

Docker

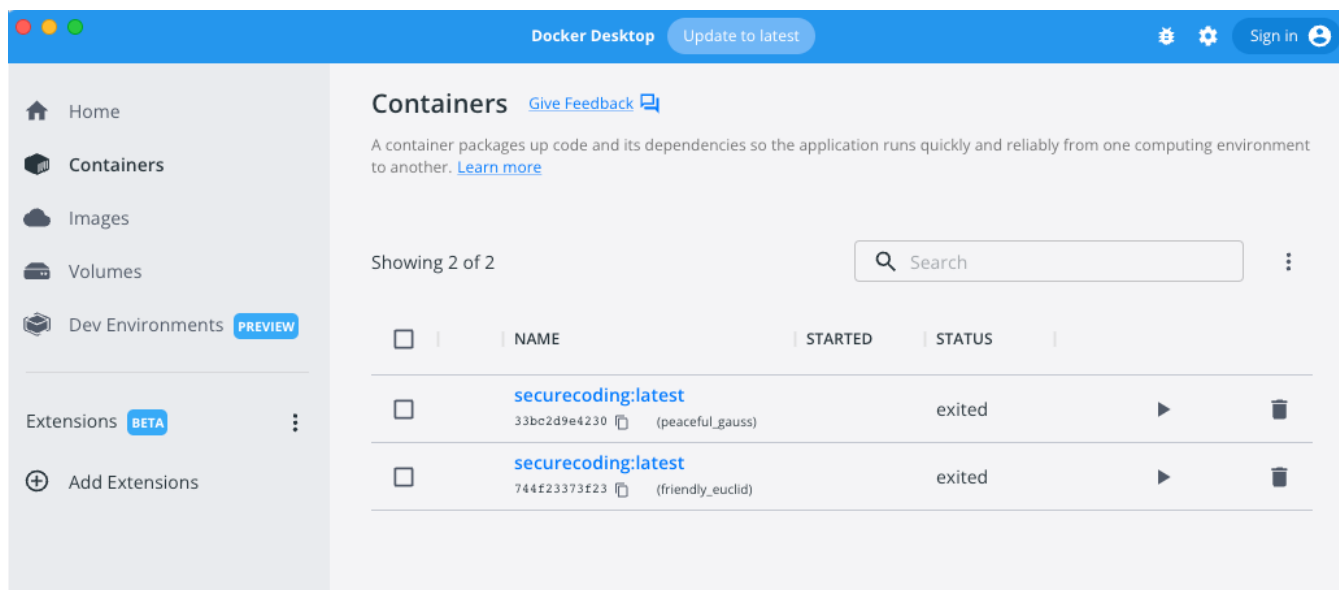
Docker es un sistema de contenedores para microservicios. La idea es reducir las aplicaciones a las partes más fundamentales (servidor de API, base de datos, sistemas de ticketing, colas...) y ejecutar cada uno de estos microsistemas en entornos completamente aislados llamados "contenedores". Las aplicaciones web modernas funcionan como contenedores de microservicios. Aunque hay varios sistemas de contenedores, hoy nos vamos a centrar en docker, uno de los más utilizados.

Aunque la metáfora no es perfecta, puedes entender un contenedor como una pequeña máquina virtual formada por "snapshots", fotografías del estado de una máquina virtual, a los que puedes volver en cualquier momento para reiniciar el sistema y tenerlo exactamente igual que cuando empezaste.

Docker tiene tres conceptos fundamentales:

- **Images:** son el equivalente al snapshot de una máquina virtual. Una imagen es un contenedor listo para ejecutarse.
- **Containers:** son el contenedor en sí, el equivalente a una máquina virtual ejecutándose en tu sistema. Puedes ejecutar a la vez y en paralelo varios contenedores a partir de la misma imagen. Cuando un contenedor acaba, se destruye junto con todos los datos que contenga... a menos que los datos estén en volúmenes.
- **Volume:** en nuestra metáfora imperfecta, son "pendrives virtuales" que puedes enchufar a los contenedores para guardar datos que quieras conservar.

