TEMA 5.1: DOCKER

-Definición → tecnología de virtualización 'ligera', que utiliza contenedores en vez de maq. virtuales y su objetivo es el despliegue de aplic. encapsuladas en dichos contenedores.

-Evolución: Arquitectura de un único servidor, Virtualización y **Contenedores** (mayor velocidad, mayor portabilidad y mayor eficiencia).

-**Términos frecuentes:**

- **Imagen →** archivo comprimido al que se le añaden capas con los elementos necesarios para poder ejecutar una aplicación en el sistema. No tiene estado ni cambios (salvo actualizaciones).
- **Contenedor →** imagen con instrucciones y variables de entorno que se ejecuta. Los cambios no afectan a la imagen.
- **Docker →** plataforma opensource para empaquetado y distribución de aplicaciones.
- **Docker-engine →** aplicación cliente-servidor que consta de 3 componentes:
    - Servicio docker para ejecución de contenedores.
    - API para que otras aplicaciones puedan comunicarse.
    - Docker CLI para gestionar contenedores, imágenes...
- **Docker Hub →** registro de repositorios de imágenes de la empresa Docker inc.

-COMANDOS:

➔ **sudo docker –version** (ver versión docker)
➔ **docker run docker/whalesay cowsay Hello World** (probamos que funciona sin sudo)

TEMA 5.2: DOCKER CONTENEDORES

1. GESTIÓN Y EJECUCIÓN DE CONTENEDORES.

Necesitamos imágenes. Se hace con:

➔ **docker pull nombre_imagen:version** (se descarga del repositorio una imagen con la versión indicada o la última si no se indica).

➔ **docker run [options] image [command] [arg..]** (para ejecutar contenedores)

options → opciones para el arranque (nombre, ip, puertos, volúmenes, I/O, hostname...
image → imagen base del contenedor que se buscará con docker hub o docker search.
command → orden a ejecutar en el contenedor.
arg → argumentos para la orden a ejecutar.

Pone en ejecución contenedores en base a una imagen de referencia que le indicaremos. Si no tenemos descargada la imagen se descargará automáticamente.

-Opciones vistas en los ejercicios:

docker pull ubuntu:18.04
docker run -it --name nombre_tu_contenedor ubuntu:18.04 /bin/bash
docker run --name nombre_tu_contenedor -p 8080:80 -d nginx
docker run -it -d -p 3306:3306 -e MYSQL_ROOT_PASSWORD=root mariadb

- **-it:** indica que queremos un terminal interactivo (tty). Es útil cuando se necesita interactuar con el contenedor a través de la línea de comandos.
- **-p 3306:3306:** establece un mapeo de puertos entre el puerto 3306 del host y el puerto 3306 del contenedor. MariaDB utiliza el puerto 3306 de forma predeterminada para aceptar conexiones de clientes.
- **-e MYSQL_ROOT_PASSWORD=root:** establece una variable de entorno llamada MYSQL_ROOT_PASSWORD con el valor root. Esto configura la contraseña de root de MySQL dentro del contenedor como "root". Es una práctica común para establecer la contraseña de root al iniciar un contenedor de MariaDB o MySQL.
- **--name nombre_tu_contenedor:** establece un nombre personalizado para el contenedor que se va a crear. En este caso, "nombre_tu_contenedor" es el nombre que se le asignará al contenedor.

Otras opciones:

- **-h o --hostname** para establecer el nombre de red parar el contenedor.
- **--help** para obtener ayuda de las opciones de docker.

- **-d o --detach** para ejecutar un contenedor (normalmente porque tenga un servicio) en background.
- **-e o --env** para establecer variables de entorno en la ejecución del contenedor.
- **--interactive o -i** para mantener la STDIN abierta en el contenedor.
- **--ip** si quiero darle una ip concreta al contenedor.
- **--net o --network** para conectar el contenedor a una red determinada.
- **-p o --publish** para conectar puertos del contenedor con los de nuestro host.
- **--restart** que permite reiniciar un contenedor si este se "cae" por cualquier motivo.
- **--rm** que destruye el contenedor al pararlo.
- **--tty o -t** para que el contenedor que vamos a ejecutar nos permita un acceso a un terminal para poder ejecutar órdenes en él.
- **--user o -u** para establecer el usuario con el que vamos a ejecutar el contenedor.
- **--volume o -v** para montar un bind mount o un volumen en nuestro contenedor.
- **--wordirk o -w** para establecer el directorio de trabajo en un contenedor.

