# Docker cookbook

## Install

### mac

```
brew install docker
brew install docker-compose
```

### ubuntu

```
# remove the old docker
sudo apt-get remove docker docker-engine docker.io
# install repo
sudo apt-get update
# install packages to allow apt to use a repository over HTTPS:
sudo apt-get install apt-transport-https ca-certificates curl software-properties-common

# add Docker’s official GPG key:
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

# add repo
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

# install docker
sudo apt-get update
sudo apt-get install docker-ce


# install Compose 1.23.1
https://docs.docker.com/compose/install/#master-builds

sudo curl -L "https://github.com/docker/compose/releases/download/1.23.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

## Try it

### pull latest ubuntu

`docker pull ubuntu:latest`

### start a container and login

`docker container run -it ubuntu:latest /bin/bash`

`docker container run` tells the Docker daemon to start a new container. The `-it` flags tell Docker to make the container interactive and to attach our current shell to the container's terminal. The image is latest ubuntu. The contain will run `/bin/bash` when the container starts.

docker will give your newly created container a random name if you don't assign one for it. Use `--name` to add name for it
`docker container run --name webserver -it ubuntu:latest /bin/bash`

**Attaching to a running container**

1. check which containers are running
   `docker container ls`
2. attach a bash to it
   `docker container exec -it (name or id of the runing container) bash`

## Dockerfile

Dockerfile is a plain-text document describing how to build an app into a Docker image.

```
FROM alpine
LABEL maintainer="xxxx"
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN  npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

build a image based on Dockerfile.
`docker image build -t test:0.01 .`
add a name and optionally a tag for the image. `.` means docker image builder should find a Dockerfile in the current location.

check the newly created image
`docker image ls`

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
test                0.01                551943a101d3        2 minutes ago       68.1MB
```

create a container on top of it
`docker container run -d --name web1 --publish 8080:8080 test:0.01`

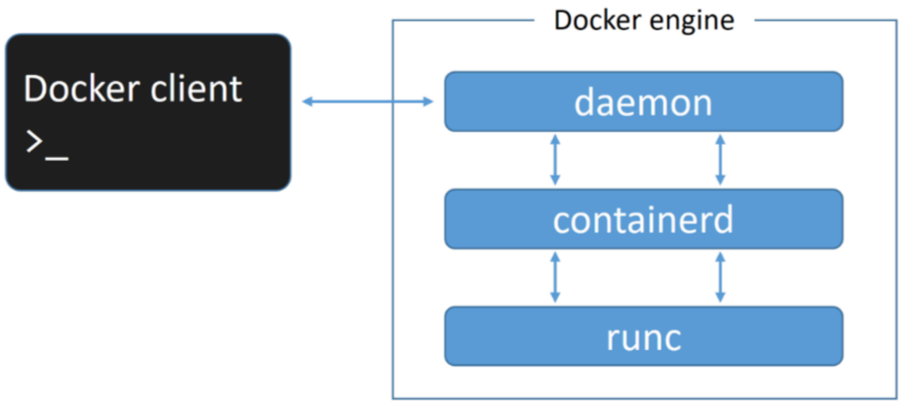
check if the new container is running

