

docker

Profesor: Víctor Villegas Borge

Con la estrecha ayuda del profesor: Óscar Rodríguez García.

1. Contexto (desarrollo de aplicaciones y administración de sistemas).

El desarrollo de aplicaciones y la administración de sistemas han sufrido en los últimos años una gran evolución. Se ha pasado de entornos encapsulados en un único equipo o servidor a entornos mucho más complejos en los que la existencia de diferentes aplicaciones, servicios y sistema ejecutándose al mismo tiempo aumentan el grado de complejidad de la infraestructura existente.

Y esto sin contar con la infinidad de sistemas operativos disponibles y utilizados en una misma empresa, las distintas versiones del software utilizado a distintos niveles, entornos de desarrollo, de pruebas o de producción o incluso la existencia de aplicaciones alojadas íntegramente o solo en parte en alguno de los tipos de nubes que existen en la actualidad.

Las aplicaciones son cada vez más complejas y necesitan de una serie de elementos interrelacionados que permiten, por un lado, su desarrollo como los distintos frameworks, IDEs o lenguajes de programación, o bien su uso y ejecución como pueden ser bases de datos, sistemas operativos o librerías necesarias para que una determinada aplicación pueda funcionar sobre un determinado sistema operativo.

En este contexto surge, hace ya unos cuantos años, la idea de aislar las aplicaciones del resto de los sistemas y servicios que se están ejecutando en un determinado equipo. Surge así el concepto de contenedor que tan de moda está en la actualidad como ese elemento que permite encapsular de

alguna manera todo lo que necesita una aplicación para ejecutarse en un equipo y mantenerlo separado del resto de las aplicaciones.

2. ¿Qué es Docker?

Docker es una palabra que se utiliza para referirse a varias cosas diferentes, así que vamos a intentar aclararlo un poco.

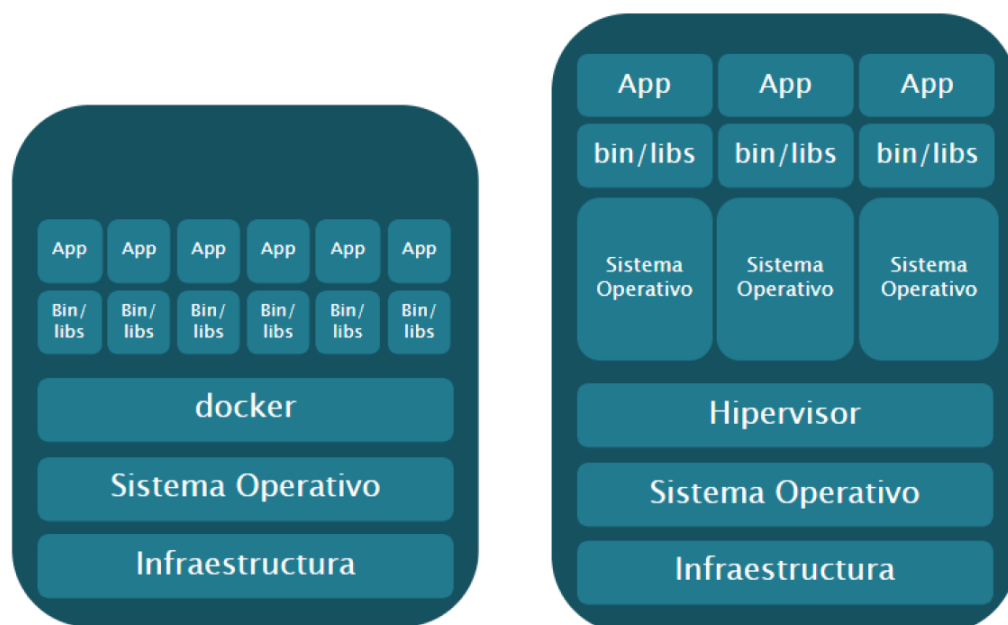
Por un lado, Docker se utiliza para referirse a un proyecto *Open Source* denominado Docker y respaldado por una gran comunidad que se encarga de desarrollar y mantener las herramientas disponibles en ese proyecto.

En segundo lugar, Docker también se refiere a las propias aplicaciones y herramientas generadas por este proyecto *Open Source*, y que sirven como base para la gestión de contenedores de una forma simplificada y unificada en distintos tipos de sistemas.

