Virtualización III

Miguel Ortuño
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

Noviembre de 2024

Contenidos

Plataformas para ejecuta docker

Docker tiene arquitectura cliente servidor

- El cliente acepta la entrada del usuario, le muestra la salida, maneja los ficheros con los que preparar las imágenes
- El servidor ejecuta el contenedor

El servidor solo está disponible para Linux 64 bits
Hay versiones para macOS y Microsoft Windows, donde el cliente se ejecuta en nativo contra un servidor dentro de una máquina virtual

- Esta virtualización es transparente para el usuario

Docker dentro de una máquina virtual

Si vamos a ejecutar docker dentro de una máquina virtual,

- guest y host deben tener arquitectura 64 bits
- Es necesario que el host tenga soporte para Intel VT-x
  - Los equipos muy antiguos o muy baratos no lo permiten (VT-x es del año 2006, pero no se generaliza en los equipos de gama básica/media hasta varios años después)
  - Muchos equipos actuales tienen esta opción deshabilitada por omisión en la BIOS/UEFI

Algunos conceptos

Imágenes:

- La imagen de un contenedor (o simplemente *imagen*) es un fichero en el sistema de ficheros del host
- Un contenedor se ejecuta a partir de una imagen

Esto es análogo a un proceso que se ejecuta a partir de un fichero

Manejaremos diversos identificadores, que no debemos confundir

- Nombre de la imagen. Ejemplos:
  debian
  test/c01
- Identificador de la imagen. Ejemplo:
  cc8393a39248
- Nombre de contenedor. Si no lo indicamos explícitamente, docker usará nombres aleatorios como *focused_yonath* o *wonderfuld_goldberg*
- Identificador de contenedor
  Ejemplo: 18009dd9f349
- Nombre de host
  Nombre de máquina que se percibirá dentro del contenedor. (variable de entorno `$HOST`, nombre en el *prompt*, fichero `/etc/hostname`, etc)
  Atención: Este *host* de Docker se corresponde con lo que en VirtualBox sería el *guest*

Nombres de imagen

El nombre de la imagen

- Un nombre sin prefijo, por ejemplo *debian* indica una imagen oficial aprobada por docker
- Un nombre con prefijo, por ejemplo *test/c01* es una imagen no oficial. El prefijo puede ser una etiqueta que hayamos definido o un nombre de usuario en un registro de imágenes

Instalación de docker

- Podemos instalar el paquete incluido en nuestra distribución de ubuntu

```
apt update; apt upgrade -y ; apt install docker.io
```

- Si por algún motivo ese paquete resulta antiguo y necesitamos la última versión estable disponible de docker, ejecutamos el script disponible en `https://get.docker.com`

```
wget https://get.docker.com -O get-docker.sh  #letra "O" mayúscula
bash get-docker.sh
```

Lanzar una imagen

Para ejecutar docker, tenemos dos opciones

- Añadir nuestro usuario al grupo docker

```
addgroup docker
adduser $USER docker
# (abrir una nueva sesión)
```

- Ejecutar docker con sudo

Para comprobar que la instalación ha sido correcta, lanzamos una imagen sencilla

```
docker run debian echo "hola,mundo"
```

Esto busca en el *registry* oficial de docker una imagen llamada *debian*, ejecuta en ella la orden indicada, muestra su salida por stdout y concluye

Otro holamundo

```
koji@mazinger:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
5b0f327be733: Pull complete
Digest: sha256:1f19634d26995c320618d94e6f29c09c6589d5df3c063287a00e6de8458f8242
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash
```

Servidor docker remoto

- En la configuración más sencilla, el servidor de docker está en la misma máquina que el cliente, el ejecutable incluye ambas funciones
- Pero también puede ubicarse en una máquina remota. Esto es útil, por ejemplo
  - Cuando el cliente no es linux 64 bits
  - Cuando el cliente no tiene privilegios de root en la máquina local
  - Para arquitecturas distribuidas, equilibrio de carga, en la nube, etc