2. GESTIÓN Y EJECUCIÓN DE CONTENEDORES.

Una vez tenemos un contenedor corriendo, vamos a ejecutar órdenes dentro de este usando el comando **docker exec.** También podemos copiar archivos del contenedor a la máq. anfitriona y viceversa.

- ➔ **docker exec -it web /bin/bash** (obtener un terminal en un contenedor que ejecuta servidor Apache(httpd) y que se llama web)
- ➔ **docker exec web ls /usr/local/apache2/htdocs** (mostrar contenido de esa carpeta del contenedor llamado web)
- ➔ **docker cp prueba.html web:/usr/local/apache2/htdocs/index.html** (copiar un fichero 'prueba.html' al fichero 'web:/...' de un contenedor llamado web)
- ➔ **docker cp web:/usr/local/apache2/htdocs/index.html $HOME/test.html** (copiar el fichero index.html que se encuentra en '/usr/..' de un contenedor llamado web en un fichero llamado 'test.html' en el directorio HOME de la máq. anfitriona.)

3. OBTENCIÓN DE INFORMACIÓN DE CONTENEDORES.

- ➔ **docker ps** (mostrar contenedores en ejecución)

- ➔ **docker ps -a** (mostrar todos los contenedores)
- ➔ **docker ps -a -s** (añadir información del tamaño)
- ➔ **docker ps -a -l** (mostrar información del último contenedor creado)
- ➔ **docker ps –filter publish=8080** (mostrar contenedor según puerto)
- ➔ **docker inspect (id/nombre_contenedor)** (para obtener más información)
- ➔ **docker logs (id/nombre_contenedor)** (para obtener más información sobre un contenedor que está fallando)
- ➔ **docker stop (id/nombre_contenedor)** (parar contenedor)
- ➔ **docker rm (id/nombre_contenedor)** (eliminar contenedor)
- ➔ **docker start (id/nombre_contenedor)** (iniciar contenedor que estaba parado previamente)
- ➔ **docker restart (id/nombre_contenedor)** (reiniciar contenedor que previamente estaba en ejecución)

TEMA 5.3: DOCKER IMÁGENES

1. DESCARGA IMÁGENES.

- ➔ **docker pull nombre_imagen:version** (se descarga una imagen con la versión indicada o la última si no se indica). (Recuerda que si ejecutas un repositorio y la imagen no está descargada lo hará automáticamente).

2. LISTADO IMÁGENES.

- ➔ **docker images** (ver listado). Se obtiene:
  - ◆ Repository → nombre de la imagen en el repositorio
  - ◆ Tag → versión de la imagen descargada
  - ◆ Imagen Id → identificador único
  - ◆ Created → cuando se creó
  - ◆ Size → tamaño

3. BORRAR IMÁGENES.

- ➔ **docker rmi (id/nombre)**
- ➔ **docker image rm (id/nombre)**
- ➔ **docker rmi -f (id/nombre)** (para forzar borrado)
- ➔ **docker image prune -a/--all** (borrar imágenes no usadas por el contenedor)

➔ **docker image prune -f/- -force** (para que no nos solicite información, nos puede borrar muchas a la vez)
➔ **docker image prune --filter** (para especificar filtros)

Dos ejemplos:
Borrar todas las imágenes sin usar: ***docker image prune -a***
Borrado de la imágenes creadas hace más de 10 días: ***docker image prune --filter until="240h"***

4. OBTENCIÓN INFORMACIÓN IMÁGENES.

Tenemos dos fuentes:

- **Página Docker Hub**, que recoge información como:
  - Descripción o servicio de la imagen.
  - Lista de versiones TAGs disponibles.
  - Variables de entorno.
  - Como ejecutar la imagen.
- **docker image inspect / docker inspect**
  - el id y el checksum de la imagen,
  - los puertos abiertos,
  - la arquitectura y el sistema operativo de la imagen,
  - el tamaño de la imagen, los volúmenes, las capas...

