Advanced Docker Training

Anton Weiss
Otomato Software Delivery
http://otomato.link

Module 1

Docker in a Nutshell. Installation options.

Docker Overview

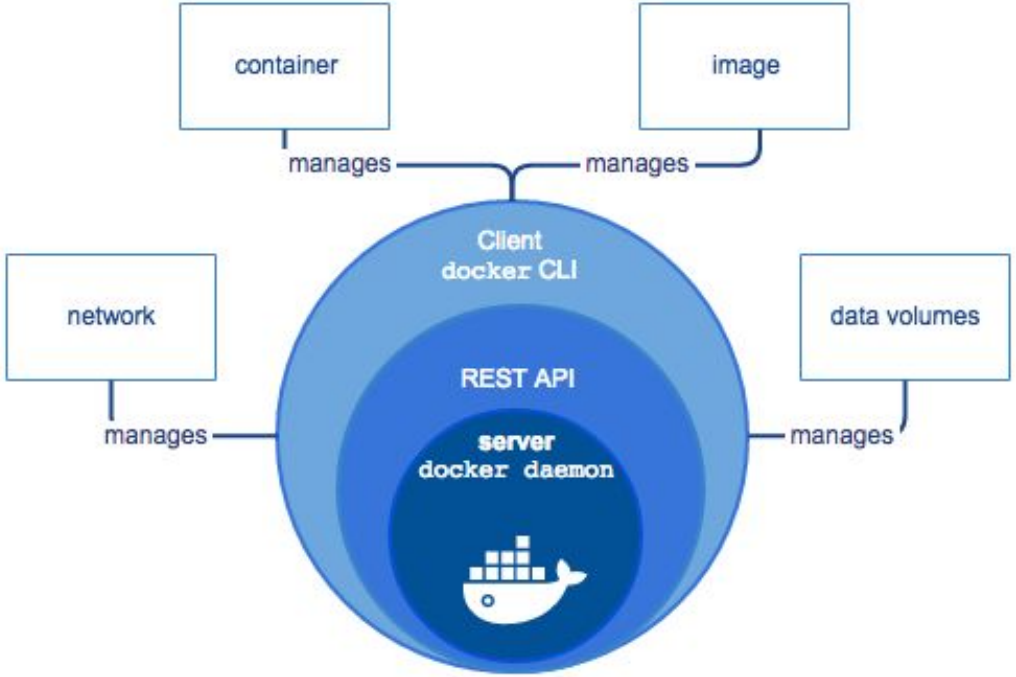
Docker provides tooling and a platform to manage the lifecycle of your containers:

- Encapsulate your applications (and supporting components) into Docker containers
- Distribute and ship those containers to your teams for further development and testing
- Deploy those applications to your production environment, whether it is in a local data center or the Cloud

What Is Docker?

- Configuration Management Tool?
- OS Level Virtualization?
- Application Packaging Format?
- Deployment Method?
- Deployment Platform?

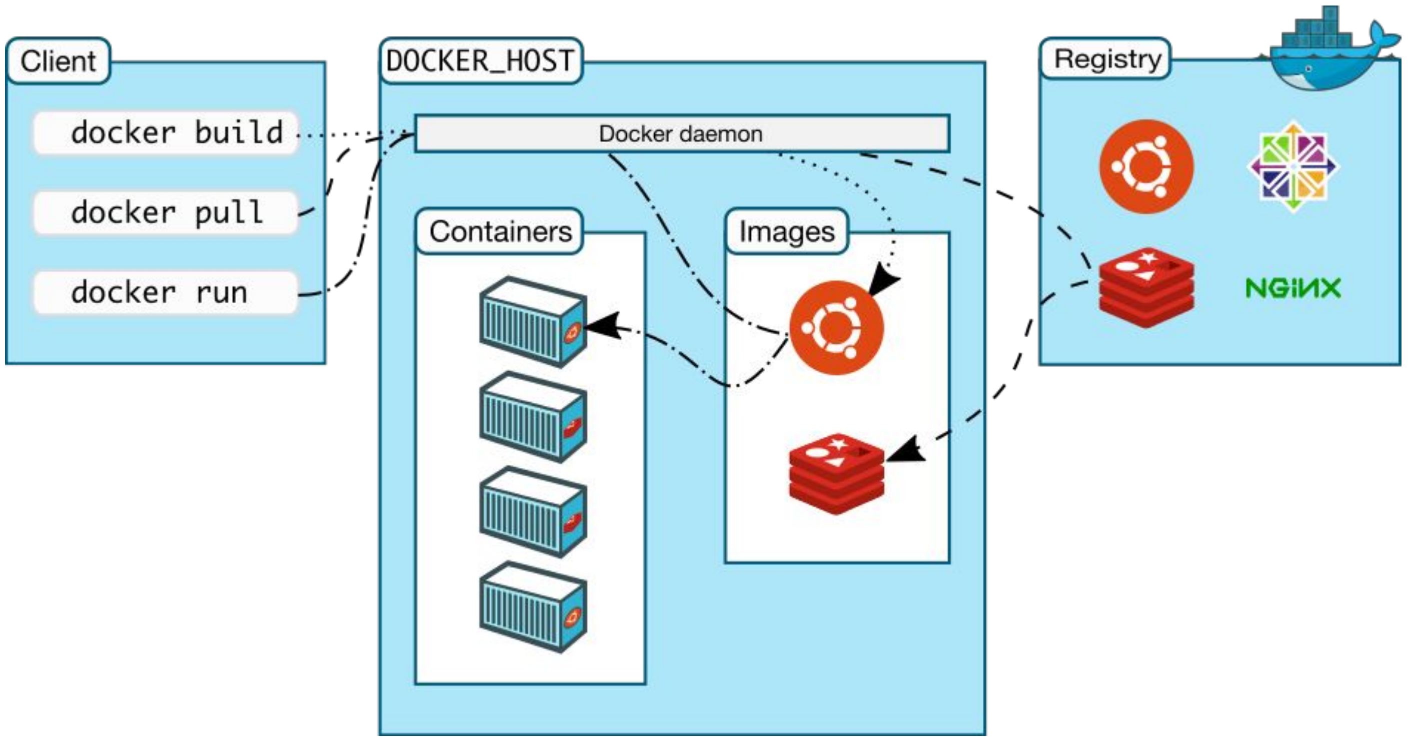
Docker Host Architecture



Underlying Technology

- Namespaces
- Cgroups
- Capabilities
- COW file systems

Docker Platform Architecture



Install Docker

Windows 7: Docker Toolbox

Windows 10: Docker for Windows

Mac OS X: Docker for Mac

RedHat Linux:

