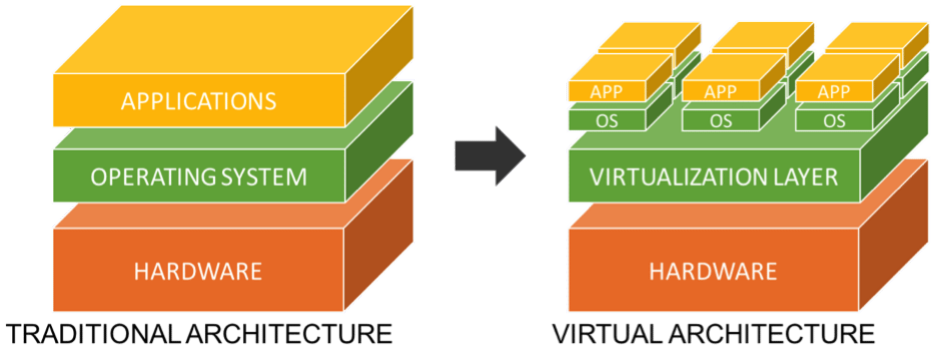
1.1 Tecnología de contenedores

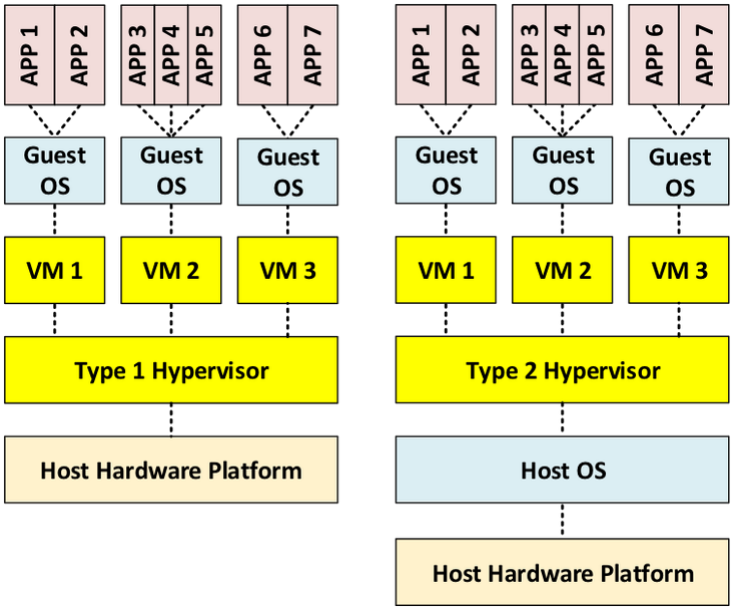
Virtualización tradicional

- Es una abstracción de la capa de hardware
- Despliegues más sencillos
- Ahorro costes
- Aislamiento
- ...



Hypervisor

- Es un monitor que orquesta el acceso de varios SO a los recursos de un servidor físico.



Hypervisor type 1

- Nativo o Bare Metal Hypervisor
- Corre directamente en el hardware de la máquina, hacen la función de HAL (Hardware Abstraction Layer)
- Ej: VMWare ESXI, Microsoft Hyper-V, Citrix/Xen Server

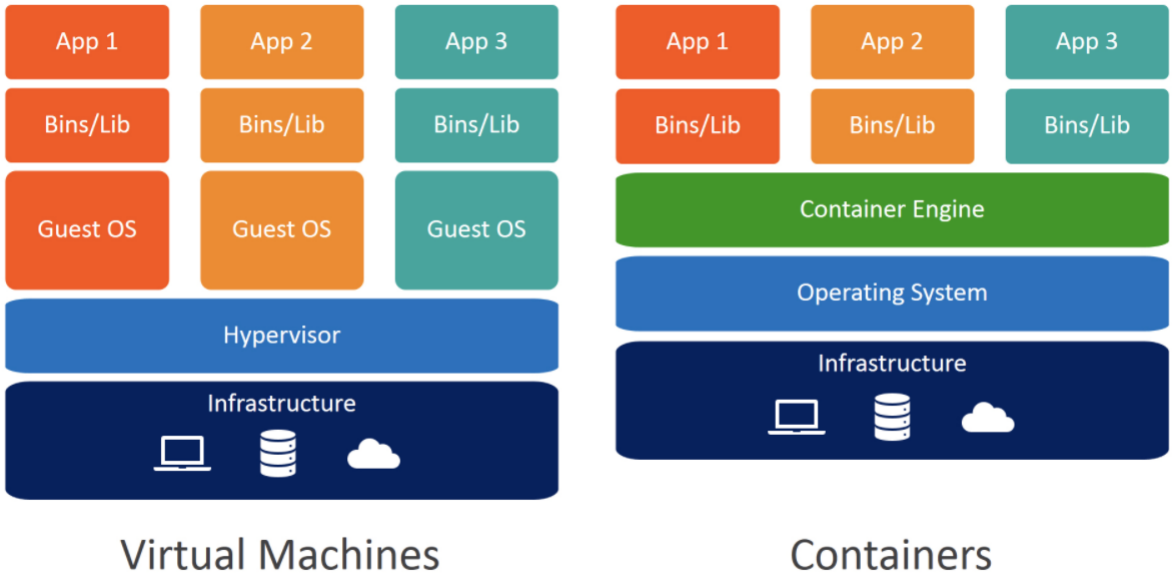
Hypervisor type 2

- Host OS Hypervisor.

- Corre sobre el sistema operativo, como una aplicación más.
- Ej: VMware Workstation, VMware Player, VirtualBox, Parallel Desktop (MAC)
- No es adecuado cuando hay un workload elevado: Active Directory, bbdd...
- Adecuados para entornos de test
    - Más baratos
    - Instalación más sencilla

Contenedores

- Son una abstracción de la capa de aplicación



Ventajas contenedores

- Más ligeros
    - Portabilidad
        - Contrato entre el sysadmin y el developer
        - Despliegues más rápidos
    - Mas eficientes -> menor coste
- Son efímeros

Desventajas

- Menor seguridad y aislamiento
    - Depende del host
- Snapshots
- Migraciones en caliente (VMWare vMotion)
- Son efímeros

Cómo se consigue

- Usando características del kernel de Linux como:

- Chroot y CGroups
- Kernel namespaces
- Apparmor
- Union Filesystem

chroot y cgroups

- Chroot: Un mecanismo para cambiar el directorio raíz de un proceso y sus hijos, y por tanto ejecutarlo en un entorno enjaulado (1999 - FreeBSD)
- CGroups:
  - En el kernel de Linux desde 2008.
  - Permiten agrupar procesos (control groups) compartiendo memoria, CPU y sistema de archivos.

kernel namespaces

- Docker crea un conjunto de namespaces diferente para cada contenedor
  - Proporciona una capa de aislamiento
- Se utilizan los siguientes namespaces de Linux:
  - PID namespace: procesos
  - NET namespace: interfaces de red
  - IPC namespace: recursos IPC (Inter-Process Communication)
  - MNT namespace: puntos de montaje
  - UTS namespace (Unix Time-Sharing): hostname y domain

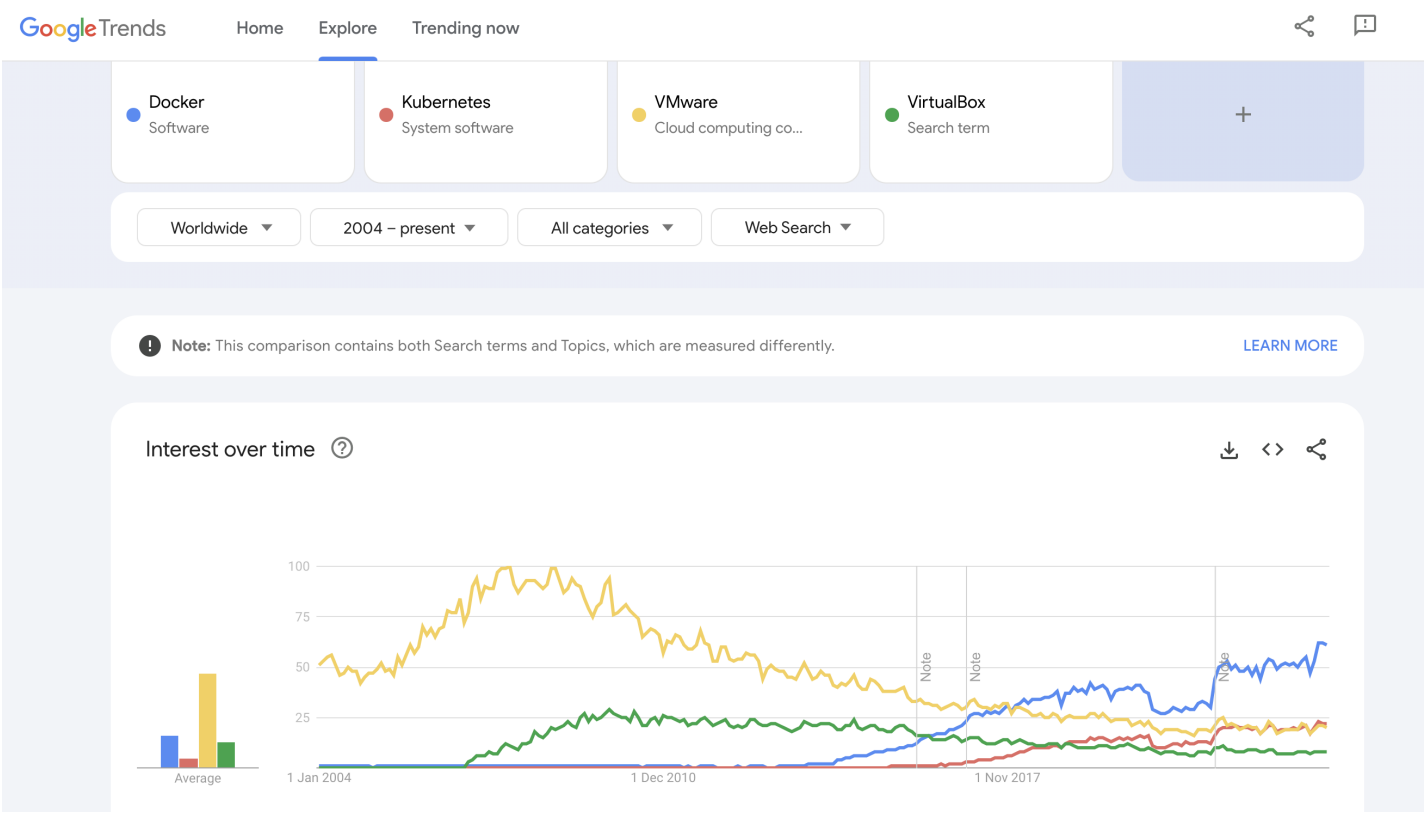
AppArmor

- Módulo de seguridad del kernel de Linux que proporciona control de acceso para confinar programas a un conjunto limitado de recursos
- Se pueden definir 2 tipos de perfiles:
  - **enforcement**: impiden la acción y además reportan el intento
  - **compliance**: únicamente se registra el acceso.
- Docker crea un perfil AppArmor llamado **docker-default** y lo carga en el kernel.
  - Es un perfil de protección moderada, a la vez que proporciona la máxima compatibilidad de aplicaciones.
  - Ver la plantilla a partir de la cual se crea la plantilla

Union Filesystem

- Se montan varias fuentes de ficheros en directorios comunes
  - Por ej al montar un CD o un servicio NFS remoto sobre un /home
  - Es una característica disponible en muchos tipos de sistemas de ficheros

Google Trends



Cloud Native Computing Foundation

- Organización sin fines de lucro que se dedica a impulsar la adopción de tecnologías de código abierto para la computación en la nube.
- Se crea en el año 2015
- Fundadores: Google, CoreOS, Mesosphere, Red Hat, Twitter, Huawei, Intel, Cisco, IBM, Docker, Univa, and VMware.
- Explorar CNCF

Casos de uso

1. **Desarrollo y despliegue de aplicaciones**:
   - Entorno consistente y reproducible para desarrolladores
   - Es un contrato entre el developer y el administrador de sistemas
2. **Escalado Horizontal**:
   - Son ideales para implementaciones escalables, ya que pueden crearse y eliminarse rápidamente según la demanda.
   - Permite manejar picos de tráfico y cargas de trabajo variables.
3. **Microservicios**:
   - Cada componente de la aplicación se empaqueta y ejecuta de manera independiente en su propio contenedor.
   - Esto facilita la modularidad, el mantenimiento y la escalabilidad de las aplicaciones.
4. **Entornos de Pruebas y QA**:
   - Los contenedores proporcionan entornos de pruebas aislados y reproducibles que pueden ser fácilmente creados y desechados según sea necesario.

