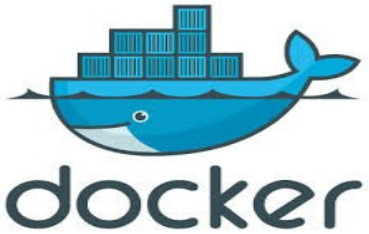


Docker: dietro le quinte

Alessandro Dentella

THUX

THUX

Alessandro Dentella
adentella@thux.it

Obiettivo

Comprendere le tecnologie alla base dei docker per avere più chiaro ogni passaggio. La differenza fra immagine e container diventerà chiarissima, così come la differenza fra virtualizzazione ed isolamento dei processi

Non Obiettivo

Non è un tutorial su come usare i docker. Ci sono tanti tutorial che servono allo scopo. Non parlo di docker Windows, non li conosco, mai usati.

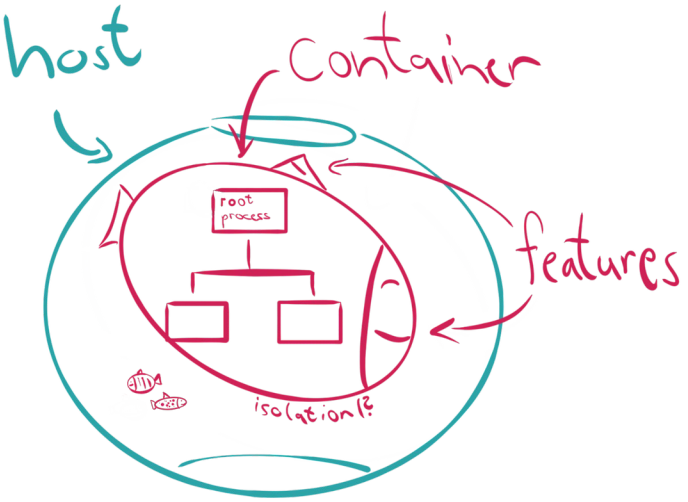
Alessandro Dentella
adentella@thux.it

Definizione: container

I container sono solo gruppi isolati di processi in esecuzione su un singolo host, che soddisfano un insieme di funzionalità 'comuni'.

Con questa chiacchierata volgiamo comprendere

- su singolo host
- gruppi di processi. I processi vivono in una struttura ad albero, quindi esiste un root process…
- isolati: cosa significa esattamente?
- hanno funzionalità comuni: quali?



THUX

Alessandro Dentella
adentella@thux.it

Metodo

- Come fareste con un orologio meccanico, per capire come funziona, proveremo a smontarlo per poi rimontarlo a mano
- Seconda metafora: se volete capire come funziona un motorino la cosa più utile è smontarlo, prendere confidenza con ogni singola parte. Questo NON vi garantisce di sapere andare in motorino, solo di saperlo aggiustare, capire.

Alessandro Dentella
adentella@thux.it

Target

- Chi ha già iniziato ad usare i docker ma non ha chiaro cosa succede veramente quando eseguiamo:
  - docker build
  - docker run/exec
- Chi vuole capire questa tecnologia di cui tutti parlano
- Chi è convinto di conoscerli bene ma non sa cosa siano i namespace del kernel, i cgroups, un filesystem a strati
- Chi NON si spaventa ad usare la riga di comando

Alessandro Dentella
adentella@thux.it

Letture consigliate

Sascha Grunert

Molti dei contenuti qui presentati sono presi da

Demistifying containers

- Kernel
- Runtime
- Image
- Security

https://github.com/saschagrunert/demystifying-containers?tab=readme-ov-file

Argomenti

Prima parte

- Presentazione
- mount –bind
- creazione immagini
  - Filesystem a strati
  - Dockerfile
  - analisi immagine

Seconda parte

- Namespaces
- run
- volume
- cgroup
- containerd

THUX

Alessandro Dentella
adentella@thux.it

Docker… personaggi in ordine sparso

- **docker**: comando (cli) e toolkit per gestione container
- **container**: una immagine in esecuzione
- **docker compose**: permette di configurare il setup di più containerin modo dichiarativo e molto chiaro
- **swarm**: gestione di tanti nodi docker che collaborano
- **kubernetes**: altra gestione (orchestrazione) di container

Alessandro Dentella
adentella@thux.it

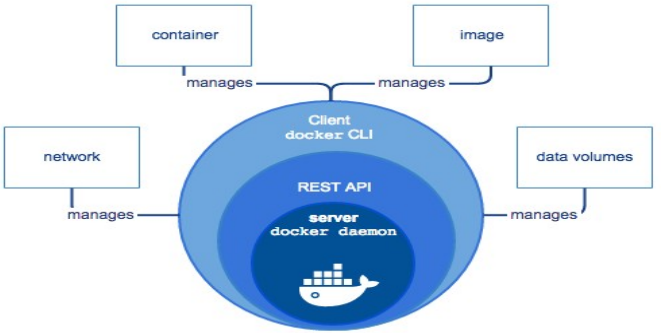
Obiettivi del progetto Docker

- **Superare lo spreco di risorse implicito nella virtualizzazione**
- **Facilitare la distribuzioni delle applicazioni** in contenitori leggeri, autosufficienti, facili da eseguire.
- Facili da scalare
- Isolati