La interfaz principal de docker es la línea de comandos, pero en Windows y OSX puedes encontrar una interfaz que sirve al menos para comprobar qué imágenes, contenedores y volúmenes tienes en tu sistema, y hacer algunas operaciones básicas con ellos.



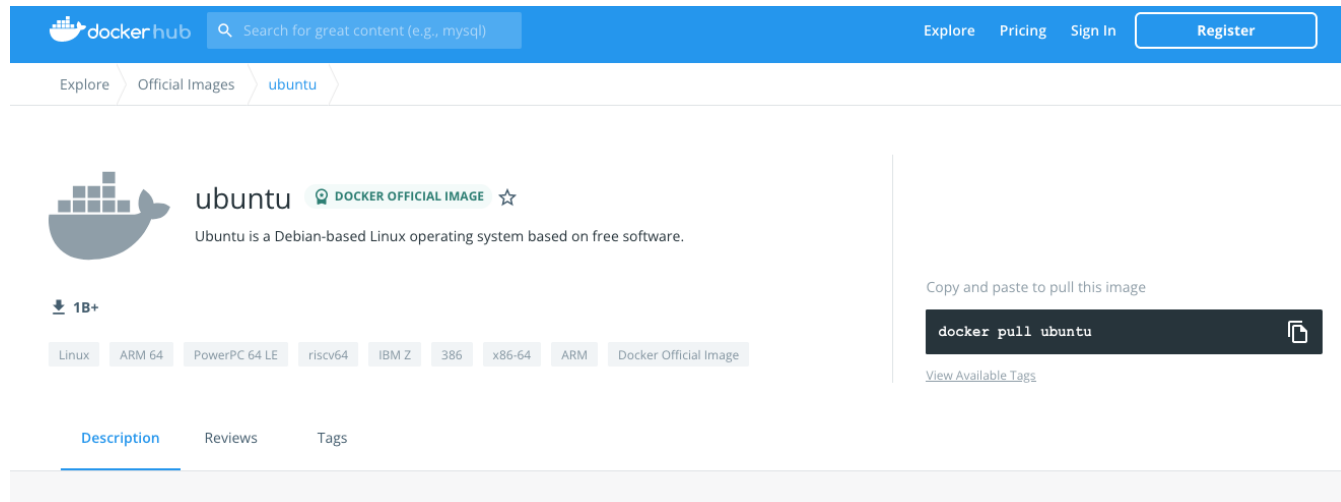
Aún así, la interfaz principal para comunicarse con docker es la línea de comandos y una aplicación llamada "docker-compose"

```
23:54 ejercicios $ docker image ls
REPOSITORY              TAG          IMAGE ID       CREATED        SIZE
securecoding            latest      01f13f5d5d39   5 days ago    992MB
docker/getting-started  latest      157095baba98   2 months ago  27.4MB
nabo.codimd.dev/hackmdio/hackmd  2.4.1      f1290e44dd49   13 months ago 723MB
postgres               11.6-alpine 94d455984111   2 years ago   150MB
23:54 ejercicios $
```

Dónde conseguir imágenes

¿Dónde podemos conseguir imágenes, es decir, contenedores listos para ejecutarse? Principalmente, en la página <https://hub.docker.com/> (<https://hub.docker.com/>). No es necesario registrarse para descargar imágenes, solo para subirlas o poder tener imágenes privadas.

Hay imágenes de cualquier cosa que puedas imaginar. Busca, por ejemplo: "ubuntu", "mariadb" o "elasticsearch". Recuerda: no es necesario registrarte si simplemente quieres descargar imágenes.



