Docker Provisioning

Build, ship & run

Rubén Gómez García

Version 1.2.0 2017-05-24

Table of Contents

Chapter 1. Introducción

- Docker es un motor open-source que automatiza el despliegue de aplicaciones en contenedores
- Sus inicios fueron el 15 de Marzo del 2013
- Docker apareció tras una charla de 5 minutos en las conferencias de desarrollo Python en Santa Clara, California.
- Su código fue publicado en GitHub para colaborar y ampliar el proyecto.
- En pocos meses, la industria comenzó a aceptar el software como una revolución de como se debía construir, entregar y ejecutar el software.
- Docker es una herramienta que promete la encapsulación de la creación de un artefacto distribuible para cualquier aplicación, desplegandolo en escala en cualquier entorno.

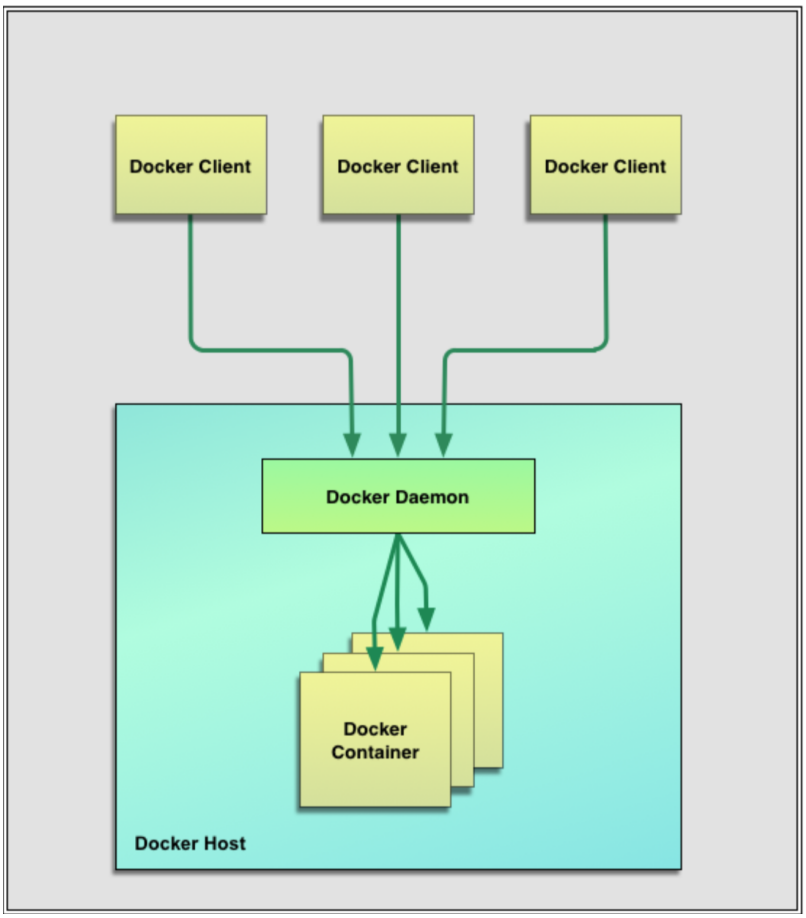
1.1. Virtualización

- Docker es más que una plataforma de virtualización.
- Docker incluye tecnologías de KVM, xen, OpenStack, Mesos, Capistrano, Fabric, Ansible, Chef, Puppet, SaltStack, etc.
- Debemos separar el concepto de hipervisor
  - Una o más máquinas virtuales se ejecutan virtualmente en el hardware físico por medio de una capa intermedia
  - Los contenedores se ejecutan en el espacio de usuario, por encima del kernel del sistema operativo.
  - Los contenedores permiten ejecutar distintas instancias en los espacios de usuario totalmente aislados

1.2. Docker

- Docker agrega un motor de despliegue de aplicaciones por encima de un entorno de ejecución de contenedores virtualizado.
- Docker es rápido, se pueden dockerizar aplicaciones en minutos
- Muchos contenedores tardan menos de un segundo en ejecutarse
- Los componentes que componen Docker
  - Servidor y cliente
  - Imagenes
  - Registros
  - Contenedores
- Docker es una aplicación cliente-servidor
- El cliente habla con el servidor o daemon, que realiza todas las tareas
- Docker viene con un comando *docker* y un api RESTful completo

Arquitectura Docker



1.3. Imágenes

- Docker almacena las imágenes en registros.
- Pueden ser públicos o privados
- Docker posee un registro público llamado DockerHub donde nos podemos registrar
- Podemos almacenar imágenes de Docker en privado
- Más adelante, aprenderemos a crear nuestro repositorio privado.
- Docker permite contruir y desplegar contenedores
- En los contenedores empaquetamos aplicaciones y servicios

1.4. Contenedores

- Un contenedor es:
  - Un formato de imagen
  - Un conjunto de operaciones
  - Un entorno de ejecución
- Cada contenedor posee una imagen de software, y permite que una serie de operaciones se ejecuten.
- Permite crear, iniciar, parar, reiniciar y destruir

1.5. Objetivos de Docker

- Aceleración de desarrollo y flujo de construcción rápido, eficiente y ligero
- Ejecución de servicios y aplicaciones consistente en múltiples entornos
- Util para micro-servicios y arquitecturas orientadas a servicios
- Creación de instancias aisladas para ejecución de pruebas como las realizadas por CI (Integración Continua) como Jenkins
- Construcción y pruebas de aplicaciones complejas y arquitecturas en un host local
- Entornos con sandboxing para desarrollo, pruebas, formación, etc.
- SAAS (Software As A Service) como Memcached
- Alto rendimiento

Chapter 2. Gestión de Docker

- Comprobamos primero que docker existe y está en funcionamiento

Estracto de salida

```
$ docker info
Containers: 0
Images: 0
Total Memory: 1.954 GiB
Name: moby
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): true
Registry: https://index.docker.io/v1/
```

2.1. New container

- Para crear el contenedor usaremos el comando run:
- El comando permite operar con todas las capacidades de ejecución del contenedor
- Creamos un nuevo contenedor:

```
$ docker run -i -t ubuntu /bin/bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu

952132ac251a: Pull complete
82659f8f1b76: Pull complete
c19118ca682d: Pull complete
8296858250fe: Pull complete
24e0251a0e2c: Pull complete
Digest: sha256:f4691c96e6bbaa99d99ebafd9af1b68ace2aa2128ae95a60369c506dd6e6f6ab
Status: Downloaded newer image for ubuntu:latest
root@5a034b4eb4cb:/#
```

- Analicemos el comando:

```
$ docker run -i -t ubuntu /bin/bash
```

- Con docker run, hemos pedido iniciar el contenedor
- con -i mantiene el STDIN abierto del contenedor. Es necesario para ejecutar shells interactivos.
- -t indica a Docker asignar un tty al contenedor que se crea.
- Por eso al final aparece un nuevo prompt asignado.
- Hemos creado un contenedor que nos muestra una línea de comandos del mismo.