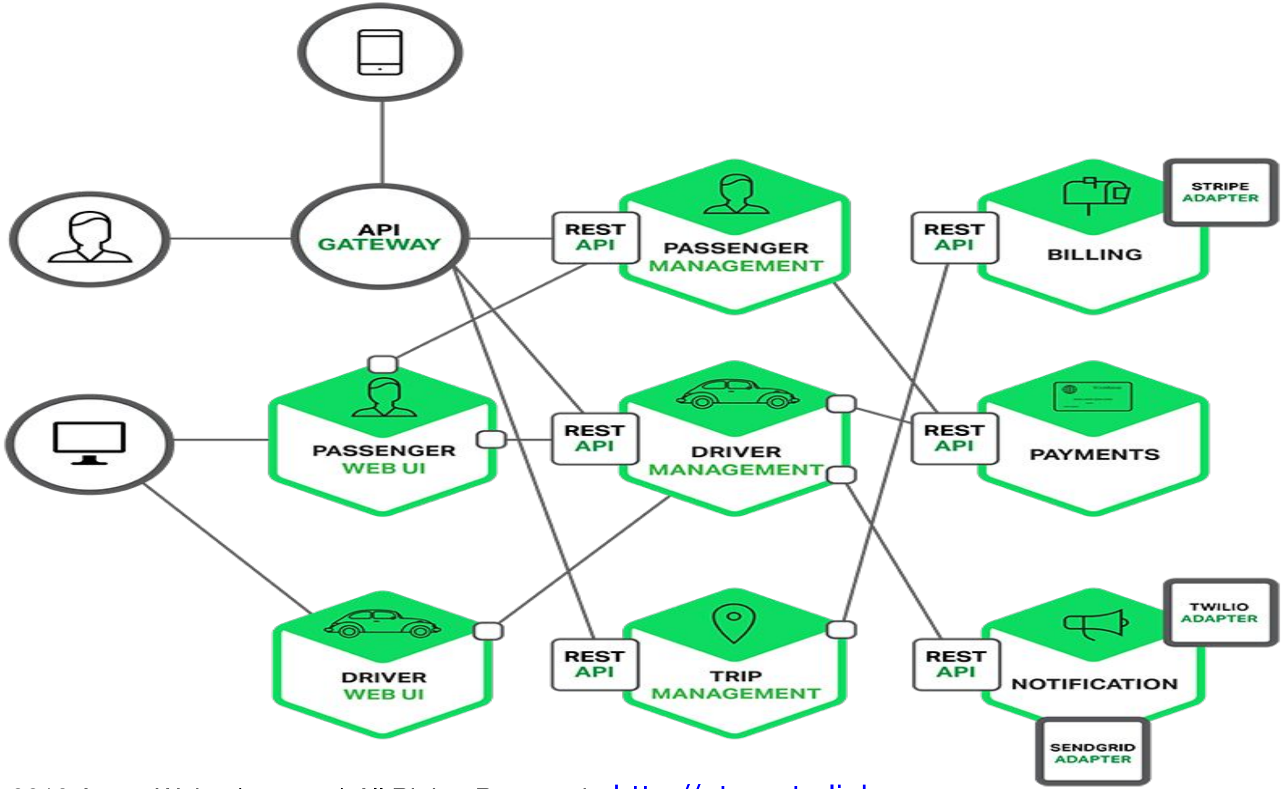
```
sudo yum -y remove docker
sudo yum install -y yum-utils
sudo yum-config-manager \
      --add-repo \

  https://docs.docker.com/engine/installation/linux/repo_files/centos/docker.repo
sudo yum makecache fast
sudo yum -y install docker-engine
```

Module 2

Microservices

Microservices and Containers

Microservices?



Microservices - the Advantages

- Smaller Application Footprint
- Comprehensibility
- Shorter Development Time
- Continuous Delivery
- Scalability
- Polyglossia
- Efficient Organisational Structure

The 12-Factor App

- I. **Codebase**
  - One codebase tracked in revision control, many deploys
- II. **Dependencies**
  - Explicitly declare and isolate dependencies
- III. **Config**
  - Store config in the environment
- IV. **Backing services**
  - Treat backing services as attached resources
- V. **Build, release, run**
  - Strictly separate build and run stages
- VI. **Processes**
  - Execute the app as one or more stateless processes

The 12-Factor App

- **VII. Port binding**
  - Export services via port binding
- **VIII. Concurrency**
  - Scale out via the process model
- **IX. Disposability**
  - Maximize robustness with fast startup and graceful shutdown
- **X. Dev/prod parity**
  - Keep development, staging, and production as similar as possible
- **XI. Logs**
  - Treat logs as event streams
- **XII. Admin processes**
  - Run admin/management tasks as one-off processes

Docker and Microservices

- Stateless/Ephemeral Applications.
- Horizontal scaling.
- Self-Contained Immutable Infrastructure.
- Continuous Delivery.
- Fast startup/graceful shutdown.
- Build image once, run anywhere. (finally)

Module 3

Docker Compose

Docker-compose

- A Tool for Defining and Running Multi-container Systems
- YAML-based configuration
- Basic units of deployment:
  - Services
  - Volumes
  - Networks

Docker-compose

- git clone https://github.com/antweiss/oto-products
- cd oto-products
- cat docker-compose.yml

Docker-compose

```
version: '3'
services:
  oto-products:
    build: .
    ports:
      - 5000
  mongo:
    image: mongo
    ports:
      - 27017
```

Docker-compose Commands

- `docker-compose up (--no-start)`
- `docker-compose down`
- `docker-compose stop`
- `docker-compose start`
- `docker-compose rm`
- `docker-compose run`
- `docker-compose logs`

`Reference: https://docs.docker.com/compose/reference/`

Docker-compose for httpd

```yaml
services:
 web:
   image: httpd
   ports:
     - 80
```

```
> docker-compose up -d
```

Exercise

- Create a Dockerfile:
  - Image based on httpd:latest
  - Create a file index.html with content:
    - <html><body><h1> Hello, Docker! </h1></body></html>
  - Copy index.html to /usr/local/apache2/htdocs/ on the image
- Create a docker-compose.yml that:
  - Builds your image
  - Runs it as service named ‘myweb’ on port 8080 of your host.

