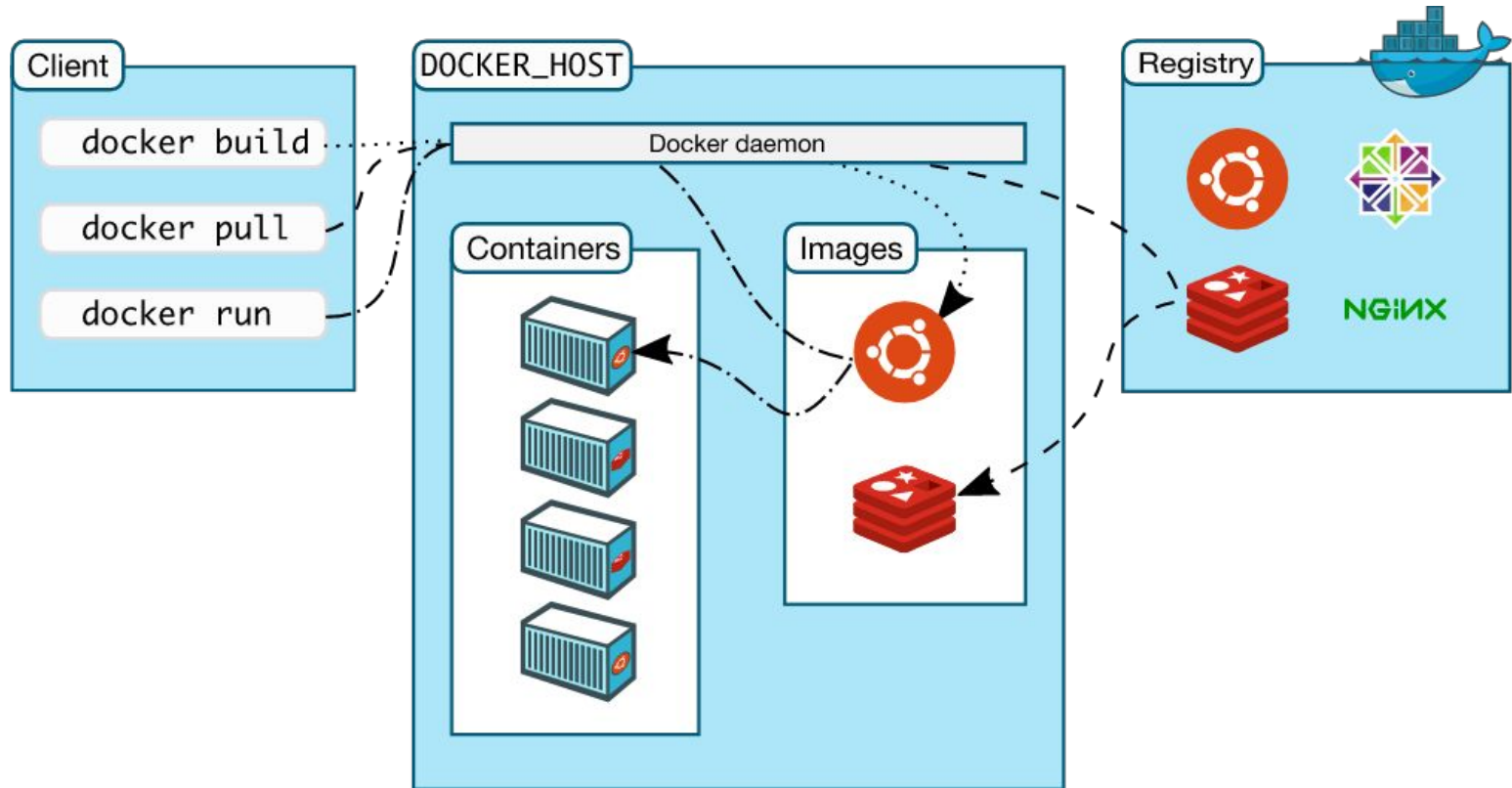
Same Docker UI and commands

User interacts with the Docker Engine

Engine communicates with containerd

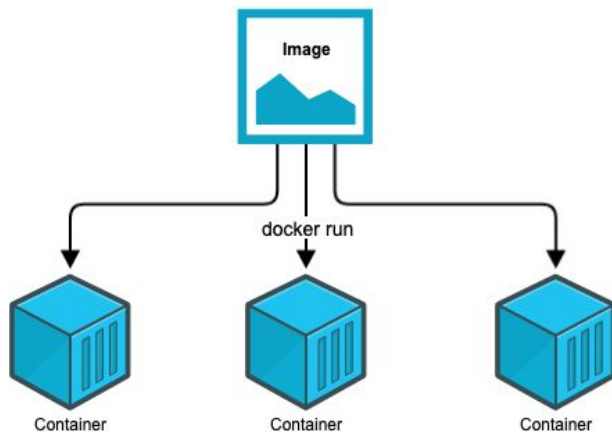
containerd spins up runc or other OCI compliant runtime to run containers