Alessandro Dentella
adentella@thux.it

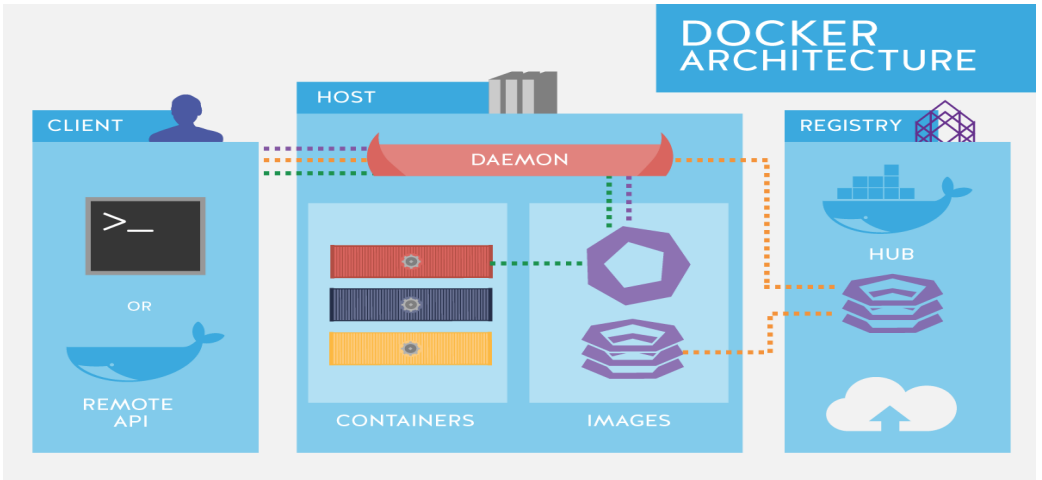
Componenti base

- **Image**: un file contenente un filesystem che ha tutto il necessario per eseguire un programma. Lo si crea con una *ricetta* chiamata `Dockerfile`
- **Container**: l'immagine in esecuzione.
- **Data volume**: il docker deve potere sparire ed essere sostituito. I nostri contenuti stanno in un "volume"
- **Network**: l'insieme di device e regole del kernel per regolare la visibilità dei container



Alessandro Dentella
adentella@thux.it

Achitettura



- `docker pull <myimage>`
- `docker image list`
- `docker build`
- `docker run`
- `docker exec`
- `docker ps`
- `docker inspect`
- `[docker compose …]`

THUX

Alessandro Dentella
adentella@thux.it

Docker Registry

- es.: `hub.docker.com`
- è gratuito, permette di immagazzinare le immagini pubbliche e private ad un modico costo
- gitlab ne ha uno privato che si inserisce bene nelle pipeline cd/ci

Alessandro Dentella
adentella@thux.it

Tecnologie usate

- Filesystem a layer (overlay/unionfs/aufs)
- Kernel condiviso (in contrapposizione alle macchine virtuali che devono emulare l'hardware)
- Namespaces (isolamento risorse net, volume, processi,...)
- Cgroups (limitazione risorse ram, cpu)
- Bind-mounting
- DNS
- Molto iptables (firewalling del kernel)

Alessandro Dentella
adentella@thux.it

mount --bind

Il bind mounting è una operazione in cui una cartella di un filesystem viene resa visibile/utilizzabile (anche) in un altro punto, offrendo quindi una *vista alternativa* dell'alberatura delle cartelle

```
mount --bind /some/where /else/where
mount -o bind /some/where /else/where
```

Usato per condividere dati fra host e guest
Usato per condividere /proc...

Alessandro Dentella
adentella@thux.it

Filesystem in un singolo file

Per prima cosa occorre familiarizzare con il fatto che un filesystem, ovvero la struttura logica che permette di gestire i file (coordinate per raggiungerlo fisicamente, permessi, link,…) non è legata solo ad un device fisico ma può essere creata in un file. Un esempio che tutti abbiamo in mente sono le immagini .iso dei sistemi operativi che possiamo ad esempio creare così, creando prima una chroot:

```
mkdir new-root
sudo debootstrap new-root
sudo chroot new-root
ping 8.8.8.8
```

generiamo una immagine .iso
==========

```
# genisoimage -o chroot_image.iso -r -J new-root/
# mkdir mnt
# mount chroot_image.iso ./mnt
# mount chroot_image.iso mnt/
mount: /home/docker/prove-sandro/mnt: WARNING:
source write-protected, mounted read-only.
```

THUX

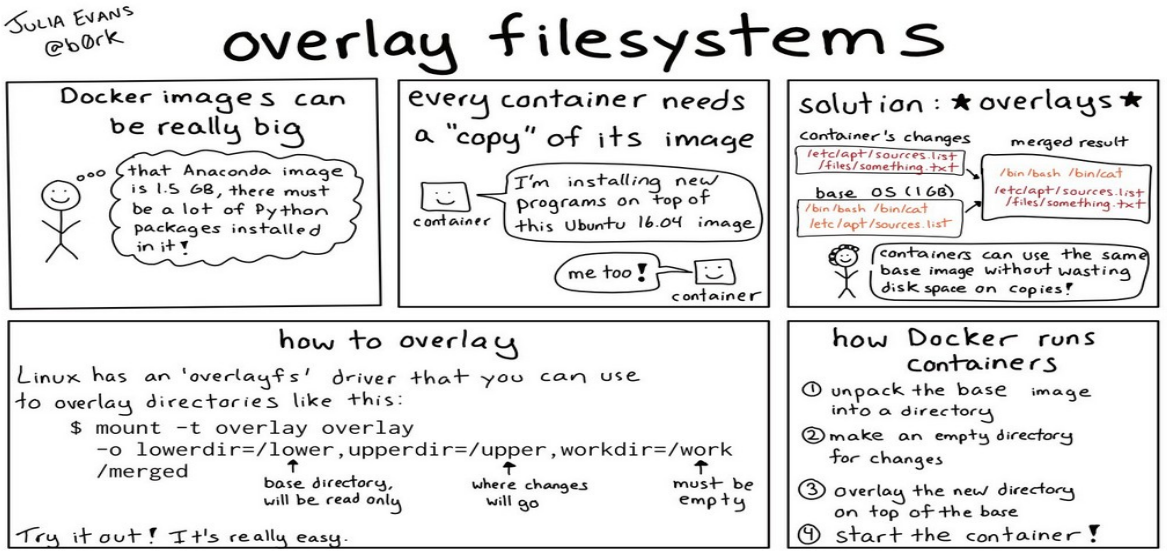
Alessandro Dentella
adentella@thux.it

rw

- Nell'esempio appena fatto, il Warning ci avvisa che i filesystem iso da noi utilizzato non permette la scrittura, per potere usare un filesystem iso per un cd utilizzabile anche in scrittura occorre un ulteriore elemento

Alessandro Dentella
adentella@thux.it

overlay



https://jvns.ca/blog/2019/11/18/how-containers-work--overlayfs/

Alessandro Dentella
adentella@thux.it

L'idea

Fogli di acetato sovrapposti, ciascuno con il suo disegno, permetto di comporre in modo efficiente un disegno