- Con *ubuntu* hemos dicho que use del registro de **Docker Hub** la imagen *ubuntu*
- Por defecto, Docker busca imágenes en local, y luego en **Docker Hub** .
- Docker usa la imagen para crear el contenedor en el sistema de ficheros
- El contendor posee red, dirección ip, y una interfaz puente para hablar con localhost.
- Por último, */bin/bash* es el comando a ejecutar en el contenedor.
- Podemos observar ciertas características del nuevo contenedor
- Nombre

```
root@5a034b4eb4cb:/# hostname
5a034b4eb4cb
```

- O el fichero /etc/hosts

```
root@5a034b4eb4cb:/# cat /etc/hosts
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2  5a034b4eb4cb
```

- Docker posee una entrada para nuestro contenedor con su dirección ip

Estracto de salida

```
root@5a034b4eb4cb:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
```

NOTE	Se debe actualizar el contenedor: apt-get update Sino no puedes instalar las net-tools que son necesarias para poder lanzar comandos como ping, ifconfig u otros apt-get install net-tools

- Podemos comprobar cuales son los procesos ejecutandose en el contenedor

```
root@5a034b4eb4cb:/# ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.1  18240  3308 ?        Ss   10:05   0:00 /bin/bash
root        15  0.0  0.1  34424  2952 ?        R+   10:31   0:00 ps -aux
```

- Una vez que salgamos del prompt, el contenedor se parará.
- Podemos comprobar los contenedores en ejecución:

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                           PORTS
NAMES
5a034b4eb4cb        ubuntu              "/bin/bash"         36 minutes ago      Exited (0) About a minute ago
infallible_austin
```

- Por defecto, los contenedores poseen un nombre. En nuestro caso infallible_austin es el suyo. Equivalente a el container-id 5a034b4eb4cb (se puede abreviar , como 5a0 o 5a, mientras no haya otros)
- Podemos darle nuevo nombre al contenedor en su ejecución

```
$ docker run --name ubuntu-bash -i -t ubuntu /bin/bash
root@74382b322414:/# exit
```

- Si comprobamos el contenedor:

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                       PORTS
NAMES
74382b322414        ubuntu              "/bin/bash"         3 seconds ago       Exited (0) 1 seconds ago
ubuntu-bash
5a034b4eb4cb        ubuntu              "/bin/bash"         39 minutes ago      Exited (0) 4 minutes ago
infallible_austin
```

- Podemos iniciar manualmente el contenedor, en vez de crear uno nuevo cada vez y conectarnos a él

```
$docker start ubuntu-bash
$docker attach ubuntu-bash
root@74382b322414:/#
```

2.2. Creación de un nuevo contenedor como servicio

- Podemos crear contenedores daemon o como servicio.
- Son contenedores que no necesitan sesiones interactivas
- Usamos la opción -d para indicar que pase a segundo plano

```
$ docker run --name test_daemon -d ubuntu /bin/sh -c "while true; do echo hola mundo;
sleep 1; done"
fc90b1bd1312ac3c81cef3e0b603fe40c469dec86a8878579793480a6a731b33
```

- Podemos consultar la salida del contenedor:

```
$ docker logs -f test_daemon
hola mundo
hola mundo
hola mundo
```

- El comando log permite generar entradas con timestamp

```
$ docker logs -ft test_daemon
2016-09-01T09:23:51.700058936Z hola mundo
2016-09-01T09:23:52.707106869Z hola mundo
2016-09-01T09:23:53.708836849Z hola mundo
2016-09-01T09:23:54.715426920Z hola mundo
```

- Podemos analizar los procesos de un contenedor

```
$ docker top test_daemon
PID                 USER                TIME                COMMAND
3098                root                0:00                /bin/sh -c while true; do echo hola mundo; sleep 1; done
4007                root                0:00                sleep 1
```

- Para parar un contenedor podemos usar dos formas
  - SIGTERM - docker stop - Recomendado
  - SIGKILL - docker kill - En caso de que no se pare. Debemos ser conscientes de las consecuencias

2.3. Consulta de información de contenedores

- Un contenedor muestra su información a partir del atributo inspect

```
$ docker inspect test_daemon
[
    {
        "Id": "60d64f741ef6f754da49fd5bbfc6ec637f665ce32523a169e7444b1a1bcf7158",
        "Created": "2016-09-01T09:29:48.168378161Z",
        "Path": "/bin/sh",
        "Args": [
            "-c",
            "while true; do echo hola mundo; sleep 1; done"
        ],
...
```

- Podemos filtrar la información de inspección para localizar datos concretos

```
$ docker inspect --format='{{ .NetworkSettings.IPAddress}}' test_daemon
172.17.0.3
```

- Se trata de una plantilla GO

https://golang.org/pkg/text/template/

- Podemos pedir distinta información en el mismo comando

```
docker inspect --format='Running: {{ .State.Running }} Network:{{ .NetworkSettings.IPAddress}}' test_daemon
Running: true Network:172.17.0.3
```

- Podemos consultar distintos contenedores

```
$ docker inspect --format='Running: {{ .State.Running }} Network:{{ .NetworkSettings.IPAddress}}' test_daemon
otro_test_daemon
Running: true Network:172.17.0.3
Running: true Network:172.17.0.4
```

NOTE	Si se acumulan demasiados contenedores, es conveniente destruir todos los contenedores de un solo comando: # docker rm $(docker ps -a -q) -f

2.4. Gestión de imágenes

- Podemos buscar imágenes desde los repositorios de Docker Hub

```
# docker search alpine
NAME                            DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
alpine                          A minimal Docker image based on Alpine Lin...   1774      [OK]
anapsix/alpine-java             Oracle Java 8 (and 7) with GLIBC 2.23 over...   176                  [OK]
frolvlad/alpine-glibc           Alpine Docker image with glibc (~12MB)          58                   [OK]
container4armhf/armhf-alpine    Automatically built base images of Alpine ...   44                   [OK]
mhart/alpine-node-auto          Automated build of mhart/alpine-node — a...     35                   [OK]
kiasaki/alpine-postgres         PostgreSQL docker image based on Alpine Linux   29                   [OK]
zzrot/alpine-caddy              Caddy Server Docker Container running on A...   29                   [OK]
davidcaste/alpine-tomcat        Apache Tomcat 7/8 using Oracle Java 7/8 wi...   11                   [OK]
```

- La lista indica nombre, descripción, popularidad de la imagen, si es oficial y si se construye automáticamente por medio del Docker Hub
- Para bajar una imagen, lo indicamos de la siguiente forma:

```
$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
0a8490d0dfd3: Pull complete
Digest: sha256:dfbd4a3a8ebca874ebd2474f044a0b33600d4523d03b0df76e5c5986cb02d7e8
Status: Downloaded newer image for alpine:latest
```