```
CONTAINER ID  IMAGE     COMMAND          CREATED        STATUS       PORTS     NAMES
7cf67b047493  test:0.01 "node ./app.js"  4 minutes ago  Up 4 minutes 0.0.0.0:8080->8080/tcp
web1
```

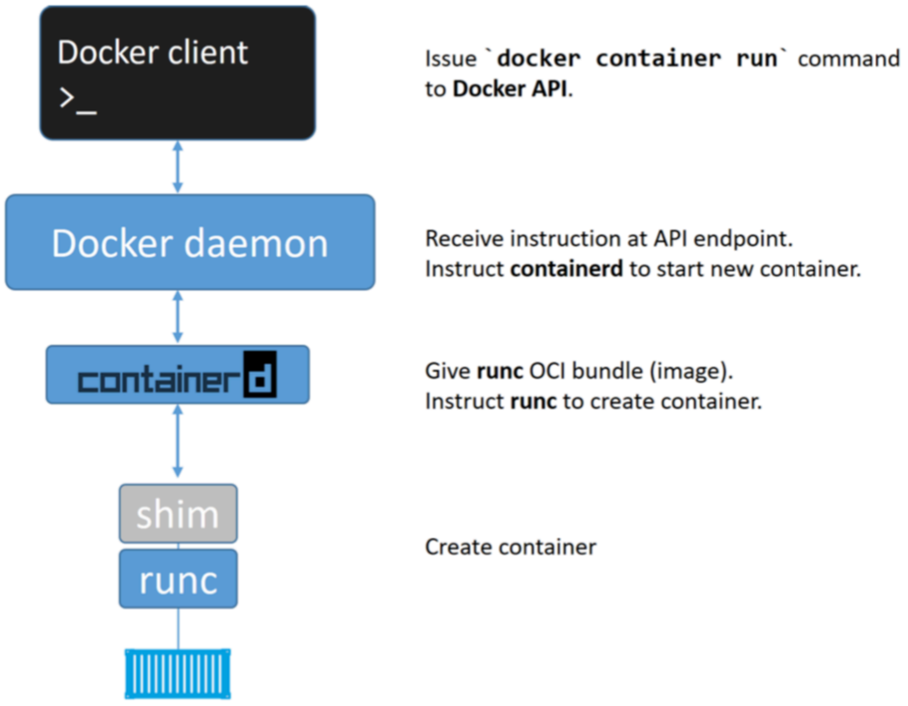
explanation of the arguments in the command:

1. `-d`: run container in background and print container ID
2. `--name`: assign a name to the container
3. `--publish , -p`: publish a container's port(s) to the host

## docker engine



how they work toghter



Issue `docker container run` command to **Docker API**.

Receive instruction at API endpoint. Instruct **containerd** to start new container.

Give **runc** OCI bundle (image). Instruct **runc** to create container.

Create container

## docker image

Blueprint for the container.

- Image must contain all OS and application files required to run the app/service.
- Image don't contain a kernel — all containers running on a Docker host share access to the host's kernel.

- Docker image is just a bunch of loosely-connected read-only layers.

Cannot delete the image until the last container using it has been stopped and destroyed.

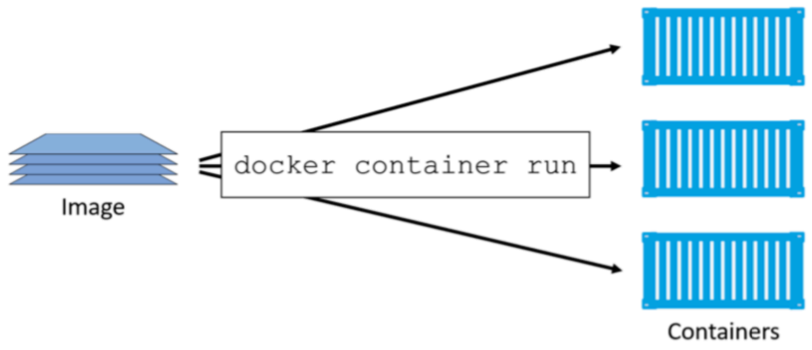
Docker images are stored in image registries.

pull latest ubuntu
try `docker image pull ubuntu:latest`. If you do not specify an image tag after the repository name, Docker will assume you are referring to the image tagged as latest. However, `latest` is an arbitrary tag and is not guaranteed to point to the newest image in a repository. Add `-a` to pull all images in the repo.

- pull: `docker image pull <repository>:<tag>`
- remvoe dangling images: `docker image prune`
- filter: `docker image ls --filter key=value`
- format: `docker image ls --format "{{.Size}}"`
- search for repo: `docker search *****`
- inspect image: `docker image inspect`
- remove image: `docker image rm`
- remove all images: `docker image rm $(docker image ls -a -q) -f`
- build new image based on dockerfile: `docker image build -t test:0.01 .`
- push image to registry: `docker image push username/web:latest`

### docker container

A container is the runtime instance of an image.



killing the main process in the container will also kill the container. Ctrl-PQ to exit the container without terminating it.

Example:
`docker container run -it ubuntu /bin/bash`
`-it` flags will make the container interactive and connect your current terminal window to the container's shell.
Containers run until the app they are executing exits.
Attaching to a container will create a new process.

#### lifecycle

- start a new container: `docker container run <options> <image>:<tag> <app>`
- start a stoped a container: `docker container start`
- stop a container: `docker container stop`
- remove a container: `docker container rm`
- reattach to a container: `docker container exec -it <container> bash`

#### restart policy

Enable Docker to automatically restart containers after certain events or failures have occurred.

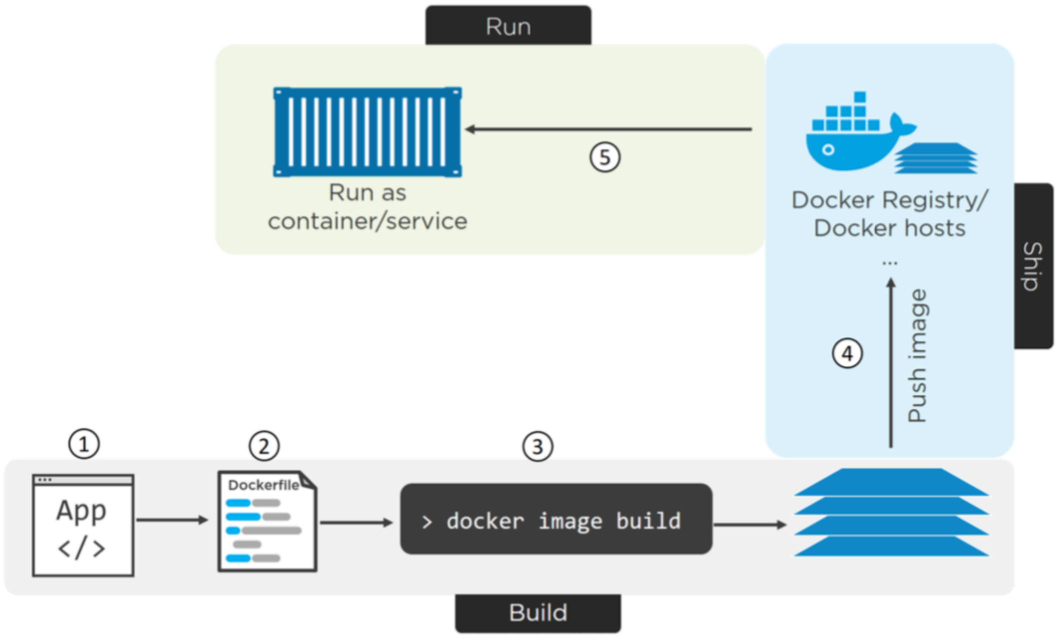
- `--restart always`: will always restart a stopped container unless it has been explicitly stopped, and the stopped container will be restarted when the Docker daemon starts.
- `--restart unless-stopped`: will always restart a container except when the container is in exited state.
- `--restart on-failure`: will restart a container if it exits with a non-zero exit code.

