Docker

Overview

Docker provides the ability to package a run an application in a loosely isolated environment called a container. Since the containers share the host machines kernel while still be isolated, they do not require a hypervisor like VMWare making them very lightweight, scalable, and quick to setup.

Since docker containers are isolated and run in standardised environments, they fit in nicely with CI/CD workflows such as TravisCI. Devs can run development containers locally and then push them to the CI pipeline for testing and deployment.

# Architecture

General

Docker uses a client-server architecture:

* Daemon - Builds, runs, and distributes containers. Runs on client server listening for Docket API requests.
* Client - Talks to daemon to tell it what to do, use can interface with the client using scripting. The client and daemon can run on the same machine, communicating through a UNIX socket, or they can be remote.
* Registries - A docker registry stores docker images, for example Docker hub is a public registry which anyone can use and is where docker looks for images by default. It is possible to run a private registry of use the Docker Datacenter.

Namespaces are used to provide isolated workspaces called the container. When a container is created a set of namespaces are also created, each for a different layer of the container. Each different aspect runs in a different namespace, allowing for isolation. On Linux, some of the namespace are:

* pid - Process isolation
* net - Network management
* ipc - Interprocess communication
* mnt - filesystem mount points
* uts - Kernel and version identifiers

Control group (cgroups) are also used to limit the application to a specific set of resources. This allows the Docker engine to share hardware resources among containers, enforcing limits and constraints.

Union file systems operate by creating layers, making them very lightweight and fast. This is how Docker create building blocks for containers. Various UnionFS are available: AUFS, btrfs, vfs, and DeviceMapper.

Docker Objects

The daemon creates and manages Docker objects such as:

* Image - Read only template with instructions for creating the docker container. Most images are based off other, such as a server image based off the ubuntu image, modified to run a reverse proxy. To create an image a Dockerfile is made, with each step creating a layer in the docker image.
* Container - Runnable instance of an image which can be started, stopped, moved, and deleted.
* Services - Services allow for multiple Docker daemons to run containers in a load-balanced swarm. The services allows for a system admin to define how many containers of each service need to be running, and the load-balancing between them.
* Networks -
* Volumes - Persistent linked storage on the host machine

Docker Engine

The docker engine is the client-server application with the major components to run the system:

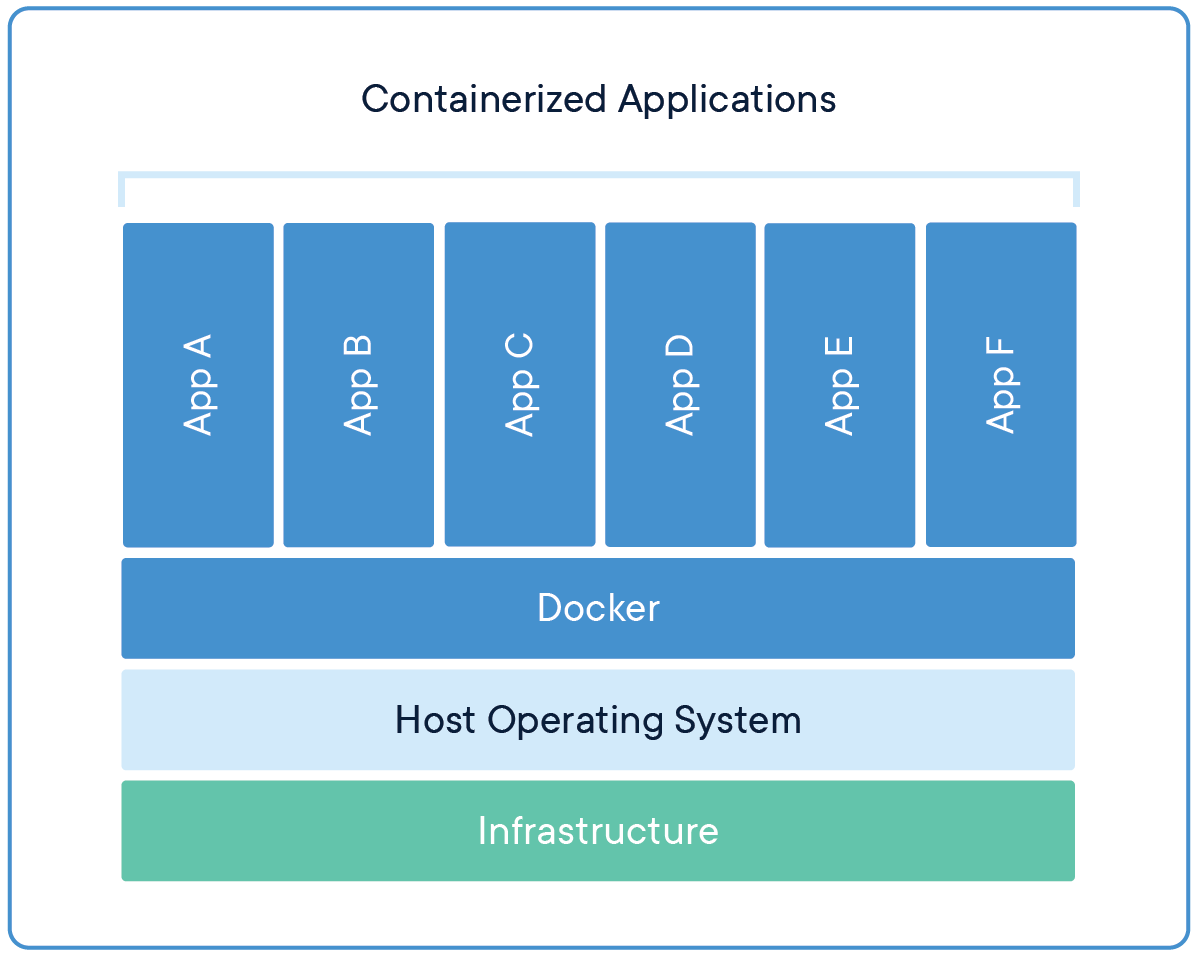
* Server - Long running daemon process (dockerd)
* REST API which specifies the interface that programs can use to talk to the daemon
* CLI - docker command which uses the REST API

Container Format

The Docker Engine combines the namespaces, control groups, and UnionFS into a wrapper called a container.

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

The default format is libcontainer.



In general, containers should do one thing and do it well, this is due to many reasons:

* Front end APIs and backends applications (such as databases) often need to be scaled at different rates
* Separate containers for different applications allow for updates to be applied in isolation
* Backend services like databases which are normally a single application in development can be swaped for managed services when deployed for production
* Running multiple processes requires a process manager, containers only start one process, so more complexity will be added at startup and shutdown

# Docker CLI

Build Image from Dockerfile

To build a container image from a Dockerfile, run the build command:

docker build -t <image-name> <dockerfile-loc>

Args:

* -t <image-name> - give the container image a descriptive name
* <dockerfile-loc> - path to dockerfile to build container image from, generally '.' when the terminal is in the root directory of the project
* -v <volume-name>:<mapping-path> - map a host machine volume to the container mapping path
* -e <env-variable>=<value> - add environment variables to a container

Run Container from Image

The 'docker' command is used to build and run docker containers. To run a built container use the run command:

docker run -d 80:80 <image-file>

Flags:

* -d - detached mode, container runs in background
* -p 80:80 - maps port 80 on the host to port 80 of the container environment
* <image-file> - path to image file to mount

Swap Images

After updates, docker containers can be rebuilt and swapped by stopping and restarting the new image. To do this get the container id of the currently running image:

docker ps

Stop the container which is to be swapped:

docker stop <container-id>

Remove the old container using the remove command:

docker rm <container-id>

Then start the new container using the docker run command as before. The above command can be combined using the force flag on the rm command:

docker rm -f <container-id>

Run Command in Container

To run a command in a docker container from the terminal use the exec command:

docker exec <container-id> <command>

The command stdout will be piped to the current terminal

Tails Logs

To see the stdout log of a docker container in realtime, run the command:

docker logs -f <process-id>

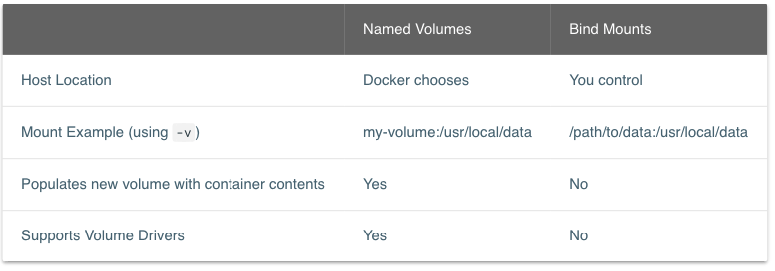
# Volumes

General

Since each docker container is its own isolated system, each image has its own local storage and filesystem. Therefore, local fs changes in one docker container, will not affect fs of another identical container running on the same machine.

Volume provide the ability to connect specific filesystem paths of a container back to the host machine. If the same directory is mounted between different containers, they will share the same files.

There are two main types of volume which docker uses:



Named Volumes

Named volumes are like a bucket of data which docker maintains on the host machine. To use a named volume, only the name is required.

Create a named volume using the volume create command:

docker volume create <volume-name>

The volume can then be mapped to a container image in the run command using the -v flag:

docker run -dp 3000:3000 -v <volume-name>:<mapping-path> <image-name>

The volume will then be mapped to the <mapping-path> inside the docker container.

The docker named volume can checked using the inspect command:

docker volume inspect <volume-name>

Note: when using docker desktop, the docker system is run inside a VM on the host machine, therefore paths may not be exact, since they are relative to the VM.

Note: when starting a container, if the volume tag is passed but the named volume has not yet been created, so long as the name does not match a directory, docker will automatically create the named volume

Bind Volume

A bind volume is similar to a named volume, expect the location on the host machine is directly specified and not maintained by the docker engine.

-v <host-dir>:<mapped-volume>

Bind volumes can be useful for injecting source code into the container while being able to watch it from the host machine. For example, setting up a dev mount and using nodemon to reload the docker image each time a change is detected. This can be done by loading the app source as the working directory of the container:

docker run -dp 3000:3000 -w /app -v ${PWD}:/app node:12-alpine sh -c "yarn install && yarn run dev"

# Networks

General

Networks can be used to link containers and allow them to interact with eachother, such as a frontend api interacting with a database. For containers to interact, they must be on the same network.

Networks can be created by the docker engine using the network create command:

docker network create <network-name>

Once a network is created, there are two ways to put a container onto a network, assign it at start or connect to already running. To add on start use the --network flag:

docker run -d --network <network-name> --network-alias <network-alias> <image-name>

Adding a network alias, effectively add a dns A record to the network, allowing the container to resolve the ip address by alias name instead of ip address.

Default Bridge

All containers started without a network specified are added to the default bridge. This means they can communicate with any other containers on the network, which can be an issue for effective isolation.

Netshoot

The nicolaka/netshoot container contains a lot of network tools which are useful for debugging network issues. Start the container using:

docker run -it --network <network-to-debug> nicolaka/netshoot

# Environment

Secrets

While using environmnet variables to set application setting in development, it is discouraged for use in production. This is because passing application secrets as environment variables makes them prone to being accidentally exposed due to a variety of reasons:

* environment is implicitly available to the process, making it hard to track access and how contents get exposed
* applications often dump the environment in error reporting, leading to secrets in plain text logs etc
* environment is passed to child process, meaning any spawned process such as third party apps will have access to the environment

Therefore, secrets should be managed by the container orchestrator such as kurbernetes or swarm. For example, swarm uses the docker secret command:

docker secret create secure-secret

Another option is if the application supports file environment variables, where the environment variables is stored in a file which can be mounted securely and accessed via application the variable \_file, such as MYSQL\_PASSWORD\_FILE.

# Project Setup

General

To create a new container image which can be deployed, run, and maintained, several files must be created to manage its creation and the environment it will run in.