- Creamos el contenedor y accedemos al bash para lanzar un comando de prueba

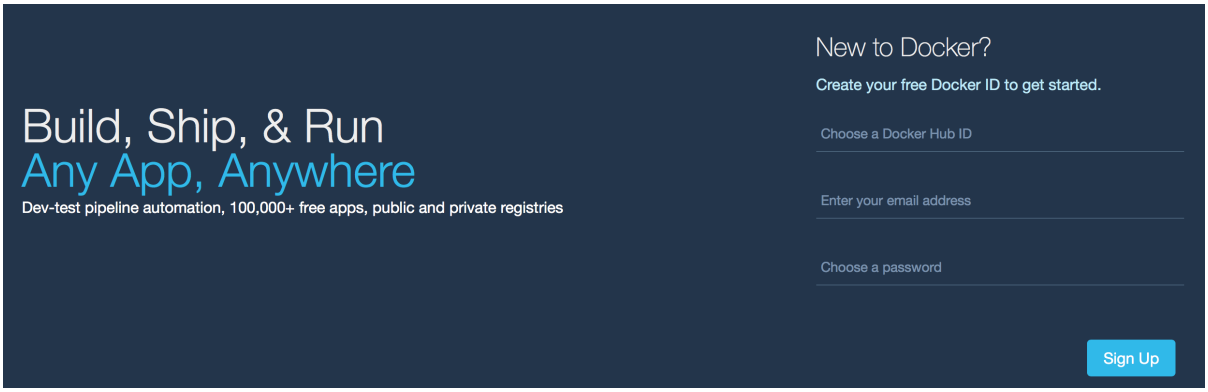
```
[vagrant@localhost ~]$ docker run -it alpine sh
/ # ls -la
total 24
drwxr-xr-x   18 root     root          4096 Jan 12 10:31 .
drwxr-xr-x   18 root     root          4096 Jan 12 10:31 ..
-rwxr-xr-x    1 root     root             0 Jan 12 10:31 .dockerenv
drwxr-xr-x    2 root     root          4096 Dec 26 21:32 bin
drwxr-xr-x    5 root     root           380 Jan 12 10:31 dev
drwxr-xr-x   14 root     root          4096 Jan 12 10:31 etc
drwxr-xr-x    2 root     root             6 Dec 26 21:32 home
drwxr-xr-x    5 root     root          4096 Dec 26 21:32 lib
drwxr-xr-x    5 root     root            41 Dec 26 21:32 media
drwxr-xr-x    2 root     root             6 Dec 26 21:32 mnt
dr-xr-xr-x  221 root     root             0 Jan 12 10:31 proc
drwx------    2 root     root            25 Jan 12 10:31 root
drwxr-xr-x    2 root     root             6 Dec 26 21:32 run
drwxr-xr-x    2 root     root          4096 Dec 26 21:32 sbin
drwxr-xr-x    2 root     root             6 Dec 26 21:32 srv
dr-xr-xr-x   13 root     root             0 Jan 12 10:31 sys
drwxrwxrwt    2 root     root             6 Dec 26 21:32 tmp
drwxr-xr-x    7 root     root            61 Dec 26 21:32 usr
drwxr-xr-x   12 root     root           115 Dec 26 21:32 var
```

- Podemos crear nuestras propias imágenes con contenidos personalizados
  - docker commit
  - docker build con un Dockerfile
- Docker commit no está recomendado para realizar nuevas imágenes

NOTE	Hay que crear nuevas cuentas de dockerhub para ello

- Debemos crear una cuenta en DockerHub para dar de alta los usuarios

Página oficial de Docker Hub



- Nos conectamos a través de la shell

```
$ docker login
Username: curso
Password:
Login Succeeded
```

2.4.1. commit

- Uso de docker commit
- Para ello vamos a crear e instalar software en un contenedor

```
$ docker run -it ubuntu /bin/bash
root@5b0dd54d36ce:/# apt-get -yqq update
root@5b0dd54d36ce:/# apt-get -yqq install apache2
```

- Hemos lanzado el contenedor a partir de la imagen de ubuntu
- Hemos actualizado e instalado apache
- Podemos salvar el estado actual para que no sea necesario actualizar e instalar el software
- Para ello usamos el comando *commit*

```
$ docker commit 5b0dd kane_project/webserver
sha256:2d9f67b3bc9d539f636a02e2fea34264aa2ecc7d5991b9db3a3dd00e447f76cb
```

- Comprobamos que el id del contenedor ha cambiado
- Docker commit solo guarda las diferencias entre la última imagen de contenedor y el estado actual.
- Las actualizaciones son muy ligeras.

```
$ docker images kane_project/webserver
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
kane_project/webserver  latest              2d9f67b3bc9d        2 minutes ago       264.8 MB
```

- Podemos agregar más información a la imagen
- Como el autor, una descripción y un tag a la imagen.
- Por defecto, el tag es latest

```
$ docker commit -m="Apache 2.4 Web Server" --author="Ruben Gomez" 5b0dd kane_project/webserver:apache24
sha256:733221df1d57da85d88ec16e6eeabb8c5afddd7dd3e564a4d1ffdd2190501bfd
```

- Podemos comprobar los detalles de la modificación

```
$ docker inspect kane_project/webserver:apache24
[
    {
        "Id":
"sha256:733221df1d57da85d88ec16e6eeabb8c5afddd7dd3e564a4d1ffdd2190501bfd",
        "RepoTags": [
            "kane_project/webserver:apache24"
        ],
        "Comment": "Apache 2.4 Web Server",
        "Author": "Ruben Gomez",
    }
]
```

2.4.2. Dockerfile

- La forma recomendada es Dockerfile, no commit
- Uso de un DSL con instrucciones de constucción para generar las imágenes Docker
- Uso del comando build para construir la imagen desde el fichero Dockerfile
- Vamos a crear nuestro primer Dockerfile
- Creamos un directorio con un fichero en su interior llamado Dockerfile

Contenido del Dockerfile

```
# Version: 0.0.1
FROM ubuntu:14.04
MAINTAINER Ruben Gomes "rgomez@pronoide.es"
RUN apt-get update
RUN apt-get install -y nginx
RUN echo '<marquee>Mira, un nuevo contenedor!<marquee/>' \
    >/usr/share/nginx/html/index.html
EXPOSE 80
```

- El fichero Dockerfile contiene una serie de instrucciones con argumentos
- Las instrucciones deben escribirse en mayúsculas, seguido de sus argumentos.
- Las instrucciones se ejecutan de forma secuencial