Fíjate que cada entrada tiene una descripción más o menos amplia, una ayuda de cómo usarse, una lista de etiquetas (label), que son las versiones disponibles, y el comando para descargar la imagen:

```
docker pull ubuntu
```

Eso descargará la última versión de ubuntu (etiqueta *latest*), pero puedes descargar versiones determinadas con las etiqueta. Por ejemplo:

```
docker pull ubuntu:22.04
docker pull ubuntu:bionic
```

Hay otros lugares donde conseguir imágenes, y tu empresa puede tener su propio repositorio. Además, tú puedes construir tus propias imágenes basadas en otras imágenes, como veremos al final de estos ejercicios.

Nuestro primer contenedor

Ejecuta este comando:

```
docker run -d -p 80:80 docker/getting-started
```

Este comando `docker run` empieza un contenedor a partir de la imagen `docker/getting-started`. Como es muy posible que esta imagen no esté en tu sistema, la descargará de dockerhub. Es decir: `docker run` ya incluye antes un `docker pull` si no lo has hecho tú mismo/a ¡Qué cómodo!

Esta será la estructura de los comandos docker:

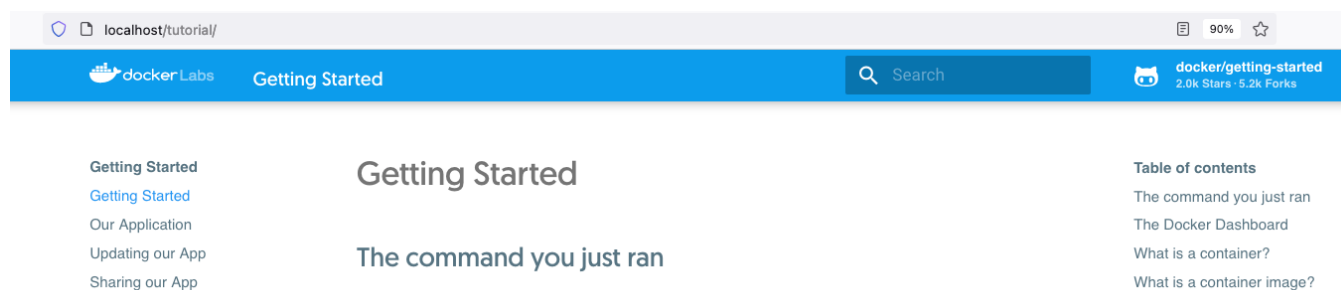
```
docker COMANDO PARAMETROS-DEL-COMANDO IMAGEN-O-CONTENEDOR PARAMETROS-DEL-CONTENEDOR
```

OR

En este caso, hemos ejecutado el comando `run` con los parámetros `-d -p 80:80` y la imagen `docker/getting-started`, que es una imagen de bienvenida de docker. Hay decenas de parámetros diferentes, veremos algunos poco a poco:

- `-d` : empieza el contenedor y vuelve inmediatamente a la línea de comandos. Si no lo usas, verás los mensajes del contenedor. No usarlo es muy útil para debug y ver los errores.
- `-p 80:80` : mapea las conexiones al puerto 80 del PC real al puerto 80 del contenedor. Es decir: hace un NAT.

Ahora puedes visitar el enlace <http://localhost:80> (<http://localhost:80>), que es lo que hace nuestro contenedor: presentar esta página web.



Congratulations! You have started the container for this tutorial! Let's first explain the command that you just ran. In case you forgot, here's the command:

```
docker run -d -p 80:80 docker/getting-started
```

You'll notice a few flags being used. Here's some more info on them:

- `-d` - run the container in detached mode (in the background)
- `-p 80:80` - map port 80 of the host to port 80 in the container
- `docker/getting-started` - the image to use

Pro tip

You can combine single character flags to shorten the full command. As an example, the command above could be written as:

```
docker run -dp 80:80 docker/getting-started
```

Ese contenedor contiene un tutorial de "primeros pasos con docker", y lo vamos a seguir más o menos en este ejercicio.