Configuración del servidor remoto

- Es necesario el paquete `docker.io`
- El usuario tiene que poder entrar por ssh en la máquina remota
- Debería poder autenticarse por ssh sin escribir la contraseña cada vez (lo contrario sería muy incómodo)
- El usuario debe pertenecer al grupo `docker`

Configuración del cliente

- Es necesario el paquete `docker.io`
- El usuario necesita la variable de entorno

```
DOCKER_HOST="ssh://jperez@servidor_remoto"
```

Recuerda que:

- Puedes crear la variable de entorno en `~/.bashrc`. Pero los cambios no son inmediatos, es necesario una nueva sesión o leer el fichero explícitamente con `source`
- Puedes comprobar las variable de entorno con `env`

Podman sobre NFS

Para usar Podman sobre NFS, como en el laboratorio, es necesario configurar el sistema para que los volúmenes vaya a un disco local, p.e. `/tmp/MILOGIN`
Para ello, creamos un fichero
`~/.config/containers/storage.conf`
con el siguiente contenido:

```
[storage]
  driver = "vfs"
  runroot = "/run/user/NNNN"
  graphroot = "/tmp/MILOGIN/.local/share/containers/storage"
  [storage.options]
    ignore_chown_errors = "true"
    mount_program = ""
```

- Reemplazando `MILOGIN` por nuestro nombre de usuario
- Reemplazando `NNNN` por nuestro uid[1]

[1] que podemos conocer con el comando id

Contenedores dentro de máquina virtual VirtualBox / Vagrant

- Un desarrollador trabajando en su propio portatil, puede crear y lanzar contenedores con tranquilidad
- En un entorno en producción, como el del laboratorio, esto no es conveniente
  - Hay muchos alumnos de muchas asignaturas
  - Está basado en NFS
- En estos casos, los normal es lanzar los contenedores dentro de una máquina virtual
  En esta asignatura, dentro de una máquina VirtualBox lanzada con Vagrant

Recuerda que si en el laboratorio tenemos
`~/lagrs/vbox01`
Y desde allí lanzamos una máquina virtual de VirtualBox con Vagrant

- Dentro de la máquina vagrant el directorio `/vagrant`
- Se corresponderá con el directorio del laboratorio `~/lagrs/vbox01`

Repositorio de imágenes

Además de guardarse localmente, las imágenes están disponibles en los *registry*

- Registro (Registry)
  Servicio responsable de almacenar y distribuir imágenes. El registro por omisión es `https://hub.docker.com`
  Aunque hay otros similares, públicos. Y quien lo desee puede establecer su propio registro
- Repositorio (Repository)
  Una colección de imágenes relacionadas, normalmente ofrecen diferentes versiones de la misma aplicación o servicio
- Etiqueta (Tag)
  Identificador alfanumérico asociado a una única imagen

Docker run

Esta instrucción lanza un contenedor a partir de una imagen

`docker run <opciones> <imagen>`

- Observa que las opciones deben escribirse antes del nombre de imagen. De lo contrario, docker las ignora
- La imagen se puede identificar mediante su nombre o mediante su id
- Las opciones `-i` y `-t` normalmente se usan juntas, para indicar que queremos una sesión interactiva en un terminal
- `--name <nombre_contenedor>`
- `-h <nombre_host>`
  `--hostname=<nombre_host>`

Ejemplo

```
docker run -it --name c01 -h c01 test/im01
```

Consulta de imágenes y contenedores

- `docker ps`
  Muestra los contenedores
- `docker images`
  Muestra las imágenes
- `docker inspect <imagen>`
  Muestra un json con descripción detallada del contenedor
- `docker diff <imagen>`
  Muestra los cambios en el sistema de ficheros del contenedor
- `docker logs <imagen>`
  Muestra las instrucciones ejecutadas en el contenedor

Exited containers

Cuando un contenedor finaliza su ejecución, queda en estado *exited*, al que informalmente se suele llama *parado*

- `docker ps -a`
  Muestra los contenedores, incluyendo los parados
- `docker rm <contenedor>`
  Borra un contenedor
- `docker rmi <imagen>`
  Borra una imagen