- Cada instrucción agrega una nueva capa y luego genera la imagen.
- Orden
  - Docker ejecuta el contenedor de la imagen seleccionada
  - Se ejecuta una instrucción que realiza cambios en el contenedor
  - Docker ejecuta el equivalente al docker commit
  - Docker ejecuta el nuevo contenedor de esa nueva imagen
  - se ejecuta otra nueva instrucción y vuelve a repetir el proceso hasta acabar
- Si por alguna razón, docker falla en alguno de los comandos, se puede acceder al último contenedor y explorar porqué no se realizó el proceso
- Si explicamos el contenido del fichero, la primera instrucción que no es un comentario, debe ser siempre FROM
- FROM especifica una imagen existente, donde el resto de comandos se van a ejecutar
- A esta imagen se la llama imagen base
- Nosotros hemos elegido la imagen Ubuntu 14.04
- Otra instrucción es MAINTAINER
- Permite indicar quien es el autor de la imagen y su correo.
- Después tenemos instrucciones de tipo RUN
- Primero actualizamos, luego instalamos nginx y por último creamos un fichero con contenido web
- Por defecto, el comando RUN se ejecuta como si fuera un comando /bin/sh
- Podemos ejecutarlo como un exec, evitando la cascada de información

```
RUN [ "apt-get","-y","update"]
```

- Por último, lanzamos el comando EXPOSE
- Permite ejecutar el contenedor con un puerto específico
- Esto no implica que el puerto esté expuesto.
- Docker no abre los puertos automáticamente, espera a que lo hagamos en el comando run
- Sirve para ayudar a enlazar otros contenedores
- Para construir el contenedor, lanzamos el comando build

```
$ docker build -t kane_project/nginx .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu:14.04
14.04: Pulling from library/ubuntu
862a3e9af0ae: Pull complete
...
7a1f7116d1e3: Pull complete
Status: Downloaded newer image for ubuntu:14.04
```

```
Step 2 : MAINTAINER Ruben Gomez Garcia "rgomez@pronoide.es"
 ---> Running in 7f81a83f3edd
 ---> 729d427a63a8
Removing intermediate container 7f81a83f3edd
Step 3 : RUN apt-get update
 ---> Running in b4af48374c0f
...
 ---> 14f0d2e23e33
Removing intermediate container b4af48374c0f
Step 4 : RUN apt-get install -y nginx
 ---> Running in cfd250677fe9
...
 ---> 4ce248f59373
```

```
Removing intermediate container cfd250677fe9
Step 5 : RUN echo '<marquee>Mira, un nuevo contenedor!<marquee/>'
>/usr/share/nginx/html/index.html
 ---> Running in e7be3d15d2b3
 ---> fa825b92fc9e
Removing intermediate container e7be3d15d2b3
Step 6 : EXPOSE 80
 ---> Running in 227690c0fe77
 ---> 9ec3061f6f7c
Removing intermediate container 227690c0fe77
Successfully built 9ec3061f6f7c
```

- Es recomendable usar tags para definir correctamente el estado de la imagen

```
$ docker build -t="kane_project/nginx:v2" github.com:kaneproject/docker-nginx
```

- Podemos apuntar a un repositorio de github para cargar el fichero.
- Podemos agregar un fichero .dockerignore para que no se suban al docker daemon si no son necesarios

```
$ docker build -t="kane_project/nginx:v1" github.com/kaneproject/docker-nginx
Sending build context to Docker daemon 53.76 kB
```

- Comprobemos que pasa si falla en una de las fases
- Para ello usamos un Dockerfile fallido:

```
# Version: 0.0.1
FROM ubuntu:14.04
MAINTAINER Ruben Gomes "rgomez@pronoide.es"
RUN apt-get update
RUN apt-get install -y ngin
RUN echo '<marquee>Mira, un nuevo contenedor!<marquee/>' \
    >/usr/share/nginx/html/index.html
EXPOSE 80
```

```
$ docker build -t kane_project/testing .
Sending build context to Docker daemon 53.76 kB
Step 1 : FROM ubuntu:14.04
 ---> 4a725d3b3b1c
Step 2 : MAINTAINER Ruben Gomez "rgomez@pronoide.es"
 ---> Using cache
 ---> 729d427a63a8
Step 3 : RUN apt-get update
 ---> Using cache
 ---> 14f0d2e23e33
Step 4 : RUN apt-get install -y ngin
 ---> Running in 8c012556c735
Reading package lists...
Building dependency tree...
Reading state information...
E: Unable to locate package ngin
The command '/bin/sh -c apt-get install -y ngin' returned a non-zero code: 100
```

- Podemos crear un contenedor de la imagen previa al fallo y lanzar el comando fallido

```
$ docker run -it 14f /bin/bash
root@d40119b72887:/# apt-get install ngin
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package ngin
```

- Cada vez que la imagen está construida, trata las imágenes de los pasos anteriores como caché
- Las imágenes construidas en los pasos anteriores implica que ya no se procesan

- Podemos forzar a que se realice de nuevo la construcción por medio de la opción --no-cache
- Un truco para refrescar la caché cuando queramos es agregando variables de entorno
- Al agregar una variable de entorno, si esta es modificada, la caché es ignorada automáticamente

```
# Version: 0.0.2
FROM ubuntu:14.04
MAINTAINER Ruben Gomes "rgomez@pronoide.es"
RUN apt-get update
RUN apt-get install -y nginx
RUN echo '<marquee>Mira, un nuevo contenedor!<marquee/>' \
    >/usr/share/nginx/html/index.html
EXPOSE 80
```

- Ahora creamos la imagen de nuevo

```
$ docker build -t kane_project/testing .
```

2.4.3. Resultados

- Podemos observar el estado de la nueva imagen

```
$ docker images kane_project/testing
REPOSITORY             TAG                 IMAGE ID            CREATED             SIZE
kane_project/testing   latest              30d96d3de199        4 hours ago         228.3 MB
```

- Podemos observar el histórico de la creación de la imagen

```
$ docker history kane_project/testing
IMAGE              CREATED            CREATED BY                                      SIZE                COMMENT
30d96d3de199       4 hours ago        /bin/sh -c #(nop)  EXPOSE 80/tcp                0 B
c06aa252ac7       4 hours ago        /bin/sh -c echo '<marquee>Mira, un nuevo cont   46 B
2c7607683901       4 hours ago        /bin/sh -c apt-get install -y nginx             18.15 MB
1442b9191433       4 hours ago        /bin/sh -c apt-get update                       22.16 MB
8503fa4e4d43       4 hours ago        /bin/sh -c #(nop)  ENV UPDATE_AT=2016-09-01     0 B
0485842b3a0e       4 hours ago        /bin/sh -c #(nop)  MAINTAINER Ruben Gomez "rg   0 B
4a725d3b3b1c       9 days ago         /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
<missing>          9 days ago         /bin/sh -c mkdir -p /run/systemd && echo 'doc   7 B
<missing>          9 days ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/   1.895 kB
<missing>          9 days ago         /bin/sh -c rm -rf /var/lib/apt/lists/*          0 B
<missing>          9 days ago         /bin/sh -c set -xe   && echo '#!/bin/sh' > /u   194.6 kB
<missing>          9 days ago         /bin/sh -c #(nop) ADD file:ada91758a31d8de3c7   187.8 MB
```

