Docker

Feb 2018

Familiar territory ?

- Inconsistencies across environment
- Development is using JRE 8.0 but PROD is still on JRE 7.0
- Missing libraries
- Missing configuration
- Incompatible OS version

Java - Write Once and Run Everywhere or
Write Once and Debug Everywhere ?

Forget about those as I am going to give you a bundle / box / which just works

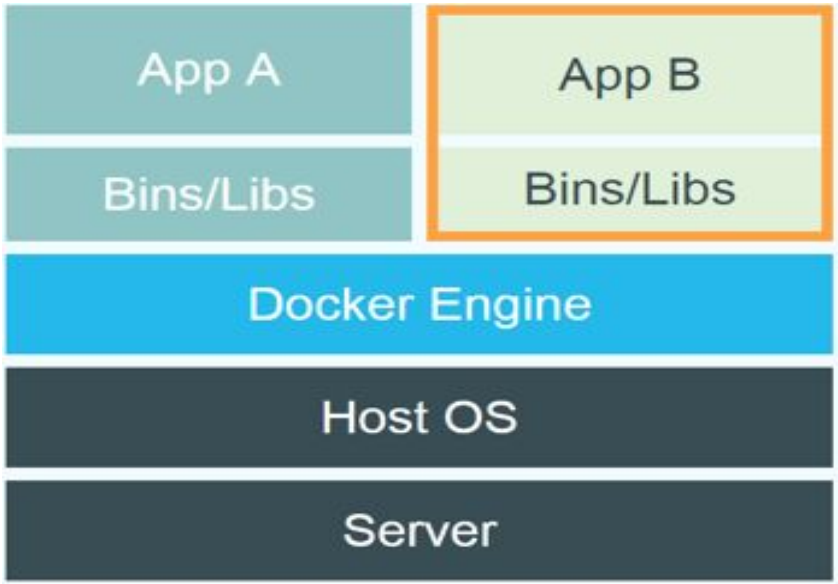
I.e. Container (e.g. Docker Container)