Fíjate en la interfaz gráfica: muestra dos contenedores que yo ejecuté en el pasado, que ya están apagados, y el que acabamos de empezar. Desde la interfaz gráfica podemos pausar o parar este contenedor.

	NAME	STARTED	STATUS	
<input type="checkbox"/>	securecoding:latest 33bc2d9e4230 (peaceful_gauss)		exited	
<input type="checkbox"/>	securecoding:latest 744f23373f23 (friendly_euclid)		exited	
<input type="checkbox"/>	docker/getting-started d704f5d982c2 (intelligent_goodall)	2 minutes ago	running	

Podemos también ver esta misma información con:

```
viu $ docker container ls -a
CONTAINER ID   IMAGE                                COMMAND                                CREATED
STATUS        PORTS                                NAMES
d704f5d982c2   docker/getting-started              "/docker-entrypoint..."            2 minutes ago
Up 2 minutes   0.0.0.0:80->80/tcp                  intelligent_goodall
```

(a tí te aparecerá un identificador diferente)

Y podemos parar y borrar este contenedor con este comando (usa el identificador que te haya aparecido a tí):

```
docker container stop d704f5d982c2
docker image ls -a
docker container rm d704f5d982c2
```

Un contenedor parado ya no sirve de nada, no puede reiniciarse. Pero puedes empezar otro igual a partir de una imagen limpia. Solo está ahí para que comprobemos qué errores ha podido tener, si lo necesitamos.

También podríamos haberle puesto un nombre a un contenedor, y decirle que lo borre automáticamente al cerrarse. Ejecuta ahora este comando:

```
docker run -d -p 80:80 --name test --rm docker/getting-started
```

Los dos nuevos parámetros significan: dale el nombre "test" a este contenedor (ya no es necesario usar identificadores) y borra el contenedor cuando se pare (`--rm`)

No lo pares todavía: vamos a entrar en el contenedor:

```
docker exec -ti test sh
```

En este caso significa: `docker exec` ejecuta un comando dentro de un contenedor que esté ejecutándose, el que se llama `test`, `-ti` significa que el comando es interactivo (es decir, podremos escribir y ver qué pasa) y el comando que ejecutamos es `sh`, es decir, una shell.

```
0:20 viú $ docker run -d -p 80:80 --name test --rm docker/getting-started
4b9188392bff6b4ee87a5939e5d43aa438a9f72157532d78e26d28d6e012b81d
0:20 viú $ docker exec -ti test sh
/ # ls
bin                etc                mnt                run
tmp
dev                home               opt               /sbin
usr
docker-entrypoint.d lib                proc                srv
var
docker-entrypoint.sh media              root                sys
/ # whoami
root
/ # exit
```

¿Has visto? Es un contenedor que está ejecutando un Linux, y además como `root`

Haz algunas pruebas: crea un archivo (`touch prueba`), sal del contenedor, páralo (`docker stop test`), comprueba que está borrado (`docker ls -a`), vuelve a empezarlo y vuelve a entrar: ¡verás que el archivo prueba ha desaparecido!

Eso es porque los contenedores se reinician a como esté la imagen cada vez que se inician. Si quieres conservar datos entre ejecuciones (por ejemplo, en bases de datos), necesitas utilizar **volúmenes**

Vamos a usar el volumen más sencillo: mapeamos un directorio llamado `datos` del sistema real en el contenedor

1. Crea un directorio llamado `datos` y un archivo dentro del directorio. En Linux o similar: `mkdir datos && touch datos/prueba.txt`
2. Ejecuta ahora el contenedor con este comando. Fíjate en el parámetro `-v $PWD/datos:/datos` : significa "mapea el directorio del PC real `$PWD/datos` en el directorio `/datos` del contenedor.

```
(Linux/OSX) docker run -d --rm --name test -p 80:80 -v `pwd`/datos:/datos docker/
getting-started
(Windows) docker run -d --rm --name test -p 80:80 -v $PWD/datos:/datos docker/get
ting-started
```

3. Entra en el contenedor y comprueba que el archivo está allí:

```
0:27 viú $ docker exec -ti test sh
/ # ls
...
datos
...
```

Aparte de carpetas reales, puedes mapear también archivos individuales o utilizar volúmenes virtuales, que se gestionan con el comando `docker volume` o la interfaz gráfica.