Module 4

Connecting Containers

Connecting Containers

- By default a container is isolated from other containers and the external network
- Docker provides a bridge - the docker0 interface

```
> /sbin/ifconfig docker0
```

- The bridge is the connection mechanism for all container connections on a single Docker host

Connecting Containers

- Let’s build our testing image!

```
> mkdir netbox
> cd netbox
> vi Dockerfile
```

Dockerfile content:

```
FROM alpine
MAINTAINER Your Name <contact@otomato.link>
RUN apk -U add netcat-openbsd
CMD ["nc", "-l", "1234"]
```

run:

```
>docker build -t mynetbox .
```

this will build an alpine-based image with netcat

```
>docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
mynetbox                      latest              c3dea9f7bbaa        2 minutes ago       8.846 MB
```

Connecting Containers

- Let’s play client and server:
- Start the server:

```
> docker run --name server -it mynetbox /bin/sh
/# nc -l 1234
```

- In another terminal:

```
> docker inspect --format='{{range
.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' server
# this will return the container ip address
> docker run -it mynetbox /bin/sh
/# nc  <server-ip-address> 1234
```

Now we make containers talk to each other…

Linking Containers

- Exit your running containers.
- Working with IPs is all fine, but they change and are hard to remember. What if we want something less ephemeral and more meaningful?
- Solution: use container links
- Start the server:

```
> docker run -it --name server mynetbox /bin/sh
/# nc -l 1234
```

- In another terminal:

```
> docker run -it --name client --link server mynetbox /bin/sh
/# nc server 1234
```

Exercise

- Turn your nc into a mini-web server:
  - Build an image based on alpine with curl installed (like on slide25 with ‘apk add curl’) - tag it **curlbox**
  - run a ‘server’ container (based on **mynetbox** image) with hostname ‘your_name_server’ (use —hostname). In the container run

```
>echo -e "HTTP/1.1 200 OK\n\n Welcome to $(hostname)" | nc -l 80
```

  - run a ‘client’ container based on **curlbox** with command ‘curl http://server’ so that it returns the expected output: **Welcome to *your_name*_server**

Module 5

Data Volumes

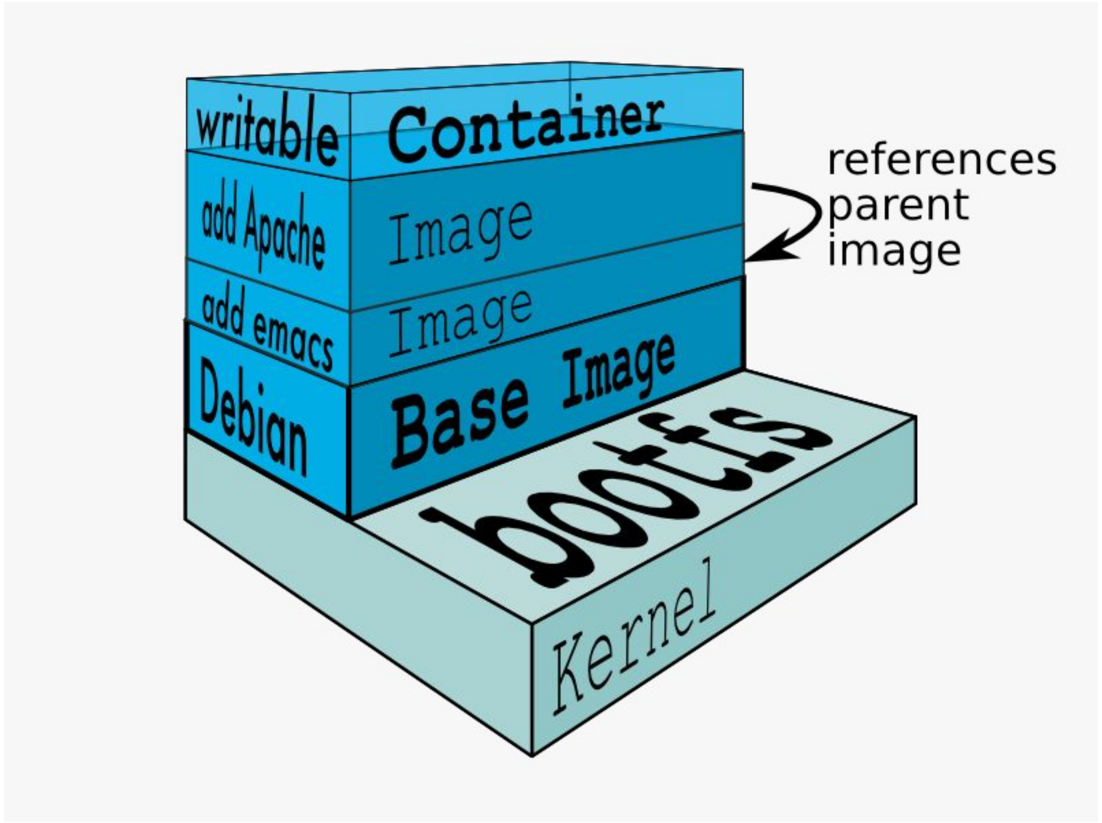
Data Volumes

A data volume is a specially-designated directory within one or more containers that bypasses the Union File System. Data volumes provide several useful features for persistent or shared data:

- Volumes are initialized when a container is created. If the container's base image contains data at the specified mount point, that existing data is copied into the new volume upon volume initialization. (Note that this does not apply when mounting a host directory.)
- Data volumes can be shared and reused among containers.
- Changes to a data volume are made directly.
- Changes to a data volume will not be included when you update an image.
- Data volumes persist even if the container itself is deleted.

