Docker

Corsi GNU/Linux Avanzati 2018

Giacomo Vercesi (1vercesig@gmail.com)

Le slide sono disponibili su https://slides.poul.org/2018/corsi-linux-avanzati/Docker

Cos’è Docker?

“Docker allows you to package an application with all of its dependencies into a standardized unit [container] for software development.”

Cos’è un containter?

Un container è una porzione di sistema isolata, con acesso limitato alle risorse (rete, filesystem, etc)

È la combinazione di chroot, cgroups, network namespaces ed altro

Un container docker **non è** una macchina virtuale

Perché dovrei usarlo?

Permette di avere un ambiente consistente e riproducibile

Per gli sviluppatori:

- Sviluppo e testing semplificati
- Deployment documentabile

Per i sysadmin:

- Isolazione delle applicazioni
- Meno dipendenze
- Manutenzione semplificata

Buzzwords

(che non useremo)

- Horizontal scaling
- One line deployment
- Cloud!
- DevOps filosophy!

Nice buzzwords there

But what can it do?

Docker crash course

Installazione di docker

Docker si può installare GNU/Linux, Mac OSX and Windows

Istruzioni per l'installazione
https://docs.docker.com/engine/installation/

Hello world

```
# docker run alpine /bin/echo "Hello, World"
```

```
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
ff3a5c916c92: Pull complete
Digest: sha256:7df6db5aa61ae9480f52f0b3a06a140ab98d427[...]
Status: Downloaded newer image for alpine:latest
Hello, World
```

Cos’è successo?

```
# docker run alpine /bin/echo "Hello, World"
```

Docker ha:

- Scaricato l’archivio di Alpine Linux (una distro Linux)
- Scompattato l’archivio
- Creato un nuovo container
- Fatto andare il comando `echo "Hello, World"` dentro al container

Per maggiori dettagli cliccare qui

Hello World Interattivo

Come eseguire una shell interattiva

```
# docker run -i -t alpine /bin/sh
```

```
/ # echo Hello, World!
Hello, World!
/ # exit
```

Daemonizzare Hello World

Con l’opzione `-d` si possono lanciare i comandi in background

```
# docker run --name "hello_world" -d alpine \
    /bin/sh -c "while true; do echo hello world; sleep 1; done"
```

```
03b37fa08977fd01c8a95677b834473bfbc0fe82bd15982dee266935ad491a7e
```

Identificare i container

Ogni container è identificato da:

- Un hash univoco
- Un nome (personalizzabile)

```
# docker ps -a
```

```
CONTAINER ID        IMAGE               COMMAND                  CREA
03b37fa08977        alpine              "/bin/sh -c 'while tr"   9 se
1857ba99f339        alpine              "/bin/sh"                21 s
a298c9e3d59d        alpine              "/bin/echo 'Hello, Wo"   37 s
```

Eliminare i container

```
# docker rm <nome_container_o_hash>
```

Ooops!

```
# docker rm hello_world
Error response from daemon: You cannot remove a running container 43a
Stop the container before attempting removal or force remove
```

```
docker ps -a
```

```
CONTAINER ID        IMAGE               COMMAND                  CREA
03b37fa08977        alpine              "/bin/sh -c 'while tr"   3 mi
```

The hello_world container is still running!

```
# docker stop hello_world
hello_world
# docker logs hello_world
Hello World
Hello World
Hello World
[...]
```

Ok, abbiamo fatto **abbastanza** Hello Worlds

Voglio dockerizzare la mia applicazione

Dockerizzare un’applicazione

Useremo come esempio l’applicazione Up1, scritta in node.js:

https://github.com/Upload/Up1

Dobbiamo creare un’immagine

Dockerfiles

Le immagini di Docker sono create tramite **Dockerfiles**

Ogni riga del Dockerfile specifica una direttiva per la creazione dell'immagine

Le direttive sono eseguite sequenzialmente

Example Dockerfile

```
FROM node:latest
LABEL maintainer=1vercesig@users.github.com
LABEL version=1.0.0

EXPOSE 9000/tcp

ENV HTTP="true" HTTP_LISTEN="0.0.0.0:9000" MAX_FILE_SIZE=50000000 [..

RUN apt-get install -y git && cd /srv && \
    git clone https://github.com/Upload/Up1 && \
    cd Up1/server && npm install && apt-get remove -y git

WORKDIR /srv/Up1/server

COPY server.conf.template server.conf.template
COPY genconfig.sh genconfig.sh
```

