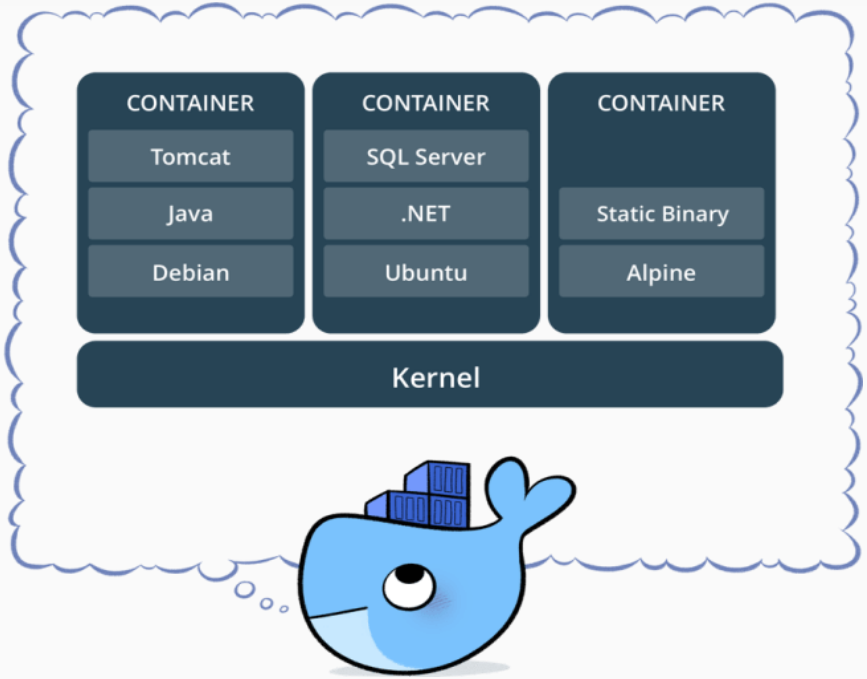
Docker and Containers

An overview

What is Docker?

“ Docker is an open platform for developers and system administrators to build, ship and run containerized applications.“



Major Infrastructure Shifts

Mainframe to PC (90's)

Bare Metal to Virtual (00's)