Data volumes are designed to persist data, independent of the container's lifecycle. Docker therefore never automatically deletes volumes when you remove a container, nor will it "garbage collect" volumes that are no longer referenced by a container.

Union File System



Data Volumes

- Adding a volume:

```
> docker run -v /data --name greeter mynetbox /bin/sh -c "echo hello docker > /data/hello”
```

This has created a data volume on host filesystem and written a file to it.

- Find the volume on host:

```
> docker inspect -f "{{range .Mounts}}{{.Source}}{{end}}" greeter
#/var/lib/docker/volumes/37ce0677fcdf4e4a15a87f969192e62bfa457f764e1e8848ef3394da09df5348/_da
ta
> cat
/var/lib/docker/volumes/37ce0677fcdf4e4a15a87f969192e62bfa457f764e1e8848ef3394da09df5348/_dat
a/hello
hello docker
```

- Now remove the container: *docker rm greeter*
- Does the volume persist?

Named Volumes vs Anonymous

Add a named volume:

```
 >docker run -v myvolume:/data mynetbox touch /data/aa
 > docker volume ls
DRIVER              VOLUME NAME
local               myvolume
>docker volume inspect myvolume
[
    {
        "Name": "myvolume",
        "Driver": "local",
        "Mountpoint": "/var/lib/docker/volumes/myvolume/_data",
        "Labels": null,
        "Scope": "local"
    }
]
> docker run -v myvolume:/data mynetbox touch /data/boo
> ls /var/lib/docker/volumes/myvolume/_data
aa boo
```

Note: named volumes are not removed when running with --rm

Managing Data Volumes

```
 > docker volume ls
DRIVER              VOLUME NAME
local               37ce0677fcdf4e4a15a87f969192e62bfa457f764e1e8848ef3394da09df5348
> docker volume inspect  <volume_name>
> docker volume rm <volume_name>
#remove all volumes
> docker volume ls -q | xargs docker volume rm
```

Mounting Host Directories

```
mkdir ${HOME}/data
echo 'take me with you' > ${HOME}/data/take.txt
docker run -v ${HOME}/data:/data mynetbox cat /data/take.txt
```

You can mount host directories in read-only mode with :

`-v <source>:<destination>:ro`

Exercise

Create directory ‘messages’ under your home directory on the host.

Run a container to write a message ‘hello from docker’ into ${HOME}/messages/message1

Data Volume Containers

If you have some persistent data that you want to share between containers, or want to use from non-persistent containers, it's best to create a named Data Volume Container, and then to mount the data from it.

```
> docker create -v /webcontent --name static mynetbox
```

Now we can run multiple containers to serve html from data volume and write output into it:

```
> docker run --volumes-from static --name web1 -p 80:80 mynetbox /bin/sh
```

Data Volume Containers

First let’s create an html file:

```
> tee /webcontent/index.html <<-’EOF’
<html>
<head>
<title>Static content</title>
</head>
<body>
Served from a data volume container
</body>
</html>
EOF
```

Data Volume Containers

Now - let’s serve it and record requests:

```
while true; do echo -e "HTTP/1.1 200 OK\n\n $(cat /webcontent/index.html) on
$(date)" | nc -l 80 >> /webcontent/output.log ;done
```

Go to http://<your_docker_vm> in a browser

And we can serve the same data from another container:

```
docker run --volumes-from static --name web2 -p 81:80 mynetbox /bin/sh
```

```
/ # while true; do echo -e "HTTP/1.1 200 OK\n\n $(cat /webcontent/index.html) on
$(date)" | nc -l 80 >> /webcontent/output.log ;done
```

Go to http://<your_docker_vm>:81 in your browser

Connecting Containers with docker-compose

- Install latest docker-compose:

```
> sudo curl -L
https://github.com/docker/compose/releases/download/1.18.0/docker-compose-`uname
-s`-`uname -m` -o /usr/local/bin/docker-compose
```

- Check:

```
> docker-compose version
docker-compose version 1.18.0, ...
```

Module 6

Connecting Containers with docker-compose

Connecting Containers with docker-compose

Create docker-compose.yml:

```
version: '2'
services:
  node1:
    image: mynetbox
    ports:
      - 1234
  node2:
    image: mynetbox
    ports:
      - 1234
```

Run:

```
> docker compose up -d
```

Connecting Containers with docker-compose

```
> docker-compose exec  node1 sh
/ # ping node2
```

When using docker-compose - there’s no need to explicitly link containers. All containers on the same user-defined network will be automatically able to access each other by service name defined in docker-compose.yml.
If no network is explicitly specified - docker-compose will place containers on a ‘compose_default’ network:

```
> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
dbd11e643100        bridge              bridge              local
3c1f81ae7807        compose_default     bridge              local
9afd1f556c68        host                host                local
86b07440c9a5        none                null                local
```

Module 7

Docker Network Function

Docker Network Function

- The default bridge network: docker0
- In addition to this network, you can create your own bridge or overlay networks.
- Bridge = single host. Overlay = cluster.