También se pueden gestionar redes (comando `docker network`) para crear varios contenedores que son capaces de comunicarse entre sí. Por ejemplo: un contenedor con la aplicación web, otro con la base de datos, otro con el sistema de autenticación....

Como puedes sospechar, el comando para ejecutar uno o varios contenedores puede complicarse mucho y no querremos tener que escribirlo siempre. Para eso se creó `docker-compose`

Antes de pasar a la siguiente sección, comprueba que has borrados todos los contenedores que estén ejecutándose

docker-compose

Docker-compose es una interfaz para evitarse tener que escribir líneas complejas de comandos que tienen varios

contenedores comunicándose entre ellos. Para ello, lee toda la configuración de un archivo llamado `docker-compose.yml`, que está escrito con un formato llamado YAML (<https://en.wikipedia.org/wiki/YAML>) (<https://en.wikipedia.org/wiki/YAML>)

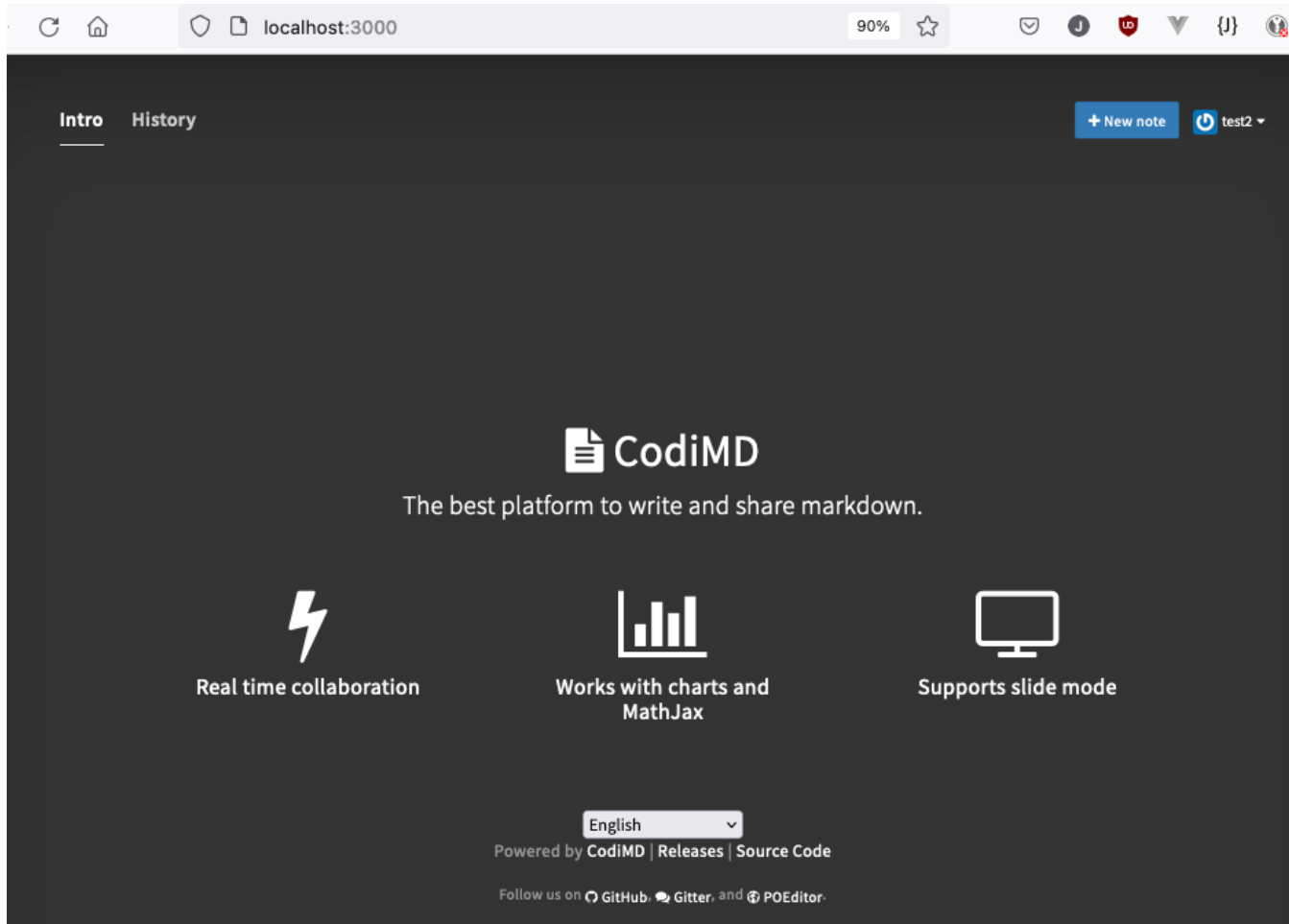
Vamos a ver cómo funciona docker-compose con la aplicación CodiMD, que es un editor de texto compartido multiusuario. Puedes consultar la ayuda y cómo configurarlo en producción real en este enlace: <https://github.com/linuxserver/docker-hedgedoc> (<https://github.com/linuxserver/docker-hedgedoc>)