- Esto facilita las pruebas de integración, las pruebas de regresión y la verificación de calidad de las aplicaciones.

5. **CI/CD (Continuous Integration/Continuous Deployment)**:
    - Son una parte fundamental de las pipelines de CI/CD, ya que permiten construir, probar y desplegar aplicaciones de manera automatizada y consistente en diferentes entornos.

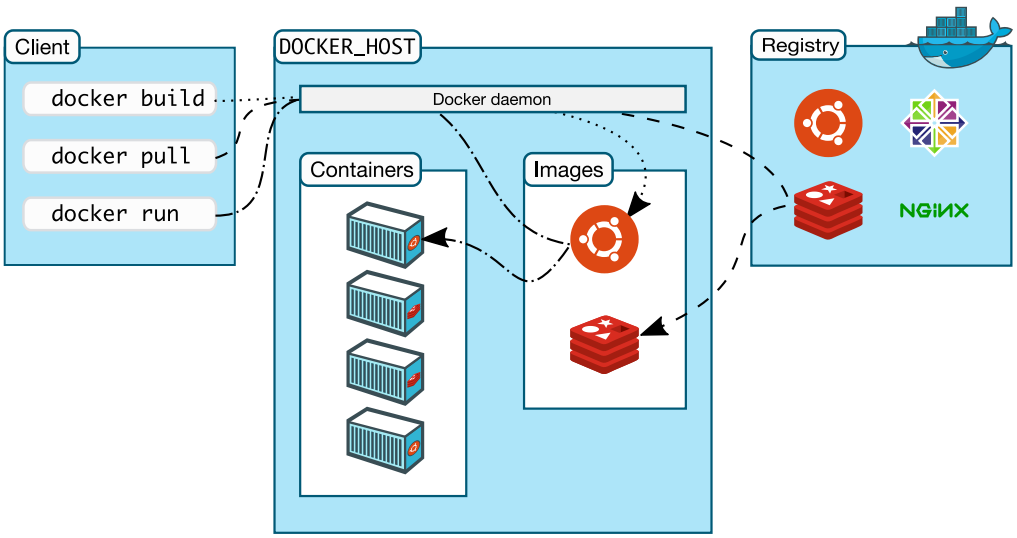
1.2 Docker intro

Qué es Docker

- Herramienta **open-source** que nos permite realizar una **virtualización ligera
- Permite **empaquetar entornos y aplicaciones** que posteriormente podremos **desplegar** en cualquier sistema que disponga de esta tecnología

Arquitectura Docker

- Es una arquitectura **cliente-servidor**
  - El servidor es el daemon (container engine) al que se accede mediante una **API REST**
  - Existen SDKs y clientes de la API para distintos lenguajes
  - El cliente habitual es el comando **docker**



- Por defecto usa **UNIX sockets**:
  - Comunicación entre procesos de la misma máquina
    - Archivo /var/run/docker.sock
  - Se maneja por el kernel
- Podemos configurarlo para que use **TCP**, cambiar el driver de almacenamiento, opciones de red... Ver documentación de dockerd

Componentes principales de Docker

- Los conceptos principales en Docker son las **imágenes**, los **contenedores**.
- Hay otros conceptos relacionados como **volúmenes**, **redes**, **servicios**, **Dockerfile**, **Docker Hub** que están relacionados y los iremos viendo a la vez.

Imagen

- Plantilla que define todas las dependencias de mi aplicación.
- Es habitual que las imágenes se creen en base a otras (herencia).
- Se definen en un fichero llamado `Dockerfile`

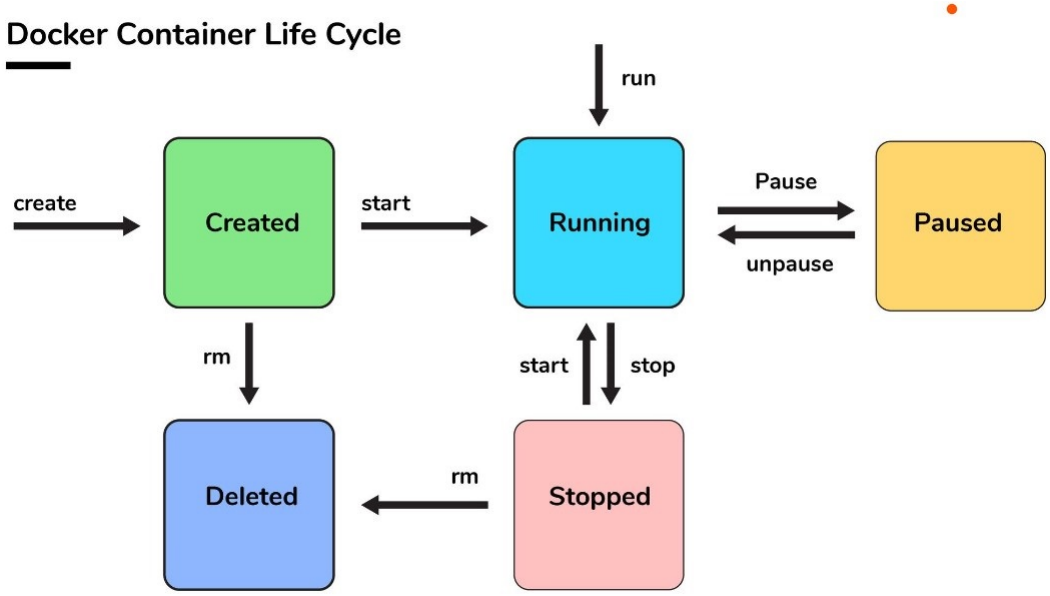
```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y nginx
CMD ["nginx", "-g", "daemon off;"]
EXPOSE 80
```

El comando nginx -g daemon-off permite arrancar nginx con una directiva global, daemon-off que indica que no arranque nginx en modo daemon, sino attached en la sesión de terminal

Contenedor

- Instancia ejecutable de una imagen
- Son **efímeros**, la persistencia se logra mediante el uso de **volúmenes**
- Cada contenedor se ejecuta en un entorno aislado (podemos controlar el nivel de aislamiento):
  - variables de entorno
  - volúmenes montados
  - Interfaces de red
- Podemos crear también una imagen a partir de un estado del contenedor.

Ciclo de vida de un contenedor



Redes

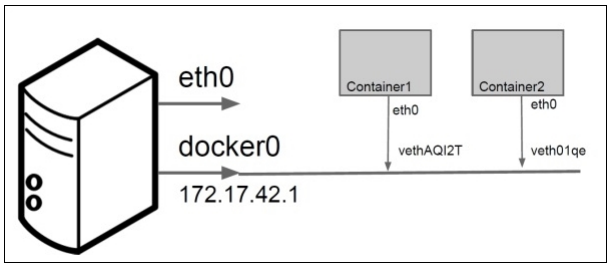
- Internamente usa las regla de enrutado del servidor
- En Linux mediante IPtables
- Es un sistema abierto, mediante el uso de drivers.

Bridge Network (Red puente)

- Es la red predeterminada en Docker.
- Cada contenedor en una red de puente tiene su propia dirección IP.
- Permite la comunicación entre contenedores en el mismo host.

    - Por IP
    - Por nombre de contenedor
    - Docker se encarga de todo :-)
- Ejecutar comandos:

```
ip  address
docker network ls
docker network inspect bridge
```



- ¿Cómo se accede al exterior?

```
$ sudo iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT) target prot opt
source destination MASQUERADE all -- 172.17.0.0/16
!172.17.0.0/16
...
```

Host Network (Red de host)

- En esta configuración, los contenedores comparten la misma red y espacio de red que el host.
- Los contenedores tienen acceso directo a los puertos del host sin necesidad de reenvío de puertos.
- Esto puede mejorar el rendimiento para algunas aplicaciones, pero puede haber conflictos de puertos si múltiples contenedores intentan usar los mismos puertos.

Overlay Network (Red de superposición)

- Permite la comunicación entre contenedores que se ejecutan en diferentes hosts.
    - Se utiliza principalmente en entornos de clúster o en infraestructuras distribuidas.
    - Utiliza un controlador de red externo (como Docker Swarm o Kubernetes) para administrar la comunicación entre contenedores en diferentes hosts.

Macvlan Network (Red Macvlan)**

- Permite que los contenedores tengan direcciones IP que son visibles en la red física subyacente.
    - Cada contenedor en una red Macvlan tiene una dirección IP propia en la misma subred que el host.
    - Útil cuando necesitas que los contenedores se comporten más como máquinas virtuales o

máquinas físicas en la red.

None Network (Red Ninguna)**

- Este tipo de red no proporciona conectividad de red para el contenedor.
  - Puede ser útil para casos de uso especiales donde no se requiere acceso a la red.

Docker en Linux

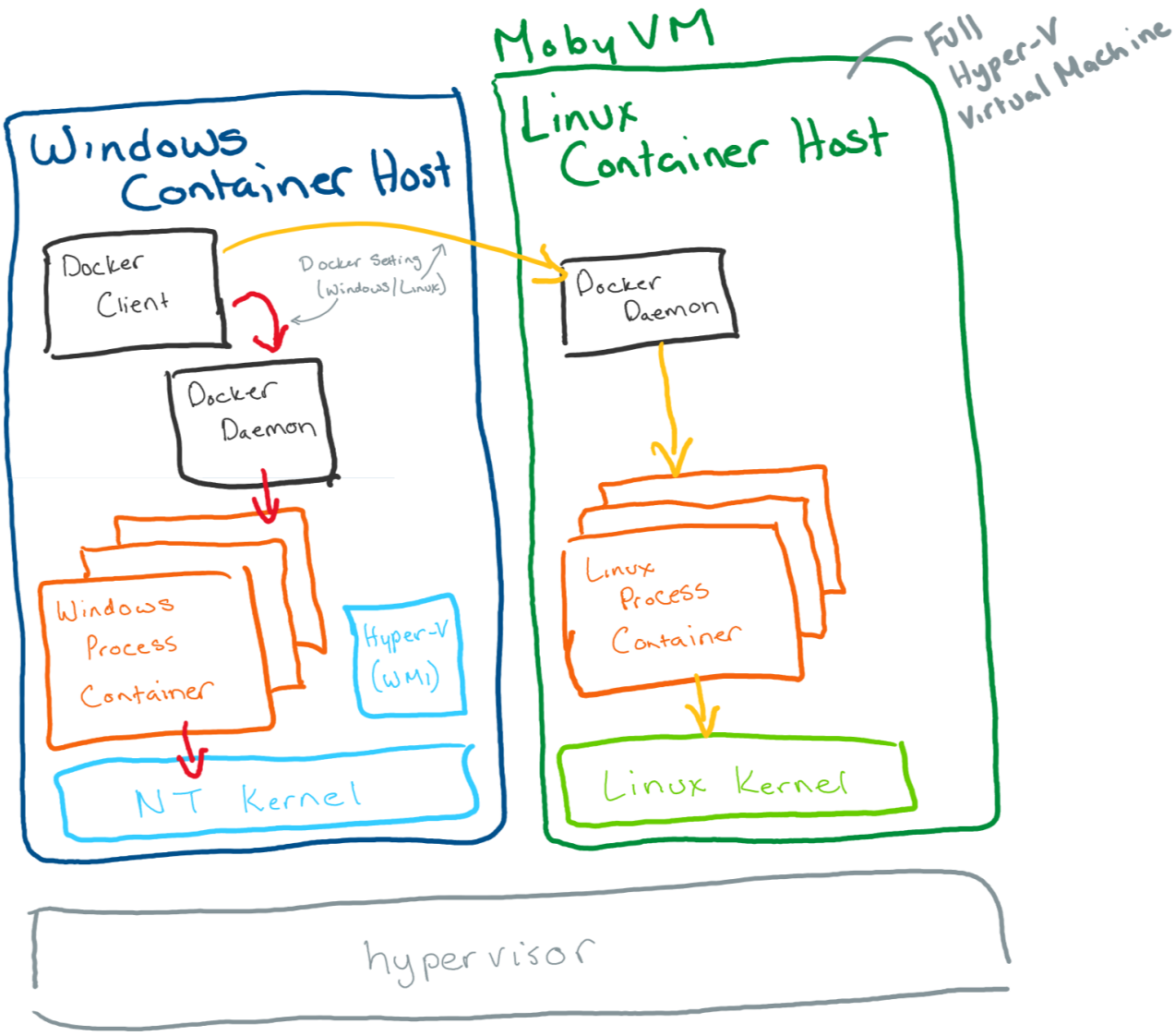
- En Linux Docker no es virtualizado, no hay un hipervisor.
- Los procesos que corren dentro de un contenedor de docker se ejecutan con el mismo kernel que la máquina anfitrión.
- Linux aisla los procesos, ya sean los propios de la máquina anfitrión o procesos de otros contenedores.
- Controla los recursos que se le asignan a contenedores pero sin la penalización de rendimiento de los sistemas virtualizados.