Create a network

```
# this creates a bridge network
> docker network create mynet
74638505cfd8e1c6e27723d7255b84acaa84c3858ce0657932ffa236070c79d0


> docker network inspect mynet
...
```

Connect Containers to a Network

```
# create an isolated network
> docker network create -d bridge --subnet 172.25.0.0/16 isolated_nw
#run 2 containers
> docker run -itd --name c1 mynetbox
> docker run -itd --name c2 mynetbox
#connect c2 to the network
> docker network connect isolated_nw c2
> docker network inspect isolated_nw
```

Connect Containers to a Network

```
# now start a new container connected to the network. Specify an ip address:
> docker run --network isolated_nw --ip 172.25.1.3 --name c3 mynetbox
# this container won't be connected to the default bridge at all
> docker inspect c3
…
"Networks": {
            "isolated_nw": {
               "IPAMConfig": {
                  "IPv4Address": "172.25.1.3"
               }...
```

DNS Resolution

The Docker embedded DNS server enables name resolution for containers connected to a given network. This means that any connected container can ping another container on the same network by its container name. From within c2, you can ping c3 by name:

```
/ # ping c3
PING c3 (172.25.1.3): 56 data bytes
64 bytes from 172.25.1.3: seq=0 ttl=64 time=0.074 ms
64 bytes from 172.25.1.3: seq=1 ttl=64 time=0.077 ms
64 bytes from 172.25.1.3: seq=2 ttl=64 time=0.076 ms
64 bytes from 172.25.1.3: seq=3 ttl=64 time=0.089 ms
```

But you can’t ping c1:

```
/ # ping c1
ping: bad address 'c1
```

Because **DNS resolution functionality is not available for the default bridge network.**

Network Aliases

```
> docker network connect isolated_nw c1 --link c3:foo

> docker exec -it c1 sh
/# ping -w1 c3
PING c3 (172.25.1.3): 56 data bytes
64 bytes from 172.25.1.3: seq=0 ttl=64 time=0.074 ms

 /# ping -w1 foo
PING foo (172.25.1.3): 56 data bytes
64 bytes from 172.25.1.3: seq=0 ttl=64 time=0.074 ms

# Start a new container with network alias
> docker run -itd  --name c4 --network isolated_nw  --network-alias bar mynetbox sh
```

Network Aliases

Multiple containers can share the same network-scoped alias within the same network. When this happens one of those containers will resolve to the alias. If that container is unavailable, another container with the alias will be resolved. This provides a sort of high availability within the cluster.

Exercise

- Create a network called **mybridge**
- Start container cont1 on **mybridge**
- Start container cont2 on **mybridge** with network alias **dop** (for doppelganger :))
- Start container cont3 on default bridge.
- Attach cont3 to **mybridge** with network alias **dop**
- Run ping from cont1 to host name **‘dop’**
- Which ip is getting pinged?
- Try nslookup on **‘dop’**
- Stop the container whose IP is getting resolved and ping again.

Port Mapping

Let’s build an apache container:

Create a Dockerfile:

```
FROM otomato/alpine-netcat
MAINTAINER Ant Weiss
RUN apk add -U apache2 && mkdir /run/apache2
EXPOSE 80 443
CMD ["/usr/sbin/httpd", "-D" ,"FOREGROUND"]
```

```
> docker build . -t myapache
```

Port Mapping

Now let’s run it with ports mapped to the host:

```
 > docker run -d --hostname web.otomato.link -P myapache
 > docker ps
...PORTS
...0.0.0.0:32786->80/tcp, 0.0.0.0:32785->443/tcp

 > curl http://0.0.0.0:32786

<html><body><h1>It works!</h1></body></html>
```

Port Mapping

Bind to a specific port:

```
-p 8000:80
```

Bind to a range:

```
-p 8000-9000:80
```

Bind to a specific interface:

```
-p 127.0.0.1::80
```

Check ports:

```
>docker port mywebserver
```

Advanced Network Functions

An *overlay* network is a multi-host network used to connect services running on different cluster nodes.

In docker swarm mode creating an overlay network is as easy as :

```
>docker network create --driver overlay --subnet 10.0.9.0/24 my-multi-host-network
```

Overlay networks for a swarm are not available to containers started with `docker run` that don't run as part of a swarm mode service.

Advanced Network Functions

Overlay network with an external key-value store:

If you are not using Docker Engine in swarm mode, the overlay network requires a valid key-value store service. Supported key-value stores include Consul, Etcd, and ZooKeeper. Before creating a network on current version of the Engine, you must install and configure your chosen key-value store service. The Docker hosts that you intend to network and the service must be able to communicate.

Using external DNS with user-defined networks

Docker run with options:

`--dns=[IP_ADDRESS...]` : The IP addresses passed via the `--dns` option is used by the embedded DNS server to forward the DNS query if embedded DNS server is unable to resolve a name resolution request from the containers. These `--dns` IP addresses are managed by the embedded DNS server and will not be updated in the container's `/etc/resolv.conf` file.

`--dns-search=DOMAIN...` : Sets the domain names that are searched when a bare unqualified hostname is used inside of the container. These `--dns-search` options are managed by the embedded DNS server and will not be updated in the container's `/etc/resolv.conf` file. When a container process attempts to access hostand the search domain `example.com` is set, for instance, the DNS logic will not only look up `host` but also `host.example.com`.

`--dns-opt=OPTION...` : Sets the options used by DNS resolvers. These options are managed by the embedded DNS server and will not be updated in the container's `/etc/resolv.conf` file.

Docker network with docker-compose

```
version: '2'

services:
  proxy:
    build: ./proxy
    networks:
      - front
  app:
    build: ./app
    networks:
      - front
      - back
  db:
    image: postgres
    networks:
      - back

#Continued on next slide
```

Docker network with docker-compose

```
networks:
  front:
    # Use a custom driver
    driver: custom-driver-1
  back:
    # Use a custom driver which takes special options
    driver: custom-driver-2
    driver_opts:
      foo: "1"
      bar: "2"
```

Logging and Performance Monitoring

Module 8

Logging Considerations

- Log from the application
- Store logs on a dedicated data volume
- Use docker logging functionality for recording stdout/stderr
- Log through a dedicated logging container (e.g: logspout, fluentd

Logging Containers

Simple logging - with default driver called ‘json-file’:

```
> docker run --name err mynetbox sh -c '>&2 echo error; >&1 echo log'
error
log
> docker logs err
error
log
```