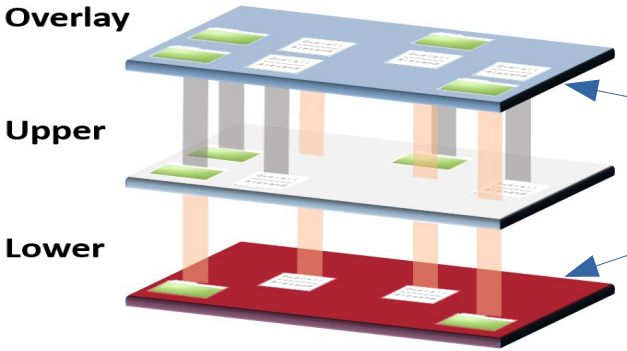



THUX

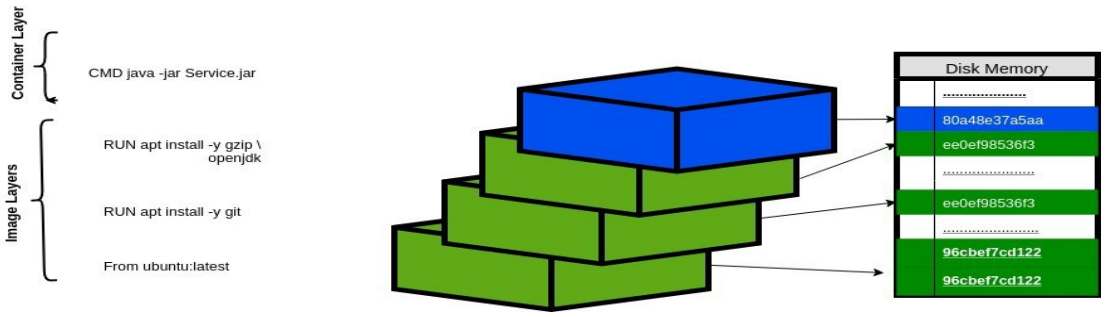
Alessandro Dentella
adentella@thux.it

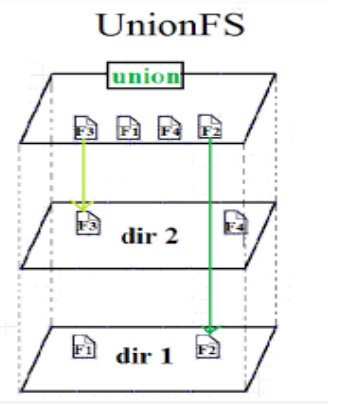
Layered Filesystem

strumentale, necessaria per rw,
i file della lowerdire in modifica
vengono prima scritti qui e poi spostati
con operazione atomica




```
sudo mount -t overlay anyname -o \
  lowerdir=./lower,upperdir=./upper,workdir=./work \
  ./overlay
```





THUX

Alessandro Dentella
adentella@thux.it

overlay in the shell

Mostriamo in azione il fielesystem overlay usando

- un layer in `ro` '`lower`'
- un layer in `rw 'upper'`
- una dir per il risultato `'merged'`
- `workdir`: overlay ha poi bisogno di una cartella temporanea vuota per gestire le operazioni di merge

Il comando da utilizzare è:

```
sudo mount -t overlay anyname -o \
   lowerdir=./lower1,upperdir=./upper,workdir=./work ./merged
```


Alessandro Dentella
adentella@thux.it

Esempio

creiamo una struttura con

questi comandi, poi applichiamo il merge

```
mkdir -p lower1/frutta \
          lower2/frutta \
          upper/frutta \
          upper/formaggi \
          work \
          merged

 touch lower1/frutta/ciliegia \
       lower1/frutta/mela \
       upper/formaggi/grana \
       upper/formaggi/zola

 echo "in lower1" > lower1/frutta/pera
 echo "in lower2" > lower2/frutta/pera
 echo "in upper" > upper/frutta/pera
```

```
$ tree lower1/ upper/ merged/
lower1/
└── frutta
    ├── ciliegia
    ├── mela
    └── pera
upper/
├── formaggi
│   ├── grana
│   └── zola
└── frutta
merged/
```

```
$ tree merged/
merged/
├── formaggi
│   ├── grana
│   └── zola
└── frutta
    ├── ciliegia
    ├── mela
    └── pera
```

Eliminazione

Procediamo ad eliminare un file:
`rm merged/frutta/mela`
Il risultato a fianco mostra:

- `merged` non ha più il file eliminato
- `upper/frutta` ha ora un file `'mela'`

  che mostra una `'c'`(character),
  questo file indica ad overlay
  che il file 'mela' va eliminato
  dal risultato del merge

```
$ tree upper merged/
upper
├── formaggi
│   ├── grana
│   └── zola
└── frutta
    └── mela
merged/
├── formaggi
│   ├── grana
│   └── zola
└── frutta
    ├── ciliegia
    └── pera
```

THUX

```
$ rm merged/frutta/mela
$ ls -l upper/frutta/mela
c--------- 1 root root 0, 0 mag  6 09:33 upper/frutta/mela
```

analisi di server docker

Questo è un esempio di un server reale dove ho lanciato:

```
mount | grep overlay
```

```
overlay on /var/lib/docker/overlay2/c9499ad3a1bf60ab81bac01d9ac443d44f769e50ca2711608a16310edad837ed/merged type overlay
(rw,relatime,lowerdir=/var/lib/docker/overlay2/l/DGOT6MGOQQVIUUSPLB3S43L3UG:/var/lib/docker/overlay2/l/46DHFORQMNL4OSVLLRNWV6SJOT:/var/lib/
docker/overlay2/l/ORFSXAULPITKGQMBELERIFWIIS:/var/lib/docker/overlay2/l/34PNWDG2B2RGT6UGLCCVFXDUFU:/var/lib/docker/overlay2/l/
KL6UYPC7OB65FTS2FZMSPISQGV:/var/lib/docker/overlay2/l/O4DXBRFVZULNGQX2OFC3QZU5BH:/var/lib/docker/overlay2/l/E7UH7ZJR3D6E7MMKVRUAHXQV24:/var/
lib/docker/overlay2/l/OJPXFHZKXPESTG73BRAJ7OGCNC:/var/lib/docker/overlay2/l/U4D34IVNLSWWBB5IDCAUD3IZ24:/var/lib/docker/overlay2/l/
OMQXRPOKLMMLFN53JE4J6I66J6:/var/lib/docker/overlay2/l/QVCSDHOT7K4LKMIJIFNTUOYYU6:/var/lib/docker/overlay2/l/PFNJYQOKSCCRQQQ2MMHRDH3I7W:/var/
lib/docker/overlay2/l/BDX7ZLN5FPHRNMVPOYNOL2BLUD,upperdir=/var/lib/docker/overlay2/
c9499ad3a1bf60ab81bac01d9ac443d44f769e50ca2711608a16310edad837ed/diff,workdir=/var/lib/docker/overlay2/
c9499ad3a1bf60ab81bac01d9ac443d44f769e50ca2711608a16310edad837ed/work)
```

