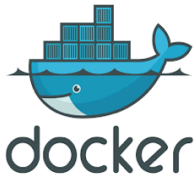
Desplegament d'aplicacions amb Docker



En este document explicarem com començar a treballar amb Docker, un gestor de contenidors open source, que automatitza el desplegament d'aplicacions en estos contenidors. Un **contenidor** és una peça de programari amb un sistema de fitxers complet, que conté tot el necessari per a poder funcionar. Els contenidors servixen per a distribuir i desplegar aplicacions, de manera portable, estandarditzada i autocontinguda (sense necessitat d'altres dependències externes). Per a ser més concrets, Docker permet gestionar contenidors Linux, permetent crear diferents entorns virtuals i executar aplicacions en ells.

1. Què és Docker?

Podríem veure Docker a priori com una alternativa a altres ferramentes de virtualització, com VirtualBox o VMWare, però existixen uns certs avantatges i diferències entre una cosa i una altra, com veurem a continuació.

1.1. Avantatges d'usar Docker enfront de màquines virtuals

Entre els principals avantatges que oferix Docker enfront d'utilitzar una màquina virtual convencional, podem citar les següents:

- Es té un **ús lleuger de recursos**, al no ser necessari instal·lar cap sistema operatiu *guest* complet sobre el sistema amfitrió (*host*). Això és possible gràcies a que, en gestionar únicament contenidors Linux, es poden aïllar les característiques bàsiques de dita kernel, compartides per diverses distribucions (Ubuntu, Xarxa Hat, CentOS...).
- És **open source i multiplataforma**, pot instal·lar-se sobre amfitrions Windows, Mac US X i fins i tot alguns en el núvol com AWS o Azure, encara que és preferible que este amfitrió siga una distribució Linux per a prendre el kernel d'ell. En estos apunts suposarem que utilitzarem Docker sobre una distribució Linux Debian.
- És **portable**, ja que totes les dependències de les aplicacions s'empaqueten en el propi contenidor, podent portar-lo i executar-lo en qualsevol *host*.
- La manera de **comunicar** contenidors entre si és més senzilla que la comunicació entre màquines virtuals. Mentres que per a comunicar màquines virtuals complexes necessitem tindre habilitats ports entre elles, en Docker s'establixen ponts de comunicació (*bridges*), que fan el procés molt més automàtic.

Així, per exemple, podríem tindre un ordinador amb un sistema Debian instal·lat, i sobre ell, mitjançant Docker, tindre un contenidor corrent CentOS, un altre corrent RedHat, o el mateix Debian, entre altres opcions. Però no caldrà instal·lar físicament ni CentOS, ni RedHat ni cap altra distribució com a màquina virtual. D'esta manera, la grandària de les imatges que es generen per als contenidors és molt més reduït que el d'una màquina virtual convencional.

1.2. Per a què s'utilitza Docker?

El principal ús de Docker es dona en arquitectures orientades a servicis, on cada contenidor oferix un servici o aplicació, facilitant així la seua escalabilitat. Per exemple, podem tindre un contenidor oferint un servidor Nginx, un altre amb un servidor MariaDB/MySQL, un altre amb MongoDB, i establir comunicacions entre ells per a compartir informació.

Qui utilitza Docker?

Actualment, Docker s'utilitza en algunes empreses o webs de rellevància, com Ebay o Spotify. A més, té per darrere el suport d'empreses importants, com Amazon o Google.

1.3. Principals elements de Docker

Abans de començar amb la instal·lació i primeres proves amb Docker, vegem quins elements principals ho componen.

El motor de Docker (*Docker engine*)

Esta ferramenta ens permetrà crear, executar, detindre o esborrar contenidors. Necessita d'un kernel de Linux per a funcionar, i internament es compon d'una aplicació client i d'un dimoni. Amb la primera, i mitjançant comandos, podem comunicar-nos amb el segon per a enviar ordes concretes (crear contenidor, posar-ho en marxa, etc.).

L'arxiu *Dockerfile*

Per a emmagatzemar la configuració i posada en marxa d'un determinat contenidor, s'utilitza un arxiu de text anomenat *Dockerfile*, encara que, com veurem, també es poden crear i llançar contenidors sense este arxiu. En ell s'inclouen diferents instruccions per a, entre altres coses:

- Determinar el sistema operatiu sobre el qual es basarà el contenidor
- Instal·lar i posar en marxa aplicacions
- Definir unes certes variables d'entorn
- ... etc.

