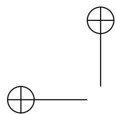
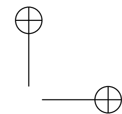



Anexo C

Docker

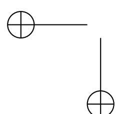
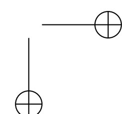
A lo largo del libro se han incluido varios ejemplos que requieren cierta infraestructura y que hemos llamado 'laboratorios'. Para implementarlos hemos utilizado `docker`, una herramienta que permite ejecutar aplicaciones dentro de contenedores. Un contenedor es un entorno de ejecución aislado, separado del SO anfitrión que tiene su propio sistema de archivos, configuración de red y procesos. Esto nos permite simular varios nodos que se comunican entre sí, obviamente muy útil cuando hablamos de redes, aunque no sea ese su propósito principal.

Al empezar a utilizar `docker` puede recordar bastante a sistemas de VM (Virtual Machine) como `VirtualBox` o `VMware`, sin embargo, hay diferencias importantes:

- El contenedor `docker` no emula el hardware de un computador, ni ejecuta un SO completo, sino que utiliza los programas sobre el SO del anfitrión. No permite ejecutar un SO Windows sobre un SO GNU/Linux o viceversa, ni siquiera de una versión diferente del núcleo.
- A diferencia de las VM, un contenedor `docker` está pensado para ejecutar un único proceso. Cuando el proceso termina, el contenedor también. Esto hace que la imagen del contenedor sea normalmente mucho más pequeña que la de una VM.
- Los contenedores son efímeros, es decir, no tienen estado. Cualquier cambio (escritura) que se aplique al sistema de archivos del contenedor se pierde al pararlo, aunque es posible montar volúmenes externos persistentes.

C.1. Imágenes

Para ejecutar un contenedor `docker` necesitas una imagen. La imagen la puedes obtener de un repositorio de imágenes o *hub* (como `https://hub.docker.com`) o bien la puedes crearla tu mismo a partir de otra.

La prueba habitual para comprobar que tienes una instalación correcta y funcional es ejecutar un contenedor con la imagen `hello-world`, que siendo la primera vez, conllevará la descarga automática de dicha imagen.

```
~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
17eec7bbc9d7: Pull complete
Digest: sha256:1d983076c7facf19a9a53a7a4855dbd736bf5206e3bfc5142a4c1da4ed44dedc
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.
    (amd64)
 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

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

La **línea 2** dice que la imagen solicitada no está en tu computador, por lo que que procede a descargarla. Las **líneas 3-6** muestran información sobre la descarga. Finalmente, a partir de la **línea 7** aparece la salida del programa `hello` que es un programa incluido en la imagen y que se ejecuta dentro del contenedor.

Cualquier imagen descargada se almacena es tu computador y se puede reutilizar. Puedes comprobarlo con el siguiente comando:

```
~$ docker images
REPOSITORY      TAG           IMAGE ID       CREATED         SIZE
hello-world     latest        1b44b5a3e06a   6 days ago      10.1kB
```

C.2. Contenedores

El contenedor representa la ejecución del programa definido en una imagen, y de hecho puedes crear varios contenedores sobre la misma imagen. Puedes probar un contenedor que ejecuta un servidor web (`nginx`). De hecho eso, es lo habitual: `docker` se suele usar para ejecutar un servidor o un servicio que queda en ejecución indefinidamente.

```
~$ docker run -d -p 8000:80 -v .:/data nginx
fc085a4e035a6d07d0ab22ed6d03887d99a0ad3c6ff84cc221ee6b4eb09dbac5
```

Los argumentos de ese comando son los más frecuentes:

- `-d`: ejecuta el contenedor en segundo plano. La secuencia alfanumérica que ves en este caso es el identificador del contenedor.
- `-p 8000:80`: consigue que el puerto TCP 80 del contenedor esté accesible a través del 8000 de tu computador y con ello dar acceso al servicio desde el exterior. Puedes probar a acceder a ese servidor web cargando `http://localhost:8000` en tu navegador y verás la página de bienvenida de `nginx`.
- `-v .:/data`: monta el directorio actual (filename.) de tu computador en el directorio `/data` del contenedor. Eso da al contenedor acceso a los archivos de ese directorio y además le otorga persistencia.
- `nginx`: es el nombre de la imagen, de forma análoga al `hello-world` del ejemplo anterior.

Puedes comprobar el estado de los contenedores y sus procesos asociados con `docker ps` o `docker container ls`, aunque este segundo comando muestra también los contenedores parados.

```
~$ docker container ls
CONTAINER ID   IMAGE   COMMAND                  PORTS                  NAMES
fc085a4e035a   nginx   "/docker-entrypoint.…"   0.0.0.0:8000->80/tcp   hardcore_cannon
```

En esta sencilla información de estado del contenedor aparece el comando que se ejecuta en el interior del contenedor (llamado menudo *entrypoint*), el puerto mapeado 8000->80 y el nombre del contenedor (`hardcore_cannon`) que se genera aleatoriamente y no debes confundir con el nombre de la imagen.

El contenedor seguirá en ejecución indefinidamente hasta que lo detengas u ocurra un error grave. En este situación `docker` te permite ejecutar otros comandos del contenedor con la orden `docker exec`. Por ejemplo, esto ejecuta una shell interactiva (gracias a la opción `-ti`) dentro del contenedor:

```
~$ docker exec -ti hardcore_cannon bash
root@1ad721a249fe:/#
```

Por último puedes parar un contenedor activo con:

```
~$ docker stop hardcore_cannon
hardcore_cannon
```

C.3. Dockerfile

Si quieres crear una imagen propia, o mejor dicho, personalizar una imagen existente puedes crear un archivo `Dockerfile`. Por ejemplo, supongamos que necesitas tener Python disponible en el contenedor anterior. Puedes crear un `Dockerfile` con lo siguiente:

```
FROM nginx
RUN apt-get update && apt-get install -y python3
EXPOSE 80
```

La primera línea es la imagen base, la segunda es la ejecución de dos comandos para instalar el paquete `python3` y la tercera informa de que el contenedor expondrá el puerto 80. Para poder usar esta imagen, primero debes construirla:

```
~$ docker build -t nginx-python .
[+] Building 11.3s (6/6)FINISHED                                          docker:default
 => [internal] load build definition from Dockerfile                               0.0s
 => => transferring dockerfile: 107B                                               0.0s
 => [internal] load metadata for docker.io/library/nginx:latest                    0.0s
 => [internal] load .dockerignore                                                  0.0s
 => => transferring context: 2B                                                    0.0s
 => [1/2] FROM docker.io/library/nginx:latest                                      0.1s
 => [2/2] RUN apt-get update && apt-get install -y python3                        10.2s
 => exporting to image                                                             0.9s
 => => exporting layers                                                            0.8s
 => => writing image sha256:b1141380ef52995902f0c568a82feb4fcfe122d8a...           0.0s
 => => naming to docker.io/library/nginx-python                                    0.0s
```

El parámetro `-t` asigna un nombre a la imagen y el punto final se refiere al directorio actual, que es donde debe buscar el archivo `Dockerfile`. Ahora la nueva imagen aparecerá en la lista:

```
~$ docker images
nginx-python            latest        b1141380ef52   2 minutes ago    248MB
nginx                   latest        ad5708199ec7   44 hours ago     192MB
hello-world             latest        1b44b5a3e06a   6 days ago       10.1kB
```

Y con esto puede arrancar un contenedor con la imagen recién creada:

```
~$ docker run -d -p 8000:80 nginx-python
```

C.4. Docker Compose

Cuando se necesitan varios contenedores que deben cooperar o comunicarse entre sí, es mucho más cómodo utilizar `docker-compose`. Este programa leer un archivo llamado `docker-compose.yml` que indica la configuración de varios contenedores y sus relaciones.

```
services:
  nginx:
    image: nginx
    container_name: web-server
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html:ro
    depends_on:
      - db

  db:
    image: postgres
    container_name: db
    environment:
      POSTGRES_USER: usuario
      POSTGRES_PASSWORD: password
      POSTGRES_DB: storage
    ports:
      - "5432:5432"
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```

Este archivo define dos servicios (contenedores). Veámoslos en detalle:

- `nginx` es el servidor web, equivalente al del ejemplo de la sección anterior.
  - `ports` define redirecciones de puertos como el parámetro `-p` en el comando `run`.
  - `volumes` define un punto de montaje, como la opción `-p` aunque es este caso de indica que es solo lectura (`ro`), de modo que el contenedor no podrá modificar el contenido de este directorio.
  - `depends_on` dice que el contenedor `db` debería arrancar antes que este.
- `db` ejecuta la base de datos PostgreSQL (versión 15).
  - `environment` define variables de entorno. En este caso se define el nombre de la base de datos y el nombre de usuario y clave con el que se accede. Las variables de entorno son la forma más habitual de configurar servicios `docker`.
  - `volumes` en este caso especifica un volumen gestionado por `docker`. Fíjate que la parte antes del : no es una ruta. Es un identificador que aparece al final en la sección `volumes`.

Para arrancar los contenedores basta con ejecutar el siguiente comando en el directorio dónde está `docker-compose.yml`:

```
~$ docker-compose up -d
[+] Running 4/4
 - Network compose-example_default   Created                                              0.0s
 - Volume "compose-example_db_data"  Created                                              0.0s
 - Container db                      Started                                              0.3s
 - Container web-server              Started                                              0.4s
```

Además, `docker` crea automáticamente una red interna con direccionamiento privado a la que conecta todos los contenedores. El nombre de la red se basa en el directorio que contiene el archivo `docker-compose.yml`. Puedes obtener información de las redes disponibles con:

```
~$ docker network ls
NETWORK ID     NAME                      DRIVER    SCOPE
32ef38990d8e   bridge                    bridge    local
dc3b83ef6011   compose-example_default   bridge    local
059b37dd8ae6   host                      host      local
030f578f0ff0   none                      null      local
```

Y puedes ver detalles sobre esa red y las direcciones asignadas a los contenedores:

```
~$ docker network inspect compose-example_default
[...]
        "IPAM": {"Config": [{
            "Subnet": "172.19.0.0/16",
            "Gateway": "172.19.0.1"
         }]},
[...]
        "Containers": {
            "3748abed03d58138129ee0f17d6f1a5a9a4df22584fef184fd1d7968f8c7aeed": {
                "Name": "db",
                "MacAddress": "c2:d7:a3:24:c0:b5",
                "IPv4Address": "172.19.0.2/16",
            },
            "5e6cb60767f0b5e00ba8e60eec9692dd073da56826af27ea3cc8db659d3151e6": {
                "Name": "web-server",
                "MacAddress": "86:47:2d:6b:5a:20",
                "IPv4Address": "172.19.0.3/16",
            }
        },
[...]
```