# Docker Architecture



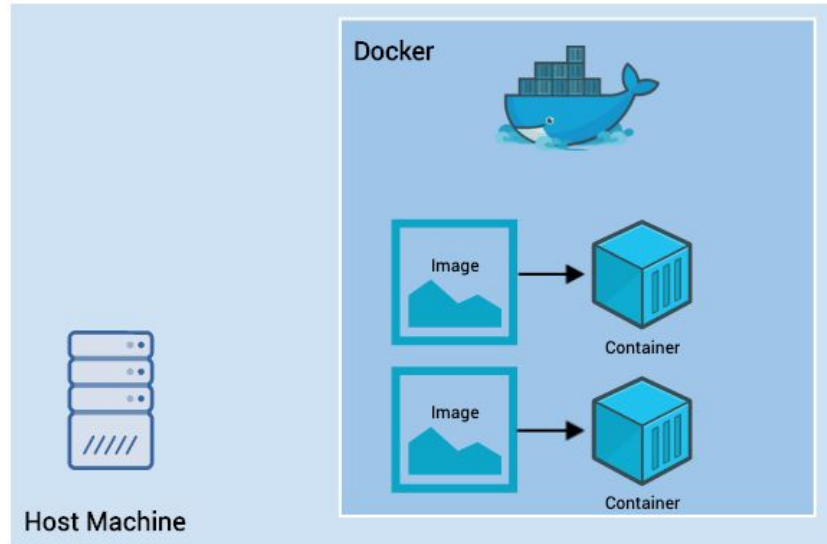
# Images and Containers

- A docker image is an archive containing all the data needed to run the application
- When you run an image, it creates a container, which you can start and stop and delete without it affecting the image
- You can have many containers running the same image
- You can think of a Docker image as like a class in Object-Oriented Programming, and a docker container as like an object



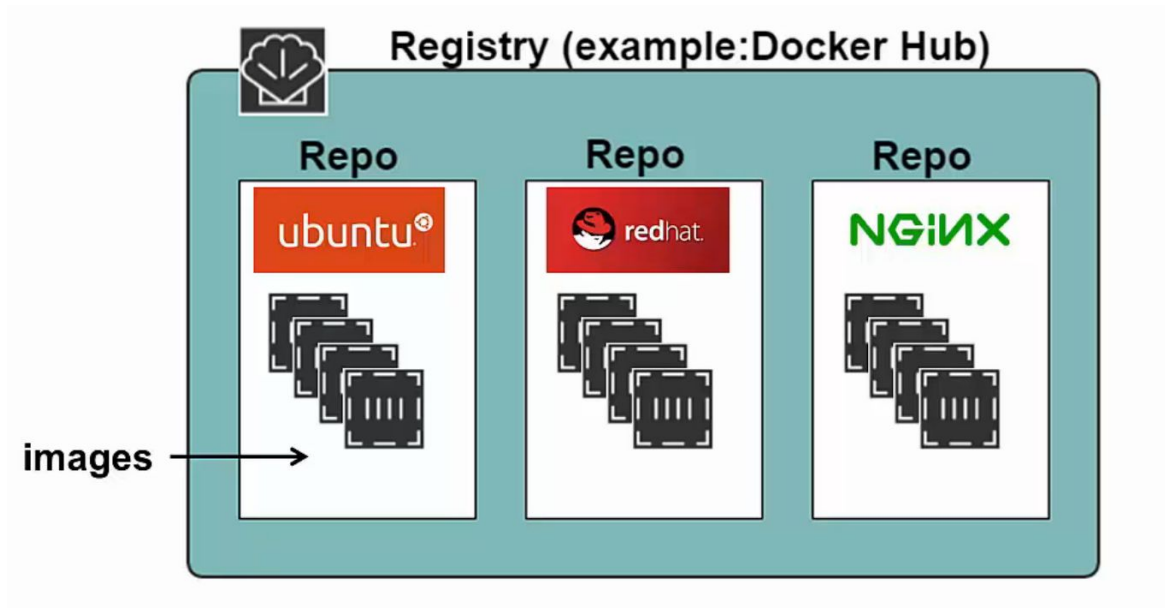
# Docker Host

The host machine is the machine running Docker, on which images and containers are stored



