Dockerfile

fichero de scripting para generar un contenedor

```
FROM debian:latest
RUN apt-get update && apt-get install -y wget
```

`docker build -t <nombre> .`: genera una nueva imagen con repositorio <tag>

> `vagrant rsync` : sincroniza el directorio «compartido» entre el host y la imagen vagrant También se puede lanzar un proceso vagrant rsync-auto que se encarga de ir haciendo este proceso automaticamente

instrucciones

- FROM : imagen-plantilla
- ENV : variables de entorno (modificar PATH)
- ARG : variables de Dockerfile (no en el contenedor)
- COPY : copiar ficheros en la imagen
- ADD
- RUN : ejecutar comandos
- VOLUMEN
- EXPOSE
- ENTRYPOINT
- CMD
- USER
- WORKDIR = cd

FROM

imagen de referencia

- `FROM <image>`
- `FROM <image:tag>`
- `FROM <@id_imagen>`

RUN

ejecuta comandos en el contenedor (linux)

CMD

`CMD [«echo»,«hello world»]`: comando que se ejecuta por defecto

> al separar las líneas, el primer RUN es un layer y quedará cacheado (y usado en otras imágenes), lo que hará que nuestro sistema no se actualice. Por eso la línea RUN del update incluye la instalación de soft

```
FROM debian
RUN apt-get update
RUN apt-get install -y nginx
CMD ["nginx","-g","daemon off;"]
```

OPTIMIZAR

lo que se tenga que borrar, se tiene que hacer en la misma layer (RUN)

```
FROM debian
RUN apt-get update \
    && apt-get install -y nginx \
    && rm -rf /var/lib/apt/lists/*
CMD ["nginx","-g","daemon off;"]
```

COPY

- `COPY file /path/to/file`
- `COPY [–chown=<user||uid>:<group||gid>] file /path/to/file`
- `COPY file file2 file3 /tmp/`
- `COPY fil* /tmp/`
- `COPY . /tmp/` : no usar *

> UNION FILE SYSTEM - relación con layers https://en.wikipedia.org/wiki/UnionFS

LAB2

```
FROM debian:latest
RUN apt update && apt install -y nginx
CMD ["nginx","-g","daemon off;"]
COPY index.html /var/www/html/index.html
```

importante que el COPY esté por debajo del RUN (LAYERS!!!)

volumes

- `docker run -v $PWD:/var/www/html/ <imagen>` : mapea $PWD al directorio indicado dentro del contenedor

    - `docker run -v $(pwd):/var/www/html <imagen>` : uso «alternativo» de **pwd**
- `docker exec -it <contenedor_id> bash` : acceder al contenedor

los volúmenes se usan para persistencia, ya que un contenedor cuando se para, todo lo que hubiese se pierde.

label

- `label key=value`: etiquetas a las imágenes (para posterior filtrado)

arg

- `arg key=value`: variables de uso interno

env

- `env key=value`: variables de entorno en el contenedor (para nuestra app)

add

- `add <origen> <destino>`: copia un fichero o una URL en el destino
    - es un copy on steroids
    - también descomprime un tgz o similar directamente
    - CREA UNA LAYER (como RUN y COPY)

expose

- `expose 80/tcp`: informativo (de cara a quien trabaje con esa imagen y a docker para trabajar con el `-P` (mapeo automático de puertos))

workdir

- `workdir /app`: cd al directorio

imágenes alpine

- **nginx:latest** VS **nginx:1.15-alpine**
    - las versiones **alpine** están basadas en la imagen **alpine**, muy ligera
        - usan **sh** y no **bash**
    - también se usa **slim** para hacer entender que es una versión ligera

user

- USER `<user||uid>`: cambiar de usuario (buena práctica usar UID)
- RUN `useradd -u 1001 -m -g root dani`
    - en **openshift** siempre ha de estar el usuario añadido al grupo de **root**

    - ```
      #!/bin/bash fix-permissions.sh
      for arg in "$@"
      do
      ```
      ```

```
    find "$arg" -exec chgrp 0 {}\;
    find "$arg" -exec chmod g+rw {}\;
    find "$arg" -type d -exec chmod g+x {}\;
done
```

si extendemos de una imagen que ha cambiado el USER y nosotros hemos de realizar cambios, hemos de hacer el cambio a USER root, hacemos lo que necesitemos y lo volvemos a dejar con el usuario que toque.

context

- `docker build -t <imagen> .` : el **.** es el contexto donde buscar todos los ficheros a los que hacemos referencia, incluido **Dockerfile** al demonio docker
  - para evitar el problema del **Dockerfile**, podemos pasarle el parámetro `-f <ubicación_Dockerfile>`
  - para hacerlo más límpio, usar el **.** siempre y todo referenciado desde ahí

bbdd

bien para desarrollo, no para producción

es un cuello de botella, ya que para mantener la persistencia hay que montar un volumen por NFS (red), lo que hace que sea lento

los DBA ullen de las máquinas virtuales, quieren servidores físicos, así que de docker...

volumes

- `VOLUME /myvol`: informamos a Docker que ahí habrá un volumen (implica persistencia)
  - funcionamiento parecido a EXPOSE y -P
  - si no especificamos nosotros con `-v` se gestiona «internamente»
    - `docker volume ls`
    - `docker volume inspect <hash_volumen>`
    - `docker inspect <contenedor_id>`
  - FROM ... RUN mkdir /app RUN echo "..."

```
 /app/hello
VOLUME /app
RUN echo "..." > /app/hello2
```

entrypoint

- es lo primero que se ejecuta (antes que CMD)
- `ENTRYPOINT [«echo»]`
  - a este comando le llega por parámetro lo que viene desde CMD
- `ENTRYPOINT [«/entrypoint.sh»]`
  - ejecución en tiempo de ejecución

- ejecuta todo lo que recibe a través del CMD:

```
#! /bin/bash
exec "$@"
```

los entrypoint y los cmd hay que hacerlo a la manera JSON [«comando»,«comando»] para evitar ejecuciones extrañas

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT ["exec_entry", "p1_entry"]
No CMD	*error, not allowed*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd", "p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD ["p1_cmd", "p2_cmd"]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

proxy-pass

configuración de apache para que al recibir una petición delegue en otros contenedores

imagen creada de docker donde se le pasa por variable de entorno el destino al que ha delegar, ayudado por el `entrypoint`

.dockerignore

permite expresiones regulares, como **.gitignore** y metes todos aquellos ficheros que no han de formar parte de la imagen