Ací podem veure un exemple molt senzill d'arxiu Dockerfile:

```
FROM ubuntu:latest
MAINTAINER Juan Pérez <juan.perez@gmail.com>
RUN apt-get update
RUN apt-get install -i apache2
ADD 000-default.conf /etc/apache2/sites-available/
RUN chown root:root /etc/apache2/sites-available/000-default.conf
EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Analitzant amb una mica de detall les seues línies, podem veure que:

- La primera línia especifica que el sistema operatiu a emprar per al contenidor serà Ubuntu. La versió s'indica després dels dos punts, i en este cas és l'última, però podríem indicar alguna concreta pel seu nom de pila, com per exemple "trusty" (versió 14) o "precise" (versió 12).
- La segona línia al·ludix al creador de l'arxiu Dockerfile i el seu contacte, en cas de tindre algun dubte o problema amb este.
- Les següents dos línies actualitzen els repositoris i instal·len Apatxe, respectivament. Es poden posar comandos en línies separades, o enllaçats en una sola línia.
- Les següents dos línies afigen el virtual host per defecte a la carpeta de llocs disponibles d'Apatxe, i li canvien el seu propietari a root.
- La línia amb la instrucció EXPOSE fa públic el port que s'indique (el 80, en este cas), de manera que puga ser accessible des d'aplicacions fora del contenidor i fins i tot del sistema amfitrió.
- L'última línia (CMD) ha de ser única en cada contenidor. Indica el que s'executarà en iniciar-se. En este cas, arranquem apatxe en mode FOREGROUND (és a dir, en primer pla). Si ho arrancàrem com a servici (per exemple, amb CMD["service apache2 start"]), llavors el contenidor es tancaria immediatament, al no tindre cap procés executant en primer pla. Si hi haguera més d'una instrucció CMD definida en l'arxiu *Dockerfile*, només es considerarà l'última d'elles.

Existixen altres instruccions, com ENV per a definir variables d'entorn, però la idea de l'estructura i utilitat d'un arxiu *Dockerfile* pot haver quedat clara amb este exemple senzill.

2. Instal·lació i posada en marxa

Vegem ara com instal·lar Docker. En la web oficial podem trobar-ho per als principals sistemes operatius:

- En el cas de *Windows* i *Mac* instal·lem una ferramenta anomenada *Docker Desktop*, que ens permet configurar i posar en marxa els diferents contenidors que necessitem des d'estos sistemes. És necessari, no obstant això, algun sistema que simule un kernel de Linux com *host*, sobre el qual secundar Docker. Les últimes versions de *Docker Desktop* ja incorporen este sistema Linux virtualizado. Una vegada instal·lat, podrem usar diferents instruccions de Docker des de línia de comandos.
- En el cas de *Linux* (especialment si estem utilitzant un servidor remot sense interfície gràfica), el que s'instal·la és directament el *Docker Engine*, per a poder configurar i llançar contenidors directament des del terminal. La instal·lació és molt més lleugera, al no necessitar una màquina virtual addicional sobre la qual executar Docker (ja que utilitzarà la infraestructura del propi sistema Linux en què s'instal·la).
- De manera addicional, també pot instal·lar-se i configurar-se Docker en el núvol (sistemes AWS, Azure...).

2.1. Instal·lar *Docker Engine* en Linux Debian

En estos apunts ens centrarem en la instal·lació de *Docker Engine* sobre Linux, per ser el més habitual. En concret ho instal·larem sobre un sistema Debian. Podem trobar més informació respecte a la instal·lació en la documentació oficial de Docker. Podem comprovar la versió actual del nostre sistema Linux amb el comando `cat /etc/us-release` o bé amb el comando `lsb_release -cs`. També podem determinar l'arquitectura de processador amb el comando `uname -m`. Així podrem determinar si el nostre sistema és

compatible amb Docker actualment o no, consultant el llistat de sistemes disponibles en la web anterior d'instal·lació.

Seguint els passos de la documentació oficial que esmentem, primer hem de desinstal·lar versions antigues de Docker (anomenades *docker*, *docker-engine* o *docker.io*), si n'hi haguera, amb este comando :

```
for pkg in docker.io docker-doc docker-compose podman-docker \
containerd runc; do sudo apt-get remove $pkg; done
```

Després ja podem instal·lar Docker, de diverses formes: des de repositori (recomanat), instal·lant manualment el paquet *.deb*, o des d'alguns scripts que automatitzen la instal·lació, i que suposen l'única via d'instal·lació en sistemes com Raspberry Pi (Raspbian). En el nostre cas farem una instal·lació des de repositori, per la qual cosa podem seguir els passos i executar les comandes que s'indiquen ací.

Una vegada que ja hem instal·lat Docker, podem provar que tot ha anat correctament executant la imatge de prova `hello-world`, que mostrarà un missatge de salutació per pantalla i finalitzarà.