But where is json?

```
>  docker inspect err -f '{{ .LogPath }}'
/var/lib/docker/containers/91ee976...c1-json.log

> cat /var/lib/docker/containers/91ee976...c1-json.log
{"log":"error\n","stream":"stderr","time":"2017-01-26T08:24:36.018189054Z"}
{"log":"log\n","stream":"stdout","time":"2017-01-26T08:24:36.018398255Z"}
```

Log Rotation Options

- --log-opt max-size=[0–9+][k|m|g]
- --log-opt max-file=[0–9+]

Exercise

- Build a docker image:
  - based on ‘alpine’
  - Add the following shell script to /opt/logging.sh

```
#! /bin/sh
while (true)
do
>&1 echo out
>&2 echo error
     date
     sleep 10
done
```

  - Make /opt/logging.sh the CMD of the image
- Run the image in background with maximum logfile size = 1k and maximum 2 log files
- Inspect container log storage to verify log rotation

Additional Drivers

none

No logs will be available for the container and `docker logs` will not return any output.

json-file

The logs are formatted as JSON. The default logging driver for Docker.

syslog

Writes logging messages to the `syslog` facility. The `syslog` daemon must be running on the host machine.

journald

Writes log messages to `journald`. The `journald` daemon must be running on the host machine.

gelf

Writes log messages to a Graylog Extended Log Format (GELF) endpoint such as Graylog or Logstash.

Additional Drivers

fluentd

Writes log messages to fluentd (forward input).

awslogs

Writes log messages to Amazon CloudWatch Logs.

splunk

Writes log messages to splunk using the HTTP Event Collector.

etwlogs

Writes log messages as Event Tracing for Windows (ETW) events. Only available on Windows platforms.

gcplogs

Writes log messages to Google Cloud Platform (GCP) Logging.

nats

NATS logging driver for Docker. Publishes log entries to a NATS server.

Additional Drivers

Check daemon’s default logging driver:

```
>docker info | grep Logging
Logging Driver: json-file
```

Check logging driver for container:

```
>docker inspect <container> -f '{{.HostConfig.LogConfig.Type}}'
```

Logging to syslog

```
> docker run --log-driver=syslog --name syslogc -d alpine sh -c 'while (true) do >&2
echo error; date; >&1 echo log;date;sleep 10; done'
> tail -f /var/log/messages
```

Centralized Cluster Logging

Logspout is a log router for Docker containers that runs inside Docker. It attaches to all containers on a host, then routes their logs wherever you want. It also has an extensible module system.

E.g in a docker swarm cluster:

```
docker service create --name logspout \
    --mode global \
    --mount
"type=bind,source=/var/run/docker.sock,target=/var/run/docker.sock" \
    -e SYSLOG_FORMAT=rfc3164 \
    gliderlabs/logspout syslog://logstash:51415
```

Performance Monitoring

- Instrument your applications
- Monitor container resource usage
- Alert on service performance, not container performance. (Use orchestration infra metrics)
- Cloud-aware monitoring (transient entities)
- Monitor APIs

Container Resource Constrains

-m, --memory=""
Memory limit (format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of b, k, m, or g. Minimum is 4M.
-c, --cpu-shares=0
CPU shares (relative weight) (By default new containers have `1024` shares)
--device-read-iops=""
Limit read rate (IO per second) from a device (format: `<device-path>:<number>`). Number is a positive integer.
--device-write-iops=""
Limit write rate (IO per second) to a device (format: `<device-path>:<number>`). Number is a positive integer.
--oom-kill-disable=false
Whether to disable OOM Killer for the container or not.

Full list of resource constraint flags

Performance Monitoring

docker stats:

A built-in utility for live monitoring of container resource consumption

Run a cpu-intensive container:

```
> docker run -d --name cpueater mynetbox dd if=/dev/zero of=/dev/null
> docker stats #displays stats for all containers
> docker stats cpueater #stats for a specific container
```

Performance Monitoring

Stats with Docker remote API:
By default docker daemon listens on /var/run/docker.sock
To call API on the socket use:

```
curl --unix-socket /var/run/docker.sock
http:/v.1.24/containers/load/stats
```

To make API available remotely - edit /etc/default/docker

```
DOCKER_OPTS='-H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock'
> systemctl restart docker
```

Then run:

```
curl http://localhost:4243/containers/load/stats
```

CAdvisor

A daemon that collects, aggregates, processes, and exports information about running containers in a web dashboard.

Run cAdvisor:

```
docker run -d --name=cadvisor -p 8080:8080 \
--volume=/var/run:/var/run:rw \
--volume=/sys:/sys:ro \
--volume=/var/lib/docker/:/var/lib/docker:ro \
google/cadvisor:latest
```

Access in browser : http://<your docker host >:8080

cAdvisor dashboard shows data for the last 60 seconds only, but it CAdvisor also exposes a metrics API that can be scraped by Prometheus at: http://<your docker host >:8080/metrics

netdata

netdata is *scalable, distributed real-time performance and health monitoring*
Docker image: https://hub.docker.com/r/titpetric/netdata/
netdata collected metrics can be pushed to central time-series databases (like **graphite** or **opentsdb**) or scraped by **Prometheus** for archiving.

netdata provides powerful alarms and notifications.

```
> docker run -d --cap-add SYS_PTRACE -v /proc:/host/proc:ro  -v
/sys:/host/sys:ro -v /var/run/docker.sock:/var/run/docker.sock -p
19999:19999 titpetric/netdata
```

```
http://<your-docker-host>:19999
```

netdata

Metrics:
http://<your-docker-host>:19999/api/v1/allmetrics?format=shell
http://<your-docker-host>:19999/api/v1/allmetrics?format=prometheus

Preconfigured alarms are available - see on dashboard.

Getting emails on alarms:
Netdata supports forwarding alarms to an email address. You can set up SSMTP by setting the following ENV variables:

- SSMTP_TO - This is the address alarms will be delivered to.
- SSMTP_SERVER - This is your SMTP server. Defaults to smtp.gmail.com.
- SSMTP_PORT - This is the SMTP server port. Defaults to 587.
- SSMTP_USER - This is your username for the SMTP server.
- SSMTP_PASS - This is your password for the SMTP server. Use an app password if using Gmail.
- SSMTP_TLS - Use TLS for the connection. Defaults to YES.
- SSMTP_HOSTNAME - The hostname mail will come from. Defaults to localhost.