Si el contenedor se lanza con la opción `--rm`, se borrará automáticamente al concluir

Borrado de imágenes y contenedores

- Borrar todas las imágenes (que no estén siendo usadas)
  `docker rmi $(docker images -a -q)`
- Borrar todos los contenedores detenidos
  `docker rm $(docker ps -a -f status=exited -q)`
- Borrar todos los contenedores creados (y nunca ejecutados)
  `docker rm $(docker ps -a -f status=created -q)`

Creación de imágenes

La orden `docker build` nos permite construir imágenes.
Para construir una imagen, normalmente usaremos tres cosas:

- Un directorio contexto, que será un directorio vacio en nuestra máquina, donde iremos añadiendo los ficheros necesario para construir la imagen
- Un fichero `Dockerfile` dentro del directorio contexto, con las instrucciones para crear la imagen
- Un fichero `entrypoint.sh`, que será un script de shell que
  - Crearemos en el directorio contexto
  - Llevaremos a la imagen
  - Se ejecutará cada vez que se lance un contenedor con esa imagen

- Si la imagen es muy sencilla, puede que no necesite `entrypoint.sh`
  Ejemplo:

```
FROM ubuntu:24.04
RUN apt update && apt upgrade -y
ENTRYPOINT /bin/bash
```

- También es posible crear una imagen sin usar un fichero Dockerfile
  Para ello basta con
  1. Entrar en el contenedor
  2. Configurarlos: instalar paquetes, añadir ficheros, modificar ficheros...
  3. `docker commit <CONTENEDOR> <IMAGEN>`
     `<CONTENEDOR>`: Nombre o id del contenedor que será punto de partida da la imagen
     `<IMAGEN>`: Nombre que tendrá la imagen

Ejemplo: banner

Vamos a crear una imagen llamada *test/banner* basada en la orden *banner* que al ejecutarse mostrará los siguiente:

```
koji@mazinger:~/lagrs/banner$ docker run -h c01 --name c01 test/banner
```

```
#####      #    ######  #    #  #    #  ######  #    #     #    #####    ####
#    #     #    #       ##   #  #    #  #       ##   #     #    #    #  #    #
#####      #    #####   # #  #  #    #  #####   # #  #     #    #    #  #    #
#    #     #    #       #  # #  #    #  #       #  # #     #    #    #  #    #
#    #     #    #       #   ##   #  #   #       #   ##     #    #    #  #    #
#####      #    ######  #    #    ##    ######  #    #     #    #####    ####
```

```
  ##
 #  #
#    #
######
#    #
#    #
```

```
         ###      #
 ####   #   #    ##
#    # # # #   #  # #
#      #  #  #    #
#      #   # #    #
#    #  #   #     #
 ####    ###    #####
```

Creamos un *directorio contexto* en el host, y en él escribimos un fichero `entrypoint.sh`

```
#!/bin/bash
banner bienvenido
banner a
banner $HOSTNAME
```

Cuando sea posible, es muy conveniente probar este script antes de construir la imagen, los errores aquí son uno de los problemas más habituales preparando contenedores

En el directorio contexto también creamos un fichero `Dockerfile`

```
FROM ubuntu:24.04
RUN apt update && apt upgrade -y && apt install -y sysvbanner
COPY entrypoint.sh /
ENTRYPOINT ["/entrypoint.sh"]
```

- La instrucción FROM indica la imagen de partida
- La instrucción RUN indica las modificaciones a realizar en la imagen
  Puede haber más de un RUN, pero eso crea imágenes intermedias, por lo que lo habitual es encadenar varias órdenes de shell con `&&`
- La opción `-y` en
  `apt upgrade`
  `apt install`
  es imprescindible (contesta a todas las preguntas con *yes*)

- La instrucción `COPY origen destino` es muy simipar al comando `cp origen destino` de la shell, pero copia un fichero desde el directorio contexto (que está en el host) hasta el sistema de ficheros del futuro contenedor que se ejecute a partir de la imagen
- El origen puede ser un fichero, o un directorio (a diferencia de `cp`, no es necesario indicar `-r` para copia recursiva)
- El origen siempre estará dentro del diretorio contexto, no se permite copiar ficheros desde otro sitio[2]
- Normalmente el destino lo indicaremos con un trayecto absoluto[3]