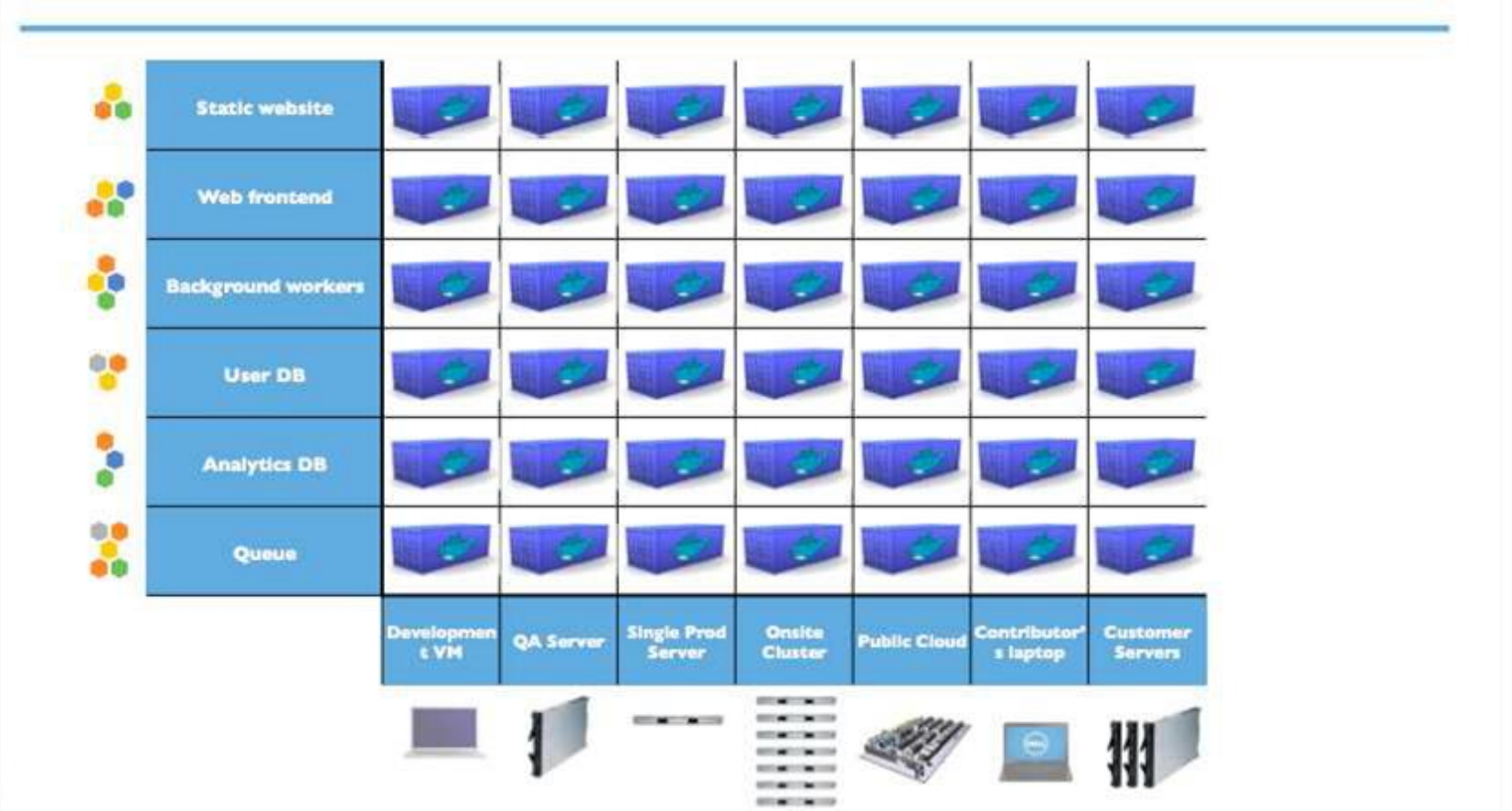
Datacenter to Cloud (10's)

Host to Container (Serverless)

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	?	?	?	?	?	?	?

Solving the matrix: Container



Benefits of Docker:

Speed

- Develop Faster
- Build Faster
- Test Faster
- Deploy Faster
- Update Faster
- Recover Faster

Portability

- Less dependencies between process layers
- Ability to move between infrastructures

Efficiency

- Less OS Overhead
- Improved VM Density

Docker Editions

Docker Community Edition

- Free; community-supported product for delivering a container solution
- Intended for: Software development and test

Docker Enterprise Edition

- Subscription based commercially supported products for delivering a secure software supply chain
- Certified on specific platforms
- Intended for: Production Deployments and Enterprise Customers