```
restart_policy:
  condition: always | unless-stopped | on-failure
```

#### map port

`-p <host port>:<container port>`: maps ports on the Docker host to ports inside the container.

### containerize an app



```
# base layer is alpine
FROM alpine
# add a label
LABEL maintainer="nigelpoulton@hotmail.com"
# RUN create a new temperary image as a layer with nodejs and npm installed in it
RUN apk add --update nodejs nodejs-npm
# copy files into a new image as a layer
```

```
COPY . /src
# set the working directory for the rest of the instructions in the file
WORKDIR /src
# install dependencies in the new image as a layer
RUN npm install
# export tcp 8080
EXPOSE 8080
# Entrypoint sets the command and parameters that will be executed first when a container is run.
ENTRYPOINT ["node", "./app.js"]
```

If an instruction is adding content such as files and programs to the image, it will create a new layer. If it is adding instructions on how to build the image and run the application, it will create metadata.

`docker container run -d --name c1 -p 80:8080 web:latest`
create a container named c1 that is based on web:lastest image, and map 80 of host to 8080 of c1.

`docker image history <docker image>` checks how the image is built. Some instructions in the dockerfile create intermediate layers, sizes of the images are showed in the result.

General process to build an image

1. create a temprary container
2. run a line of the instructions in the dockerfile inside of the container
3. save the result as a new image layer
4. remove the temprary container

Optimization:

- combind multiple RUN commands as part of a single RUN instruction, glued together with double-ampersands (&&) and backslash () line-breaks

### Multi-stage Builds

```
FROM node:latest AS storefront
WORKDIR /usr/src/atsea/app/react-app
COPY react-app .
RUN npm install
RUN npm run build

FROM maven:latest AS appserver
WORKDIR /usr/src/atsea
COPY pom.xml .
RUN mvn -B -f pom.xml -s /usr/share/maven/ref/settings-docker.xml dependency:resolve
COPY . .
RUN mvn -B -s /usr/share/maven/ref/settings-docker.xml package -DskipTests

FROM java:8-jdk-alpine
RUN adduser -Dh /home/gordon gordon
WORKDIR /static
# copy the build file to this image
```

```
COPY --from=storefront /usr/src/atsea/app/react-app/build/ .
WORKDIR /app
# copy the compiled file to this image
COPY --from=appserver /usr/src/atsea/target/AtSea-0.0.1-SNAPSHOT.jar .
ENTRYPOINT ["java", "-jar", "/app/AtSea-0.0.1-SNAPSHOT.jar"]
CMD ["--spring.profiles.active=postgres"]
```

The final image does not include `node:latest` and `maven:latest`, so it is much smaller.

The ENTRYPOINT specifies a command that will always be executed when the container starts.
The CMD specifies arguments that will be fed to the ENTRYPOINT. The CMD instruction in the dockerfile can be overrided by the the argument in the docker run command.

https://docs.docker.com/v17.09/engine/userguide/eng-image/multistage-build/#before-multi-stage-builds

#### cache

When docker excutes instructions in the dockerfile, it will try to find cache that matches the instruction. If the match can be found, docker will skip the instruction, link to that existing layer, and continues the build with the cache in tact. However, if any instruction results in a cache-miss, the cache is no longer used for the rest of the entire build.

Docker performs a checksum against each file being copied, and compares that to a checksum of the same file in the cached layer. If the checksums do not match, the cache is invalidated and a new layer is built.

`--no-cache=true` forces docker to ignore cache

Optimization:

- Try and build them in a way that places any instructions that are likely to change towards the end of the file.

#### squash

`docker image build --squash` can squashes all layers into one single layer. Squashed images do not share image layers

#### no-install-recommends

Use the `no-install-recommends` flag with the `apt-get install` command.

### docker compose

Compose uses YAML files to define multi-service applications.

```
version: "3.5"
services:
  web-fe:
    build: .
    command: python app.py
    ports:
      - target: 5000
        published: 5000
    networks:
      - counter-net
    volumes:
      - type: volume
        source: counter-vol
        target: /code
  redis:
    image: "redis:alpine"
    networks:
      counter-net:

networks:
  counter-net:

volumes:
  counter-vol:
```

- version: defines the version of the Compose file format [https://docs.docker.com/compose/compose-file/compose-versioning/](https://docs.docker.com/compose/compose-file/compose-versioning/)
- services: defines the different application services
- networks: creates new networks
- volumes: creates new volumes

Each service will become a container.
`web-fe` will build the image that defined in the Dockerfile that is in the current location, then create a container and run `python app.py` as the main app. The port 5000(-target) in the container will be mapped to port 5000(published) inside the host.

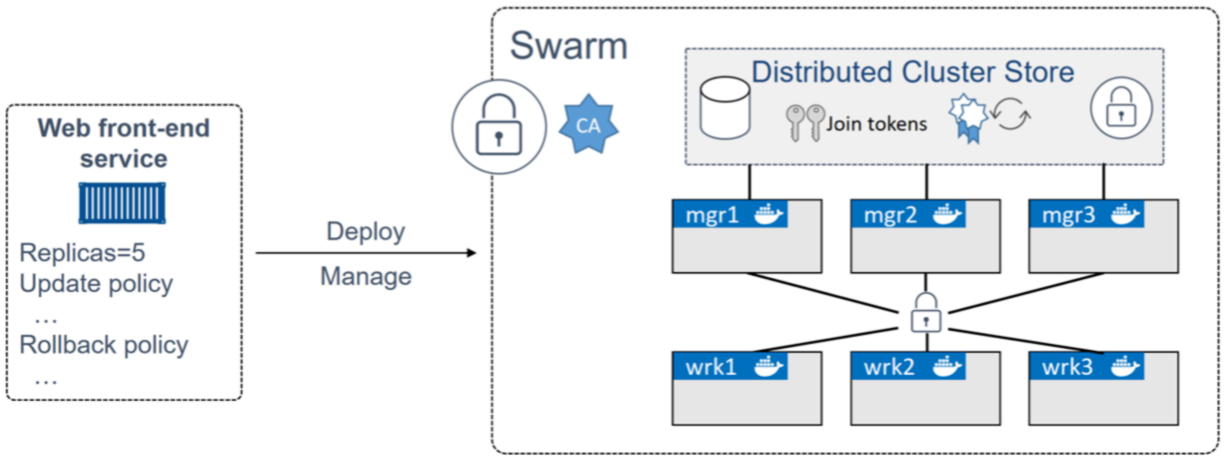
- `docker-compose up`: deploy a Compose app. and `&` to give the terminal back
- `docker-compose stop`: stop all of the containers in a Compose app without deleting them from the system.
- `docker-compose rm`: delete a stopped Compose app. It will delete containers and networks, but it will not delete volumes and images.
- `docker-compose restart`: restart a Compose app that has been stopped with docker-compose stop. If you have made changes to your Compose app since stopping it, these changes will not appear in the restarted app. You will need to re-deploy the app to get the changes.
- `docker-compose ps`: list each container in the Compose app.
- `docker-compose down`: stop and delete a running Compose app. It deletes containers and networks, but not volumes and images.

## Swarm

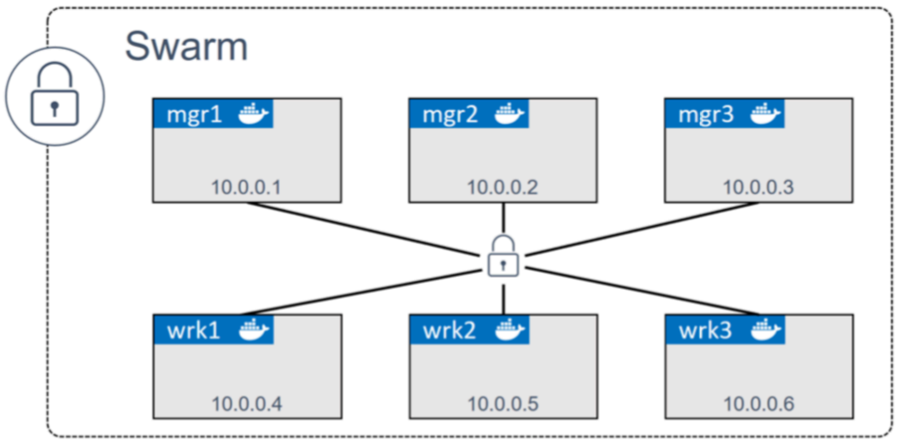
Docker Swarm is two things: an enterprise-grade secure cluster of Docker hosts, and an engine for orchestrating microservices apps.

a cluster has one or more Docker nodes. Nodes are configured as managers or workers. Managers look after the control plane of the cluster, meaning things like the state of the cluster and dispatching tasks to workers. Workers accept tasks from managers and execute them.

orchestration manages the lifecycles of containers.



**build a secure swarm cluster**



prerequisites:

- 2377/tcp: for secure client-to-swarm communication
- 7946/tcp and 7946/udp: for control plane gossip
- 4789/udp: for VXLAN-based overlay networks

1. create one manager and one worker

```
docker-machine create -d xhyve --xhyve-boot2docker-url https://github.com/boot2docker/boot2docker/releases/download/v18.06.1-ce/boot2docker.iso mgr1
docker-machine create -d xhyve --xhyve-boot2docker-url https://github.com/boot2docker/boot2docker/releases/download/v18.06.1-ce/boot2docker.iso wrk1
```

2. log on the mgr1 and initialize swarm.

```
docker-machine ssh mgr1
#logged
docker swarm init \
--advertise-addr <ip of mgr1>:2377 \
--listen-addr <ip of mgr1>:2377
```

3. log on wrk1 and join the swarm as worker

```
docker-machine ssh wrk1
#logged
docker swarm join --token <token> <ip of mgr1>:2377 \
--advertise-addr <ip of wrk1>:2377 \
--listen-addr <ip of wrk1>:2377
```

4. check the swarm in the mgr1