Yo voy a usar este archivo, que es muy similar a un sistema en producción para un pequeño grupo de programadores:

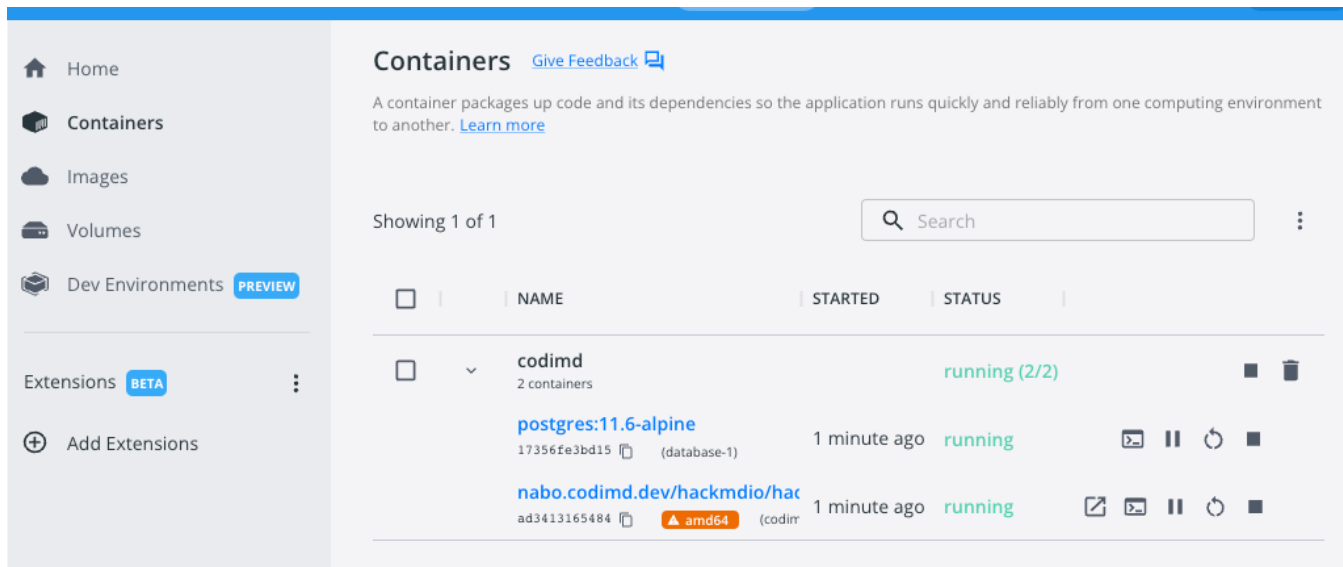
```
version: "3"
services:
  database:
    image: postgres:11.6-alpine
    environment:
      - POSTGRES_USER=hackmd
      - POSTGRES_PASSWORD=c48iiabh48nfd
      - POSTGRES_DB=codimd
    volumes:
      - database:/var/lib/postgresql/data
    networks:
      backend:
    restart: always
  codimd:
    # Esta imagen es codimd, la versión opensource original de hackmd
    image: nabo.codimd.dev/hackmdio/hackmd:2.4.1
    # Esta imagen es HedgeDoc, un fork de codimd. CAMBIA TAMBIÉN LOS VOLÚMENES MÁS ABAJO
    # image: quay.io/hedgedoc/hedgedoc:1.8.2
    environment:
      - CMD_DB_URL=postgres://hackmd:c48iiabh48nfd@database:5432/codimd
      - CMD_USECDN=true
      - CMD_ALLOW_ANONYMOUS=false
      - CMD_ALLOW_ANONYMOUS_EDITS=false
      - CMD_ALLOW_ANONYMOUS_VIEWS=true
      - CMD_SESSION_SECRET=6aj+3bes+az
      - CMD_IMAGE_UPLOAD_TYPE=filesystem
      # Comenta las siguientes líneas para pruebas en local sin SSL
      #- CMD_DOMAIN=hackmd.mycompany.com
      #- CMD_URL_ADDPORT=443
      #- CMD_PROTOCOL_USESSL=true
    depends_on:
      - database
    ports:
      - "3000:3000"
    networks:
      backend:
    volumes:
      - ./uploads:/home/hackmd/app/public/uploads
      - ./default.md:/home/hackmd/app/public/default.md
      - ./features.md:/home/hackmd/app/public/docs/features.md
      # Usa estas para hedgedoc