```
sudo docker run hello-world
```

> **NOTA**: observeu en els missatges que es mostren en executar la comanda que la imatge no està disponible de manera local, però Docker la descarrega automàticament.

També podem verificar que estiga correctament instal·lat executant `docker` a seques (que traurà una llista amb les opcions a indicar juntament amb la comanda), o `docker version` (que mostrarà la versió actualment instal·lada).

Per defecte, el dimoni de Docker s'inicia automàticament després de la instal·lació. Podem habilitar o deshabilitar que s'inicie amb el sistema amb estes comandes, respectivament:

```
sudo systemctl enable docker
sudo systemctl disable docker
```

2.2. Desinstal·lar Docker

Per a desinstal·lar Docker del sistema, executem el comando:

```
sudo apt-get purge docker-ce docker-ce-cli containerd.io
```

També podem eliminar les imatges, contenidors, etc del sistema, esborrant manualment la carpeta `/var/lib/docker` :

```
sudo rm -rf /var/lib/docker
```

3. Creació i ús de contenidors

Ja hem vist què és Docker i com instal·lar-ho. Vegem ara com crear diferents tipus de contenidors i quines operacions útils podem emprar en ells. Convé tindre present que la majoria de comandes `docker` que s'indiquen a continuació requeriran dels permisos adequats (*root/sudo*) per a poder-los utilitzar.

3.1. Les imatges: el *hub* de Docker

Per a crear un contenidor necessitem disposar d'una imatge Docker. Existixen multitud de llocs des d'on descarregar imatges de sistemes utilitzables en Docker, però potser el lloc més interessant de tots és el **hub de Docker** (*Docker Hub*), en esta URL.

En esta web els usuaris pugen imatges creades o personalitzades per ells mateixos, perquè la resta de la comunitat les puga utilitzar i/o adaptar. Existixen alguns repositoris importants, com el d'Apatxe, Nginx, Node, MongoDB o MySQL, amb estes aplicacions ja instal·lades i llistes per a poder-se utilitzar. Veurem més tard algun exemple sobre això.

Per a buscar alguna imatge que ens puga interessar, podem emprar la comanda `docker search` amb la paraula clau (per exemple, "mongodb"), i veure quines imatges hi ha disponibles sobre este element, juntament amb la seua valoració (*STARS*), i alguna informació addicional. Les imatges apareixen ordenades per valoració:

```
docker search mongodb

NAME            DESCRIPTION           STARS  OFFICIAL
mongo           MongoDB document...   10099  [OK]
mongo-express   Web-based MongoDB...   1411  [OK]
bitnami/mongodb Bitnami MongoDB...      245
...
```

Des del propi client/terminal de Docker podem descarregar les imatges amb:

```
docker pull nom_imatge
```

Per a veure les imatges que tenim disponibles en el sistema, podem escriure:

```
docker images
```

I apareixerà un llistat amb el nom de la imatge, la seua versió, i l'espai que ocupa, entre altres dades. Per exemple:

```
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
busybox       latest    5b0d59026729   3 weeks ago    1.15MB
alpine        latest    05455a08881e   5 weeks ago    7.38MB
hello-world   latest    f2a91732366c   3 months ago   1.85kB
```

Docker permet albergar un nombre elevat d'imatges en local, però encara així, si volem eliminar una imatge del llistat perquè ja no l'anem a utilitzar (i sempre que no estiga sent utilitzada per cap contenidor), podem fer-ho amb `docker rmi`, seguit de l'aneu de la imatge (segons el llistat anterior):

```
docker rmi f2a91732366c
```

També podem escriure res més el principi de l'aneu (en este i altres comandos de *docker*), sempre que servisca per a diferenciar-ho de la resta:

```
docker rmi f2a
```

3.2. Descàrrega d'imatges

Provarem de descarregar una imatge per a unes primeres proves de comandes senzills. Provarem amb una imatge molt lleugera, anomenada ***alpine***, que conté un conjunt de ferramentes bàsiques per a treball sobre un kernel de Linux. Descarreguem la imatge amb:

```
docker pull alpine
```

Al no indicar versió, ens informarà per terminal que es descarregarà l'última (*latest*). En el cas de voler especificar versió, es pot indicar a mode de *tag*, posant dos punts després del nom de la imatge, i després el nom de la versió:

```
docker pull alpine:3.5
```

La imatge d'Alpine és una imatge molt lleugera (uns pocs MB), que podem comprovar que està descarregada amb la comanda `docker images` vista anteriorment:

```
docker images

REPOSITORY  TAG     IMAGE ID     CREATED      SIZE
alpine      latest  05455a08881e 5 weeks ago  7.38MB
...
```

3.3. Execució senzilla de contenidors

Vegem ara com crear i llançar algun contenidor senzill a partir de les imatges que obtinguem dels repositoris anteriors. La següent comanda **llança un contenidor** amb la imatge *alpine* descarregada prèviament, **i executa una comanda** `echo` sobre este contenidor:

```
docker run alpine echo "Hola món"
```

> **NOTA**: si en executar `docker run` la imatge no es troba disponible en el nostre equip, es descarregarà com a pas previ, abans de posar en marxa el contenidor sobre ella.

En este cas, obtindrem per pantalla "Hola món" i finalitzarà el procés. Podem consultar els contenidors actius amb `docker ps`. Si ho executem no obtindrem cap en la llista, ja que la comanda anterior ja va finalitzar. Però podem consultar els contenidors creats (detinguts o en execució) amb `docker ps -a`:

```
CONTAINER ID  IMAGE   COMMAND              CREATED       ...
3fd9065eaf02  alpine  "echo 'Hola món'"    2 minutes ago ...
```

Amb este exemple que hem vist, podem començar a entendre com s'usen i per a què servixen els contenidors. La seua filosofia és molt senzilla: executar una comanda o programa i finalitzar (o deixar-ho en execució). D'esta manera, podem llançar diversos contenidors sobre una mateixa imatge, i cadascun s'encarregarà d'executar una tasca o procés diferent (o el mateix diverses vegades).

Este altre exemple llança un contenidor amb la imatge *alpine*, i **obri un terminal interactiu** en ell, de manera que podem escriure comandes sobre el terminal que actuaran sobre la imatge emmagatzemada en el contenidor i el seu sistema d'arxius. Per a finalitzar l'execució del contenidor, n'hi ha prou que escriguem `exit` en el terminal:

```
docker run -it alpine /bin/sh
```

Podem **especificar el nom** (*name*) que se li dona, per a no haver d'estar buscant després l'*id* o el nom que Docker genera automàticament. Per a això, utilitzarem el paràmetre `--name`:

```
docker run -it --name alpine_nacho alpine /bin/sh
```

Una altra opció que pot resultar útil és **executar un contenidor sense emmagatzemar el seu estat en la llista**, amb el que s'esborra automàticament en finalitzar. Això s'aconseguix amb l'opció `-rm`:

```
docker run -rm alpine echo "Hola món"
```

En el cas de voler **executar un contenidor en segon pla** (com si fora un dimoni), es fa amb el paràmetre `-d`. En este cas, no mostrarà cap resultat per pantalla, i alliberarà el terminal per a poder fer altres operacions. Esta opció és útil per a llançar contenidors relacionats amb servidors (bases de dades, servidors web...), i deixar-los així en marxa sense bloquejar el terminal.

```
docker run -d imatge
```

Associat amb la comanda anterior, si pel que fora es produïra algun missatge d'error o d'eixida, per a poder-lo consultar emprarem la comanda `docker logs`, seguit de l'*id* del contenidor executat:

```
docker logs f2e5a1629beb
```

3.4. Parada i esborrament de contenidors

Per a finalitzar un contenidor actualment en execució podem llançar la comanda `docker stop` juntament amb l'*id* del contenidor (o el nom), obtingut amb el propi `docker ps` anterior:

```
docker stop f2e
```

Podem eliminar el contenidor de la llista de contenidors, sempre que no estiga actiu, amb `docker rm`, passant-li l'*id* del contenidor (o el principi del mateix):

```
docker rm f2e
```

En el cas que el contenidor estiga en la llista de contenidors i haja finalitzat la seua execució (visible amb `docker ps -a`), podem tornar-lo a llançar fàcilment amb `docker start`, seguit de l'*id* o del nom del contenidor:

```
docker start f2e
```

3.5. Associar recursos del *host* al contenidor

És molt habitual en unes certes ocasions associar algun recurs del sistema *host* al contenidor, de manera que des del contenidor es puga accedir o comunicar amb este recurs del *host*. En concret, veurem a continuació com associar ports i carpetes.

3.5.1. Associar ports entre *host* i contenidor

Per a llançar un contenidor i **associar un port** del sistema *host* amb un port del contenidor, s'empra el paràmetre `-p`, indicant primer el port del *host* que es compartix, i després, seguit de dos punts `:`, el port del contenidor pel qual es comunica. Podem repetir este paràmetre més d'una vegada per a associar més d'una parella de ports.

Per exemple, la següent comanda associa el port 80 del *host* amb el port 80 del contenidor, i el port 8080 del *host* amb el 3000 del contenidor, de manera que les peticions que arriben al *host* pels ports 80 i 8080, respectivament, s'enviaran al contenidor.