[2] y un nombre de fichero absoluto se entiende desde el directorio contexto

[3] si es relativo, se entiende que cuelga del WORKDIR, que por omisión es el raiz

- La instrucción ENTRYPOINT especifica el fichero que se ejecutará al iniciar cada contenedor
  Es habitual llamarlo `entrypoint.sh` y colocarlo en el directorio raiz del contenedor
- En el Dockerfile se pueden crear comentarios con el caracter #
- El contenido del Dockerfile es *case insensitive*, aunque el convenio es usar mayúsculas para las instrucciones
- Si no existe un fichero `Dockerfile`, docker busca un fichero `dockerfile`

Una vez preparados los ficheros, construimos la imagen

- Desde el directorio padre del directorio contexto ejecutamos `docker build -t test/banner directorio_contexto`

Recuerda que los nombres de las imágenes que crearemos siempre llevarán prefijo (puesto que no son imágenes oficiales)
Almacenamiento de la configuración:

- La configuración de docker se guarda en /var/lib/docker
- Las imágenes, depende del driver que docker use para el almacenamiento. Por omisión se usa aufs, que guarda las imágenes en `/var/lib/docker/aufs`

La sentencia COPY

- Con mucha frecuencia querremos tener cierto fichero dentro de la imagen. En el Dockerfile escribiremos algo como

```
COPY  ejemplo /home/jperez
```

  Esto es, indicamos nombre del fichero (en el directorio contexto) y directorio destino (en la imagen). El nombre del fichero no cambia.

- Otras veces será conveniente que el nombre del fichero en el directorio contexto sea distinto al nombre en la imagen. Típicamente si se trata de un fichero de configuración oculto

```
COPY  ejemplo_config  /home/jperez/.ejemplo_config
```

  Esto es, el primer argumento de COPY será el nombre del fichero, no oculto, y el segundo argumento, el trayecto completo del fichero *en su sitio*, con el nombre precedido por un punto para que esté oculto.

El fichero entrypoint

- El fichero *entrypoint* normalmente será un script de shell, con el nombre `entrypoint.sh`. Aunque podría ser cualquier otro fichero que indiquemos en la instruccion `ENTRYPOINT` del Dockefile
- Como cualquier script de shell
  - Necesita permiso de ejecución
  - Su primera línea debe ser exactamente
    `#!/bin/bash`
    Y ninguna otra cosa
    ```
    #! /bin/bash   # MAL
    !#/bin/bash    # MAL
    /bin/bash    # MAL
    ```

- Si queremos que el usuario trabaje de manera interactiva dentro del terminal, añadimos una llamada a la shell dentro del entrypoint

```
#!/bin/bash
/bin/bash
```

  Recuerda que, además, es necesario añadir las opciones `-it` en `docker run`

- Naturalmente, si modificas el entrypoint (o cualquier otro fichero del directorio contexto) es necesario volver a construir la imagen para que los cambios tengan efecto

- Para que se ejecute algo en el terminal *antes* de que el usuario empieze a trabajar, lo añadimos en el entrypoint *antes* de la llamada a la shell

```
#!/bin/bash
echo Esto se ejecuta ANTES
/bin/bash
echo Esto se ejecuta cuando concluye la sesión
echo justo antes de finalizar el contenedor
```

Normalmente querremos que `/bin/bash` sea la última línea del entrypoint

Gestión de datos en docker

El sistema de ficheros interior al contenedor es volátil

- Todo lo escrito durante la ejecución del contenedor se pierde al borrar el contenedor
- Es complicado acceder a esos datos sin usar el mismo contenedor

Podríamos guardar datos en una nueva capa creando una nueva imagen, pero sería poco práctico, no es recomendable

Un contenedor no debería tener estado. O en su defecto, el mínimo estado posible
Docker ofrece 3 mecanismos para la persistencia de datos

- Bind mounts
- Volumes
- tmpfs

Por supuesto, dentro del contenedor se puede usar cualquier otro protocolo o servicio no específico de Docker: NFS, sshfs, SMB, rsync, almacenamiento en la nube, bases de datos relacionales, bases de datos no relacionales...