5. OTROS COMANDOS.

➔ **docker image build** (construir una imagen desde un fichero Dockerfile)
➔ **docker image history** (mostrar la evolución de la imagen)

TEMA 5.4: DOCKER (DATOS CONTENEDORES, VOLÚMENES)

1. DATOS CONTENEDORES (persistencia de los datos).

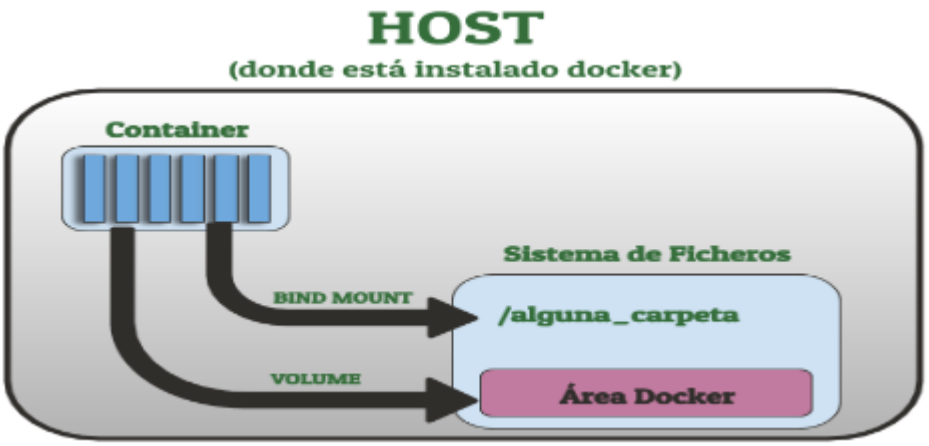
A tener en cuenta:

- Los datos, configuraciones y archivos de un contenedor mueren con él.

- Los datos de los contenedores no se mueven fácilmente ya que están fuertemente acoplados con el host en el que el contenedor está ejecutándose.
- Escribir en los contenedores es más lento que escribir en el host ya que tenemos una capa adicional.

Para persistir datos de contenedores, Docker nos ofrece varias soluciones:

- Los volúmenes docker.
- Los bind mount.



- ☐ <u>VOLÚMENES:</u> los datos de los contenedores que decidamos almacenar en una parte del sistema de ficheros gestionada por docker (solo docker tiene acceso por los permisos que tiene)
  Esta zona reservada de docker cambia según el SO y forma de instalación.
  - Si estamos en Linux es: **/var/lib/docker/volumes**
    - **Solo podemos acceder con el superusuario (su)**

  Estos volúmenes se usan para:
  - Para compartir datos entre contenedores. Tendrán que usar el mismo volumen.
  - Para copias de seguridad ya sea para que sean usadas posteriormente por otros contenedores o para mover esos volúmenes a otros hosts.
- ☐ <u>BIND MOUNT:</u> mapeamos una parte del sistemas de ficheros del host, de la que normalmente yo tengo el control, con una parte del sistema de ficheros del contenedor.
  Este mapeado me permite:
  - Compartir ficheros entre el host y los contenedores

- Que **otras aplicaciones que NO sean docker tengan acceso a esos ficheros**, ya sean código, ficheros...

El **bind mount es el mecanismo que vamos a preferir para la fase de desarrollo** ya que:

• Las aplicaciones que podrán acceder a esos ficheros serán los IDEs o editores de código.

• Estaremos modificando con aplicaciones locales código que a la vez se encuentra en nuestro equipo y en el contenedor.

• Y desde mi propio equipo estaré probando ese código en el entorno elegido, o en varios entornos a la vez sin necesidad de tener que instalar absolutamente nada en mi sistema.

2. GESTIÓN DE VOLÚMENES Y OBTENCIÓN DE INFORMACIÓN.

☐ CREACIÓN DE VOLÚMENES → **docker volume create [options] [volume]**

- ➔ options → opciones para el volumen (driver, metadatos y opciones para el driver).
- ➔ volume → nombre del volumen (si no se especifica se le dará como nombre un ID).

Entre las opciones que podemos incluir a la hora de crear los volúmenes están:

- **--driver o -d** (especifica el driver elegido para el volumen, si no especificamos nada el driver utilizado es el local que es el que nos interesa desde el punto de vista de desarrollo porque desarrollamos en nuestra máquina)
- **--label** (para especificar los metadatos del volumen mediante parejas clave-valor.)
- **--opt o -o** (para especificar opciones relativas al driver elegido.)
- **--name** (para especificar un nombre para el volumen. Es una alternativa a especificarlo al final)

Algunos ejemplos:
Creación de un volumen llamado data: **docker volume create data**
Creación de un volumen data especificando el driver local: **docker volume create -d local data**

☐ ELIMINAR VOLÚMENES: Dos opciones:

- ➔ **docker volume rm id/nombre** (para eliminar un volumen en concreto)

- ➔ **docker volume prune** (para eliminar volúmenes que no están siendo usados por ningun contenedor)

☐ OBTENCIÓN DE INFORMACIÓN:

- ➔ **docker volume ls** (lista de volúmenes creados e información: driver usado, nombre y si no tiene, se muestra ID)
- ➔ **docker volume inspect** (información más detallada: fecha de creación, tipo driver, etiquetas, nombre, opciones y ámbito)

3. ASOCIACIÓN DE ALMACENAMIENTO A UN CONTENEDOR.

Para usar los volúmenes y bind mounts en los contenedores usaremos dos flags de la orden docker run:

- **docker run --volume/-v** (lo utilizaremos para establecer bind mounts)
- **docker run --mount.** (para establecer bind mounts y para usar volúmenes previamente definidos)

Cuando usamos volúmenes o bind mount el contenido de lo que tenemos sobreescribirá la carpeta destino en el sistema de ficheros del contenedor en caso de que exista.

Si no existe y hacemos un bind mount esa carpeta se creará pero lo que tendremos en el contenedor es una carpeta vacía (tener cuidado porque las carpetas pueden tener datos y configuraciones).

Si usamos imágenes de Docker Hub, debemos leer la información que cada imagen nos proporciona en su página ya que esa información suele indicar cómo persistir los datos de esa imagen, ya sea con volúmenes o bind mounts, y cuáles son las carpetas importantes en caso de ser imágenes que contengan ciertos servicios (web, base de datos etc...).

Algunos ejemplos:

- ➔ docker run --name apache -v /home/usuario/web:/usr/local/apache2/htdocs -p 80:80 httpd (la carpeta web será el directorio raíz del servidor apache)
- ➔ docker run --name apache -p 80:80 --mount type=**bind**,src=/home/usuario/web,dst=/usr/local/apache2/htdocs httpd (la carpeta web será el directorio raíz del servidor apache, se crea si no existe)
- ➔ docker run --name apache -p 80:80 --mount type=**volume**,src=Data,dst=/usr/local/apache2/htdocs httpd (mapear un volumen previamente creado y que se llama Data en la carpeta raiz del servidor)

➔ docker run --name apache -p 80:80 --mount type=**volume**,dst=/usr/local/apache2/htdocs httpd (igual que el anterior, pero al no tener src(el nombre del volumen) se crea uno automáticamente)

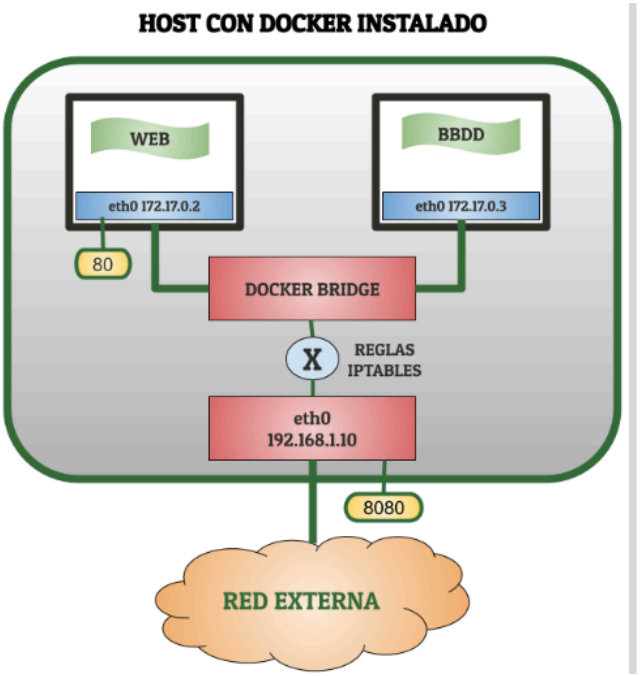
TEMA 5.5: DOCKER REDES

1. TIPOS DE REDES EN DOCKER.

➔ BRIDGE → es el driver por defecto. Mi equipo actúa de puente del contenedor con el exterior y como medio de comunicación entre distintos contenedores (en ejecución) de una misma red.
➔ HOST → el contenedor usa la red de la máquina.
➔ OVERLAY → conecta distintos servicios docker de máquinas diferentes. (para docker Swarm).
➔ MacVlan → permite asignar un MAC a nuestro pc.
➔ NONE → si queremos que le contenedor no tenga conectividad ninguna.