Por último, Docker también se refiere a la empresa Docker Inc., que apoya este proyecto *Open Source* y a la comunidad que lo respalda.

Al estar el proyecto Docker respaldado tanto por una empresa privada como por la comunidad *Open Source* vamos a encontrarnos con dos herramientas diferentes: Docker CE (Community Edition) y Docker EE (Enterprise Edition), que si bien son similares la primera está respaldada por la comunidad open source y por tanto es gratuita y la otra está respaldada por la empresa y lleva unos costes asociados.

Nosotros vamos a utilizar la versión de la comunidad de Docker.



En el fondo, Docker es una plataforma para que tanto desarrolladores como administradores puedan desarrollar, lanzar y ejecutar aplicaciones en un entorno aislado denominado contenedor permitiendo separar las aplicaciones

de la infraestructura donde se desarrollan o ejecutan y trabajar así más cómoda y rápidamente.

Podemos entender a Docker como un tipo de virtualización en el que las aplicaciones y todo lo que necesitan para poder ser ejecutadas está encapsulado en ese contenedor que se ejecuta directamente sobre el kernel del sistema operativo anfitrión.

De forma rápida y sencilla, digamos que Docker nos ofrece la posibilidad de ejecutar varias y muy distintas aplicaciones, de forma aislada e independiente. Por ejemplo, nos da la posibilidad de tener una instancia funcionando con php7, otra con php5 y otra con NodeJS 8.

Cada contenedor se ejecuta directamente sobre el kernel del sistema operativo instalado en la máquina (host o anfitrión) por lo que son mucho más ligeros que las máquinas virtuales. Esto implica que cada contenedor no necesita tener un sistema operativo completo corriendo para poder ejecutarlo, sino que comparte el sistema operativo con el anfitrión, reduciendo de esta forma el tamaño de los contenedores que sólo necesitan una pequeña parte de código para ejecutarse. También contienen el sistema de ficheros y algunas librerías que permiten su ejecución como si fuera un sistema diferente del instalado sobre el anfitrión.

De esa manera podemos ejecutar muchos más contenedores sobre el mismo hardware que si éstos fueran maquina virtuales. Al prescindir del sistema operativo son muy ligeros, pudiendo “pesar” la mayoría menos de 100Mb frente a los varios Gb de las máquinas virtuales.

Cuando trabajas con múltiples clientes, desarrollos o proyectos, es realmente útil. Es muy normal que estés desarrollando una web para un cliente, funcione todo correctamente, y cuando lo publicas en el servidor de producción del cliente ves que ocurren fallos por incompatibilidad de las versiones. Porque puede ser que en tu entorno de desarrollo utilices PHP 7 y el servidor del cliente tenga PHP 5 por ejemplo, o una versión diferente también de MySQL.

Con estos contenedores, podemos probar rápidamente nuestra aplicación web, por ejemplo, en múltiples entornos distintos al que estamos utilizando para nuestro desarrollo. Realmente es mucho más rápido que hacerlo en una máquina virtual, puesto que reduce el tiempo de carga y el espacio requerido por cada uno de estos contenedores.

Los contenedores son útiles en los servidores, pero también para pruebas de software, ya que incluyen todos los elementos que necesitan las aplicaciones, lo que facilita la instalación y elimina los problemas de incompatibilidad de versiones en bibliotecas y aplicaciones auxiliares.

Cada uno de estos contendores están basados en imágenes que contienen una configuración base de un sistema operativo o de una aplicación o incluso de ambos. Estas imágenes serán la base para los contenedores que las utilizarán como “modelos” o plantillas en cuanto a funcionalidad y configuración.

Las imágenes se pueden crear a partir de otras imágenes más básicas incluyendo software adicional en forma de capas. Cada capa adicional va enlazada con la anterior, pero presenta una identidad propia. Todos los contenedores creados a partir de una imagen contienen el mismo software, aunque en el momento de su creación se pueden personalizar algunos detalles.

Las imágenes se almacenan en una caché local de cada uno de los equipos que ejecutan Docker y que sirve como origen para la creación de los distintos contenedores que se basen en ellas.