docker.com/get-started

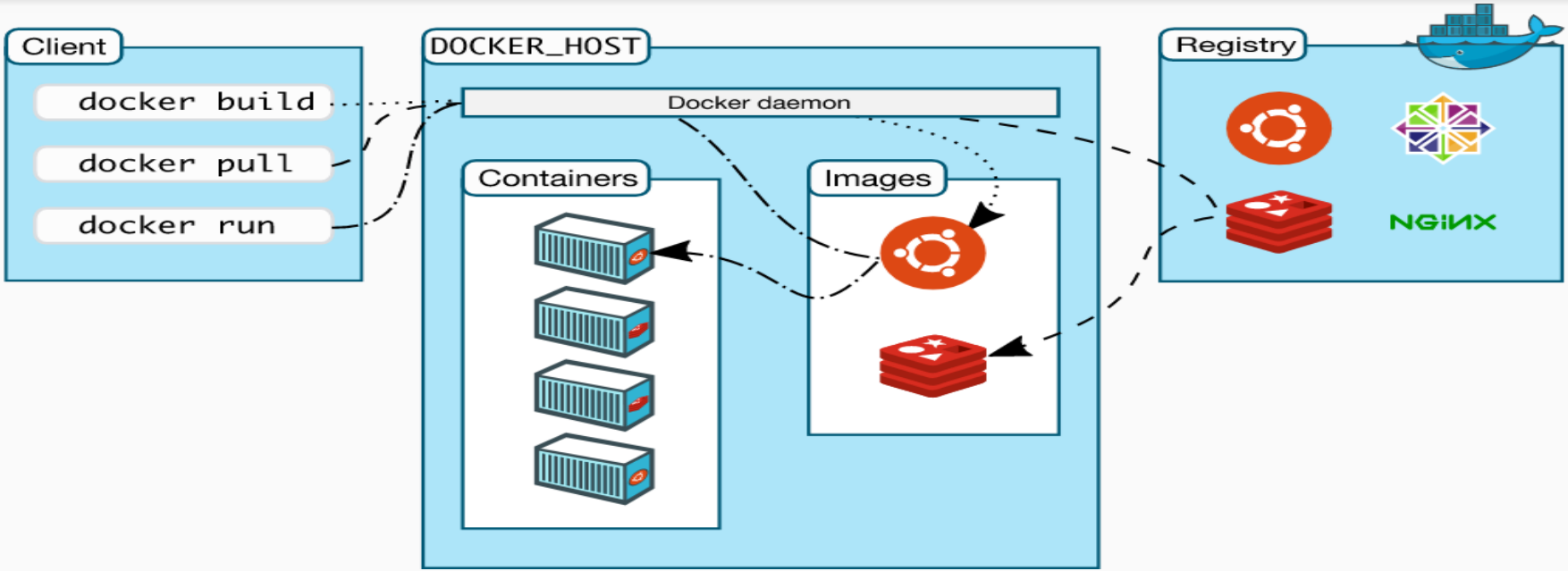
Docker on Linux:Setup

- Best native experience
- Three ways to install:
  - Script, store or ***docker-machine***
- get.docker.com script (latest Edge release)
  - ***curl -sSL https://get.docker.com/ | sh***
- store.docker.com has instructions for each distro
- RHEL officially only supports Docker EE (paid), but CentOS will work
- Installing in a VM, Cloud Instance, all are the same process
- May not work for unlisted distros (Amazon Linux, Linode Linux, etc.)
- Don't use pre-installed setups (Digital Ocean, Linode, etc.)

Installing Docker on Ubuntu

- Installation Steps
- Register with Docker Hub

Docker Architecture:



Source: https://docs.docker.com/engine/docker-overview/

Docker Architecture: (Contd.)

- Client-Server architecture having 3 main components:
  1. Docker Client
  2. Docker Host
  3. Docker Registry
- Docker Objects:
  - Images
  - Containers
  - Volumes
  - Networks

Docker Architecture: (Contd.)

Docker Client:

- The interface through which users interact with the docker using the docker commands.
- 2 basic ways to interact with Docker:
  - Docker CLI
  - Docker APIs
- Docker Client can communicate with more than one daemon

Docker Architecture: (Contd.)

Docker Host:

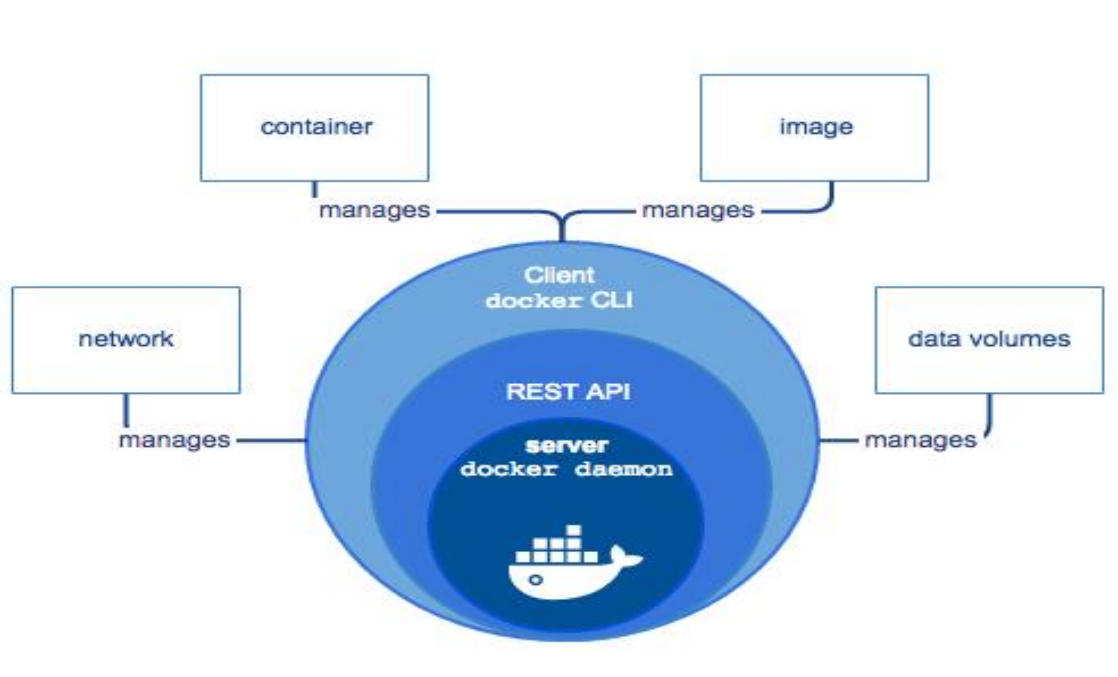
- Runs docker daemon which listens to and performs actions requested by Docker Client.
- Manages Docker objects such as images, containers, networks and volumes
- The Docker client and daemon can run on same systems or can connect docker client to a remote docker daemon
- Docker daemon is responsible for building dockerfiles, generating images and running containers

Docker Architecture: (Contd.)

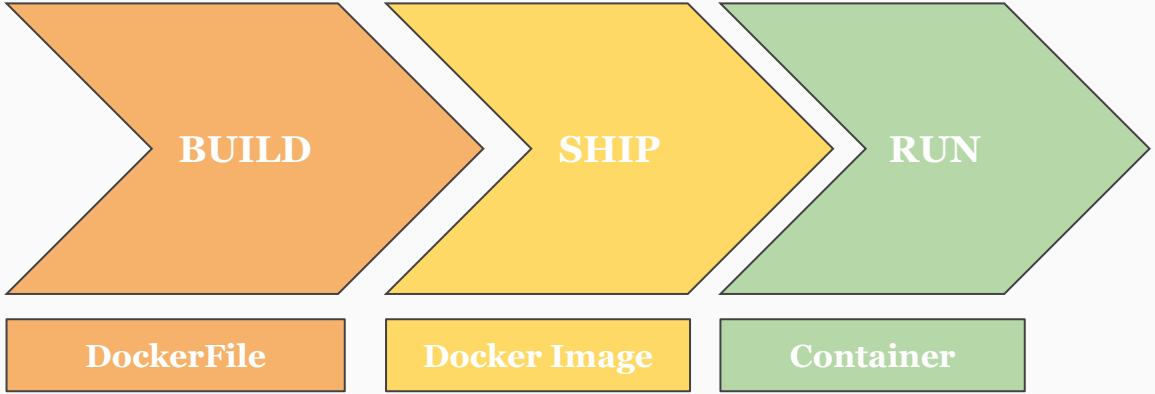
Registry:

- Simplest component of docker architecture
- A universal place to store docker images and make them available to others
- Docker Hub is a public registry that anyone can use (https://hub.docker.com/)
- Docker is configured, by default, to look for docker images in Docker Hub
- *docker pull* and *docker push* are the commands used to pull and push the images from and to the configured registry