Docker en Windows

- El ecosistema de contenedores fundamentalmente utiliza Linux
- Como los contenedores comparten el kernel con el host, no se pueden ejecutar directamente en Windows, hace falta virtualización
- HyperV en Windows 10 pero versiones PRO o ENTERPRISE
- HyperV y VirtualBox no se llevan bien
- Debemos deshabilitarlo mediante el siguiente comando de PowerShell:

```
Disable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V-All
```

  - Y además puede ser necesario reiniciar :-(
- Docker Client se ejecuta en Windows pero llama al Docker Daemon de una máquina virtual Linux

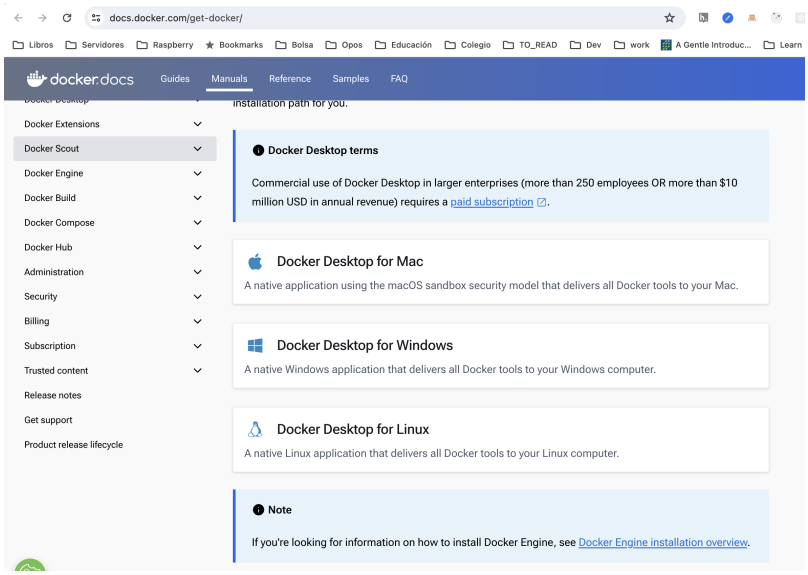

Moby VM
Full Hyper-V Virtual Machine
Windows Container Host
Docker Client
Docker Setting (Windows/Linux)
Docker Daemon
Windows Process Container
Hyper-V (WMI)
NT Kernel
Linux Container Host
Docker Daemon
Linux Process Container
Linux Kernel
hypervisor

1.3 Fundamentos de Docker

Instalación y configuración de Docker

Tipos de instalación

- Ir a la web de Docker
  - Se puede instalar el Docker Desktop en Windows, Mac o Linux
    - Gestiona los recursos del host para los contenedores
    - Ofrece Docker Scout integrado
    - Se comunica con el Docker Engine (se instalan juntos)
    - Ofrece extensiones
  - Se puede instalar el Docker Engine en Linux sin Docker Desktop ya que Docker Desktop en Linux funciona virtualizado



Instalación en Linux

- Seguir documentación oficial de Docker
- El proceso habitual de instalación es:
  - Añadir el repositorio de docker
  - Instalar Docker Engine
  - Configurar usuarios para uso de docker (sin privilegios root). Ver Linux PostInstall

Trabajar con Docker

- Utilizaremos la terminal (cliente docker)
  - Cheat sheet: https://docs.docker.com/get-started/docker_cheatsheet.pdf
- Utilizaremos Visual Studio Code
  - Terminal integrada
  - Extensión Docker

Conceptos básicos de imágenes

- Primero se define la plantilla de la imagen mediante un fichero Dockerfile con las instrucciones:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y nginx
CMD ["nginx", "-g", "daemon off;"]
EXPOSE 80
```

- Se construye la imagen:

```
docker build -t nginx-image .
```

- Observamos que aparece con un repositorio llamado igual que la imagen y un TAG latest

```
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
nginx-image  latest    794ad060cc41   15 minutes ago 122MB
```

- Podríamos haber dado una versión a la imagen:

```
docker build -t nginx-image:4.2 .
```

- Observamos que ambas tienen el mismo hash. Si borramos por hash dará error al haber más de un repositorio con la misma imagen (mismo hash). Debemos borrar por repositorio o forzar el borrado.

  ```
  docker images
  docker rmi <image-id>
  docker rmi nginx-image:latest
  docker images
  ```

- Buenas prácticas al construir imágenes: https://docs.docker.com/develop/develop-images/instructions/
  - Por ejemplo, hay que tener en cuenta la caché en el Dockerfile y liberar espacio:

```
    RUN apt-get update && apt-get install -y \
  package-bar \
  package-baz \
  package-foo  \
  && rm -rf /var/lib/apt/lists/*
```

Docker Hub y descarga de imágenes públicas

¿Qué es Docker hub?

- Registro oficial de imágenes proporcionado por Docker: Docker Hub
  - Repositorio de **imágenes oficiales de Docker**, de alta calidad.

    - Repositorio de **imágenes verificadas publicadas por terceros**.
    - Podemos crear una imagen de cero pero lo normal es usar o partir de una ya creada.
        - ¡Seguro que alguien ha preparado una imagen de nginx antes que nosotros!
- También podemos crear nuestro propio servicio.
- Ofrece los siguientes beneficios:
    - **Gestor de equipos y organizaciones**: acceso a repositorios privados.
    - **Autobuilds**: crea nuevas versiones de imágenes en base a cambios en repos de Github/Bitbucket
    - **Webhooks**: Ejecuta acciones después para integrar DockerHub con otros servicios

Limitaciones descargas

- Desde final del 2020 Docker impuso limitaciones en el uso de su registro.
    - 100 descargas de imágenes cada 6 horas para usuarios anónimos (por IP)
    - 200 descargas de imágenes cada 6 horas para usuarios autenticados
    - Cuentas **pro** y **team** para aumentar los límites.
- Conclusión: ¡¡¡Debemos hacer ***docker login***!!!

Autenticación

- Crear cuenta en Docker Hub
- Hacer login desde consola

```
docker login
```

- Se crea un fichero de configuración en *$HOME/.docker/config.json*
- Las siguientes veces que nos autentiquemos, al hacer *docker login* leerá directamente el fichero
- Observa que si cerramos la sesión la entrada *auths* del fichero *config.json* queda vacía

Descarga de imágenes de Docker Hub

- Buscamos algún nginx y vemos que hay un repo oficial.
- La versión actual es la 1.25.5 y hay opciones basadas en Debian y Alpine. Con o sin soporte a Alpine o Open Telemetry (soporte API para recogida de métricas y logs)

```
docker pull nginx:1.25.5
docker pull nginx:1.25.5-alpine
docker pull nginx:latest
```

- Es habitual que haya imágenes basadas en:
    - Debian
        - más universal
        - usa glibc (librería estándar de C)
    - Alpine
        - Menor peso

    - Menor superficie de ataque.
    - Usa musl (librería estándar de C)
- Observa que está disponible el Dockerfile, lo puedes analizar. Lo más importante es que las imágenes suelen tener un fichero que se suele llamar ***docker-entrypoint.sh*** con la lógica de arranque.

```
FROM ubuntu
ENTRYPOINT ["echo", "Hello,"]
CMD ["world"]
```

Si lo ejecutamos:

```
docker run my_image_name you  # Output: Hello, you
```

Versiones latest

```
docker images nginx:latest
REPOSITORY   TAG       IMAGE ID       CREATED      SIZE
nginx        latest    786a14303c96   6 days ago   193MB
```

Puede que la versión que yo tengo en local como latest, quizá no sea ya la más actual. Y... ¿qué versión de nginx tengo?
Opciones:

- Arrancar la imagen (contenedor) y usar línea de comandos para ver la versión
- Comprobar el hash con los que hay en docker hub
- Inspeccionar la imagen

```
docker image inspect nginx:latest
```

Ejecución y gestión de contenedores

- Podemos ejecutar la imagen (contenedor) desde Visual Code o mediante terminal.

```
docker run --name nginx-container1 nginx
# perdemos la consola, CTRL+C para recuperarla pero el contenedor ya no va
docker ps
docker ps -a
# lo borramos para poder arrancarlo de nuevo
docker rm nginx-container1
docker ps -a # ya no aparece
```

- Añadimos -d para recuperar la consola:

```
docker run --name nginx-container1 -d nginx
docker run --name nginx-container2 -d nginx
docker ps
```

Visualización del contenido del contenedor

Comprobación del contenido del contenedor

- Accedemos al contenedor vía Visual Code o consola

```
docker exec -it nginx-container1 bash

root@ca11087f8c0f:/# cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      ca11087f8c0f

root@8bcb7243d307:~# ls /usr/share/nginx/html/
50x.html  index.html
```

para ver la ip también podíamos haber utilizado el comando ip

```
# apt install iproute2
# ip address
ping <ip-container>
```

Comunicación entre contenedores

```
  docker exec -it <container-name> /bin/bash
  root@8bcb7243d307:~#apt update
  root@8bcb7243d307:~#apt install iputils-ping
```

Acceso vía web al contenedor

- Si estamos en Linux, el host tiene una ip en la misma red y podríamos acceder vía `http://<ip-contenedor>`
- Lo más normal sería redirigir puertos a la máquina host:

```
# paramos los contenedores
$ docker stop nginx-container1
$ docker stop nginx-container2
```

```
# los borramos
$ docker rm nginx-container1
$ docker rm nginx-container2

# los volvemos a crear poniendo el puerto 80 de cada uno al 80 y 81 de la
# máquina local para acceder vía localhost
$ docker run --name nginx-container1 -p 80:80 -d nginx
88f394b8c9c7b68258869b2b5d38083c6f31c791d1325a4acb45693a3a444606
$ docker run --name nginx-container2 -p 81:80 -d nginx
c7a827434d36d1dd36df041133207a8e466fbf648c23fdc0c722a59e8e75a471
```

- Otra opción es arrancar el contenedor en modo host:

```
docker run  --network host -d nginx
# ¿Qué sucede si lo  arrancas dos veces?
docker run  --network host -d nginx
```

Modificación ficheros

- Modificar fichero `/usr/share/nginx/html/index.html`
- Lomás sencillo desde Visual Code con la extensión de docker)
- Visualizar el `localhost:80` y `localhost:81` para ver que visualizan contenido diferente.

Persistencia de datos

- Si paramos y ararncamos el contenedor, los cambios persisten:

```
$ docker stop nginx-container2
$ docker start nginx-container2
```

- En cambio si lo tiramos y creamos de nuevo no:

```
$ docker stop nginx-container2
$ docker rm nginx-container2
$ docker run --name nginx-container2 -p 81:80 -d nginx
```

- Para tener persistencia de datos usamos volúmenes

Tipos de volúmenes

Volúmenes de host

- Este comando monta la ruta `/ruta/en/el/host` del sistema host en `/ruta/en/el/contenedor` dentro del contenedor.

`docker run -v /ruta/en/el/host:/ruta/en/el/contenedor nombre_de_la_imagen`

Volúmenes anónimos

- Docker asignará automáticamente un volumen anónimo al contenedor en la ruta especificada.

`docker run -v /ruta/en/el/contenedor nombre_de_la_imagen`

Volúmenes con nombre:

- Primero, creamos un volumen con nombre utilizando `docker volume create`, luego lo montamos en el contenedor.