- Lanzamos el nuevo contenedor usando la nueva imagen

```
$ docker run -d -p 80 --name nginx kane_project/testing nginx -g "daemon off;"
fa77bf1776b5dd52fb7df43399959436ba6e512f5ef8c57ada7d2cb9fc8e2a4f
```

- Hemos lanzado la nueva imagen con la opción -d que permite lanzarlo en segundo plano
- Con la opción -p indicamos que puerto queremos exponer
  - Si le decimos que exponga el puerto en uno al azar
  - Podemos explicitarlo para decidir que puerto utilizar
- Comprobemos el puerto y nos indica que usa el puerto 32768

```
$ docker ps -l
CONTAINER ID        IMAGE                  COMMAND                  CREATED
STATUS              PORTS                   NAMES
fa77bf1776b5        kane_project/testing   "nginx -g 'daemon off"   2 minutes ago
Up 2 minutes        0.0.0.0:32768->80/tcp   nginx
```

- Otras opciones pueden ser:
  - -p 80:80 → Permite mapear al puerto 80 del host local. No recomendable
  - -p 8080:80 → Permite mapear el puerto 80 del contenedor al 8080 externo
  - -p 127.0.0.1:80:80 → Permite mapear el puerto 80 de la interfaz localhost de la máquina host
  - -p 127.0.0.1::80 → Permite mapear a un puerto al azar de localhost
  - -P mapea automáticamente todos los puertos expuestos
- Probamos el contenedor:

2.5. Instrucciones Dockerfile

- Existen una serie de instrucciones preparadas para construir un Dockerfile y dar mayor flexibilidad
- Algunos de ellos son: CMD, ENTRYPOINT,ADD,COPY,VOLUME,WORKDIR,USER,ONBUILD,ENV

2.5.1. CMD

- Permite ejecutar comandos tras la ejecución del contenedor.
- Similar a la instrucción RUN, pero este no se ejecuta en construcción
- Una equivalencia a docker run -it <imagen> /bin/sh

```
CMD ["/bin/sh"]
```

- Podemos concatenar argumentos

```
CMD ["/bin/bash","-l"]
```

- De esta manera, al terminar de ejecutar el contenedor ya no es necesario indicarlo

```
docker run -it kane_project/testing
```

- Si indicamos el comando a ejecutar, el comando CMD se sobreescribe y no se ejecuta

```
$ docker run -it kane_project/testing /bin/ps
  PID TTY          TIME CMD
    1 ?        00:00:00 ps
$
```

2.5.2. ENTRYPOINT

- Provee de un comando que no se puede sobreescribir como CMD
- De hecho, cualquier argumento especificado en Docker Run se hará sobre el ENTRYPOINT

```
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
CMD ["-h"]
```

- El comando ejecutará sobre el entrypoint
- Podemos modificar el entrypoint, pero explicitandolo con --entrypoint

```
$ docker run -it --entrypoint /bin/sh kane_project/testing
```

2.5.3. WORKDIR

- Provee una forma de establecer un directorio de trabajo
- Permite establecer comandos a ejecutar a partir del workdir

```
WORKDIR /usr
```

- Permite realizar distintas operaciones en distintos directorios durante la construcción
- Permite usar un directorio final de uso en el contenedor

```
docker run kane_project/testing ls -la
total 28
drwxr-xr-x 10 root root   97 Dec 15 17:44 .
drwxr-xr-x 21 root root 4096 Jan 13 12:41 ..
drwxr-xr-x  2 root root 8192 Jan 13 12:36 bin
drwxr-xr-x  2 root root    6 Apr 10  2014 games
drwxr-xr-x  2 root root   26 Dec 14 01:10 include
drwxr-xr-x 26 root root 4096 Dec 14 01:10 lib
drwxr-xr-x 10 root root  105 Dec 14 01:10 local
drwxr-xr-x  2 root root 4096 Jan 13 12:37 sbin
drwxr-xr-x 61 root root 4096 Jan 13 12:37 share
drwxr-xr-x  2 root root    6 Apr 10  2014 src
```

- Se puede cambiar el workdir por medio de la opción -w

```
docker run -w /root kane_project/testing ls -la
total 12
drwx------  2 root root   35 Dec 14 01:11 .
drwxr-xr-x 21 root root 4096 Jan 13 12:41 ..
-rw-r--r--  1 root root 3106 Feb 20  2014 .bashrc
-rw-r--r--  1 root root  140 Feb 20  2014 .profile
```

2.5.4. ENV

- Se utilizan para establecer variables de entorno en el contenedor

```
ENV USR_HOME /root
```

- La variable de entorno se usará en el siguiente comando
- Las variables de entorno serán persistentes en el contenedor

```
[vagrant@localhost mi_primer_dockerfile]$ docker run -it kane_project/testing bash
root@3c9629f61a2b:/# echo $USR_HOME
/root
```

- Con la opción -e se pueden definir variables de entorno

```
$ docker run -it -e USR_HOME=/Otro kane_project/testing bash
root@ec2fb2d5b5bb:/# echo $USR_HOME
/Otro
```

2.5.5. USER

- Especifica el usuario que debe ejecutar los comandos

```
RUN useradd nginx
USER nginx
```

- Todas las siguientes líneas se ejecutan con el usuario definido.

```
[vagrant@localhost mi_primer_dockerfile]$ docker run -it -e USR_HOME=/Otro kane_project/testing bash
nginx@08bd9cd0a85b:/$ whoami
nginx
```

2.5.6. VOLUME

- Agrega volúmenes a un contenedor creado de una imagen.
- Permite definir un directorio dentro de uno o más contenedores para datos compartidos persistentes
  - Pueden ser compartidos y reutilizados entre contenedores
  - Un contenedor no tiene porqué estar ejecutandose para compartir sus volumenes
  - Los cambios en el volumen son inmediatos
  - Los volumenes persisten hasta que ninguna imagen los utilice

```
VOLUME ["/data"]
```

- Es la misma instrucción que

```
docker run -v /data
```

- Para observar donde se encuentra el punto de montaje

SHELL1

```
docker run -it --name volume-data -v /mnt kane_project/testing
nginx@9a6321b474f8:/$
```

SHELL2

```
$ docker inspect -f "{{json .Mounts}}" volume-data2
[
  {
    "Name":"91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266",
    "Source":"/var/lib/docker/volumes/91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266/_data",
    "Destination":"/mnt",
    "Driver":"local",
    "Mode":"",
    "RW":true,
    "Propagation":""
  }
]
$ docker volume inspect 91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266
[
    {
        "Name": "91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266",
        "Driver": "local",
        "Mountpoint": "/var/lib/docker/volumes/91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266/_data",
        "Labels": null,
        "Scope": "local"
    }
]
```

- Podemos acceder a la unidad montada y escribir datos en ella, que se reflejarán en el volumen del contenedor

SHELL2

```
sudo touch /var/lib/docker/volumes/91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266/_data/prueba.txt
```

- Automáticamente aparece en el nuevo contenedor:

SHELL1

```
nginx@9a6321b474f8:/$ ls /mnt
prueba.txt
```

- Otra opción más común es la de definir el punto de montaje host y guest

```
$ docker run -it -v $PWD:/mnt kane_project/testing
nginx@e9751f87d080:/$ ls /mnt
Dockerfile
```

2.5.7. ADD

- Agrega ficheros y directorios de nuestro entorno de construcción a la imagen
- Sirve para instalar aplicaciones
- Especifica un origen y un destino

```
ADD software.lic /opt/application/software.lic
ADD http://wordpress.org/latest.zip /root/wordpress.zip
ADD latest.tar.gz /var/www/wordpress/
```

NOTE	Copia fichero Descarga Descomprime. El truco está en la barra del final.