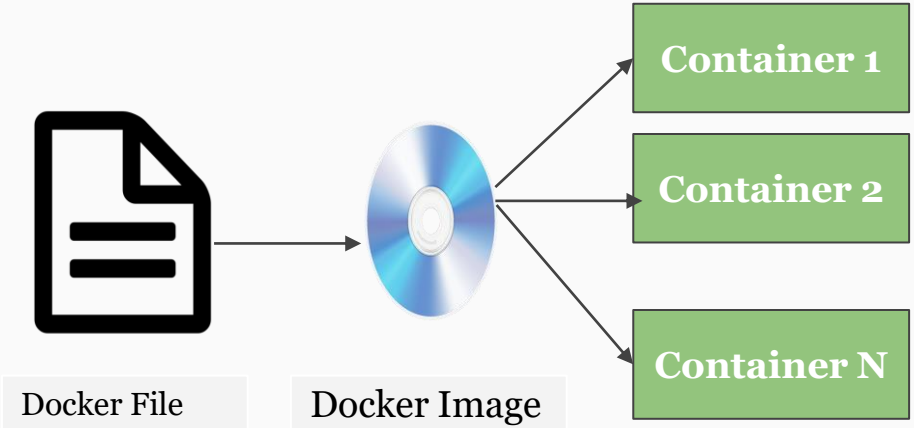
Docker Components



Containerization Stages:



Containerization Stages (Contd.):



- Multiple containers can be instantiated from same Docker image.
- One Dockerfile corresponds to one Docker Image

Image vs Container

- An image is an application we want to run
- A container is an instance of that image running as a process
- Images: building blocks of the container
- You can have many containers running off the same image

Let's Try:

docker container run -p 80:80 nginx

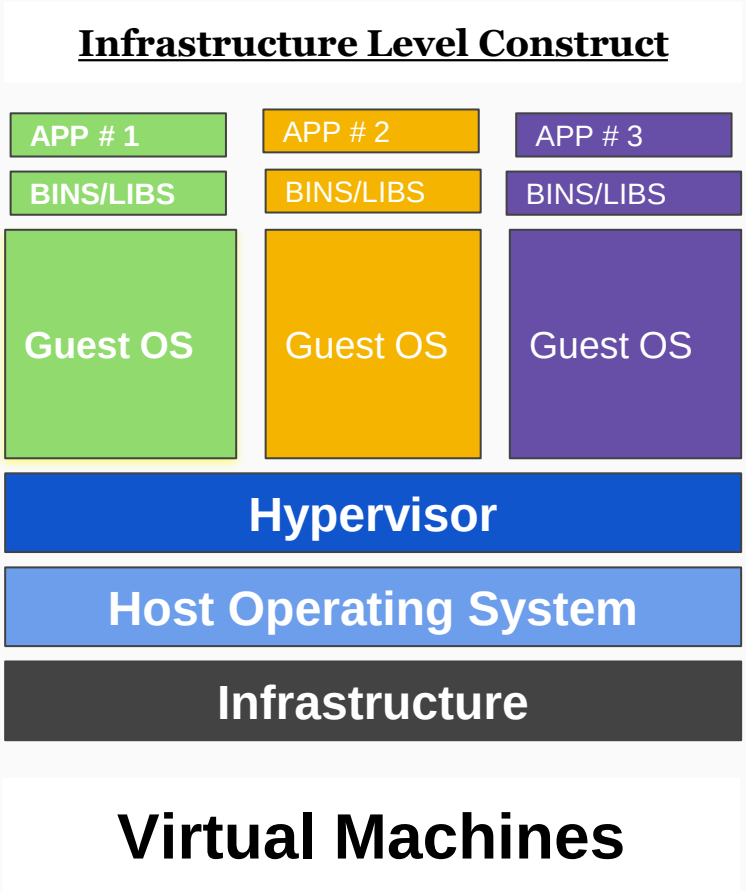
1. Download image 'nginx' from Docker Hub
2. Started a new container from that image
3. Opened port 80 on the host IP
4. Routes that traffic to the container IP, port 80