```
docker run -p 80:80 -p 8080:3000 alpine:apatxe
```

3.5.2. Associar carpetes entre *host* i contenidor

Si volem **associar una carpeta del *host*** a una carpeta del contenidor, s'empra el paràmetre `-v`, indicant primer la carpeta del *host*, seguida de dos punts `:`, i després la carpeta del contenidor. Per exemple, la següent comanda associa la carpeta `~/data` del *host* amb la carpeta `/data/db` del contenidor associat a una imatge `mongo`, comunicant els dos ports per defecte, de manera que les dades que s'afigen sobre el contenidor s'emmagatzemaran en la carpeta `~/data` del nostre sistema *host*. A més, llança el contenidor en segon pla perquè el servidor quede executant sense bloquejar el terminal.

```
docker run -d -p 27017:27017 -v ~/data:/data/db mongo
```

3.6. Arrancar contenidors Docker amb el sistema

Ara que ja sabem com llançar contenidors Docker per a allotjar els nostres servicis (servidors web, bases de dades, etc), és convenient també saber com posar-los en marxa en arrancar el sistema, per si este patix qualsevol reinici inesperat.

Per a això, farem ús del paràmetre `--restart`, passant-li l'opció de reinici que vulguem:

- `no` : per a no reiniciar el contenidor automàticament (opció per defecte)
- `on-failure` : es reiniciarà si es va tancar a causa d'un error
- `always` : es reiniciarà sempre que es detinga (encara que ho detinguem nosaltres manualment amb `docker stop`)
- `unless-stopped` : es reiniciarà sempre que no ho hàgem detingut nosaltres manualment.

Així, per exemple, reiniciaríem automàticament un contenidor amb *alpine*:

```
docker run --restart unless-stopped alpine
```

> **NOTA:** si es reinicia el sistema i tenim el dimoni de `docker` habilitat com a servici, este dimoni es reiniciarà també, i posarà en marxa tots els contenidors que tinguen configurada apropiadament la seua opció de `--restart` .

3.7. Resum de comandes útils

A continuació vam mostrar les comandes més rellevants que hem utilitzat, a mode de guia resum. Recorda anteposar la paraula *sudo* si no tens permisos suficients d'execució:

```
# Buscar imatges en Hub
docker search nom_imatge

# Descarregar imatge del Hub
docker pull nom_imatge

# Veure imatges disponibles en local
docker images

# Eliminar imatge (no utilitzada per cap contenidor)
docker rmi id_imatge

# Llançar contenidor des d'imatge (forma simple)
# La imatge es descarrega del Hub si no està en local
docker run nom_imatge

# Llançar contenidor en mode dimoni (servidors)
docker run -d nom_imatge

# Llistat de contenidors actius
docker ps

# Llistat de tots els contenidors
docker ps -a

# Detindre contenidor actiu
docker stop id_contenidor

# Reprendre contenidor
docker start id_contenidor

# Eliminar contenidor (prèviament detingut)
docker rm id_contenidor

# Associar ports a contenidor
docker run -p 80:80 -p 8080:3000 nom_imatge

# Associar carpetes locals (volums) a contenidor
docker run -v carpeta_local:carpeta_contenidor nom_imatge

# Iniciar contenidor indicant que arrancada amb sistema
docker run --restart unless-stopped nom_imatge
```

Exercici 1:

Descarrega la imatge de *mongo* amb Docker i posa en marxa un contenidor, mapejant els ports per defecte i associant la carpeta */data/db* del contenidor amb la carpeta *~/data* del teu sistema *host*. Fes-ho

> de manera que s'execute com a dimoni, i es reinicie amb el sistema llevat que el parem manualment. Prova a connectar amb el servidor des d'algun client MongoDB (Compass, extensió de VS Code...).

> **Exercici 2:**
>
> Utilitza el projecte *LlibresWebSessions* que has realitzat en esta sessió. Puja-ho al teu VPS (a través de GitHub), i posa-ho en marxa. Pots utilitzar la passarel·la *Passenger* amb Apatxe, com vam fer també en esta altra sessió. Comprova que connecta adequadament amb el servidor Mongo llançat en l'exercici anterior amb Docker.
>
> **NOTA**: hauràs de crear a mà la carpeta *public/uploads* y donar-li permisos adequats d'escriptura per provar la pujada d'imatges

4. Creació d'imatges

En la secció anterior hem vist com llançar de diferents formes contenidors de terceres parts. Però una interessant utilitat de Docker consistix a poder instal·lar el programari que necessitem sobre un contenidor, executar-lo, i també fer una imatge personalitzada del contenidor modificat, per a poder-la reutilitzar les vegades que vulguem.

Ara veurem com podem crear les nostres pròpies imatges, per a poder-les llançar en contenidors, o distribuir a altres equips. Veurem dos maneres de fer-ho: executant el contenidor i instal·lant el programari necessari, o a través d'arxius de configuració *Dockerfile*.

4.1. Instal·lar aplicacions i crear imatges personalitzades

Farem un exemple senzill a partir de la imatge *Alpine* utilitzada abans. Esta imatge per defecte no ve amb l'editor *nano* instal·lat, així que l'instal·larem, i crearem una nova imatge amb eixe programari ja preinstal·lat.

El primer que farem serà executar la imatge original en mode interactiu, com en algun exemple anterior:

```
docker run -it alpine /bin/sh
```

Una vegada dins del terminal, instal·lem *nano* amb estes comandes:

```
apk update
apk upgrade
apk add nano
```

Després d'això, podem provar d'obrir l'editor amb `nano`, i ja podem tancar el contenidor amb `exit`.

Una vegada definit el contenidor, per a crear una imatge a partir d'ell emprarem la comanda `docker commit`. Esta comanda accepta una sèrie de paràmetres, on podem especificar l'autor de la imatge,

la versió, etc. Abans de res, hem d'esbrinar l'*id* del contenidor que volem utilitzar per a la imatge (amb `docker ps -a`). Suposem per a este exemple que l'*id* és *f2e5a1629beb*. En este cas, podem crear una imatge del mateix amb:

```
docker commit -a "Nacho Iborra <nachoiborra@iessanvicente.com>" \
f2e5a1629beb nacho/alpine:nano
```

En el paràmetre `-a` indiquem l'autor i el seu e-mail, després indiquem l'*id* del contenidor a utilitzar, i finalment indiquem un nom de repositori amb uns tags opcionals que ajuden a identificar la imatge. En este cas indiquem que és una versió particular de la imatge *alpine* amb *nano* instal·lat en ella.

Després d'esta comanda, podem localitzar la imatge en el llistat d'imatges amb `docker images`:

```
REPOSITORY    TAG       IMAGE ID     ...
nacho/alpine  nano      1a3e882f228a ...
alpine        latest    3fd9065eaf02 ...
```

A partir d'este moment, podem utilitzar esta imatge per a llançar contenidors, com en este exemple:

```
docker run -it nacho/alpine:nano /bin/sh
```

Així podem crear imatges a mesura amb un cert programari base preinstal·lat.

4.2. Crear imatges a partir d'arxius *Dockerfile*

Vegem en esta secció com emprar l'arxiu de configuració *Dockerfile* per a crear imatges Docker personalitzades d'una forma més automatitzada, sense haver d'entrar per terminal en el contenidor i instal·lar els programes, com hem fet en l'apartat anterior.

Crearem una imatge similar a la de l'apartat anterior: partint d'una base *Alpine*, instal·larem *nano* en ella i crearem la imatge. En primer lloc, hem de crear un arxiu anomenat *Dockerfile* en la nostra carpeta de treball. El contingut de l'arxiu serà el següent:

```
FROM alpine:latest
MAINTAINER Nacho Iborra <nachoiborra@iessanvicente.com>
RUN apk update
RUN apk upgrade
RUN apk add nano
CMD ["/bin/sh"]
```

En la primera línia (FROM) indiquem la imatge base sobre la que partir (última versió de la imatge *alpine*), i en la segona (MAINTAINER) les dades de contacte del creador de la imatge. Després, amb la instrucció RUN podem executar comandes dins de la imatge, com hem fet prèviament. En este cas, actualitzem el sistema i instal·lem *nano*. Finalment, la instrucció CMD és obligatòria per a crear contenidors, i indica el que executaran (tots els contenidors han d'executar alguna cosa). En este cas indiquem que execute el terminal.

A partir d'este arxiu *Dockerfile*, en la mateixa carpeta on està, executem la comanda `docker build` per a crear la imatge associada.