2.5.8. COPY

- Igual que el add pero solo copia, no descomprime
- El directorio debe ser un path completo
- El UID y GID son 0
- Si el destino no está creado, es similar a mkdir -p

2.5.9. ONBUILD

- Agrega disparadores de las imágenes
- Se ejecuta cuando la imagen es base de otra imagen
- Permite ejecutar un script dependiente del entorno donde se ejecute

```
ONBUILD ADD . /miapp/src
ONBUILD RUN cd /miapp/src && make
```

- Por ejemplo, creamos una imagen base

```
FROM ubuntu:14.04
MAINTAINER Ruben Gomez "rgomez@pronoide.es"
RUN apt-get update
RUN apt-get install -y apache2
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ONBUILD ADD . /var/www/
EXPOSE 80
ENTRYPOINT ["/usr/sbin/apache2"]
CMD ["-DFOREGROUND"]
```

NOTE	docker run -it --entrypoint=/bin/bash kaneproject/webapp -s Para cambiar entrypoint y poder trabajar con la imagen

Chapter 3. Desarrollo con Docker

- Hemos aprendido a crear imagenes, lanzar y trabajar con contenedores
- Vamos a usar docker para
  - Probar un website estático
  - Construir y probar una aplicación web
  - Usar Docker para integración continua
- Los dos primeros casos se orientarán a desarrollo y pruebas individuales
- La tercera servirá como pruebas para un ciclo de vida de múltiples desarrolladores

3.1. WebSite estático

- Para un ejemplo de web estático, creamos un Dockerfile

Dockerfile

```
FROM ubuntu:14.04
MAINTAINER Ruben Gomez "rgomez@pronoide.es"
ENV REFRESHED_AT 2016-09-01
RUN apt-get update
RUN apt-get -y -q install nginx
RUN mkdir -p /var/www/html
ADD nginx/global.conf /etc/nginx/conf.d/
ADD nginx/nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
```

- Instala Nginx
- Crea un directorio para el website
- Agrega la configuración de Nginx
- Expone el puerto 80
- En la configuración obligamos a que el proceso se quede en foreground
- Construimos la imagen con los ficheros de apoyo

```
$ docker build -t cursodocker/nginx .
```

- Podemos observar todas las operaciones de la imagen realizadas

```
$ docker history cursodocker/nginx
```

- Desde el directorio donde está el Dockerfile, creamos directorios y fichero web

contenido del fichero website/index.html

```
<html><body><marquee><strong>Web Nginx</strong></marquee></body></html>
```

- Ahora conectamos el contenedor con una unidad externa

```
$ docker run -d -p 80 --name website -v $PWD/website:/var/www/html/website
cursodocker/nginx nginx
```

- Los volúmenes pueden compartirse entre unidades y montarse en modo lectura/escritura o solo lectura
- El directorio destino es creado por docker si es necesario

```
$ docker run -d -p 80 --name website -v $PWD/website:/var/www/html/website:ro
cursodocker/nginx nginx
```

- Ahora ya podemos ver el website desde nuestra máquina local buscando en localhost en el puerto externo compartido
- De hecho, ahora podemos modificar las páginas web y se verá el contenido nuevo sin necesidad de recargar

3.2. Aplicación Web

- En este caso , vamos a probar una aplicación web más compleja
- Aplicación basada en Sinatra
- Para ello, creamos otro contenedor con el siguiente Dockerfile

```
FROM ubuntu:16.04
MAINTAINER Ruben Gomez "rgomez@pronoide.es"
ENV REFRESHED_AT 2016-07-11

RUN apt-get -yqq update && apt-get -yqq install ruby ruby-dev build-essential redis-
tools
RUN gem install --no-rdoc --no-ri sinatra json redis

RUN mkdir -p /opt/webapp

EXPOSE 4567

CMD [ "/opt/webapp/bin/webapp" ]
```

- Se ha creado una imagen con ruby, con herramientas de Redis, Sinatra y Json
- Se expone el puerto por defecto del servidor WEBrick

- Se ejecuta el comando webapp
- Creamos la imagen

```
$ docker build -t cursodocker/sinatra .
```

- Tras descargar el código de aplicación en el directorio webapp, creamos el contenedor

```
$ docker run -d -p 4567 --name basewebapp -v $PWD:/opt/webapp cursodocker/sinatra
```

- Podemos ver que el proceso está ejecutándose correctamente

```
$ docker logs basewebapp
[2017-01-13 15:28:36] INFO  WEBrick 1.3.1
[2017-01-13 15:28:36] INFO  ruby 2.3.1 (2016-04-26) [x86_64-linux-gnu]
== Sinatra (v1.4.7) has taken the stage on 4567 for development with backup from WEBrick
[2017-01-13 15:28:36] INFO  WEBrick::HTTPServer#start: pid=1 port=4567
$ docker top basewebapp
UID                 PID                 PPID                C                   STIME               TTY
TIME                CMD
root                20077               20062               0                   16:28               ?
00:00:00            /usr/bin/ruby /opt/webapp/bin/webapp
```

- Podemos mostrar los puertos compartidos por el contenedor

```
$ docker port basewebapp 4567
```

- Para probar que la web es correcta, lanzamos el siguiente comando

```
curl -i -H 'Accept: application/json' -d 'nombre=Ruben&apellido=Gomez' http://localhost:32768/json
{"nombre":"Ruben","apellido":"Gomez"}
```

- Vamos a ampliar la imagen para que use otro contenedor
- Para ello, vamos a comunicar las aplicaciones por medio de un link o enlace.
- La imagen de Redis es la siguiente

```
FROM ubuntu:16.04
MAINTAINER Ruben Gomez "rgomez@pronoide.es"
ENV REFRESHED_AT 2016-09-01
RUN apt-get update
RUN apt-get -y install redis-server redis-tools
EXPOSE 6379
ENTRYPOINT ["/usr/bin/redis-server"]
```

- Construimos la imagen y la ejecutamos

- Observamos que no publicamos los puertos en ningún momento pero están expuestos

```
$ docker build -t cursodocker/redis .
$ docker run -d --name redis cursodocker/redis
```