Bind mounts

Un *bind mount* es un directorio del host que se comparte con uno (o varios) contenedores

- Muy eficientes
- Muy prácticos para compartir datos con el host
- Dependen del sistema de ficheros del host y de su estructura, con lo que tienen problemas de portabilidad
- Evidentes problemas potenciales de seguridad, al tener el contenedor acceso directo al sistema de ficheros del host

Para hacer un bind mount, basta añadir los siguientes parámetros a la orden `docker run`

- Sintaxis tradicional

  `-v <DIR_ORIGEN>:<DIR_DESTINO>`

- Sintaxis moderna, disponible a partir de Docker 17.06

  `--mount type=bind,source=<DIR_ORIGEN>,target=<DIR_DESTINO>`

- `DIR_ORIGEN` es el directorio en el host
- `DIR_DESTINO` es el directorio en el contenedor
  - En el montaje de ficheros tradicional en Unix, es necesario que exista el punto de montaje. Aquí, no
- Ambos directorios deben estar especificados con path absoluto
- No puede haber espacios antes ni después de la coma

Suponiendo que los nombres de usuario coincidan en el host y en el contenedor, podríamos hacer por ejemplo

```
docker run -it -h jperbind01  --name jperbind01 --rm \
   -v $HOME:/home/$USER  \
   jperez/bind
```

Es necesario prestar mucha atención a los montajes bind, son potencialmente peligrosos. El usuario que accede al sistema de ficheros fichero en el servidor de contenedores es el mismo que en el contenedor

Ejemplos

- Un proceso que sea root en el contenedor, también puede acceder como root al servidor. Por eso es tan delicado que un usuario pueda lanzar un contenedor
- Supongamos el *home* de un usuario del servidor configurado para que solo él tenga acceso
  - Para que un usuario del contenedor pueda acceder a este directorio con un montaje bind, el usuario dentro del docker tiene que tener el mismo id que el usuario en el servidor (no importa el nombre, solo el id)
  - Sucede lo mismo con el gid: el gid del usuario dentro del docker será el gid que vea el servidor

Una vez más: los montajes bind son potencialmente peligrosos

Diferencia importante:

- En las máquinas virtuales *tradicionales* (p.e. hipervisores) es extremadamente difícil que un proceso del *guest* consiga *escaparse* y acceder al *host*
- En los contenedores, muy fácil

Volumen

Es un disco virtual creado y gestionado por docker.
Se puede almacenar

- Como subdirectorio del host (en linux, por omisión en `/var/lib/docker/volumes`)
  Aunque no se recomienda que el host acceda directamente al volumen
- En host remotos o en la nube, Docker ofrece para ello diferentes drivers

Características:

- Son más fáciles de transportar y respaldar que los bind mounts
- Tienen mejores prestaciones para ser compartidos entre varios contenedores
- Se pueden cifrar

tmpfs

Un montaje de tipo *tmpfs* se usa para datos temporales

- Es un sistema de ficheros especialmente eficiente porque se almacena en RAM
- Si creamos una imagen a partir del contenedor, el contenido de los montajes tmpfs no se almacena

Uso de sshfs

Como hemos visto, los bind mounts permiten montar dentro de un contenedor directorios ubicados en el host docker

- Para montar directorios en cualquier otro lugar de internet, podemos usar por ejemplo sshfs
- Para ello es necesario añadir a la orden `docker run` los siguientes parámetros
  - En docker 17.10
    `--privileged`
  - En versiones más modernas de docker
    `--cap-add SYS_ADMIN --device /dev/fuse`
    `--security-opt apparmor:unconfined`