```
docker node ls
```

The swarm should have multiple managers, and only one of them is "active" leader. The leader is the only one which can issue commands against swarm. If non-active managers receive commands for the swarm, them will proxy those commands to the leader.

Note: deploy odd number of managers. 3 or 5

**prevent the restarted managers joining the cluster**

`docker swarm update --autolock=true` will lock the swarm and return a token for the restarted managers to rejoin the cluster.
`docker swarm unlock`

- `docker swarm init`: initialize a new swarm and make this node the first manager
- `docker swarm join-token worker/manager`: show commands to join the swarm

**service**

`docker service create --name web-fe -p 8080:8080 --replicas 5 <image>`

The default replication mode of a service is replicated. This will deploy a desired number of replicas and distribute them as evenly as possible across the cluster.
The other mode is global, which runs a single replica on every node in the swarm.

- `docker service ps <service name>`
- `docker service scale <service name>=10`: change the number of containers among nodes
- `docker service rm <service name>`: remvoe the service. contaienrs among the nodes will be removed too
- `docker service logs <service name>`
- `docker node ls`

## Networks

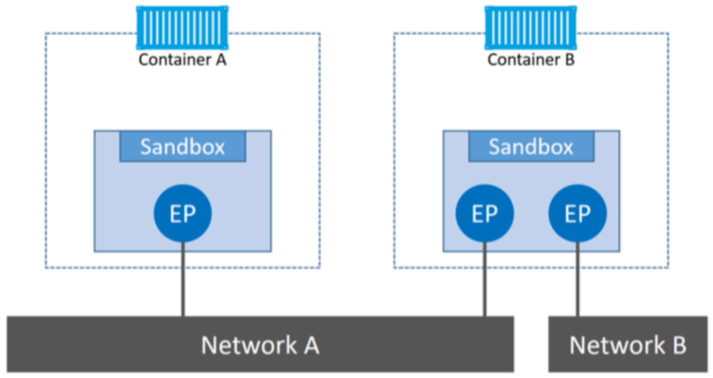
There are three default networks: bridge, null, host. and one virtual switch docker0

**Container Network Model**

design guide for Docker networking.

https://github.com/docker/libnetwork/blob/master/docs/design.md

- Sandboxes
  Isolated network stack. It includes; Ethernet interfaces, ports, routing tables, and DNS config.
- Endpoints
  virtual network interfaces that connect a sandbox to a network.
- Networks
  software implementation of an 802.1d bridge



**Libnetwork**

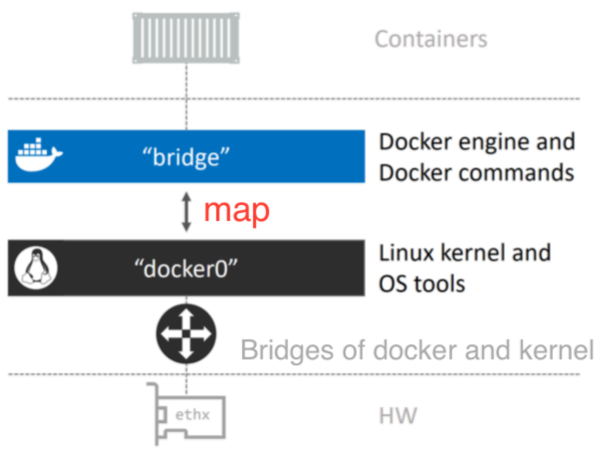
CNM implementation

**Driver**

Driver is in charge of the actual creation and management of all resources on the networks.

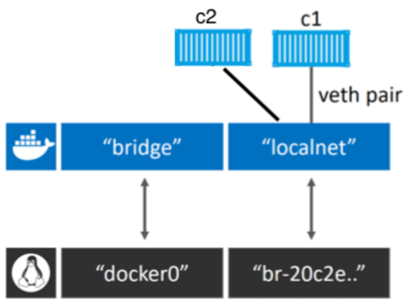
**Single-host bridge networks**

Every Docker host gets a default single-host bridge network. By default, this is the network that all new containers on this host will attach to unless other networks are specified.

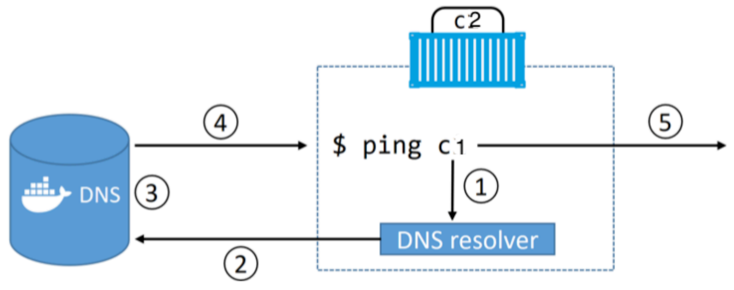


create a network and attach a container to it:

1. create a network
   `docker network create -d bridge localnet`
2. create a container c1 which attaches to the network
   `docker container run -d --name c1 --network localnet alpine sleep 1d`
3. create another container c2 which also attaches to the network