tutti i componenti stanno in:
/var/lib/docker/operlay2
ogni layer occupa un suo file.
Questo garantisce che docker diversi condividano quanto più possibile

```
overlay on /var/lib/docker/overlay2/6f7081510b69a7c51774b99b2d29c72c7fa53a0d2f78294bec29494b957436c8/merged type overlay
(rw,relatime,
lowerdir=/var/lib/docker/overlay2/l/W3UZ6VNC4EOWY7YZLIOKMGYXZM
:/var/lib/docker/overlay2/l/FTW4MKTXOECNAXLFOLZNDSFKYZ
:/var/lib/docker/overlay2/l/E4QUMEPH7CJNZLJPBEV6QPABIU
:/var/lib/docker/overlay2/l/5OFP4R5QCFCCKHFSBUPQNM3KGQ
:/var/lib/docker/overlay2/l/DFFWSDAHVFN75N77CFUY5WADEH
:/var/lib/docker/overlay2/l/LRGLB73JCC333AALPPUPYAR44D
:/var/lib/docker/overlay2/l/ZCDEIT2NPIBD5DM3DHIB6W4YHT
:/var/lib/docker/overlay2/l/FDPOHSMDDR2NAJU64GQ7OJ3OTL
:/var/lib/docker/overlay2/l/KER4VJW2NQHTA4I2UX3L2ZZ2JV
:/var/lib/docker/overlay2/l/E3NLEAJT6PIVRY5X6I5MI7YSP7
:/var/lib/docker/overlay2/l/7VFTTL7RPJIPTRFSXPGE4EKRXJ
:/var/lib/docker/overlay2/l/FRKY5ZBJO3TAU7XIVDF5PI4X2H,
upperdir=/var/lib/docker/overlay2/6f7081510b69a7c51774b99b2d29c72c7fa53a0d2f78294bec29494b957436c8/diff,
workdir=/var/lib/docker/overlay2/6f7081510b69a7c51774b99b2d29c72c7fa53a0d2f78294bec29494b957436c8/work)
```

THUX

Creiamo un filesystem

- Mettiamo a frutto quanto abbiamo visto per creare un filesystem
- Obiettivo:
  - facile da creare, aggiungendo personalizzazione a "mattoncini" standard
  - che non sprechi spazio su disco
  - che permetta di scrivere
  - che permetta la persistenza dei dati anche quando il container viene spento

Alessandro Dentella
adentella@thux.it

Dockerfile

- Sono le ricette per creare le immagini e vengono rese pubbliche
- Le ricette sono base per altre ricette

```
# syntax=docker/dockerfile:1
FROM python:3
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
COPY requirements.txt /code/
RUN pip install -r requirements.txt
COPY . /code/
```

```
# build stage
FROM node:lts-alpine as build-stage
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# production stage
FROM nginx:stable-alpine as production-stage
COPY --from=build-stage /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

THUX

RUN, COPY

RUN e COPY sono due comandi che **aggiungono layer al risultato**. Per il motivo già spiegato non è opportuno mettere in un layer ed eliminare in uno successivo. La dimensione dell'immagine non diminuisce. Meglio concatenare tutti i comandi nello stesso layer:

```
RUN \
  apt-get update \
  && DEPS=" \
    telnet \
    iproute2 \
    net-tools \
    iputils-ping \
  " \
  && seq 1 8 | xargs -I{} mkdir -p /usr/share/man/man{} \
  && apt-get update \
  && apt-get install -y --no-install-recommends $DEPS \
  && rm -rf /var/lib/apt/lists/*
```

THUX

Alessandro Dentella
adentella@thux.it

.dockerignore

Per gestire meglio i file che vogliamo copiare nell'immagine che stiamo creando possiamo usare il file .dockerignore.

Ogni file/cartella contenuta li dentro verrà ignorata nella copia.

Alessandro Dentella
adentella@thux.it

Efficenza nella creazione

```
# syntax=docker/dockerfile:1
FROM python:3
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
COPY requirements.txt /code/
RUN pip install -r requirements.txt
COPY . /code/
```

- Nel normale ciclo di sviluppo di applicazioni avremo sempre la creazione di un ambiente e la modifica del codice
- Prendiamo Python come esempio: useremo poetry o pip per installare pacchetti, poi modifichiamo codice e magari dobbiamo installare un nuovo pacchetto e poi tante modifiche al codice.
- L'operazione più lunga è l'installazione di pacchetti, quella più veloce la copia del codice. La sequenza COPY/RUN/COPY assicura che i layer con i pacchetti non vengano toccati (e quindi vengano ri-utilizzati), la cosa è vera con ogni applicazione, es:
  COPY packages.lock, /RUN yarn install

Alessandro Dentella
adentella@thux.it

CMD

CMD indica il comando che viene eseguito nel container e può essere sovrascritto dall'opzione "command" di docker compose o dall'argomento

Accetta 3 forme:

```
CMD ["executable","param1","param2"] (exec form, this is the preferred form)
CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
CMD command param1 param2 (shell form)
```

Con effetto lievemente differente, la terza lascia /bin/bash come primo processo

Alessandro Dentella
adentella@thux.it

ENTRYPOINT

è stato introdotto dopo CMD ed in parte si sovrappone a CMD nel definire il comando che viene lanciato nel docker.

Se impostato in assenza di CMD, è analogo, ovvero ha la exec form e la shell form e definisce il comando.

Più interessante quando invece viene usato assieme a CMD, allora CMD di fatto aggiunge parametri al comando definito nell'ENTRYPOINT. Questo permette di fare docker eseguibili dove i parametri vengono passati senza dovere riscrivere il comando.

Alessandro Dentella
adentella@thux.it

ENTRYPOINT as shell cmd

Un utilizzo molto tipico di `ENTRYPOINT` è impostare uno script `entrypoint.sh` che si occuperà di preparare l'ambiente (ad esempio: creare un utente db, un database, ...) prima di eseguire il db stesso.

Quando l'inizializzazione è completata, di solito terminerà con una riga:

```
exec "$@"
```

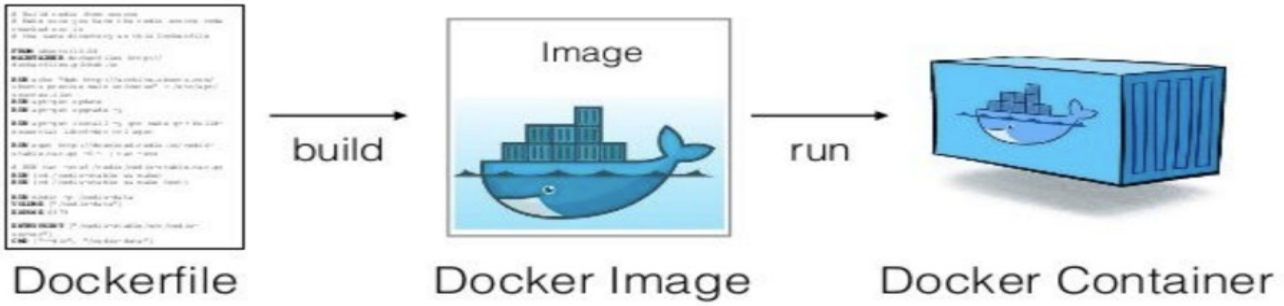
che interromperà lo script corrente ed eseguirà CMD che verrà passato come argomento.

Alessandro Dentella
adentella@thux.it

CMS interactions with ENTRYPOINT

- https://docs.docker.com/engine/reference/builder/#understand-how-cmd-and-entrypoint-interact
- https://stackoverflow.com/questions/21553353/what-is-the-difference-between-cmd-and-entrypoint-in-a-dockerfile

Alessandro Dentella
adentella@thux.it



THUX

Alessandro Dentella
adentella@thux.it

layers al lavoro

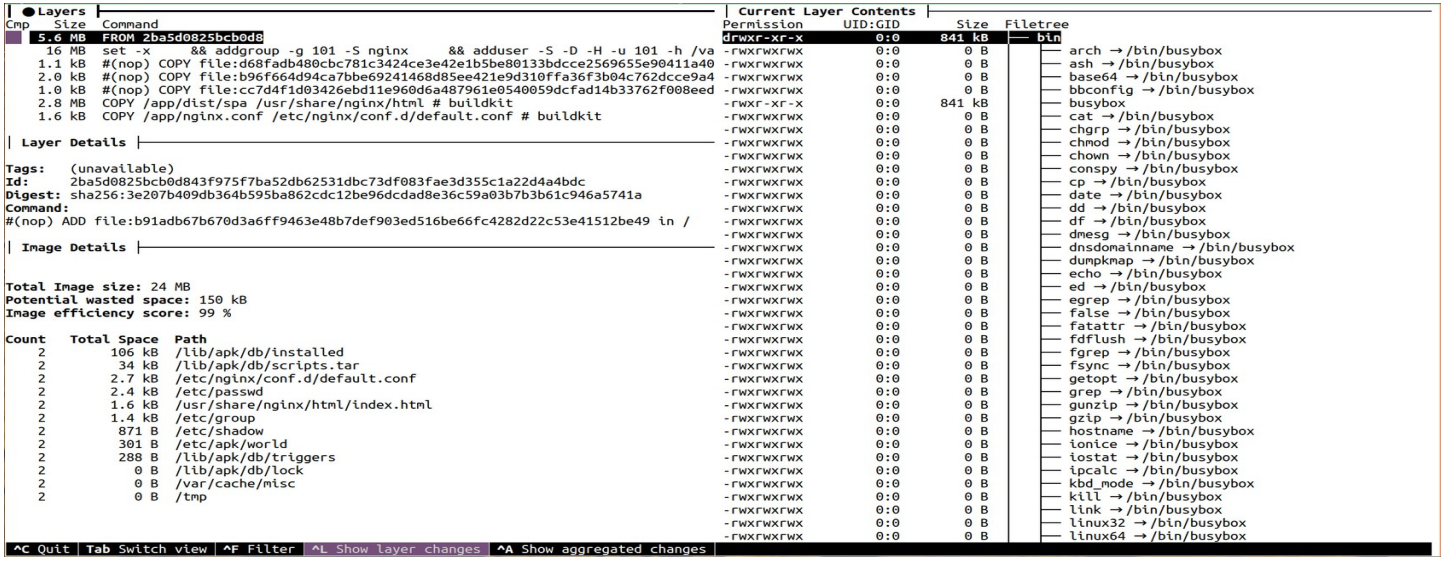
- Possiamo analizzare la struttira dei layer con il comando docker history:

```
# docker history registry.thux.dev/e-den/wikidattica/wikidattica-quasar:topic-default-dev
IMAGE          CREATED       CREATED BY                                      SIZE      COMMENT
132ca6e26918   2 years ago   CMD ["nginx" "-g" "daemon off;"]                0B        buildkit.dockerfile.v0
<missing>      2 years ago   EXPOSE map[80/tcp:{}]                           0B        buildkit.dockerfile.v0
<missing>      2 years ago   COPY /app/nginx.conf /etc/nginx/conf.d/defau…   1.58kB    buildkit.dockerfile.v0
<missing>      2 years ago   COPY /app/dist/spa /usr/share/nginx/html # b…   2.83MB    buildkit.dockerfile.v0
<missing>      2 years ago   /bin/sh -c #(nop)  CMD ["nginx" "-g" "daemon…   0B
<missing>      2 years ago   /bin/sh -c #(nop)  STOPSIGNAL SIGTERM           0B
<missing>      2 years ago   /bin/sh -c #(nop)  EXPOSE 80                    0B
<missing>      2 years ago   /bin/sh -c #(nop)  ENTRYPOINT ["/docker-entr…   0B
<missing>      2 years ago   /bin/sh -c #(nop) COPY file:cc7d4f1d03426ebd…   1.04kB
<missing>      2 years ago   /bin/sh -c #(nop) COPY file:b96f664d94ca7bbe…   1.96kB
<missing>      2 years ago   /bin/sh -c #(nop) COPY file:d68fadb480cbc781…   1.09kB
<missing>      2 years ago   /bin/sh -c set -x     && addgroup -g 101 -S …   15.6MB
<missing>      2 years ago   /bin/sh -c #(nop)  ENV PKG_RELEASE=1            0B
<missing>      2 years ago   /bin/sh -c #(nop)  ENV NJS_VERSION=0.4.1        0B
<missing>      2 years ago   /bin/sh -c #(nop)  ENV NGINX_VERSION=1.19.0     0B
<missing>      2 years ago   /bin/sh -c #(nop)  LABEL maintainer=NGINX Do…   0B
<missing>      2 years ago   /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      2 years ago   /bin/sh -c #(nop) ADD file:b91adb67b670d3a6f…   5.61MB
```


Alessandro Dentella
adentella@thux.it

layer con dive

Il comando dive ci permette di avere una introspezione molto accurata non solo dei layer e dei comandi utilizzati per crearli ma anche dei file contenuti in ogni layer



Alessandro Dentella
adentella@thux.it

running an image

- `docker build . -t presentazione`
- `docker run presentazione`
- (se l'immagine non c'è viene tentato un "`docker pull`")

  l'immagine viene eseguita e subito si esce a meno che si usi `opt -it` (interactive terminal)
- `docker run -it presentazione`
- `docker run -it -v ./src/prj:/code presentazione`
- `docker run –net web -it presentazione`

-t (tag) =
Nome immagine

crea un device interno questa rete.
Un dns interno permette di trovarsi
usando il nome del continer

THUX

Alessandro Dentella
adentella@thux.it

Isolamento

Cominciamo a capire il problema riguardando la situazione del chroot e quanto quella fallisca allo scopo di isolare:

```
chroot new-root
mount -t proc proc  /proc
ps ax   # mostra tutti i processi
ip -br addr list  # mostra tutte le interfacce
```

Chiaramente questo isolamento fallisce dal punto di vista della sicurezza che vogliamo raggiungere. Addirittura è possibile uscire dalla chroot cme utente superuser

Alessandro Dentella
adentella@thux.it

Namespaces

I **namespace** sono una caratteristica del kernel Linux introdotta nel 2002 con Linux 2.4.19.

L'**idea alla base** di un namespace è quella di avvolgere determinate risorse di sistema globali in uno strato di astrazione. Ciò fa sembrare che i processi all'interno di un namespace abbiano la propria istanza isolata della risorsa.

L'astrazione del namespace del kernel consente a **diversi gruppi di processi di avere visioni diverse del sistema.**

Completati i 7 namespace nel 2013 (kernel 3.8): mnt, pid, net, ipc, uts, user e cgroup

Alessandro Dentella
adentella@thux.it

Namespaces

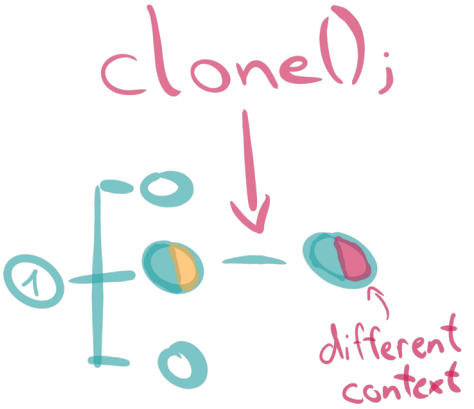
- I namespace sono gestiti dal kernel
- Ogni processo già nasce appartenendo ad un namespace (root namespace), quindi creare un nuovo namespace non comporta maggior complessità

Alessandro Dentella
adentella@thux.it

fork vs. clone

Le API namespaces aggiungono clone che permette un controllo pi`u find di quali dati del processo padre venonogno resi disponibili al figlio

- fork: tutto è visibile (modalità copy on write per efficienza)
- clone: a scelta (spazio di indirizzamento, la tabella dei descrittori di file, e la tabella dei gestori di segnali...)



THUX

Alessandro Dentella
adentella@thux.it

mount

```
sudo unshare --mount
# mkdir mnt
# mount -t tmpfs -o size=10m none ./mnt
# df -h ./mnt
File system     Dim. Usati Dispon. Uso% Montato su
none             10M     0     10M   0% /home/sandro/Archivio/thunder/corsi/Docker/esempi/mnt
# touch ./mnt/abc
```

- Verificare che l'host non vede questo mount
- possiamo vedere il mount dall'host:
  ls /proc/$(pgrep -u root bash)/mountinfo

Alessandro Dentella
adentella@thux.it

Volume

- docker ci permette di montare cartelle locali
- Possiamo montare un volume in ogni posizione
- Nel Dockerfile l'istruzione VOLUME avvisa che verrà creato un volume per conservare i dati in quella posizione (a meno che venga montato un volume dall'esterno)

Alessandro Dentella
adentella@thux.it

memoria e mount point

La memoria effettivamente utilizzata per il punto di mount si trova in uno strato di astrazione chiamato Virtual File System (VFS), che fa parte del kernel e su cui si basa ogni altro filesystem.

Se lo spazio dei nomi viene distrutto, la memoria di mount è irrimediabilmente persa.

L'astrazione dello spazio dei nomi di mount **ci dà la possibilità di creare interi ambienti virtuali in cui siamo l'utente root anche senza autorizzazioni di root**

Alessandro Dentella
adentella@thux.it

hostname - UTS

```
sudo unshare --uts
# hostname thux
```

- verificare che l'hostname della macchina host non è cambiato
- provare senza `--uts` e verificare che cambia
- NB: UTS - Unix time-sharing system

Alessandro Dentella
adentella@thux.it

inter process communication

Le IPC sono un insieme di meccanismi che consentono la comunicazione e la condivisione di dati tra i processi del sistema operativo. Alcuni dei tipi di IPC includono **code di messaggi, semafori e memorie condivise**.