- Para conectar al contenedor de Redis, no podemos aprovechar las direcciones ip
- Estas direcciones cambian según el numero de contenedores existentes.
- La mejor forma es por medio de linkado
- Para establecer el linkado, es necesario saber el nombre del contenedor
- Iniciamos un nuevo contenedor usando la plantilla redis de la aplicación web

```
docker run -p 4567 --name webapp --link redis:db -t -i -v $PWD/webapp-
redis:/opt/webapp cursodocker/sinatra /bin/bash
```

- Accediendo al prompt de la shell podemos ver que el argumento link permite
  - Establecer una relación padre-hijos
  - La relación de redis es db, en las máquinas aparecerá un recurso db para acceder a la dirección de redis
  - No es necesario exponer el puerto fuera del contenedor, solo lo ve las máquinas enlazadas
- Se puede configurar el demonio docker, para que no se admitan comunicaciones entre aplicaciones no enlazadas
- Podemos enlazar tantos contenedores como queramos

```
docker run -p 4567 --name webapp2 --link redis:db -t -i -v $PWD/webapp-
redis:/opt/webapp cursodocker/sinatra
docker run -p 4567 --name webapp3 --link redis:db -t -i -v $PWD/webapp-
redis:/opt/webapp cursodocker/sinatra
```

- podemos observar los cambios en dos puntos
- /etc/hosts donde aparece la referencia al enlace
- Podemos hacer ping a db
- Podemos comprobar las variables de entorno con el comando env

NOTE	Las imágenes por defecto tienen el software restringido Por eso se debe instalar lo estrictamente necesario apt-get install iputils-ping

- Si observamos ahora y realizamos la petición, esta se realizará desde redis

```
curl -i -H 'Accept: application/json' -d 'nombre=Ruben&apellido=Gomez'
http://localhost:32768/json
{"nombre":"Ruben","apellido":"Gomez"}
```

Chapter 4. Docker compose

- Se trata de una herramienta disponible en las docker tools.
- En los repositorios oficiales no está disponible todavía
- Permite definir aplicaciones docker multi-contenedor
- Por medio de un comando, permite crear e iniciar servicios de la configuración
- Perfecto para desarrollo y pruebas
- Se define por medio de un Dockerfile
- Se genera un fichero docker-compose
- Se inicia el servicio

Ejemplo de fichero docker-compose

```
version: '2' # indica la versión
services: # lista de servicios
  web: # nombre de servicio
    build: . # ruta de Dockerfile
    ports: # lista de puertos expuestos
    - "5000:5000"
    volumes: # lista de puntos de montaje
    - .:/code
    - logvolume01:/var/log
    links: # enlace al contenedor redis
    - redis
  redis: # nombre del servicio
    image: redis # nombre de imagen a utilizar
volumes:
  logvolume01: {}
```

4.1. Inicio de docker-compose

- La instalación más sencilla es por medio de la descarga del comando y asignación de permisos

```
$ sudo curl -L "https://github.com/docker/compose/releases/download/1.9.0/docker-compose-$(uname -s)-$(uname -m)" -o
/usr/local/bin/docker-compose
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   600    0   600    0     0    728      0 --:--:-- --:--:-- --:--:--   728
100 7857k  100 7857k    0     0  2131k      0  0:00:03  0:00:03 --:--:-- 3964k
$ sudo chmod +x /usr/local/bin/docker-compose
$ docker-compose -version
docker-compose version 1.9.0, build 2585387
```

4.2. Creación de imágenes

- Creamos una imagen para realizar una prueba de servidor con nodejs y express. .Dockerfile

```
FROM node

# Agregamos el directorio src
ADD src/ /src
# Accedemos al directorio src
WORKDIR /src
# Definimos las dependencias
RUN npm install
# Exponemos el puerto 80
EXPOSE 80
# Iniciamos el proces node
CMD ["node", "index.js"]
```

- Generamos el fichero de ejecución de nodejs para ejecutar en el contenedor de docker

src/index.js

```
var express = require('express');
var os = require("os");

var app = express();
var hostname = os.hostname();

app.get('/', function (req, res) {
  res.send('<html><body>ejemplo desde express con Node.js desde el contenedor ' +
hostname + '</body></html>');
});

app.listen(80);
console.log('Iniciado en http://localhost');
```

- Incluimos el fichero de empaquetado

src/package.json

```
{
  "name": "ejemplo-node-express",
  "private": true,
  "version": "0.0.1",
  "description": "Ejemplo de node y express con docker",
  "author": "rubengomez78@gmail.com",
  "dependencies": {
    "express": "4.12.0"
  }
}
```

- Creamos el contenedor y probamos que el servicio con express funciona correctamente

```
$ docker build -t cursodocker/express-example .
$ docker run -p 81:80 --name express -d cursodocker/express-example
```

- Ahora comprobemos el rendimiento de nuestra aplicación usando el potencial de docker y el Apache Bench

```
$ docker run --rm --network=host jordi/ab ab -n 10000 -c 10 http://localhost:81/
This is ApacheBench, Version 2.3 <$Revision: 1706008 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests


Server Software:
Server Hostname:        localhost
Server Port:            81

Document Path:          /
Document Length:        67 bytes

Concurrency Level:      10
Time taken for tests:   9.479 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      2470000 bytes
HTML transferred:       670000 bytes
Requests per second:    1054.95 [#/sec] (mean)
Time per request:       9.479 [ms] (mean)
Time per request:       0.948 [ms] (mean, across all concurrent requests)
Transfer rate:          254.47 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.3      0      12
Processing:     2    9   2.9      9      43
Waiting:        1    9   2.8      8      43
Total:          2    9   2.9      9      43
```

```
Percentage of the requests served within a certain time (ms)
  50%      9
  66%      9
  75%     10
  80%     10
  90%     12
  95%     13
  98%     16
  99%     22
 100%     43 (longest request)
```

- Estos son los datos que nos muestra la aplicación: 1054,95 peticiones por segundo.
- Aprovechemos docker compose para desplegar un cluster de servidores web y repitamos las pruebas
- Creamos un HAProxy para que haga de balanceador entre los contenedores
- Usamos la misma imagen oficialde HAProxy, por medio de el repositorio haproxy y tag alpine
- Definimos el fichero haproxy.cfg que redirige las peticiones entre los clientes

haproxy/haproxy.cfg

```
global
  log 127.0.0.1 local0
  log 127.0.0.1 local1 notice

defaults
  log global
  mode http
  option httplog
  option dontlognull
  timeout connect 5000
  timeout client 10000
  timeout server 10000

frontend balancer
  bind 0.0.0.0:80
  mode http
  default_backend aj_backends

backend aj_backends
  mode http
  option forwardfor
  # http-request set-header X-Forwarded-Port %[dst_port]
  balance roundrobin
  server express1 express1:80 check
  server express2 express2:80 check
  server express3 express3:80 check
  # option httpchk OPTIONS * HTTP/1.1\r\nHost:\ localhost
  option httpchk GET /
  http-check expect status 200
```