# Registry and Repository

- Registry is where we store images
- Registry can be private or public (Docker Hub)
- Repositories are inside Registry



# Docker CLI

Tells your operating system you are using the **docker** program

Tells Docker which *image* to load into the container

**docker**

**run**

**hello-world**

A subcommand that creates & runs a Docker container

```
$ docker run hello-world
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image *which* runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, *which* sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker Hub account:  
<https://hub.docker.com>

For more examples and ideas, visit:  
<https://docs.docker.com/engine/userguide/>

# Docker Run

To run a container, all you need to do is specify the image name, and docker will pull the image from Docker Hub, and begin running it

**docker run <IMAGE NAME>**

## Exercise

Run the following command. What does it output?

```
docker run hello-world
```

## Answer

```
Hello from Docker!
```

```
This message shows that your installation appears to be working correctly.
```

# Listing Images

Images that you have installed locally can be viewed using

**docker images**

## Exercise

Run the following command to view all images installed on your machine:

```
docker images
```

## Answer

REPOSITORY	TAG	IMAGE ID	CREATED
hello-world	latest	f2a91732366c	3 months ago



# Docker Run Flags

Docker Run is of the form:

**docker run [docker options] <IMAGE NAME> [image arguments]**

This means that arguments that affect the way Docker runs must always go before the image name, but arguments that are passed to the image itself must go after the image name

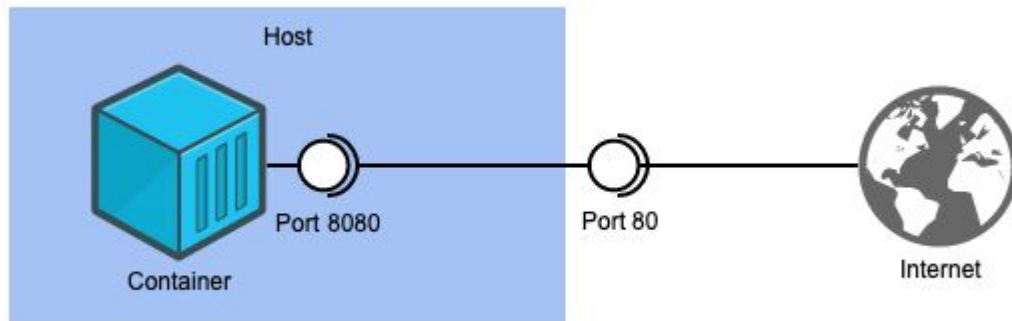
# Port Mapping

Docker containers are free to listen on whatever ports to want to, for example port 80/443 for web requests

**`docker run -p host:container <IMAGE NAME>`**

This command means "map port 8080 inside this container to port 80 on the host machine"

**`docker run -p 80:8080 <IMAGE NAME>`**



# Detached Mode

If you ever need a container to run in the background, use the `-d` docker flag. For instance, we could have run:

```
docker run -d -p 80:80 nginx
```

Docker then prints out the ID of the container, allowing you to access it later:

```
e0d19a8b903015d01a1456a8c9b2351f540b240c0f596030e1d4cd85f9d6956a
```

## Listing running containers

**docker ps** lists all currently running containers

Can also show all terminated containers with the `-a` flag

The IDs that are shown can be useful for other docker commands like `docker stop` and `docker exec`

### Exercise

Try running the command:

```
docker ps
```

### Answer

CONTAINER ID	IMAGE	COMMAND	CREATED
e0d19a8b9030	bgruening/galaxy-stable	"/usr/bin/startup"	2 minutes

# Docker Stop

Ordinarily, you can press ctrl+c to stop a container currently running in your terminal

However, if the container is running in the background (with -d), or refuses to close, you can use docker stop

**docker stop <CONTAINER ID>**

# Volumes and Bind Mounts

By default, Docker containers cannot access data on the host system. This means

- You can't use host data in your containers
- All data stored in the container will be lost when the container exits

You can solve this in two ways:

- This bind mounts a host file or directory into the container.

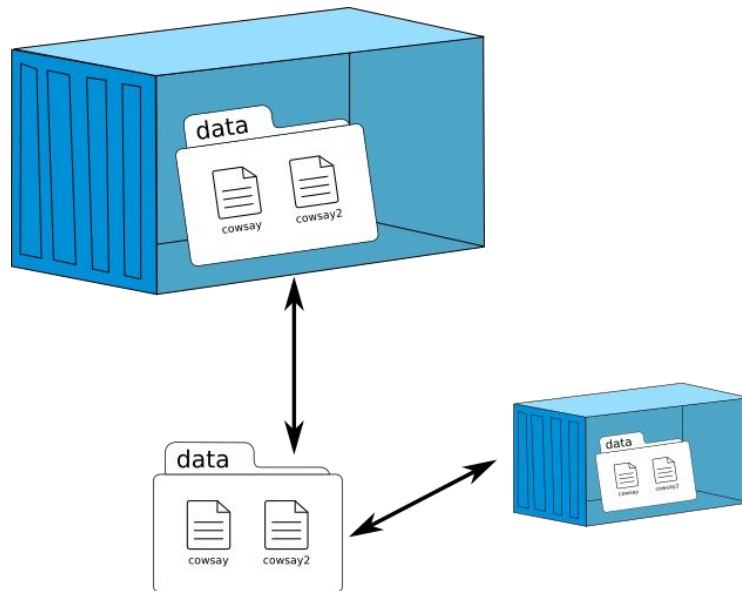
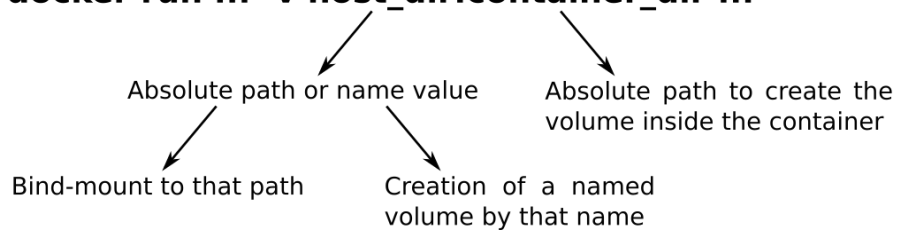
**-v /path/in/host:/path/in/container**

- This mounts a named volume into the container, which will live separately from the rest of your files.

**-v volume\_name:/path/in/container**

# Volumes and Bind Mounts

**docker run ... -v host\_dir:container\_dir ...**



# Running commands inside a container

You can run a command inside a running container using:

```
docker exec <CONTAINER ID> <COMMAND>
```

For example:

```
docker exec bd2ac6cce96f ls
```

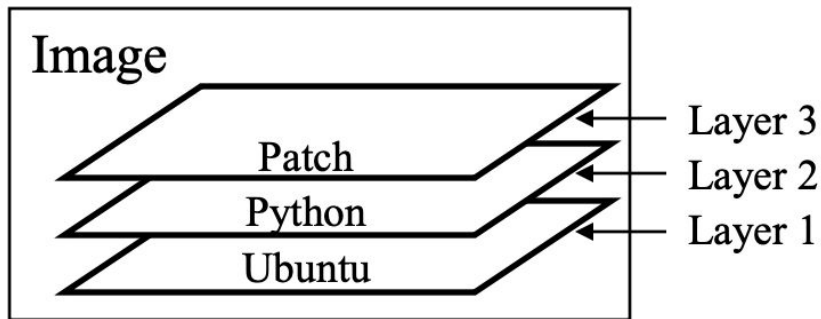
You can also run an interactive bash session inside the container with:

```
docker exec -it bd2ac6cce96f bash
```



# Layers

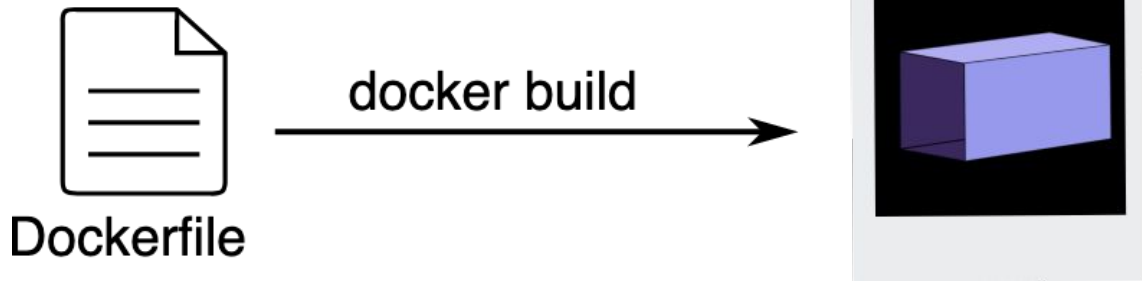
- Image is built layer by layer
- Each layer has its own 256-bit hash
- For example:
  - Ubuntu OS is installed, then
  - Python package is installed, then
  - a security patch to the Python is installed
- Layers can be shared among many containers



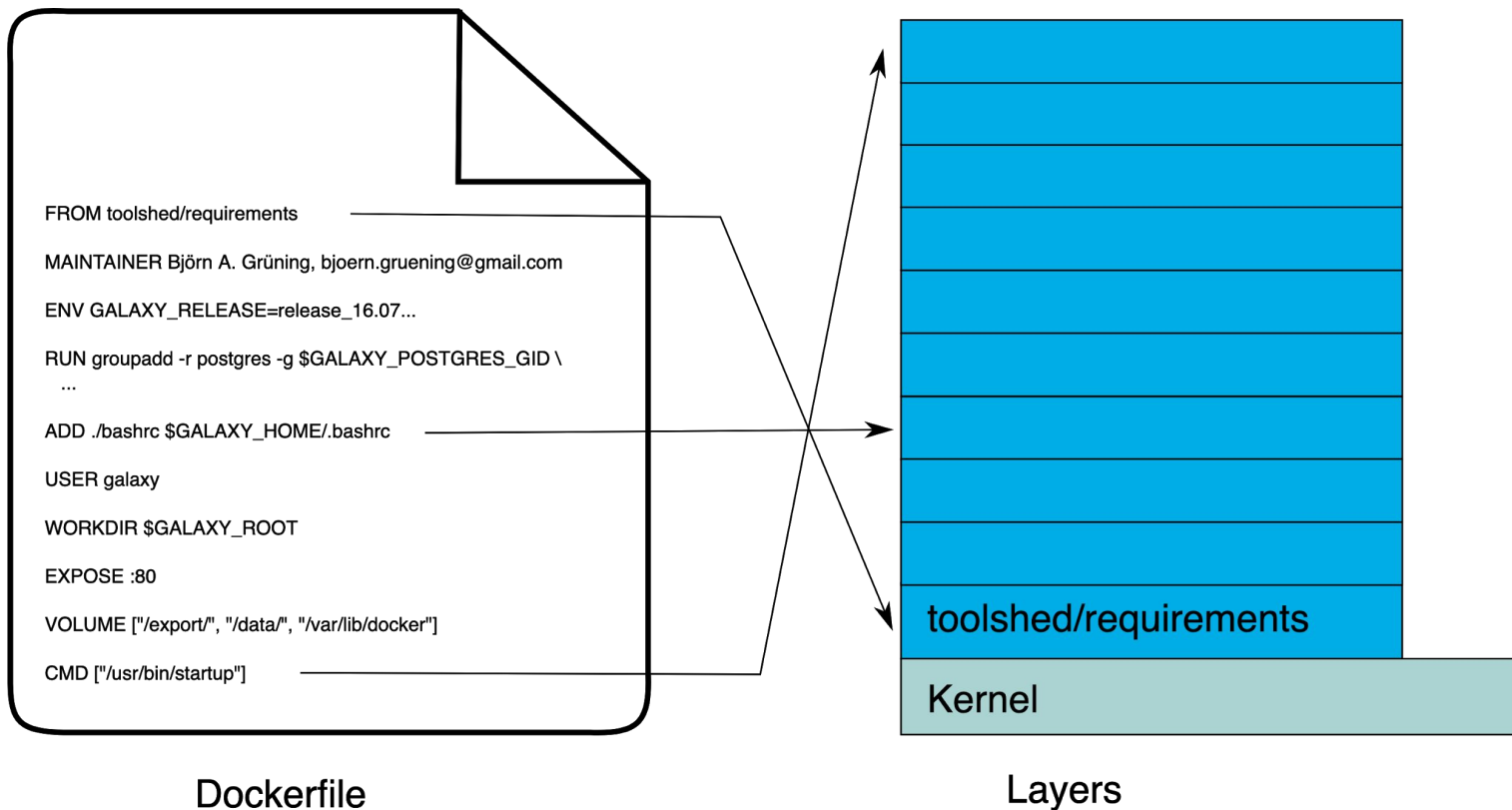
# Dockerfile

A Dockerfile is a list of commands, a lot like a shell script, that progressively builds the image:

- **FROM** lists the image to "inherit" from
- **RUN** executes a shell command
- **COPY** copies some data from the host to the image
- **ENTRYPOINT** sets the command that will be run when a container is created
- **WORKDIR**, like `cd`, sets the current working directory for the build script



# Dockerfile & Layers



# Dockerfile Example

FROM Alpine	← Start with Alpine Linux
LABEL maintainer="xx@gmail.com"	← Who wrote this container
RUN apk add --update nodejs nodejs --npm	← Use apk package to install nodejs
COPY . /src	← Copy the app files from build context
WORKDIR /src	← Set working directory
RUN npm install	← Install application dependencies
EXPOSE 8080	← Open TCP Port 8080
ENTRYPOINT ["node", "./app.js"]	← Main application to run

RUN npm install	← Layer 4
Copy . /src	← Layer 3
RUN apk add ...	← Layer 2
FROM Alpine	← Layer 1

# Docker Build

To build a Docker image from a Dockerfile, use the docker build command

You should specify an image tag/name using -t, and a directory containing the Dockerfile. For example:

```
docker build -t my_image .
```

# Networking

By default Docker comes with the following networks:

```
docker network ls
```

NETWORK ID	NAME	DRIVER
7fca4eb8c647	bridge	bridge
9f904ee27bf5	none	null
cf03ee007fb4	host	host

Docker can manage networks of different types:

- host: The host interface
- bridge: Bridged network interfaces
- overlay: Software defined multi host network (swarm only)

Manage networks with: **docker network create|rm|inspect|ls**

# Connect Network

Containers can be connected to multiple networks.

Network at startup:

```
docker run --net=<networkname> <image>
```

Connect a running container:

```
docker network connect <networkname> <containerid>
```

Disconnect a running container:

```
docker network disconnect <networkname> <containerid>
```

# Docker Compose

Docker Compose is a tool for creating and managing multi container applications

Run application:

**docker-compose up**

Stop application:

**docker-compose down**

```
version: "3.7"
```

```
services:
```

```
  app:
```

```
    image: node:12-alpine
```

```
    command: sh -c "yarn install && yarn run dev"
```

```
    ports:
```

```
      - 3000:3000
```

```
    working_dir: /docker101
```

```
    volumes:
```

```
      - ./:/docker101
```

```
    environment:
```

```
      MYSQL_HOST: mysql
```

```
      MYSQL_USER: root
```

```
      MYSQL_PASSWORD: secret
```

```
      MYSQL_DB: todos
```

```
  mysql:
```

```
    image: mysql:5.7
```

```
    volumes:
```

```
      - todo-mysql-data:/var/lib/mysql
```

```
    environment:
```

```
      MYSQL_ROOT_PASSWORD: secret
```

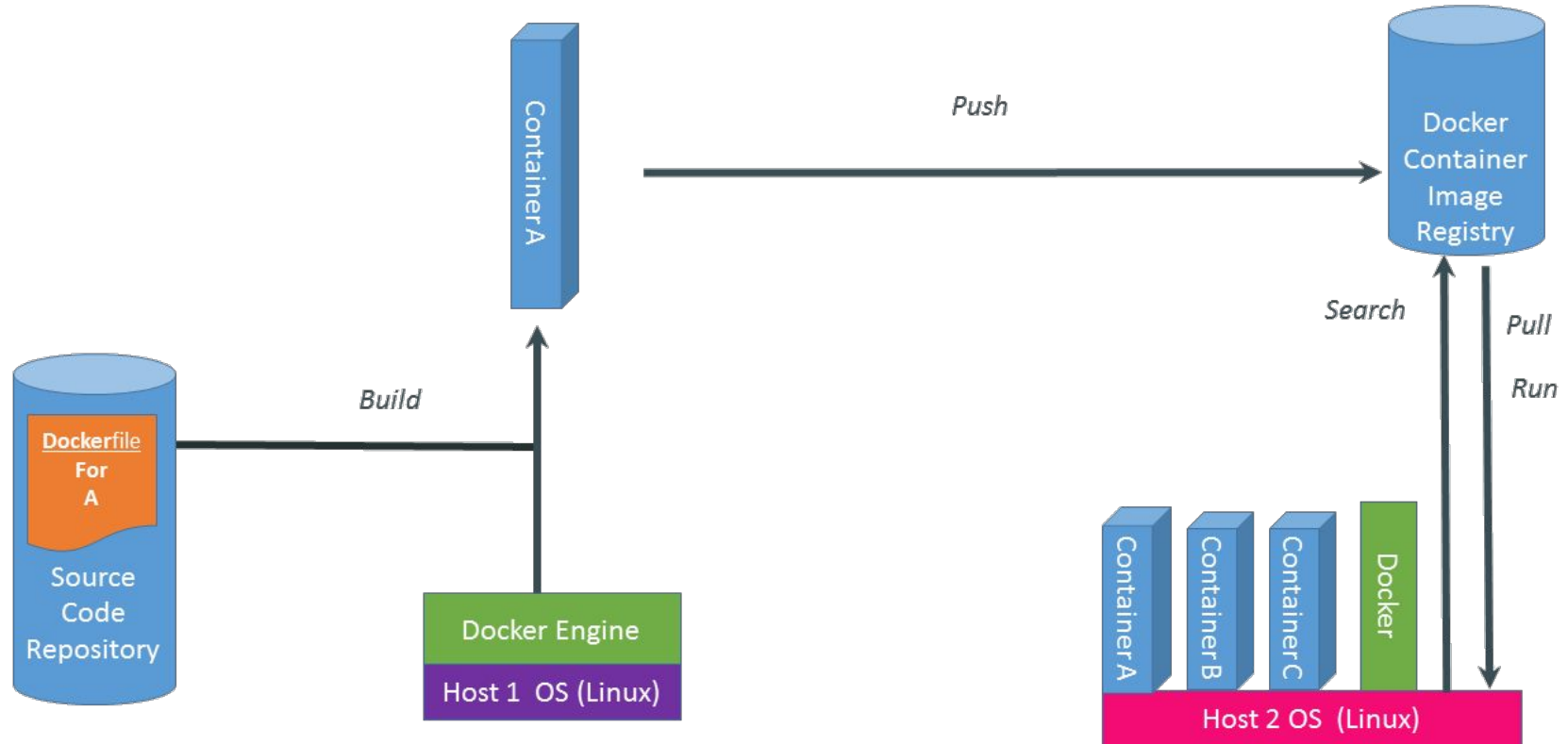
```
      MYSQL_DATABASE: todos
```