  Para averiguar tu versión de docker: `docker --version`
  - En un entorno de producción habría que usar estas opciones con precaución, puesto que incremeta mucho los privilegios del contenedor dentro del host

docker hub

Para subir nuestras imágenes al registro docker hub

1. Creamos una cuenta en `hub.docker.com`
2. Creamos nuestras imágenes usando como prefijo nuestro login en dockerhub
   `docker build -t mi_usuario/mi_imagen directorio_context`
3. Abrimos una sesión en docker hub desde la shell con la orden
   `docker login`
4. Subimos la imagen
   `docker push mi_usuario/mi_imagen`

Usuarios dentro del contenedor

La orden para crear usuario en Unix/Linux es *adduser*

- Solicita de forma interactiva contraseña, nombre, etc
- Para construir una imagen con *docker build*, usamos otra orden distinta, *useradd*, que no hace preguntas sino que permite introducir la información mediante opciones

En el Dockerfile añadimos

```
RUN useradd -rm -d /home/jperez -s /bin/bash -u 1001 jperez
```

- `-rm`
  Cuenta de sistema, con directorio *home*
- `-d`
  Directorio *home*
- `-s`
  Especifica la shell
- `-u`
  Especifica uid

Si queremos que el usuario pueda ejecutar `sudo`

- Instalamos el paquete *sudo*
- Asignamos al usuario el grupo primario *root*, y el grupo adicional *sudo*, añadiendo a `useradd` las opciones

  `-g root -G sudo`

Si queremos que al poner en marcha el contenedor con una sesión interactiva sea este el usuario, añadimos al final del *entrypoint*

```
su jperez
/bin/bash
```

Atención: todos las líneas del entrypoint después de `su` serán ejecutadas por este usuario, no por el root

Con este entrypoint tendremos el usuario ejecutando una shell sin necesidad de escribir contraseña, pero si queremos que tenga contraseña, añadimos al *Dockerfile*

```
RUN echo 'jperez:sesamo' | chpasswd
```

Configuración de red

Al instalar docker se crean 3 redes

- bridge
  Segmento privado dentro del host, `172.17.0.0/16`, al que se conectan por omisión todos los contenedores
- null
  Red nula, aisla los contenedores de la red
- host
  El contenedor comparte la red con el host, mismos interfaces, direcciones y puertos

```
koji@mazinger:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
787cf305d42c        bridge              bridge              local
256d470b6133        host                host                local
086e801223bb        none                null                local
```

- Para conectar un contenedor a una red, basta lanzarlo con `--network=<nombre_red>`
  Ejemplo

```
docker run -it -h c03 --name c03 --rm  --network=host test/im03
```

- Para crear una nueva red (un nuevo segmento), ejecutamos en la shell, directamente o en un script:

```
docker network create --subnet 192.168.12.1/24  mired
```

Servidor de SSH en el contenedor

Para un contenedor en producción, no es recomendable habilitar el demonio de ssh

- Implica tener un segundo proceso, que no es natural en Docker
- No es buena idea dejar contraseñas dentro de un contenedor ¿cómo actualizarlas?
- El código dentro del contenedor es responsabilidad del equipo de desarrollo. Pero el acceso y las políticas, compete a explotación

Sin embargo, en esta asignatura sí configuraremos un servidor de ssh dentro de un contenedor, porque el objetivo es enseñar cómo funciona el acceso por ssh, que es lo habitual en máquinas físicas y máquinas virtuales tradicionales

¿Es necesario acceder por ssh?

- Para actualizar el sistema
  No. El contenedor entonces tendría estado (las actualizaciones). Lo recomendable es crear un nuevo contenedor con la actualización.
- Para ver logs
  No. El contenedor tendría estado. Lo recomendable es llevar los logs a un volumen
- Para iniciar y detener demonios
  No. Se pueden enviar señales
- Para editar la configuración
  No. Lo recomendable es crear un nuevo contenedor
- Para depurar el servicio
  No. Se puede abrir una shell desde el servidor de contenedores