- Generamos ahora el fichero .yml que pertenece a docker compose:

docker-compose.yml

```
express1:
  build: .
  expose:
    - 80

express2:
  build: .
  expose:
    - 80

express3:
  build: .
  expose:
    - 80

haproxy:
  image: haproxy:alpine
  volumes:
    - ./haproxy/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg:ro
  links:
    - express1
    - express2
    - express3
  ports:
    - "80:80"
    - "70:70"
  expose:
    - "80"
    - "70"
```

- Ahora levantamos el servicio con docker compose

```
$ docker-compose up
Starting dockercompose_express2_1
Starting dockercompose_express1_1
Starting dockercompose_express3_1
Creating dockercompose_haproxy_1
Attaching to dockercompose_express2_1, dockercompose_express3_1,
dockercompose_express1_1, dockercompose_haproxy_1
express2_1  | Iniciado en http://localhost
express3_1  | Iniciado en http://localhost
express1_1  | Iniciado en http://localhost
haproxy_1   | <7>haproxy-systemd-wrapper: executing /usr/local/sbin/haproxy -p
/run/haproxy.pid -f /usr/local/etc/haproxy/haproxy.cfg -Ds
```

- Ahora podemos realizar peticiones contra el puerto 80 y comprobar como redirige a los distintos contenedores

```
$ curl localhost
<html><body>ejemplo desde express con Node.js desde el contenedor 22468f2328a6</body></html>
$ curl localhost
<html><body>ejemplo desde express con Node.js desde el contenedor 2f9d169e7f6b</body></html>
$ curl localhost
<html><body>ejemplo desde express con Node.js desde el contenedor 07431198daef</body></html>
$ curl localhost
<html><body>ejemplo desde express con Node.js desde el contenedor 22468f2328a6</body></html>
```

- Podemos realizar de nuevo la prueba del cluster de servidores, aunque en el caso de virtualización no ofrecerá los resultados deseados. No siempre por tener más es mejor!

```
$ docker run --rm --network=host jordi/ab ab -n 10000 -c 10 http://localhost/
This is ApacheBench, Version 2.3 <$Revision: 1706008 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests


Server Software:
Server Hostname:        localhost
Server Port:            80

Document Path:          /
Document Length:        92 bytes

Concurrency Level:      10
Time taken for tests:   12.528 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      2730000 bytes
HTML transferred:       920000 bytes
Requests per second:    798.21 [#/sec] (mean)
Time per request:       12.528 [ms] (mean)
Time per request:       1.253 [ms] (mean, across all concurrent requests)
Transfer rate:          212.80 [Kbytes/sec] received
```

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.4      0      19
Processing:     2   12   3.9     12      84
Waiting:        2   12   3.8     11      84
Total:          2   12   4.0     12      85

Percentage of the requests served within a certain time (ms)
  50%     12
  66%     13
  75%     14
  80%     14
  90%     15
  95%     18
  98%     22
  99%     25
 100%     85 (longest request)
```

4.3. Ventajas e inconvenientes de docker-compose

- Por defecto, genera una interfaz de red única por aplicación
- No usa el link entre contenedores, sino la interfaz
- No usa network de tipo overlay, y no despliega en distintos nodos en swarm
- Si se configura un puerto fijo en swarm, los contenedores repetidos no iniciarán al usar todos el mismo puerto

4.4. Ejemplos en version 3

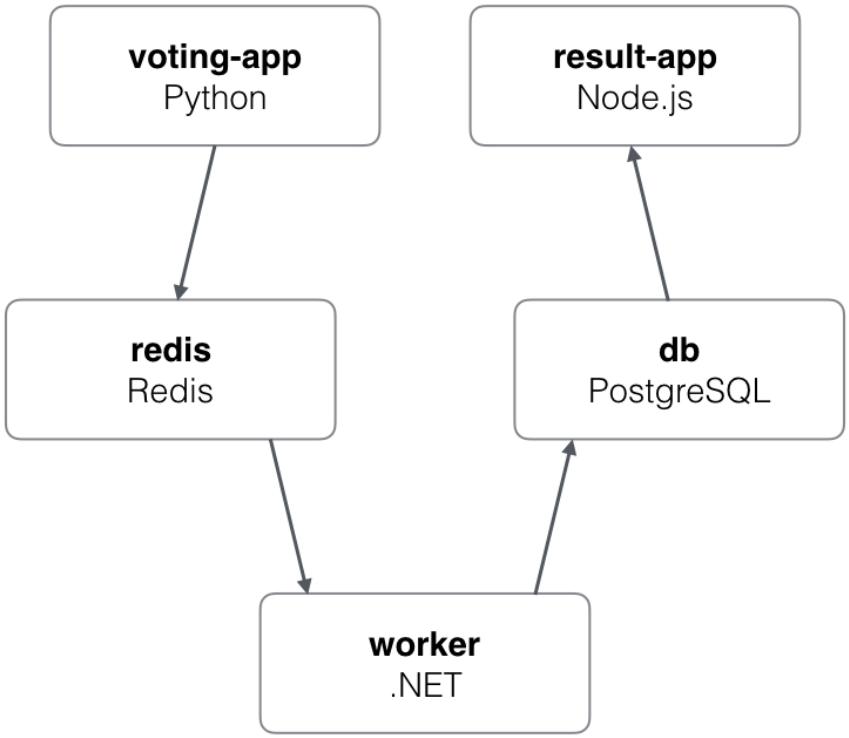
```yaml
version: "3"

services:
  vote:
    build: ./vote
    command: python app.py
    volumes:
     - ./vote:/app
    ports:
      - "5000:80"
    networks:
      - front-tier
      - back-tier

  result:
    build: ./result
    command: nodemon --debug server.js
    volumes:
      - ./result:/app
```

```
    ports:
      - "5001:80"
      - "5858:5858"
    networks:
      - front-tier
      - back-tier

  worker:
    build:
      context: ./worker
    networks:
      - back-tier

  redis:
    image: redis:alpine
    container_name: redis
    ports: ["6379"]
    networks:
      - back-tier

  db:
    image: postgres:9.4
    container_name: db
    volumes:
      - "db-data:/var/lib/postgresql/data"
    networks:
      - back-tier

volumes:
  db-data:

networks:
  front-tier:
  back-tier:
```

- Para iniciar el proyecto, podemos bajarlo de
  - https://github.com/dockersamples/example-voting-app

```
$ git clone https://github.com/dockersamples/example-voting-app
```

- La arquitectura de la aplicación es:



- Podemos ver los resultados de la app en :
  - http://localhost:5000 (app)
  - http://localhost:5001 (resultados)

NOTE	Si no inicia por fallo del comando mv de la imagen de nodejs, editar el Dockerfile y usar: RUN cp -R node_modules /