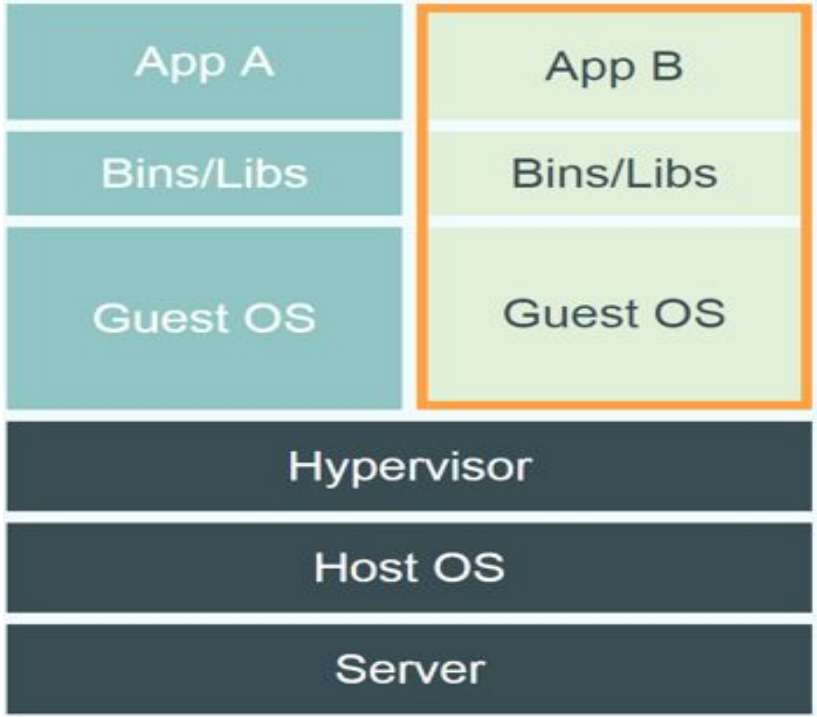
Overview of Docker

Docker Platform

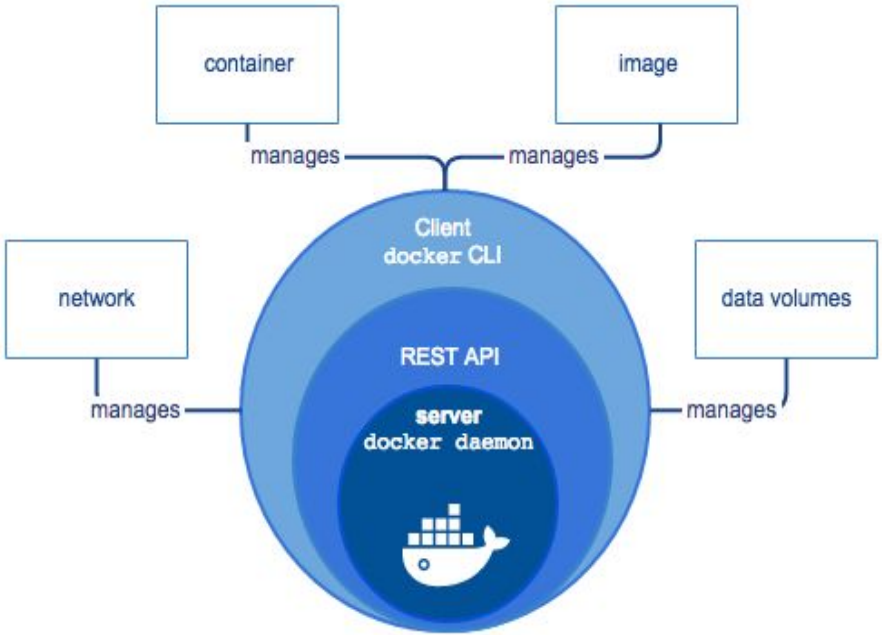
- Build, Ship and Run Methodology
- Provides ability to package and run an application in a loosely isolated environment called a container
- Isolation and security allow to run many containers simultaneously on a given host
- Containers are lightweight because they don’t need the extra load of a hypervisor
- Run directly within the host machine’s kernel
- More containers on a given hardware combination than if you were using virtual machines
- Tooling & Platform to manage lifecycle of containers
- Works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Docker Vs Virtual Machine