Build it

```
# git clone https://github.com/vercesig/Up1-docker.git
# cd Up1-docker
# docker build -t vercesig/up1 .
```

Crea l’immagine up1

Nel Frattempo…

Comandi Utili:

Comando	Descrizione
run	crea ed esegue un container
[re]start/stop/kill	[re]start/stop/kill un container
ps	elenca tutti i container
images	elenca tutte le immagini
build	crea un'immagine da un Dockerfile
rm/rmi	cancella un continer/immagine
rename	rinomina un container

Google, `man` e `docker --help` per il resto

Run it

```
# docker run -e "API_KEY=random1" -e "DELETE_KEY=random2" \
--name up1 -p 8080:9000 -v /tmp/up1:/srv/Up1/i/ \
fcremo/up1
```

Opzione	Significato
`--name up1`	Assegna al container il nome `up1`
`-p 8080:9000`	Espone la porta 9000 del container sulla porta 8080
`-e "KEY=VAL"`	Imposta la variabile d'ambiente KEY a VAL
`-v /host/:/cont/`	Monta `/host/` (lato host) a `/cont/` (lato container)

Di default le porte sono esposte solo per la macchina host

Usando `-p 0.0.0.0:8080:9000` si espone il servizio su tutte le interfacce

Bonus: docker usa iptables per fare networking
Da sfruttare a vostro rischio e pericolo

Come passo la configurazione al container?

Varie opzioni:

- Variabili d'ambiente
- Script sed/awk/bash
- https://github.com/jwilder/dockerize
- Usando i volumi

Se avete da fare `docker build` ad ogni cambio di configurazione (**probabilmente**) state facendo qualcosa di sbagliato

DEMO

Per gli sviluppatori di applicazioni

Docker funziona bene con applicazioni

- modulari
- (facilmente) configurabili
- autoaggiornabili

NB: i container sono effimeri - possono essere rimpiazzati in qualsiasi momento

E per sistemi con più componenti?

Mattermost

App di chat open source
Ha bisogno di un database
Useremo postgres

Docker Hub ha già entrambe le immagini

```
# docker pull postgres
# docker pull jasl8r/mattermost
```

Vogliamo che i due container si parlino tra di loro

Ma siano isolati con il resto dei container

Creare una nuova rete

```
# docker network create mattermost-net
```

Avviamo Postgres

e lo colleghiamo alla rete `mattermost-net`

```
# docker run --name mattermost-postgres -d \
--net mattermost-net \
--env 'POSTGRES_USER=mattermost' --env 'POSTGRES_PASSWORD=pwd' \
--volume /tmp/postgres:/var/lib/postgresql postgres
```

Controlliamo la configurazione di rete

```
# docker inspect mattermost-postgres | \
    jq ".[].NetworkSettings.Networks"
```

```
# docker run --rm --net mattermost-net alpine \
    /bin/ping -c 3 mattermost-postgres
```

Avviamo Mattermost

```
# docker run --name mattermost -d --publish 8080:80 \
    --net mattermost-net \
    --volume /tmp/mattermost:/opt/mattermost/data \
    --env 'DB_ADAPTER=postgres' \
    --env 'DB_HOST=mattermost-postgres' \
    --env 'DB_USER=mattermost' --env 'DB_PASS=pwd' \
    jasl8r/mattermost
```

I container possono connettersi per nome

#itsmagic

No, è il server DNS di Docker che risolve i nomi dei container con i relativi ip

Se volessimo scrivere meno al terminale?

Enter docker-compose

Tool per la coordinazione dei container

Example

```
mysql:
  restart: always
  image: mysql:latest
  environment:
    - MYSQL_USER=mattermost
    - MYSQL_PASSWORD=password
    - MYSQL_DATABASE=mattermost
    - MYSQL_ROOT_PASSWORD=password
  volumes:
    - /srv/docker/mattermost/mysql:/var/lib/mysql

mattermost:
  restart: always
  image: jasl8r/mattermost:3.4.0-1
  links:
    - mysql:mysql
```

Deploy mattermost

Get docker-compose.yml

```
$ wget "https://raw.githubusercontent.com/jasl8r/docker-mattermost/ma
```

execute

```
# docker-compose up
```

Done!

Grazie per l'attenzione

https://slides.poul.org/

Slides originali di Filippo Cremonese
Modifiche di Giacomo Vercesi