```
volumes:
```

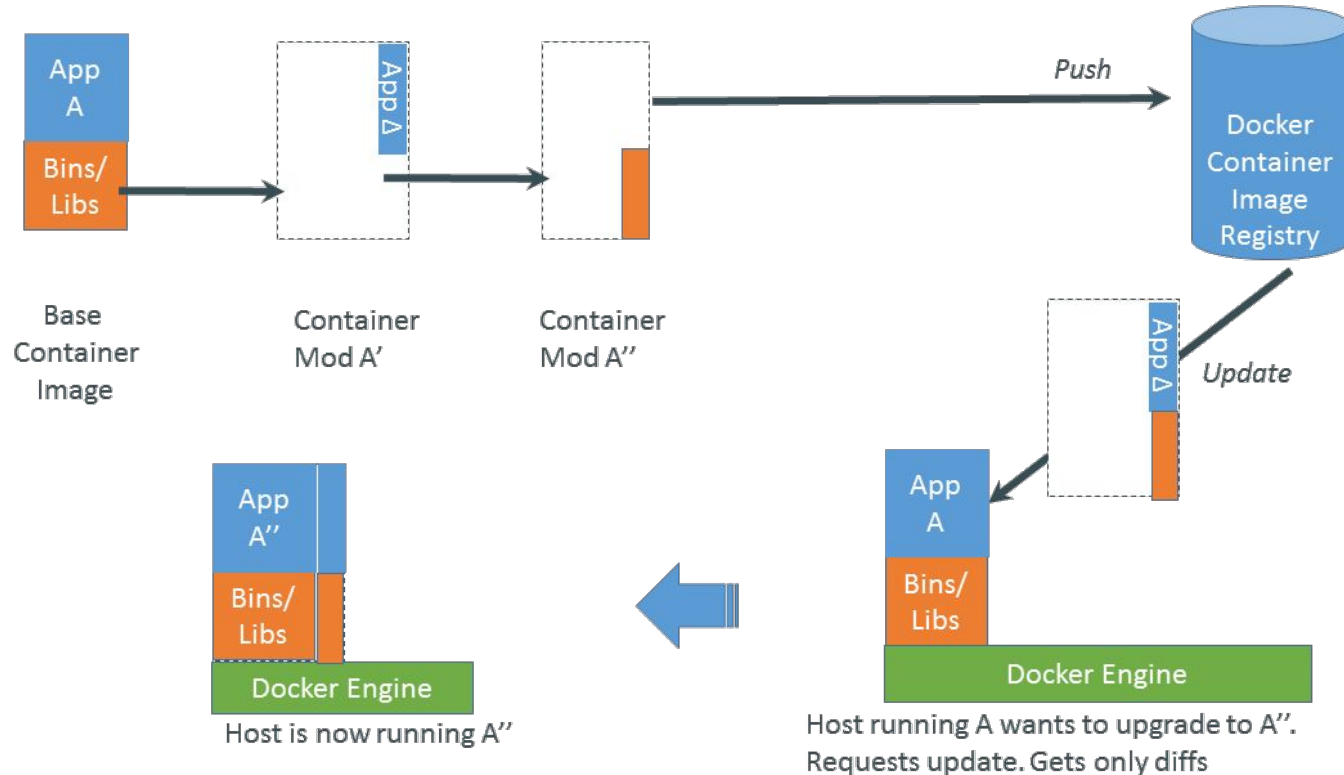
```
  todo-mysql-data:
```



# Docker Workflow



# Docker Workflow - Changes & Updates

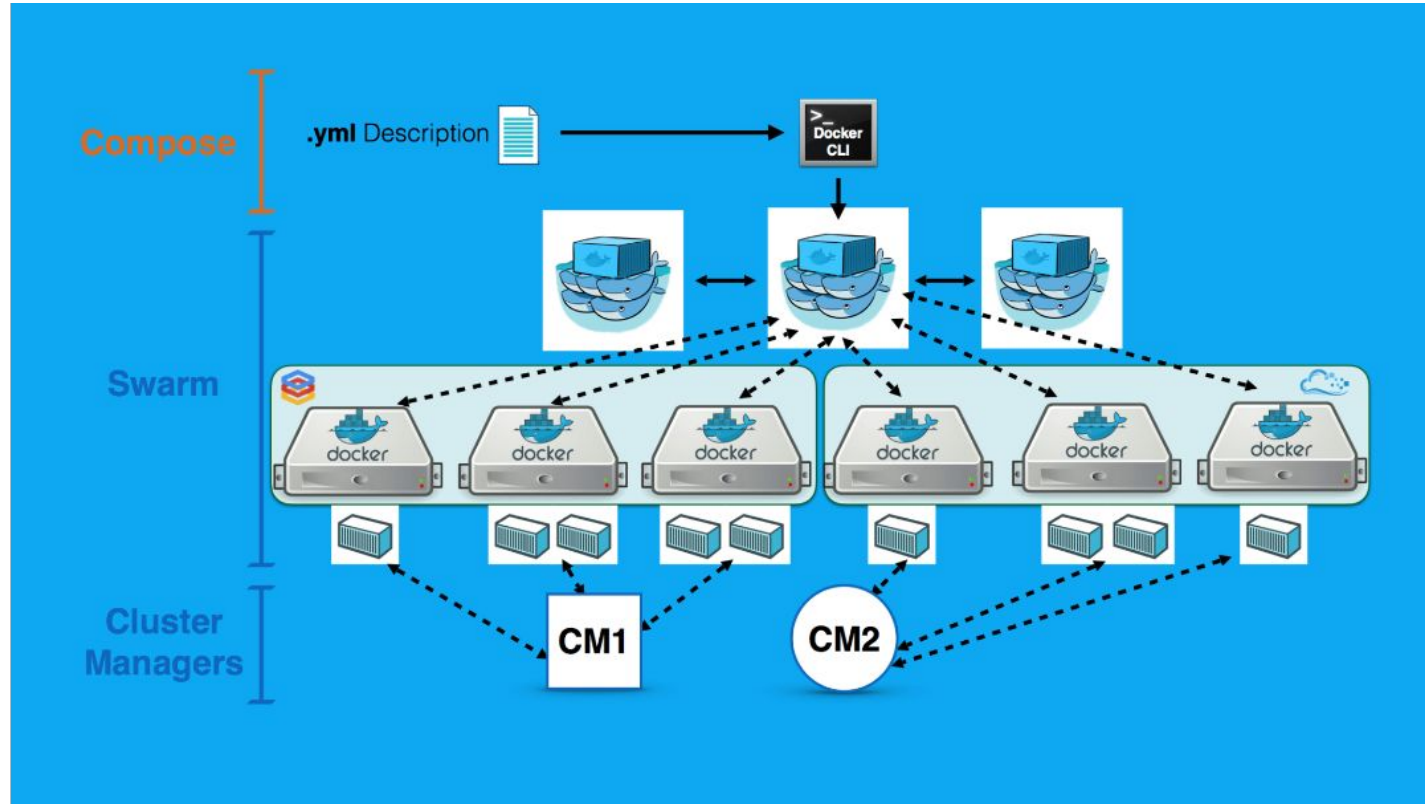


# Docker Swarm

Swarm is a build in Docker clustering mode. It provides:

- Single master for communication with the cluster
- Monitoring and failover between node
- Scaling of containers
- Load balancing of published ports

# Docker Swarm



Demo

# Conclusion

Docker provides

- It works on my machine!
- I have new developers to onboard.
- We are adopting a microservices architecture.
- I need to move my legacy apps to containers.