Docker Engine



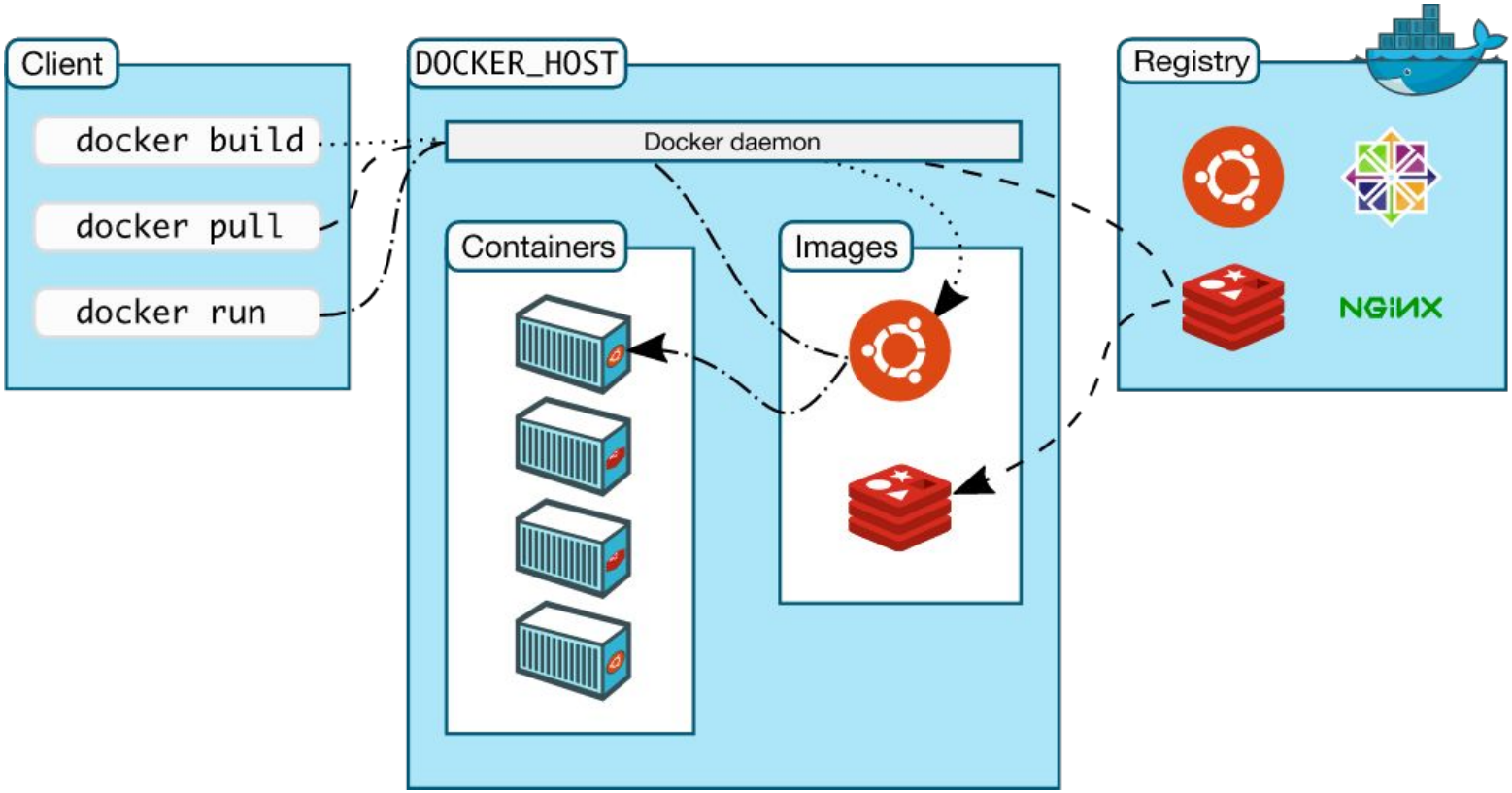
Client Server application

- Docker daemon (`dockerd`)
- REST API
- `Docker CLI`

Docker daemon manages objects such as

- `Images`
- `Containers`
- `Volumes`
- `Neworks`

Docker Architecture



Docker Daemon

- A.k.a `dockerd`
- listens for Docker API requests and manages Docker objects such as
  - Images
  - Containers
  - Networks &
  - Volumes

Docker Client

- `[docker] -` primary interface for connecting with docker daemon

Docker Registry

- Collection of Repositories
- Repository is a collection of images
  - Latest
  - V9.0
  - V8.0
  - …
  - V1.0 and so on
- Push/Pull images to/from repository
- Notation
  - `username/repository:tag`
- `Tag - typically used for versions`

Images

- read-only template with instructions for creating a Docker container
- *based on* another image, with some additional customization
  - E.g. 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.
- Image is built using *Dockerfile*
  - Provides 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 the image change and rebuilt only those layers which have changed are rebuilt. This makes images lightweight, small, and fast

Container

- Runtime instances of portable image
- Primary runtime unit - defined by image and any runtime parameters
- Container States - Start , Stop, Remove
- When a container is removed, any changes to its state that are not stored in persistent storage disappear - *ephemeral storage*
- Can create image from running container
- What happens when container is created - for details refer https://docs.docker.com/engine/docker-overview/#docker-objects

Services

- Allows to scale containers across multiple Docker daemons, which all work together as a *swarm* with multiple *managers* and *workers*
- Each member of a swarm is a Docker daemon, and the daemons all communicate using the Docker API
- A service allows you to define the desired state, such as the number of replicas of the service that must be available at any given time
- By default, the service is load-balanced across all worker nodes
- To the consumer, the Docker service appears to be a single application.
- Docker Engine supports swarm mode in Docker 1.12 and higher

Deploying applications using Docker

Container...

- Runtime instances of portable image
- **Image**
  - Contains what goes inside the container
  - Images are created based on Dockerfile
  - **Tag** - identifies the version of image
  - Shared via Registry
- **Dockerfile**
  - Defines what goes inside the container
  - Series of instructions
  - Access to networking elements, disk drives
  - Port mapping
  - E.g. EXPOSE - exposes the container port

...Container

- Container Commands
  - Running container
  - Running container in detached mode or background

Services

- Defines how containers behave in production
  - How many replicas to run
  - Restart behaviour
  - Limits on resources
  - Ports to be used
  - Network to be used
- Scale containers across multiple docker daemons
- By default, the service is load-balanced across all worker nodes.
- To the consumer, the Docker service appears to be a single application
- Uses `docker-compose`

```yaml
services:
  web:
    # replace username/repo:tag with your name and ima
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "80:80"
    networks:
      - webnet
networks:
  webnet:
```

...Services

- Changing the replicas results in IN-PLACE deployment

Stack

- Group of interrelated services that
  - share dependencies at the top level &
  - can be orchestrated and scaled together
- Capable of defining entire application
- Peer Service
  - Service at the same level
  - No dependency on other service

docker-compose

Docker Compose

- Tool for defining and running multi-container docker application
- YAML based
- Single Command to create and start all the services
- Start, Stop, Rebuild services