Cluster monitoring

Need to aggregate data

Evaluate whole cluster health

For kubernetes: use **heapster**
https://kubernetes.io/docs/user-guide/monitoring/

Module 9

Docker Security Features

Docker Security

There are four major areas to consider when reviewing Docker security:

- the intrinsic security of the kernel and its support for namespaces and cgroups;
- the attack surface of the Docker daemon itself;
- loopholes in the container configuration profile, either by default, or when customized by users.
- the “hardening” security features of the kernel and how they interact with containers.

Intrinsic Security

- **Namespaces** provide the first and most straightforward form of isolation - process level
- Each container also gets its own **network stack** - a container doesn’t get privileged access to the sockets or interfaces of another container
- **CGroups** implement resource accounting and limiting. (run systemd-cgls to see existing control groups) - they can prevent denial of service attacks

Docker Daemon Attack Surface

- Daemon runs under ‘root’
- Only trusted users should be allowed to control your Docker daemon
- Daemon listens on unix socket to limit access
- REST API over http has to be secured with HTTPS and certs
- Use signed docker images (see Content Trust)

Linux Kernel Capabilities

- By default, Docker starts containers with a restricted set of capabilities.
- Capabilities turn the binary “root/non-root” dichotomy into a fine-grained access control system.
- “root” within a container has much less privileges than the real “root”
- the best practice for users would be to remove all capabilities except those explicitly required for their processes.

Linux Kernel Capabilities

- List of default container capabilities :

```
"CAP_CHOWN",
"CAP_DAC_OVERRIDE",
"CAP_FSETID",
"CAP_FOWNER",
"CAP_MKNOD",
"CAP_NET_RAW",
"CAP_SETGID",
"CAP_SETUID",
"CAP_SETFCAP",
"CAP_SETPCAP",
"CAP_NET_BIND_SERVICE",
"CAP_SYS_CHROOT",
"CAP_KILL",
"CAP_AUDIT_WRITE"
```

Linux Kernel Capabilities

```
 > docker run --cap-add <CAP> # add capability
 > docker run --cap-drop <CAP> #remove capability
```

Example:

```
> useradd otomato
> touch /tmp/myfile
> id otomato
uid=1001(otomato) gid=1001(otomato) groups=1001(otomato)
> docker run -it --cap-drop CHOWN -v /tmp:/tmp:rw  alpine sh
/ # chown 1001:1001 /tmp/myfile
chown: /tmp/myfile : Operation not permitted
```

Exercise

Run mynetbox container without granting it `CAP_NET_BIND_SERVICE` capability.

Run ‘nc -l 80’ inside the container

Does that work?

Docker Content Trust

- Docker content trust enables enforcing client-side signing and verification of image **tags.**
- Image publishers sign images.
- Image consumers verify that the images they pull are signed.

Docker Content Trust

- Docker image record: `[REGISTRY_HOST[:REGISTRY_PORT]/]REPOSITORY[:TAG]`
- Content trust is associated with the *TAG* portion of the record.
- Some tags in a repository can be signed while others are not.
- Allows publishers iterating over a tag before releasing and signing.

Docker Content Trust

The commands that operate with content trust are:

- push
- build
- create
- pull
- run

Docker Content Trust

Trust for an image tag is managed through the use of signing keys. A key set is created when an operation using content trust is first invoked. A key set consists of the following classes of keys:

- an offline key that is the root of content trust for an image tag
- repository or tagging keys that sign tags
- server-managed keys such as the timestamp key, which provides freshness security guarantees for your repository

Docker Content Trust

Enable content trust:

```
> export DOCKER_CONTENT_TRUST=1
```

(use the *--disable-content-trust* flag to run individual operations on tagged images without content trust on an as-needed basis.)

```
 >docker pull otomato/alpine-netcat:0.1
Error: remote trust data does not exist for docker.io/otomato/alpine-netcat:
notary.docker.io does not have trust data for docker.io/otomato/alpine-netcat
 > docker pull --disable-content-trust otomato/alpine-netcat:0.1
0.1: Pulling from otomato/alpine-netcat
3690ec4760f9: Pull complete
4192bf5631f7: Pull complete
Digest: sha256:016eb23d3a39c26aa5b1e31cf7615adbe56050712d08bd6464e569bf1607cd45
Status: Downloaded newer image for otomato/alpine-netcat:0.1
```

Docker Content Trust

Publish Signed Images:

With DOCKER_CONTENT_TRST enabled

```
> docker tag mynetbox:latest <dockerhub_user>/mynetbox:trusted
> docker push <dockerhub_user>/mynetbox:trusted
...
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with id a1d96fb:...
```

Docker Content Trust

- Inspect generated keys in ~/.docker/trust
- If a publisher loses keys it means losing the ability to sign trusted content for your repositories. If you lose a key, contact Docker Support (support@docker.com) to reset the repository state.
- It is very important that you backup your keys to a safe, secure location. **Loss of the repository key is recoverable; loss of the root key is not.**
- The Docker client stores the keys in the ~/.docker/trust/private directory. Before backing them up, you should tar them into an archive:

```
> umask 077; tar -zcvf private_keys_backup.tar.gz ~/.docker/trust/private; umask 022
```

Module 10

Container Scheduling and Orchestration

Container Scheduling & Orchestration

Motivation:

- manage multiple containers on multiple container hosts
- treat an entire host cluster as a single deployment target
- schedule multi-container workloads considering dynamic resource and allocation rules and constraints

Most Popular Platforms

- Kubernetes
- Docker Swarm
- Marathon (Mesos)
- Nomad (Hashicorp)