```
docker volume create nombre_del_volumen
docker run -v nombre_del_volumen:/ruta/en/el/contenedor nombre_de_la_imagen
docker volume ls
```

Visualización de logs

- Hacer peticiones a `localhost:80` y visualizarlas mediante los logs:

```
# últimas trazas
$ docker logs nginx-container1
# ver también trazas posteriores
$ docker logs -f nginx-container1
```

Ejercicio 1

- Crea un volumen para lograr persistencia mapeado a una carpeta del host llamada html.

Solución

- Crea en la carpeta llamada html un fichero `index.html`
  - Si la carpeta o el fichero no existe dará un `403 Forbidden`
- Recrea el nginx-container con uso de volúmenes

```
$ docker stop nginx-container2
$ docker rm nginx-container2
$ $docker run --name nginx-container2 -p 81:80 -v ./html:/usr/share/nginx/html -d nginx
```

Ejercicio 2

- Comprueba el estado de imágenes y contenedores de tu equipo
- Elimina todas las imágenes
- Elimina todos los contenedores (también los parados)

Solución

- Estado actual:

```
docker ps
docker ps -a
docker  images
```

- Borrar imágenes:

```
docker rmi $(docker images –a –q)
```

- Borrar contenedores:

```
docker rm $(docker ps –a –q)
```

2.1 Tipos de servicios

Diferentes tipos de contenedores que se pueden ejecutar para diversas aplicaciones y propósitos.

Aquí hay algunos tipos comunes de servicios de contenedores que nosotros vamos a ver en Docker:

Servicios web

- Empaquetaremos una aplicación Angular y la serviremos en un contenedor con nginx.
- Empaquetaremos una aplicación web con Express.js y la serviremos mediante un contenedor con node.js.

Bases de datos

- Veremos como inicializar una base de datos MySQL
  - Otras bases de datos de forma similar
- Veremos como configurar la espera entre contenedores (la app espera a la bbdd para poder arrancar).
- Uso de herramientas gráficas de acceso a bbdd, en nuestro caso PhpMyAdmin

Servicios de caching y almacenamiento en caché

- Varnish en Wordpress

Servicios de copias de seguridad

- Copia de seguridad de bbdd MySql en el servicio de Wordpress

Servicios de monitorización y registro

- Estos contenedores ofrecen herramientas para monitorear y registrar el rendimiento de las aplicaciones y la infraestructura, como Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana), etc.

Y muchos otros:

- Job Scheduler
- Home media server
- ...

2.2.1 Docker compose y ejemplo con Wordpress

Docker compose

- Herramienta que permite definir y gestionar aplicaciones multi-contenedor de forma sencilla y con un solo archivo de configuración en YAML, llamado `docker-compose.yml`.
- Evitamos además comandos `docker run ...` muy largos al tener que mapear puertos, volúmenes...
- Conceptos clave de Docker Compose:
  - **Servicios**: Un servicio es una definición de un contenedor Docker junto con su configuración. Por ejemplo, un servicio puede representar una aplicación web, una base de datos, un servidor de aplicaciones, etc. Cada servicio se puede configurar con diferentes opciones, como el nombre de la imagen, los puertos expuestos, las variables de entorno, los volúmenes, etc.
  - **Redes**: Docker Compose permite crear redes para conectar los contenedores entre sí. Esto es útil para comunicar servicios que necesitan interactuar entre sí.
  - **Volúmenes**: Los volúmenes en Docker Compose son utilizados para persistir datos o compartir datos entre contenedores.

Comandos más habituales de Docker Compose

`docker-compose up [<service name>] -d` : Crear e iniciar contenedores
`docker-compose down [<service name>]` : Detener y eliminar contenedores
`docker-compose logs [-f] [<service name>]` : para ver los logs de los contenedores

Configuración Wordpress

- Fichero `.env`:

```
# should be renamed .env changing sensible data

MYSQL_ROOT_PASSWORD=root
MYSQL_DATABASE=wordpress
MYSQL_USER=user
MYSQL_PASSWORD=user
```

- Fichero `docker-compose.yml`:

```yaml
services:
  db-wp:
    # see https://wordpress.org/about/requirements/
    image: mysql:latest
    hostname: db-wp
    container_name: db-wp
    volumes:
      - ./db/init-db:/docker-entrypoint-initdb.d
      - ./db/data:/var/lib/mysql
    restart: always
    environment:
```

```
      MYSQL_ROOT_PASSWORD: '${MYSQL_ROOT_PASSWORD}'
      MYSQL_DATABASE: '${MYSQL_DATABASE}'
      MYSQL_USER: '${MYSQL_USER}'
      MYSQL_PASSWORD: '${MYSQL_PASSWORD}'
    networks:
      - backend

  wp:
    # see https://wordpress.org/about/requirements/
    image: wordpress:latest
    depends_on:
      - db-wp
    hostname: wp
    container_name: wp
    restart: always
    volumes:
      - ./wp-app:/var/www/html # Full wordpress project
    environment:
      WORDPRESS_DB_HOST: db-wp:3306
      WORDPRESS_DB_USER: '${MYSQL_USER}'
      WORDPRESS_DB_PASSWORD: '${MYSQL_PASSWORD}'
      WORDPRESS_DB_NAME: '${MYSQL_DATABASE}'
    ports:
      - '80:80'
    networks:
      - frontend
      - backend

  pma:
    image: phpmyadmin/phpmyadmin
    container_name: phpmyadmin
    environment:
      PMA_HOST: db-wp
      PMA_PORT: 3306
      MYSQL_ROOT_PASSWORD: '${MYSQL_ROOT_PASSWORD}'
    ports:
      - '8080:80'
    networks:
      - frontend
      - backend

networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge
```

- Más info con proyecto más completo: https://github.com/juanda99/wordpress-docker

Ejercicio

- Crea dos proyectos de Wordpress. Uno resolverá mediante `miweb1.com` y el otro mediante `miweb2.com`
  - Como solo puedes tener un puerto 80, utilizaremos un proxy inverso llamado nginx-proxy
  - Para simular las webs, tendremos que añadir las entradas de `miweb1.com` y `miweb2.com` en el fichero `/etc/hosts` para que resuelvan a local:

```
# cat /etc/hosts
127.0.0.1   miweb1.com
127.0.0.1   miweb2.com
```

Solución

Configuración wordpress

- Creamos dos carpetas wp1 y wp2 con el siguiente `docker-compose.yml`:

```
services:
  db-wp:
    image: mysql:latest
    volumes:
      - ./db/data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: '${MYSQL_ROOT_PASSWORD}'
      MYSQL_DATABASE: '${MYSQL_DATABASE}'
      MYSQL_USER: '${MYSQL_USER}'
      MYSQL_PASSWORD: '${MYSQL_PASSWORD}'
    networks:
      - backend

  wp:
    image: wordpress:latest
    volumes:
      - ./wp-app:/var/www/html # Full wordpress project
    environment:
      WORDPRESS_DB_HOST: db-wp:3306
      WORDPRESS_DB_USER: '${MYSQL_USER}'
      WORDPRESS_DB_PASSWORD: '${MYSQL_PASSWORD}'
      WORDPRESS_DB_NAME: '${MYSQL_DATABASE}'
      VIRTUAL_HOST: miweb1.com
    networks:
      - proxy_frontend
      - backend
```

```
networks:
  proxy_frontend:
    external: true
  backend:
    driver: bridge
```

- Hemos quitado el mapeo del puerto 80 al host, ya que eso lo hará el proxy inverso que pongamos por delante.
- La red de frontend, estará configurada en el proxy y por eso la indicamos con el `external: true`. Además docker compose añade un prefijo al nombre de la red, en función del nombre de la carpeta donde esté el fichero `docker-compose.yml`
- Por último añadimos una variable de entorno nueva: `VIRTUAL_HOST: miweb1.com` que servirá para comunicar al demonio de docker los datos del contenedor que se levanta. El proxy leerá esta información del demonio de docker y se configurará adecuadamente.

Configuración proxy

- Creamos una carpeta llamada `proxy` con un fichero `docker-compose.yml` con el siguiente contenido:

```
services:
  nginx-proxy:
    image: nginxproxy/nginx-proxy
    ports:
      - '80:80'
    volumes:
      - /var/run/docker.sock:/tmp/docker.sock:ro
    networks:
      - frontend
networks:
  frontend:
    driver: bridge
```

- Arrancamos el servicio con `docker compose up -d` y posteriormente levantamos también los servicios de los directorios `wp1` y `wp2` y comprobamos que se acceda desde el navegador.

2.2.2 Ejemplo aplicación SPA

Creación de una app en Angular

```
npx -p @angular/cli ng new my-angular-app
cd my-angular-app
code . # editamos en vscode la app
npx ng serve # arrancamos la app en browser
npx ng build # v1.0.0 en dist
```

Creación del contenedor

- La aplicación necesitará un contenedor con *nginx* o *apache* para funcionar.
- Creamos el ***Dockerfile***:

```
# Use NGINX as base image
FROM nginx:1.25.5

# Copy nginx configuration file
COPY ./nginx.conf /etc/nginx/conf.d/default.conf

# Remove default nginx website
RUN rm -rf /usr/share/nginx/html/*

# Copy the built Angular app to NGINX web root
# COPY dist/<your-angular-app-name> /usr/share/nginx/html
COPY dist/my-angular-app/browser /usr/share/nginx/html
# Expose port 80
EXPOSE 80


# Command to run NGINX
CMD ["nginx", "-g", "daemon off;"]
```

Configuración nginx (nginx.conf)

- Creamos el fichero de configuración del servidor web, ***nginx.conf***:

```
server {
    listen 80;
    server_name localhost;

    location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
        try_files $uri $uri/ /index.html;
```

```
    }
}
```

- Lo más importante aquí es la directiva try_files:
  - `$uri`: Intenta servir la URI (Uniform Resource Identifier) directamente, en caso de que exista.
    - `$uri/` : Intenta servir la URI como si fuera un directorio. Si lo encuentra servirá el fichero index, por ej. `index.html`.
    - `/index.html`: Si ninguna de las opciones anteriores funciona, NGINX servirá el fichero `index.html` , que es normalmente la entrada de una aplicación SPA (Single Page Application). El enrutado en cliente entonces se encarga de gestionar el renderizado de la página.

Ejecución de la aplicación

- Compilamos la imagen:

```
docker build -t angular-nginx:1.0.0 .
```

- Arrancamos:

```
docker run -d -p 8080:80 angular-nginx:1.0.0
```

Publicación manual del contenedor en Docker Hub

- Etiquetamos la imagen:

```
docker tag <image_id> <dockerhub_username>/<repository_name>:<tag>
```

- Hacemos login y subimos la imagen:

```
docker login
docker push <dockerhub_username>/<repository_name>:<tag>
```

Publicación mediante script del contenedor en Docker Hub

- Script compilación imagen en local y publicación en DockerHub
  - Si deseas que las imágenes se etiqueten con diferentes versiones (mayor, menor, etc.) para organización y control, debes hacerlo explícitamente en tu script de implementación, Docker Hub no lo hace de forma automática.