Incluso existe un repositorio centralizado, el Hub de Docker (DockerHub), donde podemos encontrar imágenes que podemos directamente descargar y ejecutar como contenedores. De esa forma, podemos, con un solo comando, lanzar en un contenedor la aplicación con la configuración exacta que necesitamos para probar nuestro sitio web, aplicación web o cualquier desarrollo en el que estemos trabajando.

Vamos a ver ahora cómo funciona Docker. Hemos dicho ya que Docker se basa en la ejecución de contenedores que utilizan imágenes como base para definir su funcionalidad. Por otro lado, la propia plataforma Docker se basa en una arquitectura interna del tipo cliente-servidor, en la que existe un servicio o Daemon de Docker que atiende a las peticiones del cliente Docker.

Será el cliente el encargado de pedirle al Daemon lo que necesita, como descargar una imagen, lanzar un contenedor, pararlo o eliminarlo y el Daemon se encargará de mantener todo lo necesario para que tanto los contenedores como las imágenes estén disponibles y accesibles y asegurar la correcta ejecución de todos y cada uno de los contenedores y de los elementos que llevan asociados y que veremos más adelante. En la práctica, se puede apreciar claramente como la velocidad de creación y arranque de estos contenedores va a ser claramente superior a la de cualquier máquina virtual, sobre todo porque en cada contenedor podremos decidir qué capas instalar contando sólo con lo estrictamente necesario.

Hay que tener en cuenta que Docker trabaja con instancias de los contenedores, siendo cada una de estas instancias únicas y volátiles. De esta forma podemos tener en funcionamiento en nuestro sistema tantas instancias como queramos basadas en una misma imagen, por ejemplo, de un servidor web, y cada una de ellas será independiente, aislada del resto y con un identificador único que la distinguirá del resto.

En cuanto a la volatilidad, cada vez que un contenedor (instancia de una imagen) se cierra se van a perder todas las modificaciones que se hayan hecho en el mismo. Aunque esto es una característica que parece limitar las posibilidades de Docker en entornos de producción, lo cierto es que se puede utilizar sistemas externos de almacenamiento persistente como se verá más adelante.

Igualmente, el aislamiento de las aplicaciones de Docker se puede sobrellevar utilizando y configurando correctamente las conexiones de red y los puertos utilizados por las aplicaciones de los contenedores de Docker, permitiendo la redirección de puertos entre el anfitrión y los contenedores y la creación de redes virtuales para la comunicación entre los distintos contenedores.

3. Instalación de Docker y herramientas que usaremos

Docker es una tecnología multiplataforma, por lo que se puede instalar y utilizar sobre diferentes sistemas operativos y no solo sobre Linux como podría suponerse. Utiliza la virtualización como mecanismo para trabajar con los contenedores por lo que necesitaremos que nuestro equipo soporte la virtualización por hardware y tenerlo activado en la configuración de la BIOS.

Si pensamos instalar Docker en Windows vamos a encontrarnos con diferentes opciones y posibilidades dependiendo de la versión del sistema operativo que estemos utilizando. En el caso de Windows 7 y 8 necesitaremos instalar una aplicación denominada Docker Toolbox y que no es más que un conjunto de herramientas entre las que se incluye el propio Docker y varias herramientas adicionales que permiten que Windows utilice una máquina virtual Linux diminuta como base para la ejecución de Docker.

En el caso de querer utilizar Docker con Mac debemos tener en cuenta también que el hardware de nuestro Mac tiene que ser posterior a 2010 y que la versión del sistema operativo debe ser superior a la 10.14.

Si cumplimos estos requisitos podremos instalar Docker Desktop y trabajar con contenedores desde nuestro equipo.

Las versiones actuales de las distribuciones de Linux más habituales disponen en sus repositorios de los paquetes necesarios para instalar Docker. Por ejemplo, si utilizamos Ubuntu 18.04 podremos instalar Docker desde el paquete `docker.io`, mientras que si utilizamos CentOS bastará con utilizar el paquete llamado `Docker`. Utilizan Docker CE y aunque los repositorios no contienen la última versión de la plataforma, sí que disponen de una versión completamente estable y con toda la funcionalidad necesaria para utilizar Docker tanto en entornos de prueba como de producción.