   `docker container run -it --name c2 --network localnet alpine sh`
4. ping c1 from c2
   `ping c1`



Name resolution:
c2 container is running a local DNS resolver that forwards requests to an internal Docker DNS server which maintains mappings for all containers. However, default bridge network on Linux does not support name resolution and name resolution only works for containers and Services on the same network..



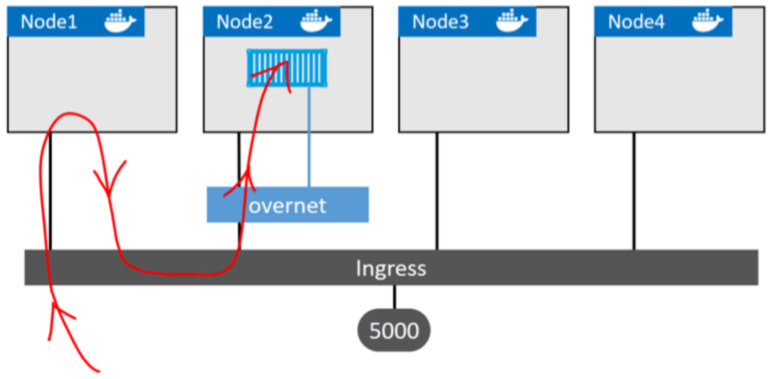
1. The `ping c1` command invokes the local DNS resolver to resolve the name "c`" to an IP address.

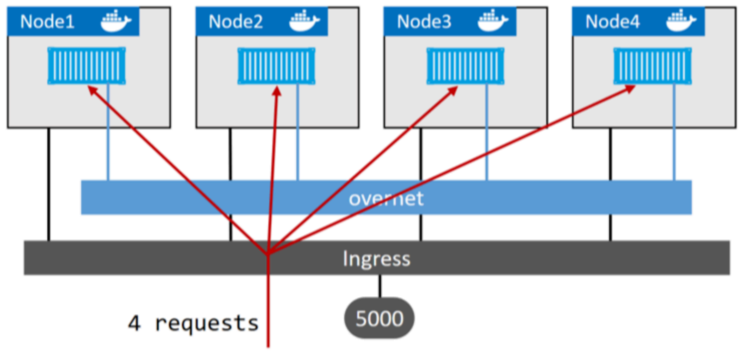
2. If the local resolver does not have an IP address for “c1” in its local cache, it initiates a recursive query to the Docker DNS server.
3. The Docker DNS server holds name-to-IP mappings for all containers created with the --name or --net-alias flags.
4. The DNS server returns the IP address of “c1” to the local resolver in “c2”.
5. The ping command is sent to the IP address of “c1”.

Port mappings let you map a container port to a port on the Docker host. Any traffic hitting the Docker host on the configured port will be directed to the container.

**Load balancing**

Services accessibility from outside of the cluster

- Ingress mode(default)
  from any node in the Swarm
  

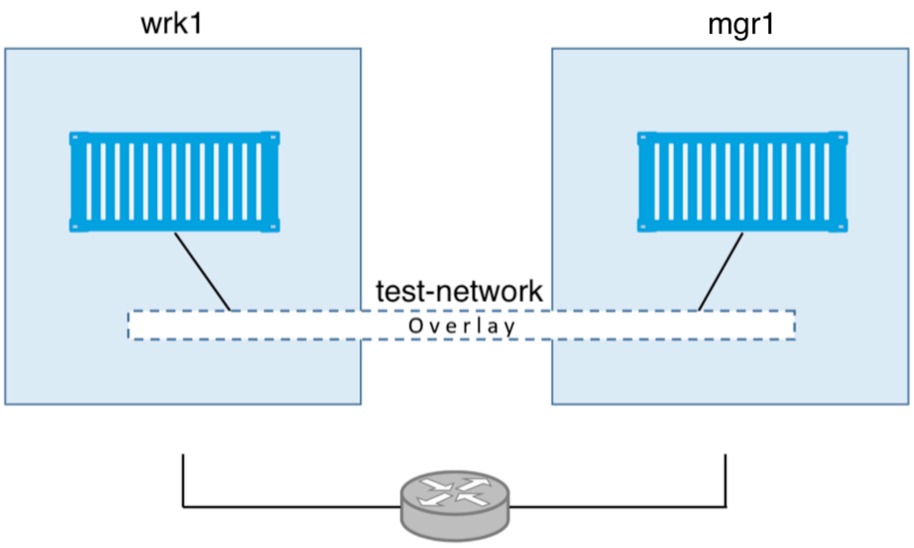
  The traffic to 5000 of node 1 will be routed to node2 which the container is running in.
  

  The traffic is balanced
- Host mode
  nodes running service replicas
- `docker network create -d <driver> <name>`
- `docker network rm`
- `docker network inspect`

### Overlay network

Build a overlay network for a swarm.

1. create a overlay network `test` on mgr1, wrk1 cannot see it because it doesn't have containers that have attached to it yet.
   `docker network create -d overlay test-network`
2. create a serviec that attaches to the new overlay network, now the wrk1 can see the new overlay network
   `docker service create --name test-service --network test-network --replicas 2 alpine sleep 1d`
3. get the ip of the newly created containers
   `docker container inspect <container id>`
4. log on whichever container and ping another one
   `docker container exec -it <container id> bash`
   `ping xx.xxx.xx.xx`



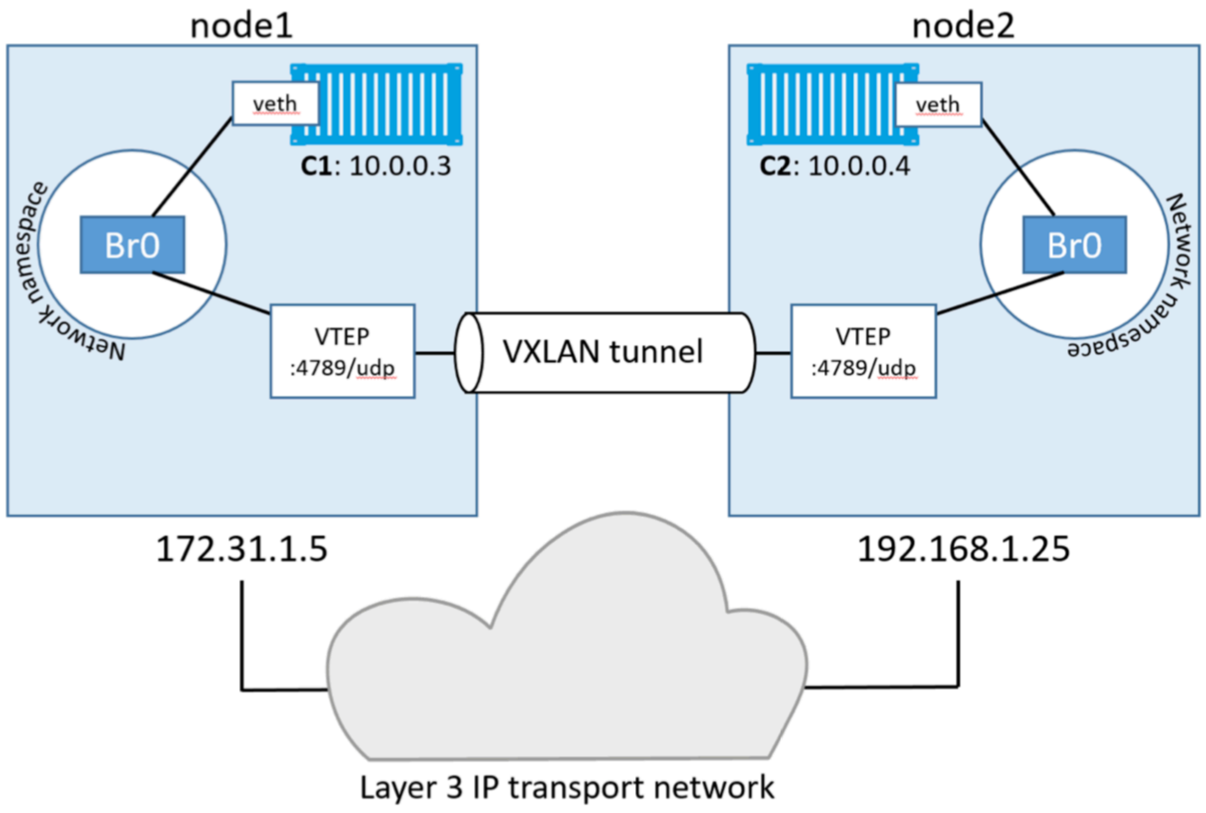
Docker overlay networking uses VXLAN tunnels to create virtual Layer 2 overlay networks. VXLANs let you create a virtual Layer 2 network on top of an existing Layer 3(IP) infrastructure.

Walkthrough:

1. When the VXLAN overlay network is created, docker creates a sandbox(a "container" for a network stack) on the host. A virtual switch (a.k.a. virtual bridge) called Br0 is created inside the sandbox.
2. VTEP and veth?
3. c1 sends the first 'ping xxx.xx.xx.xxx' package to c2.
4. The traffic goes to Br0 over veth. Br0 doesn't know the mac address of c2, so it floods the packet to all ports.
5. The VTEP responds with the mac address of c2 back to Br0, so Br0 update its mac address table.
6. Br0 forwards all 'ping' packages to VTEP, and VTEP will encapsulate the Ethernet frame by adding VXLAN header into it.
7. VTEP on node1 sends the packages to VTEP on the node2. It decapsulates the packages and sends them to

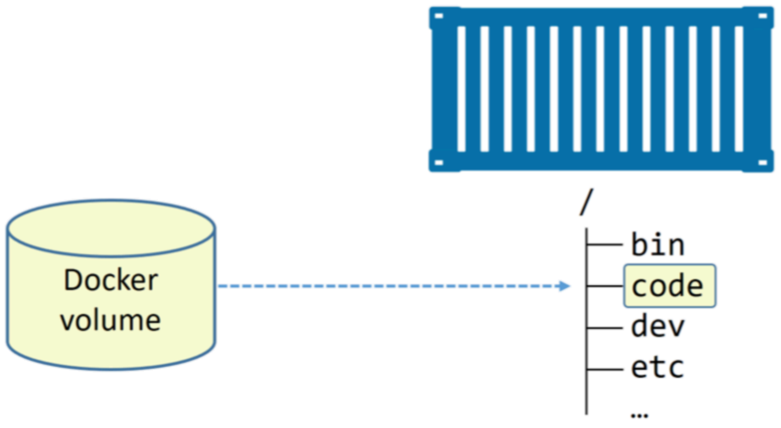
Br0 on the node2.

8. Br0 forwards packages to c2 to process.



## Volumes

Docker creates a volume, then creates a container, and docker mounts the volume into it. The volume gets mounted to a directory in the container's filesystem, and anything written to that directory is written to the volume. If you then delete the container, the volume and its data will still exist. It's not possible to specify the host directory portion in a Dockerfile. This is because host directories are, by nature, host-dependent, meaning they can change between hosts and potentially break builds.



Different drivers

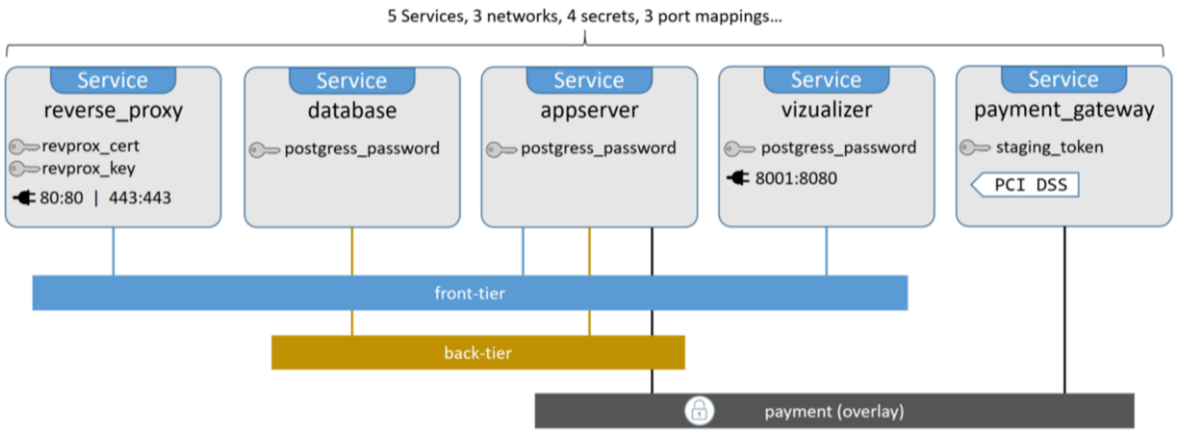
Mount a volume to a container
`docker container run -dit --name voltainer --mount source=myvol,target=/vol alpine`
Mount a volume to a service
`docker service create --name hellcat --mount source=myvol,target=/vol alpine sleep 1d`

- `docker volume create myvol`: docker creates new volumes with the built-in local driver.
- `docker volume prune/rm`: remove the unmounted volumes or remove volumes by id

## Docker stack

Docker stack simplified application management by providing; desired state, rolling updates, simple, scaling operations, health checks...

Good example: [https://github.com/dockersamples/atsea-sample-shop-app](https://github.com/dockersamples/atsea-sample-shop-app)



### Generate secret

Docker can centrally manage sensitive data and securely transmit it to only those containers that need access to it.

1. create a docker secret file which contains a secret string
   `printf "This is a secret" | docker secret create my_secret_data -`
2. create a service which has the access to the secret key
   `docker service create --name test --secret my_secret_data alpine:latest`
3. check the content of the secret
   `docker container exec $(docker ps --filter name=test -q) cat /run/secrets/my_secret_data`

### Network