Dockerfile

The Dockerfile is a text based script placed in the project root directory which is used to generate the container image. The Dockerfile script is run in order and should contain install commands for all the dependencies of the project.

There are several key parts to a Dockerfile:

* FROM <image-name> - defines which image to base the new image off, standard images a normally pulled from the dockerhub
* WORKDIR
* COPY
* RUN
* CMD ['command1', 'command2'] - The CMD directive specifies the default commands to run when starting a container from this image

Docker Compose

Docker Compose is an application which defines multi-container docker applications in a YAML file. The file can then be run with a single command to generate the system.

In the project root directory create the 'docker-compose.yml'. The first entry in the compose file is the schema version, generally use the most upto date schema version from the composite website.

version: '3.8'

Next define the services (containers) which will run in the system. Each service is placed under the 'services' key as its container name. For example, the todo app:

services:

app:

image: node:12-alpine

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

ports:

- 3000:3000

working\_dir: /app

volumes:

- ./:/app

environment:

MYSQL\_HOST: mysql

mysql:

image: mysql:5.7

volumes:

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

environment:

MYSQL\_ROOT\_PASSWORD: secret

MYSQL\_DATABASE: todos

Named volumes are then added under the volumes key:

volumes:

todo-mysql-data:

Docker compose automatically creates a network for the composed system.

To run the docker compose file, navigate to the root directory and run the command:

docker-compose up -d

To stop the system, run:

docker-compose down

add the --volumes tags to the above command to remove volumes created during the up command.

Docker compose displays logs for each container in the same log file:

docker-compose logs -f app

Wait Port

Some containers need to wait for others to be fully running before attempting to connect. One such application for Node based project is the 'wait-port' dependency.r3

# Best Practices

Image Size

One of the most challenging aspects of building docker images is keeping the image size down. Since each instruction in a Dockerfile adds a layer to the image, to build an efficient Dockerfile shell tricks and logic needs to be used to ensure each layer has the artifacts it requires to run, but nothing else.

Layer Caching

In the Dockerfile, each line creates a new layer in the container image. When a layer changes, all downstream layers have to be recreated aswell. Therefore, to speed up docker build times, add and build stable dependencies in the Dockerfile first, then add less-stable depencies and the application in later. This way the solid dependencies won't be rebuilt everytime the application changes.

For example, changing:

FROM node:12-alpine

WORKDIR /app

COPY . .

RUN yarn install --production

CMD ["node", "/app/src/index.js"]

to:

FROM node:12-alpine

WORKDIR /app

COPY package.json yarn.lock ./

RUN yarn install --production

COPY . .

CMD ["node", "/app/src/index.js"]

Will install the npm dependencies first, then copy in the application. When docker rebuilds the image again, it will pull the npm dependencies from cache, speeding up the build greatly.

Multi-Stage Builds

Mutli-stage builds can be used separate the stages which the image is built in to optimise build time. There are several advantages to this:

* Separate build-time dependencies from runtime dependencies
* Reduce overall image size by shipping only what is needed to run

For example, in a React build where there is no serverside rendering, the static assets can be served in a static ngnix container. Therefore, use a multi-stage build to first generate the assests in a node environment, then pass those assets to the next container to save container image size since no node environment is required when running the image.

FROM node:12 AS build

WORKDIR /app

COPY package\* yarn.lock ./

RUN yarn install

COPY public ./public

COPY src ./src

RUN yarn run build

FROM nginx:alpine

COPY --from=build /app/build /usr/share/nginx/html

# Docker Desktop

General

Docker desktop can be run on Macos or Windows, giving a graphical interface to the containers running on the local machine. It also provides quick access to container logs and shell into the container.

# Dockerhub

General

Dockerhub is a public registry of docker container images which can be downloaded and used or customised.

Pushing to Dockerhub

First login to dockerhub using the login command on the CLI

docker login -u <username>

To push a container image to docker, it must be namespaced with your username. To do this tag a local container image with your username namepace:

docker tag <image-name> <username>/<image-name>

Once namespaced, push the image:

docker push <username>/<image-name>