Hay que recordar que todo el proceso de instalación deberá realizarse con el superusuario, o al menos con privilegios de superusuario. Docker utiliza por defecto al usuario y grupo `Docker` para ejecutar los comandos relacionados con la gestión de los contenedores, o bien el usuario `root`. Si queremos utilizar otro usuario (lo cual es más que recomendable) habrá que añadirlo al grupo `Docker`.

Como ya había sido adelantado, para usar Docker utilizaremos de base el S.O. de Alpine Linux.

Para llevar a cabo la instalación de Docker en nuestro S.O. Alpine Linux deberemos ejecutar los siguientes comandos:

```
$ apk add --update docker openrc
```

```
$ rc-update add docker boot
```

```
$ service docker start
```

```
$ service docker status
```

Cada uno de estos comandos nos permite, instalar, habilitar el arranque de Docker junto al inicio del S.O., iniciar manualmente el servicio de Docker y comprobar si el estado de Docker está activo, respectivamente.

Por otra parte, también necesitaremos instalar otra herramienta conocida como Docker Compose. Para llevarlo a cabo, se pide que sigas la siguiente guía: <https://dev.to/docker/how-to-install-the-latest-version-of-docker-compose-on-alpine-linux-in-2022-3m4d> y modifiques este documento para que contenga los comandos que has utilizado:

```
$
```

```
$
```

```
$
```

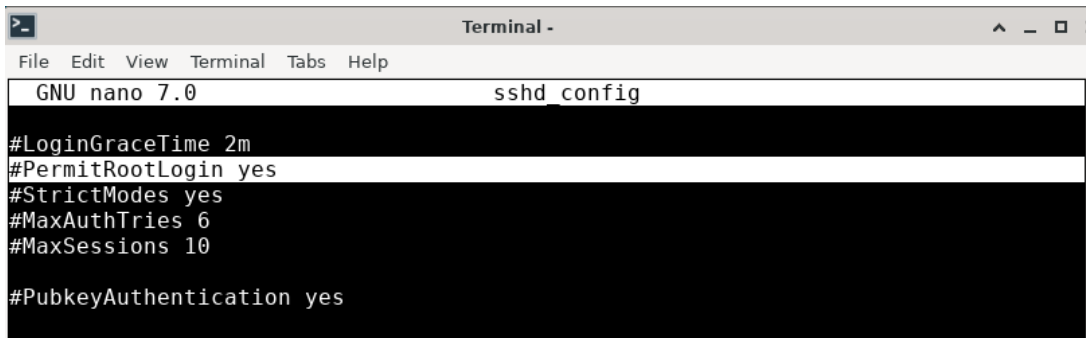
```
$
```

```
$
```

```
$
```

Una vez instalados tanto Docker como Docker Compose, estaríamos listos para poder utilizar todas las funcionalidades de Docker y crear imágenes/lanzar contenedores, sin embargo, puesto que vamos a utilizar unas herramientas llamadas Putty y WinSCP, primero deberemos modificar unos pequeños parámetros dentro de la máquina virtual de Alpine Linux.

Dentro de Alpine deberemos modificar el elemento que se encuentra en la ruta: /etc/ssh/sshd_config. En este documento, deberemos cambiar la línea en la que aparece “#PermitRootLogin prohibit-password” por: “#PermitRootLogin yes”.



```

Terminal -
File Edit View Terminal Tabs Help
GNU nano 7.0 sshd_config

#LoginGraceTime 2m
#PermitRootLogin yes
#StrictModes yes
#MaxAuthTries 6
#MaxSessions 10

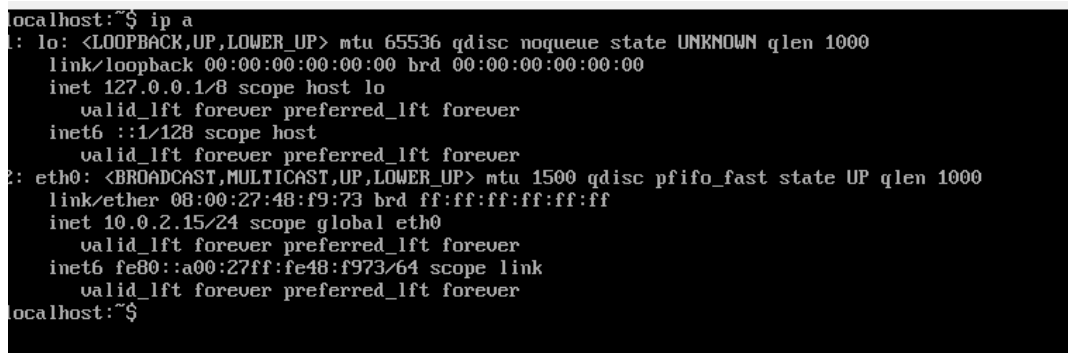
#PubkeyAuthentication yes
  