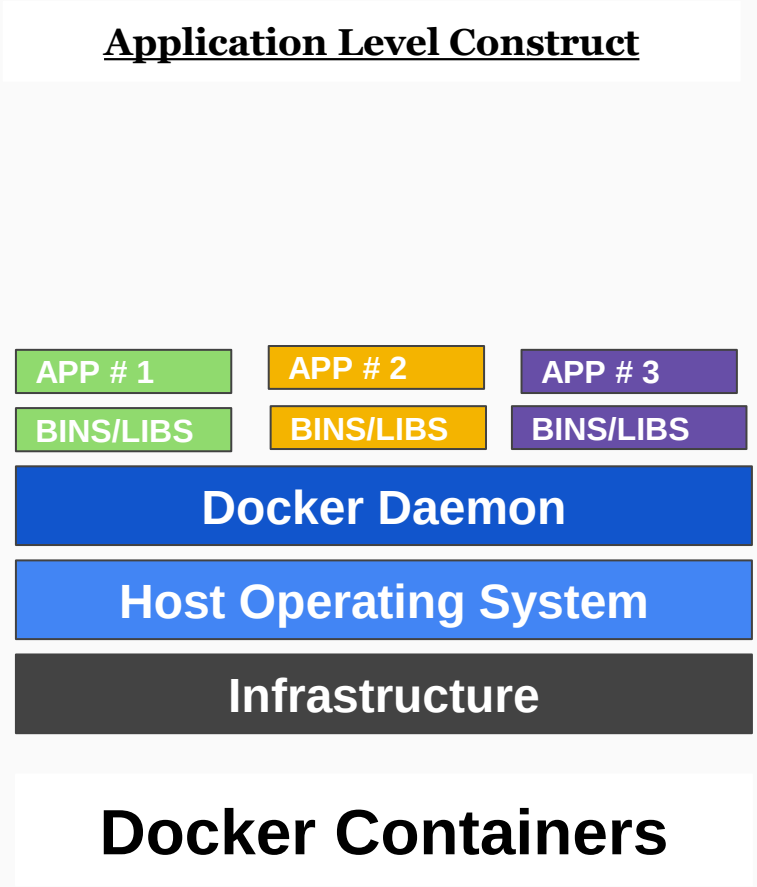
How ‘docker container run’ works?

- Looks for specified image locally in image cache.
- If it does not find the image, then it looks in remote image repository (Docker Hub by default)
- Downloads the latest version of image (nginx: latest by default)
- Creates new container based on that image and prepares to start
- Gives it a virtual IP on a private network inside docker engine
- Opens up port 80 on host and forwards to port 80 in container
- Starts container by using CMD in the image Dockerfile

Virtual Machines vs Container

Virtual Machines: vs Container





Virtual Machine vs Container

- Virtual machines are an abstraction of physical hardware turning one server into many server.
- Multiple VMs can run on single machine through Hypervisor
- Each VM has full copy of the OS
- Takes minutes to start

- Think of containers as isolated processes
- Containers are abstraction at the application layer that packages the code and dependencies together
- Each container runs in isolation with other containers;
- multiple containers can run on same machine sharing OS kernel with other containers
- Container can be up and running in milliseconds

Virtual Machine vs Container (Contd…)

Some readings on why docker is faster than virtual machine

https://www.backblaze.com/blog/vm-vs-containers/

Assignment 1: Manage Multiple Containers

- Run a nginx, a mysql, and a httpd (apache) server
- Run all of them --detach (or -d), name them with --name
- Nginx should listen on 80:80, httpd on 8080:80, mysql on 3306:3306
- When running mysql, use the --env option (or -e) to pass in MYSQL_RANDOM_ROOT_PASSWORD=yes
- Use *docker container logs* on mysql to find the random password it created on startup
- Clean it up all with *docker container stop* and *docker container rm* (both the commands can accept multiple names and IDs)
- Use *docker container ls* to ensure everything is correct before and after clean up.
- Refer docs.docker.com and *--help* for any help

What's happening within the container?

- To get process list of one container
  - ***docker container top***
- To get the configuration details of one container
  - ***docker container inspect***
- To get performance stats for all containers
  - ***docker container stats***
- Start new container interactively
  - ***docker container run -it***
- Run additional commands in existing container
  - ***docker container exec -it***
- Different Linux distros in container

Docker Networks

Section Overview:

- Review of ***docker container run -p***
- Concept of “batteries included but removable”
  - Defaults work but you can change a lot under the hood
- Quick port docker container port <container>
- Learn concepts of Docker Networking
- Understand how network packets move around Docker

Docker Network Drivers: Basics

- When you start container, we are connecting to a particular Docker Network in the background.
- The communication is managed by objects called network drivers.
- A docker network driver is a piece of software which handles container networking.
- Created using ***docker network*** command.
- **'bridge'** is the default network driver. Usually used when your application runs in standalone containers that need to communicate.