      #- ./uploads:/hedgedoc/public/uploads
      #- ./default.md:/hedgedoc/public/default.md
      #- ./features.md:/hedgedoc/public/docs/features.md
    restart: always
  volumes:
    database: {}
  networks:
    backend:
```

1. Crea un directorio nuevo llamado, por ejemplo `codimd`
2. Guarda este archivo (nombre: `docker-compose.yml`) dentro de este directorio

3. Crea otro directorio `uploads` ahí dentro. Aquí es donde guardará las imágenes que subamos
4. Ejecuta `docker-compose up -d` dentro de este directorio. Docker compose buscará el archivo `docker-compose.yml` en el directorio actual.
5. Visita: <http://localhost:3000> (<http://localhost:3000>)



¡Puedes registrar cualquier usuario con cualquier contraseña, es tu sistema!



Esto es todo lo que ha pasado:

1. Si no tenía las imágenes, las habrá descargado. Esto puede llevarle mucho tiempo.
2. Ha creado dos contenedores: uno llamado `database` que contiene la base de datos, otro llamado `codimd` que contiene la aplicación que usa la base de datos. Fíjate que el archivo `docker-compose` tiene una sección para cada uno de estos contenedores.
3. La base de datos está protegida por contraseña. La hemos puesto en el archivo, en esta línea, y además se la hemos dicho a la aplicación un poco más abajo:

```

database:
  (...)
  environment:
    - POSTGRES_USER=hackmd
    - POSTGRES_PASSWORD=c48iiabh48nfd
    - POSTGRES_DB=codimd
  (...)
codimd:
  environment:
    - CMD_DB_URL=postgres://hackmd:c48iiabh48nfd@database:5432/codimd
  (...)