```
sudo unshare --ipc
# ipcs
------ Code messaggi  --------
chiave     msqid      proprietario perms      byte utilizzati messaggi

------ Segm. Memoria Condivisa --------
chiave     shmid      proprietario perms      byte       nattch     stato

------ Matrici semafori --------
chiave     semid      proprietario perms      nsems
```


Alessandro Dentella
adentella@thux.it

ipc: cosa in concreto

Esempi di come vengono utilizzate le IPC:

- code di messaggi
- semafori
- memoria condivisa
- socket
- pipe

THUX

Alessandro Dentella
adentella@thux.it

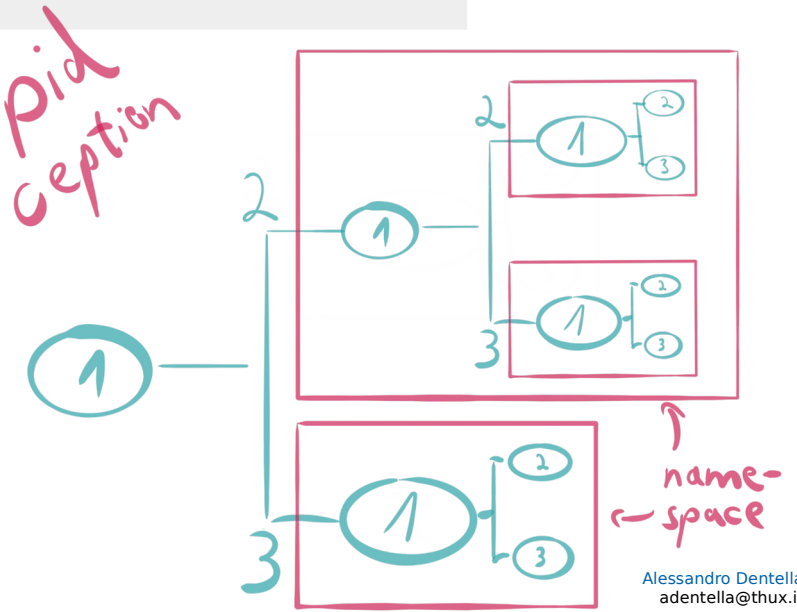
La shell ha pid 1

pid

```
sudo unshare --pid --fork --mount-proc
# ps ax
  PID TTY      STAT   TIME COMMAND
    1 pts/5    S      0:00 /bin/bash
    2 pts/5    R+     0:00 ps ax
```

Notare

- i processi dell'host non sono visibili
- la shell ha pid 1
- provare senza `-mount-proc`
- se uccido il processo 1, i figli muoiono



THUX

Alessandro Dentella
adentella@thux.it

kill pid padre

- Proviamo ad entrare nel nostro namespace usando nsenter da un'altra shell:
  `sudo nsenter -t $(pgrep -u root bash) --pid --mount`
- Vediamo che si vedono entrambe le shell (`ps ax`)
- Se uccidiamo la seconda shell, si chiude solo quella shell, se uccidiamo la prima di chiudono entrambe

network

```
sudo unshare --net -p --fork --mount-proc /bin/bash
# ip addr ls
  1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN   group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

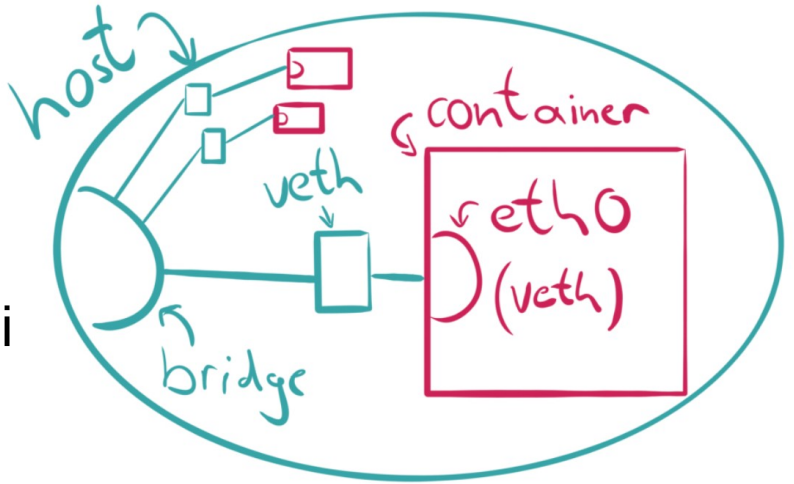
- `-n` è l'abbreviazione di `-map-newnet`
- si vede solo il device `` `lo` ``
- si vedono tutti i file del filesystem… (provare). Questo per sottolineare che noi possiamo essere in un namespace anche solo con un

Alessandro Dentella
adentella@thux.it

net

Ogni namespace contiene

- un insieme privato di indirizzi IP
- la propria tabella di routing
- elenco di socket
- tabella di tracciamento delle connessioni
- firewall e altre risorse relative alla rete.



Alessandro Dentella
adentella@thux.it

user

- Dal kernel 3.5 (2012) è stato aggiunto il namespace user che ha permesso l'isolamento di utenti e gruppi.
- Dal kernel 3.8 è possibile creare user namespace senza essere utente privilegiato.
- Lo stesso processo può essere visto con id differenti dentro e fuori dal namespace
- È interessante notare che è possibile avere un processo che è di un utente non privilegiato fuori e privilegiato dentro il container. Questo porta a dovere porre delle attenzioni alla sicurezza (per questo è stato introdotto un fie che determina se è possibile per un processo modificare i propri gruppi di appartenenza)

Alessandro Dentella
adentella@thux.it

cgroup

- L'obiettivo principale dei cgroups è quello di supportare il limitazione delle risorse, la prioritizzazione, la contabilità e il controllo.
- Un esempio è un Out-of-Memory (OOM) killer che aggiunge la capacità di terminare un cgroup come un'unica unità per garantire l'integrità complessiva del carico di lavoro.
- tramite i cgroup possiamo limitare ad esempio il numero delle cpu (anche frazioni) o la memoria totale utilizzata

Alessandro Dentella
adentella@thux.it

Namespaces

Limitano un processo e i suoi processi figli a una porzione circoscritta del sistema sui quali si basano. Al fine di incapsulare i processi, Docker utilizza Namespaces in cinque diversi ambiti:

- **System identification** (UTS): nella virtualizzazione basata su container vengono utilizzati i Namespaces UTS al fine di attribuire *a ogni container il proprio nome di host e di dominio.*
- **Process IDentifier** (PID): ogni container Docker utilizza un proprio spazio di nomi per i processi di identificazione. **I processi eseguiti all'esterno di un container non sono visibili dal suo interno.** In questo modo i processi incapsulati all'interno di container sullo stesso sistema ospitante possono possedere gli stessi PID, senza che venga però a crearsi un conflitto.
- **Comunicazione tra processi** (IPC o anche Inter-Process Communication): gli spazi di nomi IPC isolano i processi in un container in modo che *la comunicazione con i processi esterni al container venga impedita.*
- **Risorse di rete** (NET): con gli spazi di nomi di rete *è possibile assegnare risorse di rete separate a ogni container, come ad esempio* ***indirizzi IP e tabelle di routing*****.**
- **Mount Point del file system** (MNT): grazie agli spazi di nomi Mount ***un processo isolato non vede mai l'intero file system dell'host***, ma piuttosto solo una parte di esso, che corrisponde solitamente a un'immagine specifica per questo container.