```
#!/bin/bash

# Define the repository
REPO="<mi-usuario-docker-hub>/<nombre-imagen>"

# Get the version from the parameter
```

```
VERSION=$1

# If version is empty, exit with warning
if [ -z "$VERSION" ]; then
  echo "No version specified. Usage: ./deploy.sh <version>"
  exit 1
fi

# Build the project
npm run build

# Build the Docker image
docker build --platform linux/amd64 -t $REPO:$VERSION .

# Push the Docker image to Docker Hub
docker push $REPO:$VERSION

# Tag and push the Docker image as "latest"
docker tag $REPO:$VERSION $REPO:latest
docker push $REPO:latest

# Tag and push the Docker image with the major version (e.g. 1)
MAJOR_VERSION=$(echo $VERSION | cut -d. -f1)
docker tag $REPO:$VERSION $REPO:$MAJOR_VERSION
docker push $REPO:$MAJOR_VERSION

# Tag and push the Docker image with the major-minor version (e.g. 1.0)
MAJOR_MINOR_VERSION=$(echo $VERSION | cut -d. -f1-2)
docker tag $REPO:$VERSION $REPO:$MAJOR_MINOR_VERSION
docker push $REPO:$MAJOR_MINOR_VERSION
```

2.2.3 Ejemplo API en node.js

- Descarga el repositorio https://github.com/juanda99/restaurante-backend

¿Qué es importante ver?

Inicialización de la bbdd

- Mapeamos volúmenes dentro del servicio `db` en el fichero `docker-compose.yml`:

```
volumes:
    - ./data/mysql:/var/lib/mysql
    - ./data/initData:/docker-entrypoint-initdb.d
```

- El primer volumen es para tener persistencia de bbdd, mysql guarda por defecto los datos en `/var/lib/mysql`
- Por defecto si el directorio anterior está vacío y en `docker-entrypoint-initdb.d` hay algún fichero con extensión `.sql` lo carga para inicializar la bbdd. Así que mapeamos un volumen para proporcionar dicho fichero. Todas las imágenes de bbdd funcionan de forma similar y lo tienen documentado.

Espera entre contenedores

- A menudo en servicios se pone la siguiente instrucción, marcando una dependencia para que un contenedor no arranque hasta que el otro esté arrancado:

```
node-app:
    depends_on:
      - db
```

- Pero una cosa es que el contenedor esté arrancado y otra muy diferente es que el servicio esté funcionando, por eso hay que hacer cierto "trabajo" vía código. Ver `/app/app.js` y la función `connectWithRetry`

Variables para conexión

- El fichero `docker-compose.yml` define varias variables de entorno como el usuario o contraseña de bbdd.
- Como estas variables se usan en más de un servicio, es bueno definirlas, por principio DRY en un único sitio (fichero `.env`) que docker usa de forma implícita.
- Es nuestra responsabilidad en el caso de la aplicación en node poder obtener dichas variables, para lo que usamos en paquete `dotenv` obteniendo las variables de `process.env.nombreVariable`

Creación de imagen de forma dinámica

```
  node-app:
    build:
```

```
      context: ./app
      dockerfile: Dockerfile
```

- No es necesario que un servicio tenga una imagen, podemos definir el contexto donde está su configuración (fichero `Dockerfile` y otros que sean necesarios) y se compila al levantar los servicios:
  - `docker-compose up -d` generará la imagen y la ejecutará
  - `docker-compose build [--no-cache]` solo generará las imágenes

Fichero Dockerfile

- Al definir un `WORKDIR` se crea dicho directorio en el contenedor y nos situamos sobre el.
- ¿Cómo se copian los fuentes?
  - Primero copiamos el fichero de dependencias y las instalamos.
  - Luego copiamos el resto del código fuente.
  - Normalmente hay un fichero `.dockerignore` similar a un `.gitignore` para evitar que se copie el `node_modules` que es donde están instaladas todas las dependencias.

```
# Use the official Node.js image as the base image
FROM node:14

# Create and set the working directory
WORKDIR /app

# Copy package.json and package-lock.json to the working directory
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code to the working directory
COPY . .

# Expose the port your application will run on
EXPOSE 3000

# Command to run your application
CMD ["node", "app.js"]
```

2.3 Despliegue de servicios

Aplicación HolaMundo dockerizada

- Vamos a crear una aplicación express con Node.js.

```
mkdir holaMundo
cd holaMundo
npm init # npm es el gestor de paquetes de node
npm install express # instalamos dependencia
```

- Añadimos el fichero `index.js`:

```
const express = require('express')
const app = express()
const port = 3000

// Define a route for the root URL
app.get('/', (req, res) => {
  res.send('Hello, World!')
})

// Start the server
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`)
})
```

- Creamos el fichero `Dockerfile`:

```
# Use official Node.js image as base
FROM node:20

# Set the working directory inside the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json files to the working directory
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code to the working directory
# Prevent node_modules with .dockerignore
COPY . .
```

```
# Expose the port on which the app runs
EXPOSE 3000

# Command to run the application
CMD ["node", "index.js"]
```

- Creamos también un fichero `.dockerignore` con la entrada `node_modules`.
- Separando `package.json` del resto de ficheros, hace que la caché del `Dockerfile` funcione mejor (previsiblemente cambiaremos ficheros de la app, pero no tanto las dependencias).

Fichero docker-compose.yml

```
services:
  nodejs:
    container_name: my-node-app
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - '3000:3000'
      - '9229:9229' # Port redirection for debugging
    environment:
      - NODE_ENV=${NODE_ENV:-production}
    command: >
      sh -c "if [ '$NODE_ENV' = 'dev' ]; then node --inspect=0.0.0.0 index.js; else
node index.js; fi"
    restart: always
```

- Comentarios del `docker-compose.yml`:
  - `NODE_ENV=${NODE_ENV:-production}` : se fija la variable NODE_ENV como variable de entorno, y con el valor `production` si no existe.
  - `command: >` El `>` sirve para indicar en YAML que el comando es multilínea. Utilizamos shell para poner un condicional
- Accedemos al contenedor para ver que la variable está establecida:

```
docker-compose build --no-cache # no necesario
docker-compose up -d # hace el build si no existe
docker-compose exec nodejs bash
```

- Lo habitual es crear un fichero `.env` que define la variable:

```
NODE_ENV=dev
```

- Si lo creamos podemos arrancar de nuevo el servicio y ver las trazas:

```
        docker-compose down
        docker-compose up -d
        docker-compose logs -f
```

Debug de la aplicación

- Debemos arrancarla en modo dev para que reciba el parámetro `--inspect`
- Navegador: chrome://inspect
- VSCode:
  - En la opción de Run and Debug creamos un fichero `launch.json` con la siguiente información:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "attach",
      "address": "localhost",
      "port": 9229,
      "name": "Docker: Attach to Node",
      "remoteRoot": "/usr/src/app"
    }
  ]
}
```

- En el código ponemos puntos de interrupción y comprobamos que funciona.

Ficheros docker-compose.yml separados

- Fichero `docker-compose.yml` con el código común:

```
services:
  nodejs:
    container_name: my-node-app
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    restart: always
```

- Fichero `docker-compose.prod.yml` con el código específico de producción:

```
services:
  nodejs:
    environment:
```

```
      - NODE_ENV=production
    command: node index.js
```

- Fichero `docker-compose.dev.yml` con el código específico de desarrollo:

```
services:
  nodejs:
    environment:
      - NODE_ENV=dev
    ports:
      - "9929:9929"  # Port redirection for debugging
    command: node --inspect=0.0.0.0:9229 index.js
```

- Para arrancar:

```
docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d
docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d
```

- Otra opción es crear un fichero docker-compose.override.yml y que apunte al dev o al prod según como corresponda en nuestro entorno:

```
        ln -s docker-compose.prod.yml docker-compose.override.yml
        docker-compose up -d
```

- Modificación de código en dev, fichero `docker-compose.dev.yml`:

```
services:
  nodejs:
    environment:
      - NODE_ENV=dev
    ports:
      - '9229:9229' # Port redirection for debugging
    command: 'node --inspect=0.0.0.0:9929 index.js'
    volumes:
      - .:/usr/src/app
      - /usr/src/app/node_modules
```

- Mapeamos también un volumen anónimo para node_modules, de modo que no use el que tenemos en local ya que las librerías pueden ser especificas del OS.
- Cada vez que se baja el servicio y se recrea, Docker crea un nuevo volumen anónimo. Esto significa que los datos en el volumen anónimo anterior no se reutilizan, lo que puede llevar a problemas de acumulación de datos no utilizados

```
docker volume ls
docker volume rm <complete-id>
```

- Con volumen no anónimo:

```
services:
  nodejs:
    environment:
      - NODE_ENV=dev
    ports:
      - '9229:9229' # Port redirection for debugging
    command: 'node --inspect=0.0.0.0:9929 index.js'
    volumes:
      - .:/usr/src/app
      - node_modules:/usr/src/app/node_modules
volumes:
  node_modules:
```

Multistage builds

- A veces hay dependencias de desarrollo que necesitamos para generar nuestra imagen.
- Debemos montar dicho entorno, para generar nuestro ejecutable, pero luego deberíamos quitar dichas dependencias (riesgo de seguridad) y además "adelgazar" nuestra imagen.
- Creamos una imagen temporal para instalar y compilar nuestra aplicación y luego una imagen final, más pequeña, optimizada y sin las dependencias de desarrollo.

```
# Stage 1: Install dependencies and build the application
FROM node:14 AS builder

WORKDIR /app

COPY package.json package-lock.json ./

# Install dependencies for both production and development
RUN npm install

# Copy the application code
COPY . .

# Build the application
RUN npm run build

# Stage 2: Create a smaller, optimized image for production
```

```
FROM node:14-alpine

WORKDIR /app

# Copy only production dependencies from the previous stage
COPY --from=builder /app/package.json /app/package-lock.json ./

# Install only production dependencies
RUN npm install --production

# Copy the built application from the previous stage
COPY --from=builder /app/dist ./dist

# Start the application
CMD ["node", "./dist/index.js"]
```

Limitación de recursos de carga

```
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
```

- Limitamos el servicio `web` a utilizar solo el 50% de una CPU (`0.5`) y un máximo de 512 megabytes de memoria (`512M`).
- Podemos ver los recursos usados mediante `docker stats`
- Los recursos también se pueden limitar mediante Docker Desktop.