Docker Network: Defaults

- Each container connected to a private virtual network “bridge”
- Each virtual network routes through NAT firewall on host IP
- All containers on virtual network can talk to each other without -p
- Best practice: to create a new virtual network for each app:
  - Network ‘my_web_app” for mysql and php/apache containers
  - Network “my_api” for mongo and nodejs containers

Docker Networks: CLI Management

- To show networks
  - ***docker network ls***
- Inspect a network
  - ***docker network inspect***
- Create a network
  - ***docker network create --driver***
- Attach a network to container
  - ***docker network connect***
- Detach a network from a container
  - ***docker network disconnect***

Docker Networks: Default Security

- Create apps so frontend/backend sit on same Docker network
- Their intercommunication never leaves host
- All externally exposed ports closed by default
- You must manually expose via -p, which is better default security!

Docker Image

Section Overview:

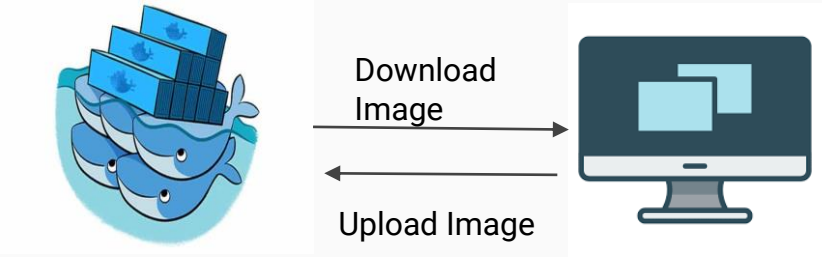
- What is an Image and image layers
- Docker Hub Registry: How to use it?
- Managing local image cache
- Build your own images

What is a Docker Image?

- Images are made up of App binaries, dependencies and metadata
- Read-only template of instructions on creating a container
- "An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime."
- Each command is a layer in the image
- Does not contain a full OS
- Store in the Docker Engine Image cache
- Permanent Storage should be in Image registry
- Create your own images or use created by others and published in registry

DockerHub

- Register at hub.docker.com
- When we create images on Docker Hub, we have to create them with our account name in front of them
- The images that have just the name of the repo are considered official.
    - Official Images: well tested, have quality documentation and obey best practice rules
- Images are tagged
    - A version of image can have more than one tag

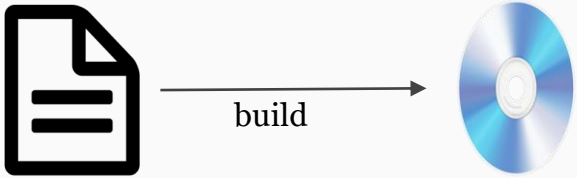


How to create Docker Image?

- Use ***Dockerfile***
- Define steps to create an image and run it
- Each instruction in Dockerfile creates a layer in image
- Move images in/out of cache via:
  - Local filesystem via tarballs
  - push/pull to remote ‘image registry” (e.g. Docker Hub)
- Images are not ideal for persistent data
  - Mount a host file system path into container
  - Use ***docker volume*** to create storage for persistent/unique data

What is Dockerfile?

- A shell script
- Defines the commands/actions to be executed
- <u>Example:</u> Issuing ***docker run*** command starts the nginx server



docker build -t test

- -t used to specify tag name for the image

The Dockerfile Keywords:

- FROM (base image)
  - Every Dockerfile starts with FROM keyword
  - Specify the image name which will act as base image for the container
  - Example: *FROM nginx*
- ENV (environment variable)
  - Specify environment variables for the Docker Image
  - Example: *JAVA_HOME*
- RUN (any arbitrary shell command)
  - Will be executed when we build the image

The Dockerfile Keywords:

- EXPOSE :
  - open port from container to the virtual network
- VOLUME:
  - Define the storage location for the unique data generated by the application
- CMD :
  - command to run when the container starts
  - This will be executed when ***docker run*** is issued

Image and its layers:

- Every image is a collection of layers.
- Each layer is uniquely identified and only stored once on a host
- This saves storage space on host and transfer time on push/pull
- container is just a single read/write layer on top of image
- ***docker image history*** and ***inspect*** commands can help us to get sense of image

Assignment 2: Build Your Own Image

- Take existing Node.js app and dockerize it
- Make Dockerfile. Build it. Test it. Push it. Remove it. Run it.
- Details in ***part3-dockerfile-assignment/Dockerfile***
- Use the Alpine version of the official 'node' 6.x image
- Expected results is web site ***"https://localhost"***
- Tag and push to your Docker Hub account (free)
- Remove your image from local cache, run again from Hub

Docker Volumes

Section Overview:

- Understanding the Persistent Data Problem
- Learn and attach volumes to the container
- Learn about Bind Mounting

Defining problem of persistent data:

- Containers are usually immutable and ephemeral
  - Immutable: a container won't be modified during its life: no updates, no patches, no configuration changes
  - Containers can be restarted, stopped or replaced easily
  - Redeploy a new container, if configuration needs to change
- Persistent Data problem: What about the unique data produced by application?
- Two solutions:
  - Volumes and Bind Mounts

Volumes: Making Docker persistent

- Volumes: Make special location outside of container UFS
- **VOLUME** command in **Dockerfile**
- Also override with ***docker run -v /path/in/container***
- Bypasses Union File System and stores in alt location on host
- Includes it's own management commands under ***docker volume***
- Connect to none, one or multiple containers at once
- Not subject to ***commit***, ***save***, or ***export*** commands
- By default they only have a unique ID, but you can make it "named volume" by assigning name to it.

Bind Mounting

- Bind Mounts: link container path to host path
- Maps a host file or directory to a container file or directory
- Basically just two locations pointing to the same file(s)
- Again, skips UFS, and host files overwrite any in container
- Can't use in Dockerfile, must be at *container run*
- *... run -v /Users/bret/stuff:/path/container* *(mac/linux)*
- *... run -v //c/Users/bret/stuff:/path/container*(*windows*)

Docker Compose

What is Docker Compose?

- A tool for defining and running multi-container docker applications
  - A YAML file is used to configure application services
  - With single command, create and start all the services from configuration
- Comprised of 2 separate but related things:
  - YAML-formatted file that describes our solution options for: **containers**, **networks** and **volumes**
  - A CLI tool docker-compose used for local dev/test automation with those YAML files
- For more learning on docker compose:
  - https://docs.docker.com/compose/

docker-compose.yml

- Compose YAML format has it's own versions: 1, 2, 2.1, 3, 3.1
- YAML file can be used with *docker-compose* command for local docker automation
- docker-compose --help
- *docker-compose.yml* is default filename, but any can be used with *docker-compose -f*

docker-compose CLI

- CLI tool comes with Docker for Windows/Mac, but separate download for Linux
- Not a production-grade tool but ideal for local development and test
- Two most common commands are
  - *docker-compose up* # setup volumes/networks and start all containers
  - *docker-compose down* # stop all containers and remove cont/vol/net
- If all your projects had a Dockerfile and docker-compose.yml then "new developer onboarding" would be:
  - *git clone github.com/some/software*
  - *docker-compose up*

Assignment 3: Writing A Compose File

- Build a basic compose file for a Drupal content management system website.
  - Use Docker Hub
- Use the *drupal* image along with the postgres image
- Use *ports* to expose Drupal on 8080 so you can **localhost:8080**
- Be sure to set POSTGRES_PASSWORD for postgres
- Walk though Drupal setup via browser
- Tip: Drupal assumes DB is localhost, but it's service name
- **Extra Credit:** Use volumes to store Drupal unique data

Additional Resources:

- Docker Homepage
- Docker Hub
- Docker Blog
- Docker Documentation
- Docker Getting Started Guide
- Docker Help on StackOverflow