```

Esta es la manera habitual de pasarle secretos a los contenedores: a través de variables de entorno. Por supuesto, tú tendrás que usar otras contraseñas.

4. Fíjate también que codimd sabe dónde está la base de datos (mira el código de antes), porque docker tiene un servicio similar a un DNS: la petición a la URL `postgres://database:5432` acabará llegando al contenedor `database`
5. Hay muchos volúmenes mapeados, tanto en la base de datos como en la aplicación

```

database:
  (...)
  volumes:
    - database:/var/lib/postgresql/data
  (...)
codimd:
  (...)
  volumes:
    - ./uploads:/home/hackmd/app/public/uploads
    - ./default.md:/home/hackmd/app/public/default.md
    - ./features.md:/home/hackmd/app/public/docs/features.md
  (...)

```

6. Los contenedores se reinician siempre, cuando se inicie tu PC: `restart: always`. Si apagas tu PC... verás que al reiniciarlo estos contenedores también se inician!
7. Solo se expone un puerto, el 3000. El PC real solo puede conectarse a este puerto, ni siquiera a la base de datos.

Crear nuestras propias imágenes: Dockerfile

Finalmente, vamos a ver cómo crear nuestras propias imágenes que luego podamos ejecutar como contenedores. Para crear una imagen, necesitamos crear un archivo llamado `Dockerfile`, con los comandos uno a uno que se necesitan para crear la imagen. **Las nuevas imágenes están basadas en imágenes anteriores.** *alpine* es una imagen muy utilizada porque es un Linux muy ligero que solo tiene lo básico. Otras imágenes que puedes usar de base son *ubuntu*, *debian*, o *node*, una imagen muy utilizada para el desarrollo de servidores en JavaScript

Este es un ejemplo de una aplicación web con Vue. Si quieres seguirlo, quizá tengas que instalar nodejs y vue en tu PC. Como este no es un curso de programación web, dejaré estos comandos como opcionales:

```

npm install -g @vue/cli
npm init vue@latest

```

Contesta Y a todas las preguntas: ahora tendrás un nuevo directorio `vue-project` que contiene el esqueleto de una aplicación web con VueJS.

Bien, vamos a crear una imagen con esta aplicación VueJS que acabamos de crear. Crea un archivo llamado `Dockerfile` (fíjate, sin extensiones, la primera letra en mayúscula) y ponlo dentro de tu aplicación:

```

FROM node:latest
WORKDIR /usr/src/app
COPY . .
RUN npm install
EXPOSE 3000
CMD ["npm", "run", "dev", "--", "--host"]

```

Ejecuta:

```
docker built -t securecoding:latest .
```

Este comando leerá el archivo `Dockerfile` que esté en el directorio actual y, si todo va bien, creará una imagen llamada `securecoding:latest` con nuestra aplicación. Para crear la imagen, ejecutará las órdenes del `Dockerfile` una tras otra:

- Es una imagen basada en la imagen `node:latest`
- Cambia (es decir, hace `cd`) al directorio `/usr/src/app`, que es donde están las aplicaciones web en la imagen `node:latest`
- Copia todos los archivos del directorio actual (es decir, nuestra aplicación) en la nueva imagen
- Ejecuta `npm install`
- Avisa que el puerto que tiene que exponer es el 3000
- Cuando se cree el contenedor, el comando que tiene que ejecutar es `npm run dev -- --host` para ejecutar la aplicación

Si todo ha ido bien, podrás ver tu nueva imagen haciendo:

```
docker image ls
```

Y ejecutarla con:

```
docker run --rm --name securecoding -p 3000:3000 securecoding:latest
```

Si tuvieses una cuenta en `dockerhub`, podrías subirla con:

```
docker login  
docker push securecoding:latest
```