Escalado de servicios

- Hay que diferenciar entre **Docker Compose** y **Docker Swarm**.
  - Docker Compose se centra en la definición y ejecución de aplicaciones de contenedores multi-servicio en **un solo host**.
  - Docker Swarm se enfoca en la orquestación de contenedores **a nivel de clúster**
  - Docker Compose no permite definir el escalado en el fichero `docker-compose.yml`
  - Docker Swarm permite definir las instancias necesarias dentro del fichero `docker-compose.yml` . Si queremos escalado automático no hay una opción out-of-the-box.

```
docker compose up -d
docker compose scale nodejs=3
```

- Pero hay problemas:
  - ¿Qué pasa con el mapeo de puertos?
  - ¿Cómo se balancea entre las distintas instancias? Necesitamos algún proxy inverso
    - HAProxy
    - Traefik
    - nginx-proxy
  - Otros requisitos:
    - Permitir **ACME**(Automated Certificate Management Environment protocol). Por ejemplo con Let's Encrypt
    - Exporter para Prometheus

Configuración de Traefik como balanceador y Proxy Inverso

- Añadimos en el /etc/hosts las siguientes entradas:

```
127.0.0.1   web1.local.es
127.0.0.1   web2.local.es
127.0.0.1   traefik.local.es
```

- Descargamos proyecto básico y lo ejecutamos:

```
git clone git@github.com:juanda99/traefik-demo.git
cd traefik-demo
docker compose up -d
```

Observabilidad

- Mediante logs, ver doc
- Métricas, ver doc
  - En nuestro caso para prometheus: http://traefik.local.es:8082/metrics
- Tracing, para desarrollo

Añadir middleware de autenticación

Así comprobamos como añadir un middleware y protegemos el Dashboard de Traefik.

- Generar contraseña: Autenticación basicAuth (se añaden 2$ para escapar el $ en el docker-compose)-

```
sudo apt-get update && sudo apt-get install apache2-utils
echo $(htpasswd -nB user) | sed -e s/\\$/\\$\\$/g
```

- Modificamos docker-compose.yml, dentro del servicio de Traefik, añadiendo las dos últimas líneas:

```
labels:
    - traefik.enable=true
    - traefik.http.routers.dashboard.entrypoints=web
    - traefik.http.routers.dashboard.rule=Host(`traefik.local.es`)
    - traefik.http.routers.dashboard.service=api@internal
        - traefik.http.routers.dashboard.middlewares=auth
        - traefik.http.middlewares.auth.basicauth.users=user:<contraseña>
```

- Reiniciamos contenedores (`docker-compose down && docker-compose up -d`) y comprobamos acceso
- Aumentamos el número de contenedores para un servicio y comprobamos vía dashboard que aumenta el número de servers sobre el que se hace balanceo:

```
docker compose scale web1=5
```

2.4 Monitorización y gestión de servicios

- Herramientas para el monitoreo de servicios
- Recolección de métricas y logs
- Gestión de la salud de los servicios

Docker stats

- Docker proporciona un comando llamado `docker stats`
- Muestra estadísticas en tiempo real de uso de los contenedores en ejecución:
  - CPU
  - Memoria
  - Red
  - E/S de los contenedores en ejecución
  - Procesos en ejecución en el contenedor
- Ventajas:
  - Facilidad de uso
  - Integrada de forma nativa
  - Monitorización en tiempo real
- Desventajas:
  - No ofrece históricos para diagnósticos más complejos
  - No ofrece un sistema de alarmas
  - No es adecuado para escenarios de escala (la salida del comando puede volverse abrumadora y difícil de gestionar)
  - Dependencia de consola

Docker Healthchecks

- Se definen healthchecks en tus archivos de Docker Compose o Dockerfile:
  - Se define un comando
  - Docker verifica el estado de salud del contenedor a partir del comando

Ejemplo con Dockerfile

```
FROM nginx:latest

# Definir el healthcheck
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD curl -f http://localhost/ || exit 1
```

- `--interval`: Frecuencia con la que Docker ejecutará el healthcheck.
- `--timeout`: Tiempo máximo permitido para que el healthcheck se complete.
- `--start-period`: Tiempo que Docker debe esperar antes de comenzar a verificar la salud después de que el contenedor se haya iniciado.

- `--retries`: Número de intentos fallidos antes de que Docker marque el contenedor como no saludable.
- `CMD`: comando que se utilizará para verificar la salud del contenedor.

Ejemplo con docker-compose.yml

```
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost/"]
      # test: ["CMD-SHELL", "curl -f http://localhost/ || exit 1"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 5s
```

Cómo testear healthcheck

- Ejecutar el comando a mano dentro del contenedor
- Parar el servicio del contenedor
- Modificar healthcheck para provocar un fallo (por ej. cambiando la url del curl)

Otras herramientas en docker

- Procesos en ejecución

```
docker  top <container_id>
```

  - Consultas específicas por contenedor

```
docker exec -it <container-id> top
```

- Configuraciones intrínsecas de los servicios (por ej. mod_status en Apache)
  - Estas mediciones se pueden recoger con servicios online como SysDig o DataDog. Ejemplo: https://docs.datadoghq.com/integrations/apache/
- Consultas al demonio de docker
  - Docker Daemon expone sus métricas a través de una API REST.
  - Estas métricas dan información sobre el rendimiento y la actividad del propio demonio de Docker, incluidos los contenedores en ejecución, imágenes, redes y volúmenes. Proporciona datos a nivel de host.
  - Se pueden exportar a Prometheus
  - Para métricas detalladas a nivel de contenedor se utiliza cAdvisor

- Proporciona una interfaz web para visualizar métricas en tiempo real sobre el uso de CPU, memoria, red, etc.

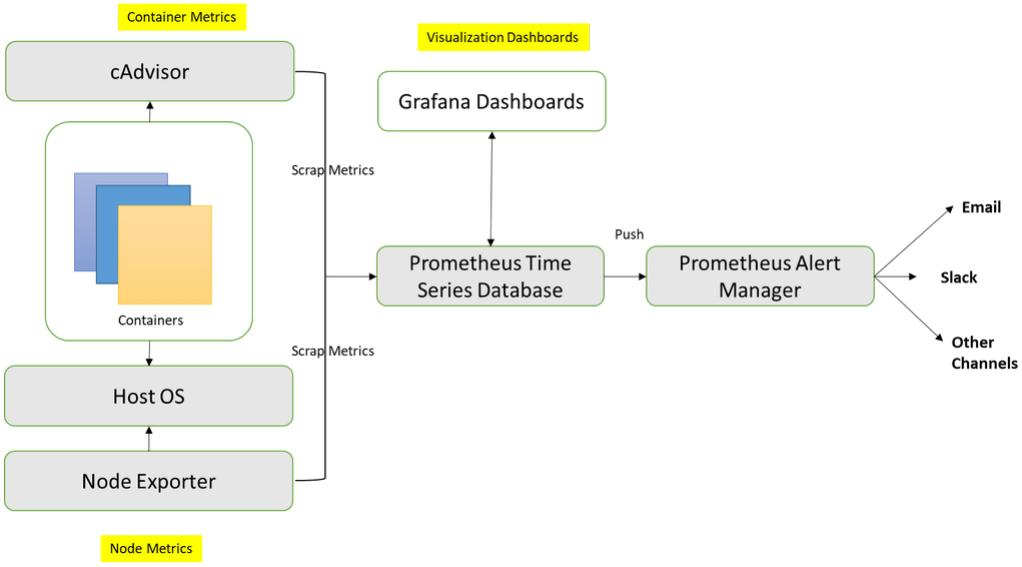
Prometheus y Grafana

- Herramientas populares de monitoreo y visualización que se pueden integrar con Docker para obtener métricas detalladas de tus contenedores y aplicaciones.
- Prometheus puede recopilar métricas de tus contenedores y servicios, mientras que Grafana te permite crear paneles de control personalizados para visualizar estas métricas de manera efectiva.

2.4.1 Prometheus y Graphana

Solución open source que provee métricas, visualización y alertas tanto de servidores físicos como máquinas virtuales o contenedores.

- Ventajas:
  - Se puede desplegar con contenedores
  - Funciona bien en sistemas distribuidos
  - Hay exporters para todo tipo de servicios
  - Hay clientes en muchos lenguajes de programación para exportar métricas a medida



Ejecución de entorno

- Ver blog y repositorio de GitHub

```
git clone https://github.com/stefanprodan/dockprom
cd dockprom

ADMIN_USER=admin ADMIN_PASSWORD=admin docker-compose up -d
```

Lista de contenedores

- Prometheus (recolector de métricas) `http://<host-ip>:9090`
- AlertManager (gestor de alertas) `http://<host-ip>:9093`
- Grafana (visualización de métricas) `http://<host-ip>:3000`
- NodeExporter (host metrics collector)
- cAdvisor (containers metrics collector)
- Caddy (proxy inverso)

Añadir hosts

- Se añaden en el fichero `prometheus.yml`

- Se corren los servicios en el host:
  - por ej vía `docker-compose.exporters.yml`
    - En este caso el network está configurado como `host`
  - A mano, vía **Node exporter**y cAdvisor
- Comprobamos las métricas de Node exporter:

```
http://localhost:9100/metrics
```

- Comprobamos las métricas de cAdvisor vía su web UI en `http://localhost:8080`. También se pueden explorar por contenedor en `http://localhost:8080/docker/<container>`.

Definición de alertas

- Ver `prometheus/alert.rules`
- Ejemplos:
  - Disparar una alerta si alguno de los servicios de monitorización (node-exporter and cAdvisor) estás caídos más de 30 segundos

```
ALERT monitor_service_down
  IF up == 0
  FOR 30s
  LABELS { severity = "critical" }
  ANNOTATIONS {
      summary = "Monitor service non-operational",
      description = "{% raw %}{{ $labels.instance }}{% endraw %} service is down.",
  }
```

- Disparar una alerta si algún contenedor tiene una carga alta de cpu durante más de 30s:

```
ALERT high_cpu_load
  IF node_load1 > 1.5
  FOR 30s
  LABELS { severity = "warning" }
  ANNOTATIONS {
      summary = "Server under high load",
      description = "Docker host is under high load, the avg load 1m is at {% raw %}{{
$value}}{% endraw %}. Reported by instance {% raw %}{{ $labels.instance }}{% endraw %}
of job {% raw %}{{ $labels.job }}{% endraw %}.",
  }
```

Importar dashboards a Grafana

1. Vamos al menú Dashboards
2. Pulsamos el botón de New y seleccionamos Import en el desplegable.

3. Seleccionamos el dashboard que nos interese de https://grafana.com/grafana/dashboards/ y copiamos el ID del mismo en el campo correspondiente (también podríamos hacerlo con el json o modificarlo).

3.1 CI CD con docker

Publicación manual del contenedor en Docker Hub

- Etiquetamos la imagen:

```
docker tag <image_id> <dockerhub_username>/<repository_name>:<tag>
```

- Hacemos login y subimos la imagen:

```
docker login
docker push <dockerhub_username>/<repository_name>:<tag>
```

Publicación mediante script del contenedor en Docker Hub

- Script compilación imagen en local y publicación en DockerHub
  - Si deseas que las imágenes se etiqueten con diferentes versiones (mayor, menor, etc.) para organización y control, debes hacerlo explícitamente en tu script de implementación, Docker Hub no lo hace de forma automática.

```
#!/bin/bash

# Define the repository
REPO="<mi-usuario-docker-hub>/<nombre-imagen>"

# Get the version from the parameter
VERSION=$1

# If version is empty, exit with warning
if [ -z "$VERSION" ]; then
  echo "No version specified. Usage: ./deploy.sh <version>"
  exit 1
fi

# Build the project
npm run build

# Build the Docker image
docker build --platform linux/amd64 -t $REPO:$VERSION .

# Push the Docker image to Docker Hub
docker push $REPO:$VERSION

# Tag and push the Docker image as "latest"
docker tag $REPO:$VERSION $REPO:latest
docker push $REPO:latest
```

```
# Tag and push the Docker image with the major version (e.g. 1)
MAJOR_VERSION=$(echo $VERSION | cut -d. -f1)
docker tag $REPO:$VERSION $REPO:$MAJOR_VERSION
docker push $REPO:$MAJOR_VERSION

# Tag and push the Docker image with the major-minor version (e.g. 1.0)
MAJOR_MINOR_VERSION=$(echo $VERSION | cut -d. -f1-2)
docker tag $REPO:$VERSION $REPO:$MAJOR_MINOR_VERSION
docker push $REPO:$MAJOR_MINOR_VERSION
```