```
networks:
  front-tier:
  back-tier:
  payment:
    driver: overlay
    driver_opts:
      encrypted: 'yes'
```

By default, front-tier and back-tier will be created as overlay networks by the overlay driver. The payment network requires a encrypted data plane.

**Secret**

```
secrets:
  postgres_password:
    external: true
  staging_token:
    external: true
  revprox_key:
    external: true
  revprox_cert:
    external: true
```

secret must already exist before the stack can be deployed.

**Service**

```
reverse_proxy:
  image: dockersamples/atseasampleshopapp_reverse_proxy
  ports:
    - "80:80"
    - "443:443"
  secrets:
    - source: revprox_cert
      target: revprox_cert
    - source: revprox_key
      target: revprox_key
  networks:
    - front-tier
```

Docker Stacks doesn't support builds. Secrets get mounted into service replicas as a regular file. The name of the file will be whatever you specify as the target value in the stack file, and the file will appear in the replica under /run/secrets on Linux.

**Database**

```
database:
  image: dockersamples/atsea_db
  environment:
    POSTGRES_USER: gordonuser
    POSTGRES_DB_PASSWORD_FILE: /run/secrets/postgres_password
    POSTGRES_DB: atsea
  networks:
    - back-tier
  secrets:
    - postgres_password
  deploy:
    placement:
      constraints:
        - 'node.role == worker'
```

The `environment` key lets you inject environment variables into services replica. database container will only be deployed on worker node.

### Appserver

```
appserver:
  image: dockersamples/atsea_app
  networks:
    - front-tier
    - back-tier
    - payment
  deploy:
    replicas: 2
    update_config:
      parallelism: 2
      failure_action: rollback
    placement:
      constraints:
        - 'node.role == worker'
    restart_policy:
      condition: on-failure
      delay: 5s
      max_attempts: 3
      window: 120s
  secrets:
    - postgres_password
```

`update_config` tells Docker how to act when rolling out updates to the service. Docker will update two replicas at a time and will perform a 'rollback' if it detects the update is failing. restart-policy tells Swarm how to restart replicas (containers) if and when they fail. Swarm will restart a replica if a replica stops with a non-zero exit code(on-failure). It will try to restart the failed replica 3 times, and wait up to 120 seconds to decide if the restart worked. It will wait 5 seconds between each of the three restart attempts.

### Visualizer

```
visualizer:
  image: dockersamples/visualizer:stable
  ports:
    - "8001:8080"
  stop_grace_period: 1m30s
  volumes:
    - "/var/run/docker.sock:/var/run/docker.sock"
  deploy:
    update_config:
      failure_action: rollback
    placement:
      constraints:
        - 'node.role == manager'
```

`stop_grace_period` property overrides this 10 second grace period in which docker stops the container and waits to perform any clean-up operations. `volume` mounts pre-created volume or host dictionary into a service replic. `/var/run/docker.sock` is the IPC socket that the Docker daemon exposes all of its API endpoints on.

### Payment_gateway

```
payment_gateway:
  image: dockersamples/atseasampleshopapp_payment_gateway
  secrets:
    - source: staging_token
      target: payment_token
  networks:
    - payment
  deploy:
    update_config:
      failure_action: rollback
    placement:
      constraints:
        - 'node.role == worker'
        - 'node.labels.pcidss == yes'
```

payment_gateway replic will only be deployed on the worker node with pcidss=yes label.

### deploy

- Create a new swarm with mgr1, wrk1, wrk2 and add a label to wrk2
  ....
  `docker node update --label-add pcidss=yes wrk2` add label into wrk2
- Create the secrets
  `openssl req -newkey rsa:4096 -nodes -sha256 -keyout domain.key -x509 -days 365 -out domain.crt`
  create docker secrets
  `docker secret create revprox_cert domain.crt`
  `docker secret create revprox_key domain.key`
  `docker secret create postgres_password domain.key`
  `echo staging | docker secret create staging_token -`
- deploy stack
  `docker stack deploy -c docker-stack.yml seastack`

`docker stack ps`
`docker stack rm`

## Log

check logs
`docker logs <container>`

```
docker compose logs
```