BRIDGE: me permite dentro de la máquina local:

- **Aislar** los contenedores de distintas subredes docker (solo podremos acceder a los equipos de esa misma subred)
- **Aislar** los contenedores de acceso **exterior.**
- **Publicar servicios** con las pertinentes reglas de iptables.



2. GESTIÓN DE REDES.

Hay que hacer una diferenciación entre **dos tipos de redes bridge**:

- La creada por defecto por docker.
- La definida por nosotros.

➔ **docker network ls** (muestra todas las redes docker) y nos muestra:

-El NETWORK ID (identificador de red)

-El DRIVER (tipo de red que voy a 'conectar' a los contenedores)

-El SCOPE (ámbito de nuestras redes).

DIFERENCIAS CON LAS REDES BRIDGE CREADAS POR NOSOTROS:

- Puedo conectar en caliente a los contenedores redes 'bridge' definidas por mi, de la otra forma tendría que parar el contenedor.
- Gestión de manera más segura del aislamiento.
- Más control en la configuración de las redes.
- Los otros, pueden provocar más conflictos con el uso de ciertas variables de entorno.

Algunos ejemplos:

➔ **docker network create red1** (Crear una red por defecto)

➔ **docker network create -d bridge --subnet 172.24.0.0/16 --gateway 172.24.0.1 red2** (Crear una red (red2) dándole el driver bridge (-d), dirección y máscara de red (--subnet) y una gateway).

➔ **docker network rm red1** (Eliminar la red1)

➔ **docker network rm 3cb4100fe2dc** (Eliminar red con su ID)

➔ **docker network prune** (Eliminar redes que no tengan contenedores asociados)

3. ASOCIACIÓN DE REDES A CONTENEDORES.

- Si al arrancar un contenedor no especificamos una red, por defecto será bridge.
- No puedo conectarlo inicialmente a más de una red.
- Cuando se crea el contenedor, puedo conectarlo a más de una red o desconectarlo, dependiendo si es por defecto no.

Algunos ejemplos:

➔ **docker run -d --name web -p 80:80 httpd** (**Arrancar** un contenedor de Apache **sin especificar red** y con conexión del exterior del puerto 80. Por defecto a la red bridge)

➔ **docker run -d --name web2 --network red1 -p 8080:80 httpd** (**Arrancar** un contenedor de Apache **conectándose a la red1** definida por el usuario y con conexión del exterior a través del puerto 8080)

➔ **docker run -d --name web2 --network red1 --ip 172.18.0.5 -p 8181:80** httpd (**Arranchar** un contenedor de Apache conectándose a la red **red1** dándole una **ip** (que debe pertenecer a esa red))

➔ **docker network connect red2 web2 (Conectar una nueva red(red2) al contenedor web2)**

➔ **docker network disconnect red1 web2** (**Desconectar la red1 del contenedor web2** (debe estar funcionando para desconectarse) **)**

Docker connect tiene más opciones que permiten, por ejemplo:

- **--dns** (para establecer unos servidores DNS predeterminados.)
- **--ip6** (para establecer la dirección de red ipv6)
- **--hostname o -h** (para establecer el nombre de host del contenedor. Si no lo establezco será el ID del mismo.)

TEMA 5.6: DOCKER

1. CONTRUCCIÓN DE IMÁGENES.

OBJETIVO DE LA PERSONALIZACIÓN→ distribuir imágenes para que puedan ser usadas por cualquier sistema con docker instalado.

La **personalización** para conseguir nuestras propias imágenes se hacen de **2 maneras**:

1.1 DESDE UN CONTENEDOR EN EJECUCIÓN.

1. Utilizando (a partir de un fichero): arrancamos contenedor, modificamos en el contenedor, **docker commit**(creas nueva imagen) **/docker save**(guardas imagen con fichero.tar) **/docker load**(cargas el fichero.tar como imagen)

2.Utilizando (a través de DockerHub): arrancamos contenedor, modificamos en el contenedor, **docker commit**(creas nueva imagen), nos identificamos en Docker Hub con docker Login y **/docker push** (distribuimos ese fichero a dockerHub)

Algunos ejemplos (EN EL TEMA)

1.2 A PARTIR DE UN FICHERO DOCKERFILE.

INCONVENIENTES CREACIÓN IMÁG. A PARTIR DE CONTENEDORES:

1. No se puede reproducir la imagen
2. No podemos cambiar la imagen base

Frente a esto, se prefiere el uso de ficheros **DockerFile**.