CI/CD

- Vamos a hacer un fork del repositorio https://github.com/juanda99/restaurante-backend, montaremos un entorno de pruebas en local, para lo que habrá que modificar varios ficheros (está todo solucionado en este repositorio: https://github.com/juanda99/restaurante-backend-tests)
- En el docker-compose.yml, añadimos el código fuente como volumen (por si hacemos modificaciones) y montamos como volumen anónimo la carpeta node_modules y mapeamos el puerto de debug.

```
services:
  db:
    image: mysql:8
    restart: always
    environment:
      LANG: C.UTF-8
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: ${DB_NAME}
      MYSQL_USER: ${DB_USER}
      MYSQL_PASSWORD: ${DB_PASSWORD}
    volumes:
      - ./data/mysql:/var/lib/mysql
      - ./data/initData:/docker-entrypoint-initdb.d

  phpmyadmin:
    image: phpmyadmin
    restart: always
    ports:
      - 9091:80
    # environment:
    #   - PMA_HOST=db

  node-app:
    build:
      context: ./app
      dockerfile: Dockerfile.dev
```

```
    ports:
      - '3000:3000'
      - '9229:9229'
    depends_on:
      - db
    volumes:
      - ./app:/app
      - /app/node_modules
    environment:
      DB_HOST: db
      DB_USER: ${DB_USER}
      DB_PASSWORD: ${DB_PASSWORD}
      DB_NAME: ${DB_NAME}
```

- Creamos una carpeta test (`app/tests`) donde colocaremos dos scripts, uno haciendo pruebas simulando el servicio de bbdd y otro sin simularlo ya que se puede montar un entorno completo para todo dockerizado, que es la idea final.
- Vamos a hacer tests de integración con bbdd, probando los endpoints de una API.
  - En principio se podrían hacer con mockups, pero no serían tan fidedignos
  - Con docker podemos reproducir el entorno de forma sencilla

```
// fichero app.test.js
const request = require('supertest')
const express = require('express')
const app = require('../app')

const categories = [
  {
    categoriaID: 1,
    categoria: 'Tapas y raciones',
  },
  {
    categoriaID: 2,
    categoria: 'Entrantes',
  },
  {
    categoriaID: 3,
    categoria: 'Pizzas',
  },
  {
    categoriaID: 4,
    categoria: 'Platos internacionales',
  },
```

```
  {
    categoriaID: 5,
    categoria: 'Bocadillos',
  },
  {
    categoriaID: 6,
    categoria: 'Guarniciones',
  },
  {
    categoriaID: 7,
    categoria: 'Carnes',
  },
  {
    categoriaID: 8,
    categoria: 'Pescados',
  },
  {
    categoriaID: 9,
    categoria: 'Postres',
  },
]

describe('GET /categories', () => {
  it('should respond with status 200 and return categories', async () => {
    const res = await request(app).get('/categories')
    expect(res.status).toBe(200)
    expect(res.body).toEqual(categories)
  })
})

// Similar tests for other endpoints: /restaurants, /dishes, /customers, /orders
```

```
// file app-mockup-db.test.js
const request = require('supertest')
const express = require('express')

const app = require('../app')

const categories = [
  {
    categoriaID: 1,
    categoria: 'Tapas y raciones',
  },
```

```
]

// Mock de la función de consulta de la base de datos
jest.mock('mysql2', () => {
  return {
    createConnection: jest.fn(() => {
      return {
        connect: jest.fn(),
        query: jest.fn((sql, callback) => {
          // Simula el comportamiento de la consulta
          if (sql === 'SELECT * FROM categorias') {
            callback(null, categories)
          }
        }),
      }
    }),
  }
})

describe('GET /categories', () => {
  it('should respond with status 200 and return categories', async () => {
    const res = await request(app).get('/categories')
    expect(res.status).toBe(200)
    expect(res.body).toEqual(categories)
  })
})

// Agrega tests similares para otros endpoints
```

- Como usamos varias librerías para desarrollo no queda otra que usar un Dockerfile para producción y un Dockerfile.dev para desarrollo:

```
# Use the official Node.js image as the base image
FROM node:20

# Create and set the working directory
WORKDIR /app

# Copy package.json and package-lock.json to the working directory
COPY package*.json ./

# Install dependencies
#  this would be for production: RUN npm install --only=production
```

```
RUN npm install

# Copy the rest of the application code to the working directory
COPY . .

# Expose the port your application will run on
EXPOSE 3000

# Command to run your application
CMD ["npm", "run", "dev"]
```

- En el package.json, añadimos el script `dev` y `test` y las dependencias de desarrollo que necesitamos:

```
{
  "name": "app",
  "version": "1.0.0",
  "main": "app.js",
  "scripts": {
    "start": "node app.js",
    "dev": "nodemon --inspect=0.0.0.0:9229 app.js",
    "test": "jest --coverage"
  },
  "dependencies": {
    "cors": "^2.8.5",
    "dotenv": "^10.0.0",
    "express": "^4.17.1",
    "mysql2": "^3.9.7"
  },
  "devDependencies": {
    "jest": "^29.7.0",
    "jest-mock": "^29.7.0",
    "nodemon": "^3.1.0",
    "supertest": "^7.0.0"
  }
}
```

- Por último para hacer pruebas, si nuestra app está arrancada , no debemos arrancarla dos veces, daría error, así que hay que cambiar algo la lógica de la app (función `startServerIfNeeded`), además de exportar la app, para usarla desde los tests. El código quedaría así:

```
// fichero app/app.js:
```

```
const express = require('express')
const cors = require('cors')
const mysql = require('mysql2')
require('dotenv').config()

const app = express()
app.use(cors())
const port = 3000
let db

// Function to establish a connection with retries
function connectWithRetry() {
  db = mysql.createConnection({
    host: process.env.DB_HOST,
    user: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_NAME,
  })

  // Attempt to connect
  db.connect((err) => {
    if (err) {
      console.error('Error connecting to MySQL:', err)
      // Retry after 5 seconds
      console.log('Retrying in 5 seconds...')
      setTimeout(connectWithRetry, 5000)
    } else {
      console.log('Connected to MySQL database')
      startServerIfNeeded()
    }
  })
}

// Start the connection with retries
connectWithRetry()

// Define your endpoints

// Get all categories
app.get('/categories', (req, res) => {
  db.query('SELECT * FROM categorias', (err, results) => {
    if (err) {
      console.error('Error executing query:', err)
```

```
      res.status(500).json({ error: 'Internal Server Error' })
    } else {
      res.json(results)
    }
  })
})

// Get all restaurants
app.get('/restaurants', (req, res) => {
  db.query('SELECT * FROM restaurantes', (err, results) => {
    if (err) {
      console.error('Error executing query:', err)
      res.status(500).json({ error: 'Internal Server Error' })
    } else {
      res.json(results)
    }
  })
})

// Get all dishes
app.get('/dishes', (req, res) => {
  db.query('SELECT * FROM platos', (err, results) => {
    if (err) {
      console.error('Error executing query:', err)
      res.status(500).json({ error: 'Internal Server Error' })
    } else {
      res.json(results)
    }
  })
})

// Get all customers
app.get('/customers', (req, res) => {
  db.query('SELECT * FROM clientes', (err, results) => {
    if (err) {
      console.error('Error executing query:', err)
      res.status(500).json({ error: 'Internal Server Error' })
    } else {
      res.json(results)
    }
  })
})
```

```
// Get all orders
app.get('/orders', (req, res) => {
  db.query('SELECT * FROM pedidos', (err, results) => {
    if (err) {
      console.error('Error executing query:', err)
      res.status(500).json({ error: 'Internal Server Error' })
    } else {
      res.json(results)
    }
  })
})

// Get all dishes for a specific order
app.get('/order/:orderId/dishes', (req, res) => {
  const orderId = req.params.orderId
  db.query(
    'SELECT pl.platoID, pl.plato, pl.descripcion, pl.precio, pp.cantidad FROM
platospedidos pp JOIN platos pl ON pp.platoID = pl.platoID WHERE pp.pedidoID = ?',
    [orderId],
    (err, results) => {
      if (err) {
        console.error('Error executing query:', err)
        res.status(500).json({ error: 'Internal Server Error' })
      } else {
        res.json(results)
      }
    }
  )
})

// Function to start the server if it's not already running so tests can run
function startServerIfNeeded() {
  if (!server) {
    server = app.listen(port, () => {
      console.log(`Server is running on port ${port}`)
    })
  }
}
module.exports = app
```

- Podríamos configurar CI/CD para las siguientes tareas:
  - Ejecutar tests después de cada push / pull request

- Ejecutar una comprobación de seguridad (trivy) después de cada push/pull si los tests son satisfactorios
- Publicar una nueva imagen si hay un nuevo tag

```
# Fichero github/workflows/tests.yml

name: Docker Image Tests and Trivy

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  tests:
    name: "Integration testing"
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - name: "Set up the environment"
      run: docker-compose -f docker-compose.yml up -d
    - name: "Wait for Express server to be ready"
      run: |
        start_time=$(date +%s)
        timeout=0
        until docker-compose logs node-app 2>&1 | grep -q "Connected to MySQL database"
|| [ $timeout -eq 100 ]; do
            timeout=$((timeout+1))
            sleep 1
        done
        if [ $timeout -eq 100 ]; then
            echo "Timeout reached waiting for Express server to be ready"
            exit 1
        fi
        end_time=$(date +%s)
        echo "Express server took $((end_time - start_time)) seconds to be ready"
    - name: "Test server"
      run: docker compose exec node-app npm run test

  trivy_scan:
    name: "Trivy Scan"
    runs-on: ubuntu-latest
```

```
    needs: tests
    steps:
      - name: "Checkout code"
        uses: actions/checkout@v4
      - name: "Install Trivy"
        run: |
          wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | sudo
apt-key add -
          echo deb https://aquasecurity.github.io/trivy-repo/deb $(lsb_release -cs)
main | sudo tee -a /etc/apt/sources.list.d/trivy.list
          sudo apt-get update && sudo apt-get install trivy
      - name: "Build Docker Image"
        run: docker build -t juanda99/node-app:latest ./app
        # This assumes your service using the built image is named 'your_service'
      - name: Run Trivy vulnerability scanner
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: 'juanda99/node-app:latest'
          format: 'table'
          exit-code: '1'
          ignore-unfixed: true
          vuln-type: 'os,library'
          severity: 'CRITICAL,HIGH'
```

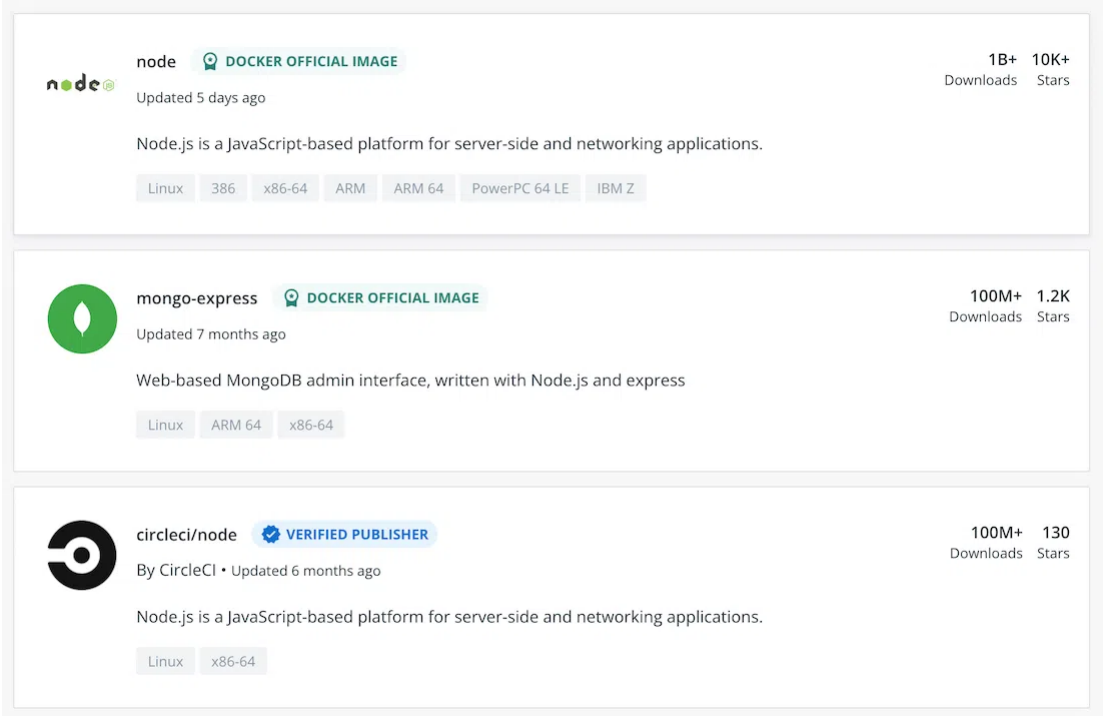
```
# Fichero github/workflows/publish.yml

name: Publish Docker
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Publish to Registry
      uses: elgohr/Publish-Docker-Github-Action@v5
      with:
        name: cateduac/arasaac-api
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}
        tag_semver: true
```

3.2 Seguridad en contenedores

Los contenedores, como Docker o Kubernetes, ofrecen ventajas significativas en términos de portabilidad, escalabilidad y eficiencia, pero también introducen nuevos desafíos de seguridad que deben abordarse adecuadamente.