Docker Swarm

- Integrated in docker core since v1.12
- Overlay networks
- Routing mesh
- Built-in security
- Service discovery and load-balancing
- Standard Docker API
- Supports docker-compose v3 syntax for deployments(since version 1.13)
- Pluggable scheduler interface

Docker Compose V3 Example

```yaml
version: "3"

services:
  voting-app:
    image: docker/voting-app-vote:latest
    ports:
      - 5000:80
    networks:
      - voteapp
    depends_on:
      - redis
    deploy:
      mode: replicated
      replicas: 2
      labels: [APP=VOTING]
      placement:
        constraints: [node.role == worker]
```

Kubernetes

- Open-sourced by Google
- Rich feature set
- Groups services in ‘pods’
- Secrets Management
- Service Discovery and Load Balancing
- Auto-scale cluster (if running on GCP)
- Dashboard UI
- Known for complexity

Marathon

- Part of Mesos, DCOS
- HA
- Multiple container runtimes
- Stateful apps support (Mesos-provisioned storage)
- Dashboard UI

Nomad

- Made by Hashicorp
- Service discovery with Consul
- HA
- Schedule batch jobs as well as containers
- Simple installation

Module 8

CI/CD with Docker containers

Docker-based Pipelines

- Build in containers for deterministic environments
- Examples:
  - GitLab Pipelines
  - Codefresh
  - Concourse CI

Jenkins integrations

[Jenkins Docker plugin]

[Yet-Another-Docker plugin]

[Docker build step plugin]

[Kubernetes Plugin]

[Nomad Plugin]

Jenkins Integrations

Docker support in Jenkins Pipeline:

```
docker.image('ruby:2.3.1').inside {

    stage("Install Bundler") {
      sh "gem install bundler --no-rdoc --no-ri"
    }

    stage("Use Bundler to install dependencies") {
      sh "bundle install"
    }
}
```

ONBUILD

Dockerfile **onbuild** instruction allows to create triggers for instructions to be executed when an image is used as a base image in **FROM.**
Useful for creating standard build scenarios:

```
FROM maven:3-jdk-7

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

ONBUILD ADD . /usr/src/app

ONBUILD RUN mvn install
```

Exercise

Create a base image from alpine so that when another image is built based on it, it echoes your name and then runs ‘apk update’

Multi-stage Dockerfiles

From 12-Factor App Principles:

- V. **Build, release, run**
  - Strictly separate build and run stages

But how do you dockerize compiled language binaries?

What are the problems of the following Dockerfile?

```
FROM maven:3.3.9

COPY . /usr/src/myapp/
WORKDIR /usr/src/myapp
 RUN mvn package
CMD ["java", "-jar",  "/usr/src/myapp/target/demo-0.0.1-SNAPSHOT.jar"]
```

Multi-stage Dockerfiles

Solution:

Multi-stage builds - separate images for build, test and run stages.

```
FROM maven:3.3.9 as build
COPY . /usr/src/myapp/
WORKDIR /usr/src/myapp
RUN mvn package

FROM java:8
WORKDIR /usr/src/myapp
COPY --from=build /usr/src/myapp/target/demo-0.0.1-SNAPSHOT.jar ./demo.jar
CMD ["java", "-jar",  "demo.jar"]
```

Exercise

Create a multi-stage docker build so that:

There’s a stage called ‘compile’ that generates a file compile.txt in a ubuntu-based image

And there’s a stage called ‘run’ that packages ‘compile.txt’ in an alpine-based image at /opt/compiled

The cmd for the resulting image should run ‘ls /opt/compiled’

Module 9

Using the Docker API

API Usages

- Running and managing containers
- Managing Swarm nodes and services
- Reading logs and metrics
- Creating and managing Swarms
- Pulling and managing images
- Managing networks and volumes

API Usages

- Can be called from any http client
- SDKs for Python and Go are available

Enable remote API

By default docker daemon listens on unix socket: /var/run/docker.sock

In order to listen on http - modify your docker daemon config:

```
sudo mkdir /etc/systemd/system/docker.service.d
sudo vi /etc/systemd/system/docker.service.d/docker.conf
```

```
[Service]
  ExecStart=
  ExecStart=/usr/bin/dockerd ... -H tcp://0.0.0.0:<port> -H
  unix:///var/run/docker.sock
```

Docker API example

```
$ curl --unix-socket /var/run/docker.sock -H "Content-Type:
  application/json" \
    -d '{"Image": "alpine", "Cmd": ["echo", "hello world"]}' \
    -X POST http:/v1.24/containers/create
  {"Id":"1c6594faf5","Warnings":null}

  $ curl --unix-socket /var/run/docker.sock -X POST
  http:/v1.24/containers/1c6594faf5/start

  $ curl --unix-socket /var/run/docker.sock -X POST
  http:/v1.24/containers/1c6594faf5/wait
  {"StatusCode":0}

  $ curl --unix-socket /var/run/docker.sock
  "http:/v1.24/containers/1c6594faf5/logs?stdout=1"
  hello world
```

Exercise

Enable docker remote API.

Run with curl:

- Pull a ‘busybox:latest’ image
- Run a container executing ‘ls /etc’
- Pull the logs from the container
- Remove the container

Use Documentation:

https://docs.docker.com/engine/api/v1.26/

Docker Healthcheck

Motivation: provide a command for continuous container health evaluation. Successful healthcheck signifies the application is ready to perform its job.

E.g:

```
HEALTHCHECK CMD curl --fail http://localhost:3000/ || exit 1
```

Docker Healthcheck

The options that can appear before CMD are:

- --interval=DURATION (default: 30s)
- --timeout=DURATION (default: 30s)
- --retries=N (default: 3)

Exercise

- Write a shell script to sleep 40 seconds and then create a file /opt/docker-course
- Build an apline-based image
- Put the script at /opt/entrypoint
- Define it as the entrypoint for container
- Define healthcheck to check for /opt/docker-course existence
- Verify that container health is reported correctly

Docker GC

```
> docker container prune
```

```
> docker image prune
```

```
> docker volume prune
```

```
> docker network prune
```

And to rule them all:

```
> docker system prune
```