Docker SWARM

Docker Swarm

- Dockerized Cluster
  - Multi machine docker containers running as a group
- Group of Docker Containers
  - Notion of SWARM Manager - Initialize Master node
  - Notion of SWARM Worker
    - Worker nodes which have joined cluster
    - Can Leave SWARM cluster
- Docker daemon can be SWITCHED ON and OFF with SWARM cluster

SWARM Manager Deployment Strategies

- EMPTIEST_NODE
  - Deploy it on the node with least number of containers
- GLOBAL
  - Equal distribution of containers across nodes
- Strategies are controlled through `docker-compose` file

Swarm Worker Node

- Just do the work
- Can’t authorize other nodes to join cluster

Consequences & Benefits of Dockerized Approach

Docker Organization Consequence

The unit of scale being an individual, portable executable has vast implications.

- CI/CD can push updates to one part of a distributed application
- System dependencies aren’t a thing you worry about
- Resource density is increased
- Orchestrating scaling behavior is a matter of spinning up **new executables, not new VM hosts.**

Managing Data in Docker

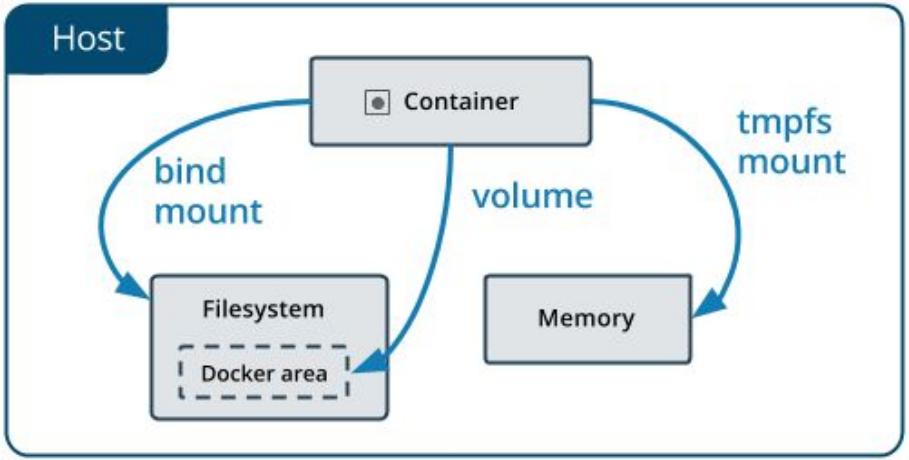
How to persist data with Docker ?

It is possible to store data within the writable layer of a container, but there are some downsides

- data doesn’t persist when that container is no longer running
- it can be difficult to get the data out of the container if another process needs it
- is tightly coupled to the host machine where the container is running. You can’t easily move the data somewhere else.
- Writing into a container’s writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel. This extra abstraction reduces performance as compared to using *data volumes*, which write directly to the host filesystem.

3 ways to persist data

- `Volumes`
  - `Stored in /var/lib/docker/volumes/`
  - `Managed by docker`
- `bind mounts`
  - may be stored *anywhere* on the host system.
- `tmpfs volumes`
  - stored in the host system's memory only



Revisit Redis Service Configuration

```
redis:
  image: redis
  ports:
    - "6379:6379"
  volumes:
    - "/home/docker/data:/data"
  deploy:
    placement:
      constraints: [node.role == manager]
  command: redis-server --appendonly yes
  networks:
    - webnet
```

Which type ?

Volume (Data Volume)

- Created & Managed by Docker
- Can be created explicitly or docker can create at the time of Service Creation
- Different lifecycle than Container
  - Even if the container is removed VOLUME remains
- Data is stored within a directory on the Docker host
- volume can be shared across containers as R,RW
- Can have Explicit NAME or anonymous (docker will give random unique name)
- Using *`volume drivers`* data can be stored at remote location (cloud providers)

Bind mounts

- Available since early days of docker
- a file or directory on the *host machine* is mounted into a container
- The file or directory is referenced by its full path on the host machine
- are very performant, but they rely on the host machine’s filesystem

`tmpfs` mounts

- Only in memory
- can be used by a container during the lifetime of the container, to store non-persistent state or sensitive information.
    - For instance, internally, swarm services use tmpfs mounts to mount secrets into a service's containers.

Which one to use ?