Si a pesar de esto queremos instalar sshd en un contenedor: Dockerfile

```
FROM ubuntu:24.04
RUN apt update && apt install -y openssh-server

# Con sshd, ENV no funciona. Para fijar una variable de entorno:
# RUN echo "export MI_VARIABLE=mivalor" >> /etc/profile

RUN service ssh start

COPY entrypoint.sh /

EXPOSE 22
ENTRYPOINT ["/entrypoint.sh"]
```

entrypoint.sh

```
#!/bin/bash
/usr/sbin/sshd
```

- La instrucción EXPOSE indica en qué puerto (TCP) atiende peticiones el contenedor
- Realmente esta instrucción no hace nada, es solo un *mensaje* del autor del contenedor para quien vaya a usar el contenedor

Configuración en español

Las imágenes base de las distribuciones son esqueletos mínimos, normalmente tendremos que personalizarlas
Por ejemplo, para configurar el idioma. En nuestro caso, español.
Instalaremos en la imagen el paquete `locales`, invocaremos a localedef con los parámetros adecuados y definiremos la variable de entorno LANG

```
FROM ubuntu:24.04
RUN apt update && apt upgrade -y && \
  apt install -y  locales  && \
  localedef -i es_ES -c -f UTF-8  \
  -A /usr/share/locale/locale.alias es_ES.UTF-8
ENV LANG es_ES.UTF-8
COPY entrypoint.sh /
ENTRYPOINT ["/entrypoint.sh"]
```

- La instrucción ENV del Dockerfile define variables de entorno dentro de la imagen

Lo habitual es usar una única instrucción RUN en cada Dockerfile, para evitar las imágenes intermedias.
Pero también podemos usar varias instrucciones, para que resulte más legible.
Ejemplo:

```
FROM ubuntu:24.04
RUN apt update && apt upgrade -y
RUN apt install -y  locales
RUN localedef -i es_ES -c -f UTF-8  \
    -A /usr/share/locale/locale.alias es_ES.UTF-8
ENV LANG es_ES.UTF-8
COPY entrypoint.sh /
ENTRYPOINT ["/entrypoint.sh"]
```

Observa que

- El slash invertido al final de línea (\) une dos líneas físicas en una misma línea lógica
- El doble ampersand (&&) separa sentencias dentro de la misma instrucción RUN

Sesiones gráficas

En un contenedor podemos lanzar aplicaciones gráficas

- Si el cliente y el servidor de Docker están en la misma máquina y ambas son Unix, podemos usar *X11 Forwarding*

```
docker run -ti --rm \
       -e DISPLAY=$DISPLAY \
       -v /tmp/.X11-unix:/tmp/.X11-unix \
       mi_imagen
```

- Una solución más general es VNC
  Aquí se describe:
  `http://gsyc.urjc.es/~mortuno/vnc.pdf`

Troubleshooting / Resolución de problemas (1)

Si tu contenedor / contenedores no funciona como debe, repasa lo siguiente

- ¿Puedes ejecutar un contenedor *holamundo*?
- ¿El usuario que lanza el contenedor pertenece al grupo *docker*?
- ¿Has probado los scripts fuera del contenedor? ¿Tienen los permisos adecuados?
- ¿Has escrito todas las opciones antes del argumento principal (la imagen) ?

```
docker run -it imagen   # ok
docker run imagen -it   # MAL
```

Troubleshooting / Resolución de problemas (2)

- Si la tarea exige privilegios de root ¿eres root o usas sudo correctamente?
- ¿Tienes claro en qué máquina estás? (máquina física, máquina virtual, contenedor). ¿Tienes claro qué ficheros van en cada cual?
- ¿Tienes claro qué va en el Dockerfile y qué en el entrypoint?
- Cuando copias un fichero desde el directorio contexto hasta la imagen del contenedor ¿lo haces correctamente? Puedes revisarlo de forma interactiva, esto es, desde una shell en el contenedor, comprobar que el fichero está en el lugar adecuado dentro del contenedor

Troubleshooting / Resolución de problemas (3)

- Un inconveniente de los contenedores es que no vemos los errores que puedan generar nuestros scripts.
  Una solución es abrir una sesión interactiva dentro del contenedor e ir ejecutando línea a línea los scripts.
  Empezando por el entrypoint. Trabajando de forma interactiva sí veremos los mensajes de error.