```
docker build -t nacho/alpine:nano .
```

El paràmetre `-t` servix per a assignar tags a la imatge. En este cas, indiquem que és la versió amb *nano* incorporada de la imatge *alpine*. També podem utilitzar este tag múltiples vegades, per a assignar diferents. Per exemple, indicar el número de versió, i que a més és l'última disponible:

```
docker build -t nacho/alpine:1.0 -t nacho/alpine:latest .
```

Una vegada creada la imatge, la tindrem disponible en el nostre llistat `docker images`:

```
REPOSITORY      TAG         IMAGE ID      ...
nacho/alpine    nano        6800a48a4d68  ...
alpine          latest      3fd9065eaf02  ...
```

I podrem executar-la en mode interactiu. En este cas no fa falta que indiquem la comanda a executar, ja que ho hem especificat en l'arxiu *Dockerfile* (el terminal):

```
docker run -it nacho/alpine:nano
```

4.2.1. Exemple: aplicació Node

Vegem ara un exemple pas a pas de com configurar un arxiu *Dockerfile* per a crear un contenidor per a una aplicació Node.js. Hem d'afegir un fitxer `Dockerfile` en l'arrel del nostre projecte Node. Podria tindre una aparença com esta:

```
FROM node:20
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY package*json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD ["npm", "start"]
```

- La primera línia `FROM node:20` especifica que necessitem incorporar al contenidor la imatge de *Node*, en concret de la seua versió 20.
- La línia `RUN mkdir -p /usr/src/app` indica que es crearà eixa carpeta dins del contenidor (creant també les subcarpetas necessàries, a través de l'opció `-p`). No té per què ser eixa ruta específicament, podem canviar-la per qualsevol altra que preferim.
- La instrucció `WORKDIR /usr/src/app` ens situa en eixa carpeta com a carpeta de treball, perquè els comandos que executem partisquen d'ací.
- La instrucció `COPY package*.json ./` còpia els fitxers de configuració JSON (*package.són* i *package-lock.json*) en la carpeta indicada abans en *WORKDIR*.
- La instrucció `RUN npm install` instal·larà les dependències d'eixos arxius en la subcarpeta *node_modules*, dins de *WORKDIR*.
- La instrucció `COPY . .` còpia tot el contingut de la nostra carpeta actual dins de la carpeta indicada en *WORKDIR*. Això pot suposar un problema, ja que podríem tindre una carpeta *node_modules*, per exemple, o altres elements que no vulguem copiar. Per a solucionar això podem crear un fitxer `.dockerignore` que indique quins elements de la carpeta del projecte han d'ignorar-se. El format és similar als arxius *.gitignore* de *Git*, i pot tindre un contingut com este:

```
node_modules
```

- La instrucció `EXPOSE 8080` exporta el port 8080, de manera que el *host* es comunicarà amb el contenidor a través d'este port. Canviarem este número de port pel qual hàgem posat a escoltar la nostra aplicació Node.
- La línia `CMD` indica el comando que finalment executarà el contenidor. En este cas executa `npm start`, suposant que tenim definit este *script* en el nostre *package.json* per a posar en marxa l'aplicació.

Per generar la imatge del projecte executarem esta comanda desde l'arrel del projecte. Canviarem el paràmetre *app_node* pel nom que vulguem donar a esta imatge:

```
docker build -t app_node .
```

Una vegada finalitzada l'execució de la comanda (pot tardar un temps si necessita descarregar la imatge de *node*), podrem veure la nostra nova imatge llistada amb `docker images`.

4.3. Ubicació de les imatges. Còpia i restauració.

Les imatges que descarreguem, i les que creem nosaltres de manera manual, se situen per defecte en la carpeta `/var/lib/docker`, encara que depenent de la font i el format de la imatge, es localitzen en l'una o l'altra subcarpeta, i amb l'un o l'altre format.

La principal utilitat que pot tindre conéixer la ubicació de les imatges és el poder-les portar d'una màquina *host* a una altra. Per a això, podem emprar la comanda `docker save`, indicant el nom de l'arxiu on guardar-la, i el nom de la imatge a guardar:

```
docker save -o <path_a_arxiu> <nom_imatge>
```

Per exemple:

```
docker save -o /home/alumne/la meua_imatge.tar nacho/alpine:nano
```

Es generarà un arxiu TAR que podrem portar a una altra part. En la nova màquina, una vegada copiat l'arxiu, podem carregar la imatge que conté en la carpeta d'imatges de Docker amb `docker load`:

```
docker load -i <path_a_arxiu>
```

5. Comunicació entre contenidors

Hem vist com crear i posar en marxa contenidors de diverses formes. En exemples i exercicis anteriors hem posat en marxa, d'una banda, un contenidor MongoDB (*Exercici 1*) i, per un altre, una aplicació Node (exemple anterior). Com podríem comunicar estos dos contenidors perquè des de l'aplicació puguem accedir a la base de dades, per exemple? En este apartat veurem alguns mecanismes que podem emprar per a comunicar contenidors.

5.1. Compartir una mateixa xarxa virtual

Una forma senzilla de comunicar contenidors és afegint-los a una mateixa xarxa virtual. Primer creem la xarxa virtual:

```
docker network create nom_xarxa
```

on canviarem el paràmetre *nom_xarxa* pel nom que vulguem donar-li a la xarxa. El següent pas és connectar a la xarxa virtual cadascun dels contenidors implicats:

```
docker network connect nom_xarxa id_contenidor1 id_contenidor2 ...
```

NOTA: també es pot usar el nom del contenidor, si el tenim accessible.

Exercici 3:

Comunicarem el projecte *tasques-nest* que desenvolupem en este apartat amb la base de dades Mongo que vam posar en marxa en el *Exercici 1*. Seguix estos passos:

1. Detingues i elimina el contenidor de Mongo. No et preocupes per les dades, perquè estaran guardades en la carpeta *data* de la teua carpeta d'usuari. Suposant que el contenidor Mongo té l'*id* 111a, les comandes serien

```
sudo docker stop 111a
sudo docker rm 111a
```

2. Crea una xarxa on conviuran els contenidors que crearem (Mongo i Nest). La cridarem per exemple *node_mongo*:

```
sudo docker network create node_mongo
```

3. Posa de nou en marxa un contenidor Mongo. Però esta vegada ho crearem associat a eixa xarxa, i li donarem un nom (per exemple "la_meua_bd_mongo").

```
sudo docker run -d --name la_meua_bd_mongo --network node_mongo \
--restart unless-stopped -p 27017:27017 -v ~/data:/data/db mongo
```

4. Modifica el projecte *tasques-nest-jwt* que esmentàvem abans. Afig un fitxer *Dockerfile* en l'arrel del projecte amb un contingut com este:

```
FROM node:20
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY package*json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

5. Afig també al projecte un fitxer *.dockerignore* perquè no es tinga en compte la carpeta *node_modules* en les comandes de Docker. Tindrà este contingut:

```
node_modules
```

6. Modifica també el fitxer *app.module.ts* del projecte perquè no connecte a MongoDB en *localhost*, sinó en el nom que li has donat al contenidor creat en el pas 3:

```
MongooseModule.forRoot('mongodb://la_meua_bd_mongo/tasques-nest'),
```

7. Puja el projecte *tasques-nest-jwt* modificat a un repositori GitHub, i clona'l en la teua carpeta del VPS.
8. Accedix a la carpeta del teu projecte, i crea la imatge amb la corresponent comanda Docker (recorda afegir el punt al final per a indicar la carpeta origen):

```
sudo docker build -t app_tasques .
```

9. Llança l'app Nest en la mateixa xarxa que el contenidor Mongo, i exposa-la pel port que vulgues (per exemple, el 3000):

```
sudo docker run -d --name la_meua_app_tasques --network node_mongo \
--restart unless-stopped -p 3000:3000 app_tasques
```

10. Ja pots accedir a l'app amb la URL *vpsxxxxx.vps.ovh.net:3000/tasca* (per al llistat de tasques).

5.2. Connexió a través de *docker compose*

Una segona alternativa que tenim per a connectar contenidors és definir-los junts en un arxiu de configuració YAML (extensió `.yml`), anomenat `docker-compose.yml` (normalment en l'arrel del projecte a

desenvolupar). Ací veiem un exemple:

```
version: "3"

services:
    web:
        container_name: app_node
        restart: always
        build: .
        ports:
            - "8080:3000"
        depends_on:
            - "mongo"
    mongo:
        container_name: bd_mongo
        restart: always
        image: mongo
        ports:
            - "27017:27017"
```

Analitzem algunes línies del fitxer:

- La primera línia `version: "3"` fa referència a la versió de *docker-compose* que es vol utilitzar, a l'efecte de sintaxi permesa.
- Dins de la secció *services* definim cadascun dels servicis o contenidors que volem posar en marxa. En este exemple tenim dos: *web* per a l'aplicació Node i *mongo* per al servidor Mongo.
- Quant al servici *web*, indiquem un nom de contenidor, el mode de reinici, la carpeta des de la qual construir l'aplicació (carpeta actual, si hem definit el fitxer *docker-compose.yml* en l'arrel del projecte Node), mapatge de ports i, en *depends_on*, a quins altres servicis necessitem connectar des d'este (al servici de base de dades posterior)
- Pel que fa al servici *mongo*, el crearem a partir de la imatge *mongo*, amb el mapatge de ports indicat.

Per a construir tots els servicis executem este comando des de l'arrel del projecte Node, en este cas:

```
docker compose build
```

Per a posar després en marxa els servicis construïts executem este altre comando:

```
docker compose up
```