Imágenes seguras

- Utiliza el Docker Hub oficial o repositorios confiables para obtener imágenes.
  - Utiliza *Docker official images* o *Verified Publisher images*



- Verifica la integridad de las imágenes descargadas utilizando sumas de verificación.
- Escanea las imágenes en busca de vulnerabilidades testeando de forma regular y utilizando varias herramientas:
  - Clair
  - Trivy
  - Docker Scout
    - Utiliza snykpor debajo.

CVE y SBOM

- **CVE (Common Vulnerabilities and Exposures)**, es un sistema de catalogación pública que proporcionando una identificación única y estandarizada para las vulnerabilidades de seguridad de la información.Se establece en el año 1999 por el MITRE.
- ***CVE-2021-34567*** podría referirse a una vulnerabilidad de seguridad descubierta en el año 2021 y catalogada con el número 34567 en la base de datos de vulnerabilidades del sistema CVE.
- El MITRE Corporation es la organización responsable de asignar y dar de alta los identificadores CVE en la base de datos oficial de vulnerabilidades de seguridad conocida como el "CVE List".

- El CVE facilita la comunicación y el intercambio de información sobre vulnerabilidades y sus correcciones entre diferentes plataformas y herramientas.
- En cualquier organización, es útil tener un inventario de software, llamado ***Software Bill of Materials (SBOM)*** para revisar las vulnerabilidades asociadas en la base de datos de CVE's.
- En el caso particular de Docker para obtener el SBOM se analizan las imágenes.

```
docker sbom <image>
```

Scanner de vulnerabilidades

- Obtienen el SBOM de las imágenes y los CVE's con su grado de severidad asociado.

Docker Scout

- Nos muestra **para cada capa de la imagen (incluyendo la imagen base)**, los paquetes asociados y sus CVE's con su grado de severidad asociado.
- Está integrado en el Docker Desktop
- Gratuito hasta 3 repositorios en Docker Hub. Gratis en local.
- Sustituye a `docker scan` (Junio 2020 - Febrero 2023).
  - Ambas usan snyk por debajo, se puede instalar el cli
- Ejemplos de uso:

```
# docker scout quickview <image>
# docker scout cves <image>
# docker scout recommendations <image>
```

Trivy

https://github.com/aquasecurity/trivy
https://trivy.dev/

- Open Source y el más popular
- Puede scanear también sistemas de ficheros y repositorios git
- Se integra bien en CI/CD

```
trivy image mysql:8
```

Grype

- Otra opción.
- A menudo es útil usar varias para detectar falsos positivos.
- https://github.com/anchore/grype

CIS Docker Benchmark

- El término *CIS Benchmark* se refiere a **conjuntos de configuraciones recomendadas** por el Center for Internet Security (CIS), una organización sin fines de lucro que se enfoca en mejorar la seguridad

en línea y proteger los sistemas informáticos.

- El CIS Docker Benchmark aborda una amplia gama de aspectos relacionados con la seguridad en entornos Docker, incluyendo la configuración de los contenedores, la gestión de imágenes, la configuración del host Docker y la seguridad de la red.
- Docker Bench for Securityse centra en escanear el host de Docker para problemas de configuración y provee una puntuación y mejoras en base a las buenas prácticas detalladas en el CIS Benchmark
- Otra opción es utilizar Dockle
  - Lista de checkpoints basados en el CIS Docker Benchmark
  - Ejecución dockerizada:

```
VERSION=$(
 curl --silent "https://api.github.com/repos/goodwithtech/dockle/releases/latest" | \
 grep '"tag_name":' | \
 sed -E 's/.*"v([^"]+)".*/\1/' \
) && docker run --rm -v /var/run/docker.sock:/var/run/docker.sock \
  goodwithtech/dockle:v${VERSION} [YOUR_IMAGE_NAME]
```

3.3 Orquestación

Qué es la orquestación

- Automatización del despliegue, escalado, operación y mantenimiento de aplicaciones distribuidas.
  - Gestión de recursos (despliegue en varios nodos) y monitorización
    - Si un contenedor falla, se lanza otro para mantener el nivel de replicas que se haya configurado.
  - Actualización de servicios sin perdida de disponibilidad (se actualizan los nodos poco a poco)
  - Administración de configuraciones y secretos
  - ...

Tipos de orquestación

- Las más populares son:
  - Kubernetes (k8s)
  - Docker Swarm
- Docker swarm es más sencillo de configurar y administrar pero no tiene de forma nativa soluciones como autoescalado.
- Kubernetes está pensado para entornos grandes
  - En k8s los Control Pane (nodos manager en terminología Docker Swarm), no ejecutan contenedores.
  - Existe k3s para escenarios más ligeros

Docker Swarm

- Comprobamos si está activo mediante `docker system info` buscando el mensaje `Swarm: active`
- Si no está activo bastará que lo añadamos como manager o worker, no es necesario instalar nada.

Inicialización como manager

```
# with vmware default port is used by vmware, so we change the port
# avertise-addr is the ip address of the manager in the chosen network
$ docker swarm init --data-path-port=7789 --advertise-addr=192.168.1.252

#without vmware
$ docker swarm init --advertise-addr=192.168.1.252
```

Inicialización como worker

- La salida el comando anterior, muestra como añadir workers:

```
Swarm initialized: current node (tjjggogqpnpj2phbfbz8jd5oq) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-
```

```
3e0hh0jd5t4yjg209f4g5qpowbsczfahv2dea9a1ay2l8787cf-2h4ly330d0j917ocvzw30j5x9
192.168.65.3:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the
instructions.
```

Creación de una red de overlay

- Para que la orquestación con swarm funcione de forma correcta, el stack que levantemos se tiene que comunicar a través de una red de tipo overlay (red entre hosts):

```
docker network create -d overlay traefik-public
```

Nodos

- En el nodo manager:
  - Se crea la definición del stack
    - Fichero `docker-compose.yml`
    - Conjunto de servicios que deben ser ejecutados a la vez
  - Se realiza la orquestación entre los distintos nodos del cluster
  - Por defecto realiza también las funciones de nodo worker
- Cada servicio define una tarea o replica de tareas que se ejecutan en uno o varios nodos

Disponibilidad de los nodos

- Los nodos están `Active` o `Drain`
- Vemos los nodos activos
  - El `*` muestra el nodo al que estás conectado:

```
$ docker node ls

ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
1bcef6utixb0l0ca7gxuivsj0    worker2   Ready   Active
38ciaotwjuritcdtn9npbnkuz    worker1   Ready   Active
e216jshn25ckzbvmwlnh5jr3g *  manager1  Ready   Active        Leader
```

- Creamos un servicio con una replica 3:

```
$ docker service create --replicas 3 --name redis --update-delay 10s redis:3.0.6
$ docker service ps redis

NAME                                IMAGE        NODE     DESIRED STATE  CURRENT STATE
redis.1.7q92v0nr1hcgts2amcjyqg3pq  redis:3.0.6  manager1 Running        Running 26
seconds
redis.2.7h2l8h3q3wqy5f66hlv9ddmi6  redis:3.0.6  worker1  Running        Running 26
```

```
seconds
redis.3.9bg7cezvedmkgg6c8yzvbhwsd  redis:3.0.6  worker2  Running        Running 26
seconds
```

- Eliminamos la disponibilidad de un nodo:

```
$ docker node update --availability drain worker1
```

- Todas las tareas se moverán al resto de nodos:

```
$ docker service ps redis

NAME                                    IMAGE        NODE      DESIRED STATE  CURRENT
STATE           ERROR
redis.1.7q92v0nr1hcgts2amcjyqg3pq       redis:3.0.6  manager1  Running        Running 4
minutes
redis.2.b4hovzed7id8irg1to42egue8       redis:3.0.6  worker2   Running        Running
About a minute
 \_ redis.2.7h2l8h3q3wqy5f66hlv9ddmi6   redis:3.0.6  worker1   Shutdown       Shutdown
2 minutes ago
redis.3.9bg7cezvedmkgg6c8yzvbhwsd       redis:3.0.6  worker2   Running        Running 4
minutes
```

- Comprobamos el estado del worker1:

```
docker node inspect --pretty worker1

ID:                     38ciaotwjuritcdtn9npbnkuz
Hostname:               worker1
Status:
 State:                 Ready
 Availability:          Drain
...snip...
```

- Ver procesos en ejecución en un nodo:

```
docker node ps --filter desired-state=Running
```

- Ver procesos en ejecución en todos los nodos:

```
docker node ps $(docker node ls -q) --filter desired-state=Running
```

- Ver e inspeccionar servicios

```
 docker service ls
 docker service inspect --pretty <service-name>helloworld
 `docker service ps <SERVICE-ID>` to see which nodes are running the service:
```

- Escalar servicios

```
docker service scale <SERVICE-ID>=<NUMBER-OF-TASKS>
```

- Borrar servicios

```
docker service rm helloworld
```

Rolling updates

Las actualizaciones se hacen de una en una a no ser que se configure paralelismo. En este caso cada 10s (una vez haya concluido la anterior). Si una falla, se pausa la actualización.

```
docker service create \
  --replicas 3 \
  --name redis \
  --update-delay 10s \
  redis:3.0.6
```

- Ver UpdateConfig:

```
docker service inspect --pretty redis

ID:             0u6a4s31ybk7yw2wyvtikmu50
Name:           redis
Service Mode:   Replicated
 Replicas:      3
Placement:
 Strategy:          Spread
UpdateConfig:
 Parallelism:   1
 Delay:         10s
ContainerSpec:
 Image:         redis:3.0.6
Resources:
Endpoint Mode:  vip
```

- Actualización:

```
docker service update --image redis:3.0.7 redis
```

- Ver errores:

```
docker service inspect --pretty redis

ID:             0u6a4s31ybk7yw2wyvtikmu50
Name:           redis
...snip...
Update status:
 State:      paused
 Started:    11 seconds ago
 Message:    update paused due to failure or early termination of task
9p7ith557h8ndf0ui9s0q951b
...snip...
```

Stack

- Un stack es un conjunto de servicios que se ejecutan juntos.
- Ejecución de un stack (equivalente al `docker compose up`)

```
docker stack deploy --compose-file docker-compose.yml <stack-name>
```

- Si actualizamos el fichero yml, al hacer el deploy del stack actualizará los cambios.
- Borrar stack (equivalente al docker compose down)

```
docker stack rm <stack-name>
```