da: digital guide ionos

Alessandro Dentella
adentella@thux.it

Namespaces names

- I namespaces prendono il nome dal PID del processo padre, nell'host
- In /proc abbiamo tutte le informazioni riguardanti i namespaces in /proc/ns
- I processi appartengono sempre ad un namespace, quindi non c'è overhead per il fatto che vengano fatti partire in un namespace

Alessandro Dentella
adentella@thux.it

unshare / nsenter

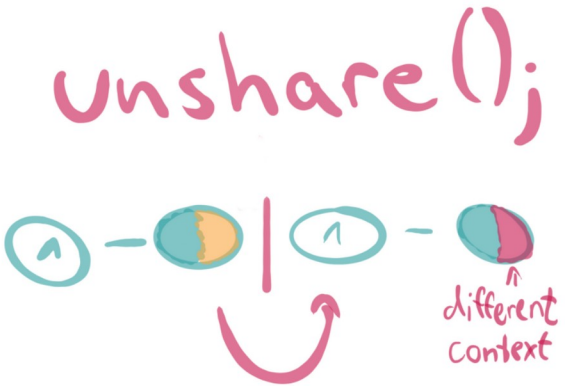
- la bash ha un comando che chiama direttamente le API del kernel per creare un namespace e poi eseguire un comando in quel namespace
- è possibile attaccarsi ad un namespace per eseguire del codice. Ad esempio possiamo fare debug
- con docker inspect possiamo trovare il namespace usato dal nostro docker:

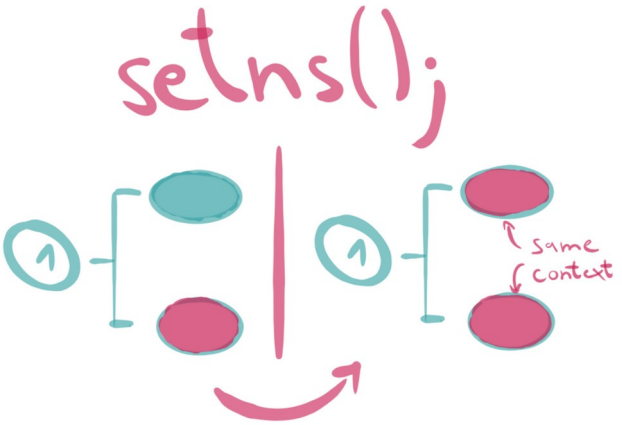
```
ns=$(docker inspect docker-name|jq '.[] .State.Pid')
nsenter --target $ns --mount --uts --ipc --net --pid
```


Alessandro Dentella
adentella@thux.it

setens <=> nsenter

- `nsenter` permette di entrare in un ns
- unshare separa una parte del contesto dal processo





THUX

Alessandro Dentella
adentella@thux.it

namespace di un processo

Per quanto chiaro, voglio sottolineare che un processo non appartiene ad un namespace solo ma appartiene ad un namespace **per ogni tipologia,** questo risulta chiaro guardando questa immagine che mostra i ns per processo.

L'id dei namespace è differente per ogni tipologia

```
bluffx:~ # cd /proc/1972/ns/
bluffx:/proc/1972/ns # ll
totale 0
lrwxrwxrwx 1 root root 0 apr 16 10:49 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 apr 16 10:49 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 apr 16 10:49 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 root root 0 apr 16 10:49 net -> 'net:[4026532008]'
lrwxrwxrwx 1 root root 0 apr 16 10:49 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 apr 16 10:49 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 apr 16 10:49 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 apr 16 10:49 uts -> 'uts:[4026531838]'
```

THUX

Alessandro Dentella
adentella@thux.it

run a container con unix tools

- https://ilearnedhowto.wordpress.com/tag/unshare/

```
$ docker export blissful_goldstine -o dockercontainer.tar
$ mkdir rootfs
$ tar xf dockercontainer.tar --ignore-command-error -C rootfs/
$ unshare --mount --uts --ipc --net --pid --fork --user --map-root-user
chroot $PWD/rootfs ash
root:# mount -t proc none /proc
root:# mount -t sysfs none /sys
root:# mount -t tmpfs none /tmp
```


Alessandro Dentella
adentella@thux.it

runc

Quello che abbiamo appena visto è il compito di runc:

- **Creazione del namespace**: runc crea i namespace del kernel necessari per il container, come il namespace PID, il namespace rete, il namespace IPC e il namespace utente.
- **Creazione del cgroup:** runc crea un gruppo di controllo (cgroup) per il container, che viene utilizzato per limitare le risorse del sistema disponibili per il container, come la memoria, la CPU e le I/O.
- **Montaggio del filesystem:** runc monta il filesystem root del container, che può essere un filesystem isolato o un filesystem overlay che condivide parti del filesystem host.
- **Configurazione dell'ambiente:** runc imposta le variabili d'ambiente e le risorse del sistema necessarie per il container, come la configurazione della rete, le variabili di ambiente e i parametri del kernel.
- **Esecuzione del processo iniziale:** runc avvia il processo iniziale del container, che può essere specificato nell'immagine del container (ad esempio, l'entrypoint in Docker) o fornito manualmente all'avvio del container.
- **Gestione dei processi:** runc gestisce i processi all'interno del container, inclusa la creazione e l'eliminazione dei processi, la gestione dei segnali e il monitoraggio dello stato dei processi.

Alessandro Dentella
adentella@thux.it

mount –bind e inode

è possibile montare cartelle o singoli file in un docker, ma il meccanismo di propagazione di una modifica del file si basa sugli inode e se il programma di edit (vi, docker login, jmacs…) invece che editare il file lo sostituisce, la propagazione non avviene.

Come escamotage è possibile creare un link simbolico, montare il link simbolico (che non cambierà) e quindi essere garantiti che le modifiche vengano propagate al container.

Alessandro Dentella
adentella@thux.it

Questo è l'inizio di una serie di lezioni sui container.

Se ti è piaciuta questa presentazione e sei interessato ad approfondimenti su questi temi contattami: adentella@thux.it

Alessandro Dentella
adentella@thux.it