- Whenever in doubt - use VOLUMES
- Use case for Volumes
  - Sharing data across containers with R / RW control
  - Storing data on remote host or with cloud provider
  - Backup , restore or migrate data to another host
  - `/var/lib/docker/volumes/<volume-name>`
- Choosing bind mounts
  - Sharing configuration files from the host machine to containers. This is how Docker provides DNS resolution to containers by default, by mounting/etc/resolv.conf from the host machine into each container

Which one to use ?

- ...Choosing bind mounts
  - Sharing source code or build artifacts between a development environment on the Docker host and a container
  - you may mount a Maven target/ directory into a container, and each time you build the Maven project on the Docker host, the container gets access to the rebuilt artifacts
- Using `tmpfs` mounts
  - Sensitive information
  - Performance reasons - Large amount of data which need not be persisted

Key Notes for Using `'volume'` mount

- If you start a container with a volume that does not yet exist, Docker creates the volume for you
- If you start a container which creates a new volume, as above, and the container has files or directories in the directory to be mounted (such as `/app/`), the directory's contents are copied into the volume. The container then mounts and uses the volume, and other containers which use the volume also have access to the pre-populated content.
- RO, RW mount flags
- Data is stored under `/var/lib/docker/volumes/<vol_name>/_data/...`

Key Notes for Using `bind` mount

- Bind propagation
  - Available only for Linux
- Bind propagation refers to whether or not mounts created within a given bind-mount or named volume can be propagated to replicas of that mount

Key Notes for Using `tmpfs` mount

- Available only for Linux
- Bind propagation refers to whether or not mounts created within a given bind-mount or named volume can be propagated to replicas of that mount

Docker Networking

Networking Layer

- Pluggable architecture for Network using Drivers
- Bridge
  - Default choice
  - Used when your applications run in standalone containers that need to communicate
  - Restricted to SINGLE HOST
- Host
  - For standalone containers
  - remove network isolation between the container and the Docker host, and use the host's networking directly
  - but you want other aspects of the container to be isolated

...Networking Layer

- Overlay
  - connect multiple Docker daemons together and enable swarm services to communicate with each other
  - For Multi host configuration
- Macvlan
  - best choice when dealing with legacy applications that expect to be directly connected to the physical network
- None
  - Disable all networking (usually works with custom driver)
- Custom
  -

Working with `bridge` Network

- Default network
  - Not recommended for PROD
  - Can communicate to other containers using only ‘docker ips’
- User defined network
  - Recommended for PROD
  - Can communicate with other container using ‘docker ips’ +
  - Can communicate with other containers using ‘names’ . This capability is called **automatic service discovery**.

Working with `overlay` network

- Enables the connectivity between containers running on different docker hosts
- Requires the manager node running in Swarm mode
- Not available to containers started with `docker run` that don't run as part of a swarm mode service
- Create one using

```
Docker network create --driver overlay --subnet 10.0.9.0/24 ol-nw
```

Container and Networks

- Container can be part of multiple networks
- Containers can be attached and detached from one network to another

Using Dockerfile - building custom image

Dockerfile

- Provides 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
- **INSTRUCTION** is not case sensitive but convention is to use CAPITAL letters
- **The first instruction must be `FROM`** in order to specify the *Base Image* from which you are building.
- When the image change and rebuilt only those layers which have changed are rebuilt. This makes images lightweight, small, and fast

Docker Use Cases

Use cases

- Development Environment
- Environments for Integration Tests
- Quick evaluation of software
- Microservices
- Multi-Tenancy
- Unified execution environment (dev,test,prod)

Docker Commands

Simple Command

```
docker run ubuntu echo Hello World
```

`Docker` : tells OS that docker is being used

`run` : creates a container to run specified image

`ubuntu` : image name to be launched inside container

`echo` : command to run inside the container

Flags

-d : detached (daemonized)

-t : pseudo terminal inside current

-i : interactive

-f : forcefully

-p : tells docker to map required port of container to host

Multiple flags can be used together for specific requirements

-it , -dt, -idt etc...

Other Parameters

--name : name of container

Log : shows the standard output of container

Start : starts the container

Stop : stops the container

rm : remove the container

rmi : remove image

Tag : tags the image

ps : shows running containers

Commit : creates image from container

Supported Platforms

Installation Platforms

- Various Linux distributions (Ubuntu, Fedora, RHEL, Centos, openSUSE, ...)
- Cloud (Amazon EC2, Google Compute Engine, Rackspace)
- 64-bit Windows 10 Pro; future release will support more Windows OS versions.

Thanks