1. Crear el fichero DockerFile.
2. **docker build** (construir la nueva imagen)
3. **docker login** (autentificarse en Docker Hub)
4. **docker push** (distribuir imagen subiéndose a Docker Hub)

UN FICHERO DOCKERFILE → conjunto de instrucciones que serán ejecutadas de manera secuencial para construir una imagen docker. Cada instrucción crea una capa. Nos permite ver esta estructura de capas de las imágenes **docker pull.**

ÓRDENES FICHEROS DOCKERFILE MÁS COMUNES:

- **FROM →** para especificar la imagen sobre la que construyo la mía **(FROM php:7.4-apache)**

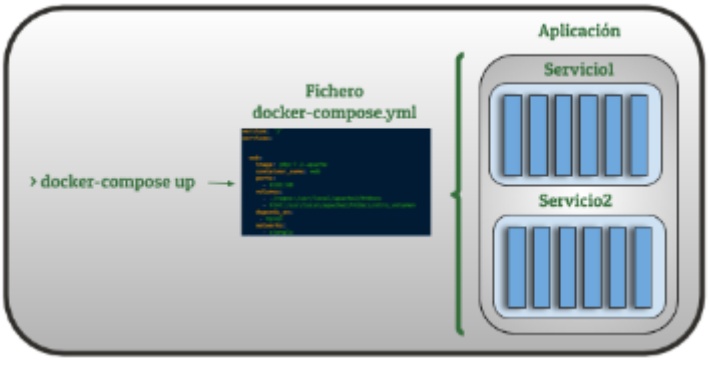
- LABEL → para añadir metadatos clave-valor (LABEL company=iesfer)
- COPY → copiar ficheros(misma carpeta) desde mi equipo a la imagen (COPY --chown=www-data:www-data myapp /var/www/html)
- ADD → especifica url y archivos comprimidos
- RUN → ejecuta una orden creando una capa (RUN apt update && apt install -y git.) (-y para que no haya interacción con el usuario en la construcción)
- WORKDIR → establece un directorio de trabajo (WORKDIR /usr/local/apache/htdocs)
- EXPOSE → da información de los puertos (EXPOSE 80)
- USER → especificar usuario (USER usuario)
- ENV → establecer variables de entorno (ENV WEB_DOCUMENT_ROOT=/var/www/html)

USO DE DOCKER BUILD Algunos ejemplos (EN EL TEMA)

TEMA 5.7: DOCKER-COMPOSE

1. APLICACIONES MULTICAPA CON DOCKER-COMPOSE

DOCKER-COMPOSE → nos permite desplegar grupo de contenedores que forman parte de una misma aplicación o entorno.



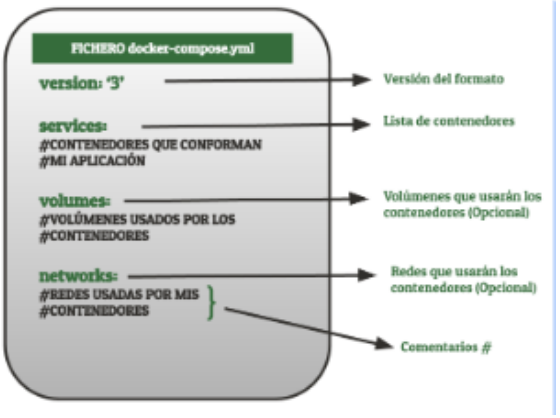
Los pasos son los siguientes:

1. Describir de manera declarativa todo los contenedores que conforman mi aplicación en el fichero **docker-compose.yml**. Este fichero tiene formato YAML.
2. Al ejecutar **docker-compose up** se levanta toda la aplicación, es decir, todos los contenedores que la conforman.

1.1 INSTALACIÓN DOCKER-COMPOSE

1.2 ARCHIVO DOCKER-COMPOSE.YML

Es un fichero YAML que contiene las instrucciones para crear y configurar los servicios que van a constituir mi aplicación.



Algunos ejemplos (EN EL TEMA)

1.3 LA ORDEN DOCKER-COMPOSE

Creado el archivo docker-compose.yml, empezamos a crear los contenedores, mediante el comando docker-compose (que debemos ejecutar en el directorio donde está el archivo correspondiente).

Algunos ejemplos (EN EL TEMA)