```

Las herramientas Putty y WinSCP, nos van a permitir acceder de forma remota a la máquina en la que se ejecute Docker. En nuestro caso, podría parecer una tontería puesto que ya tenemos a nuestro alcance la máquina virtual con el S.O. montado en VirtualBox, pero es posible que el día de mañana necesitemos realizar estas mismas labores o parecidas en máquinas a las cuales no tendremos un acceso directo.

Por ello, antes de entrar a explicar el uso de estas herramientas, es importante que cambiemos las propiedades del adaptador de red de nuestra máquina virtual para obtener una dirección IP accesible.

Primero veamos cuál es la IP de nuestra máquina virtual. Esto podemos hacerlo a través del comando:

\$ ip a

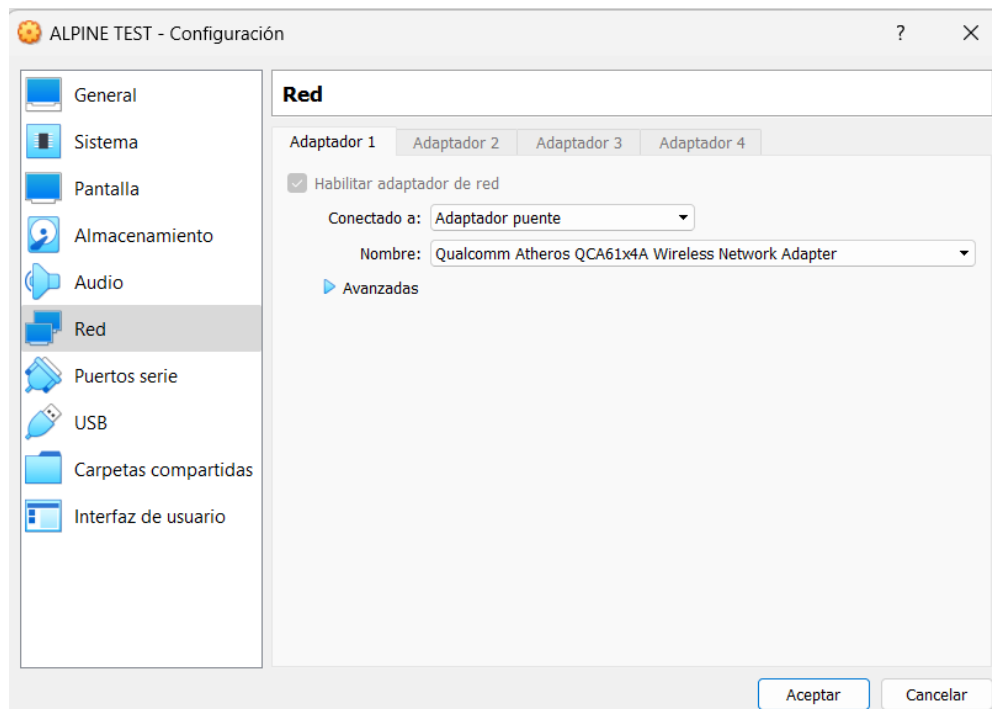


```

localhost:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:48:f9:73 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe48:f973/64 scope link
        valid_lft forever preferred_lft forever
localhost:~$
  
```

En este caso, podemos observar que la IP con la que aparece nuestra máquina por defecto es: 10.0.2.15

Vamos a acceder a las Preferencias de Red de nuestra máquina virtual a través de VirtualBox y vamos a cambiar el adaptador de red para que en vez de estar en NAT esté configurado como Adaptador Puente:



Una vez cambiado, reiniciamos el sistema y si ahora volvemos a ejecutar el comando `$ ip a`, nos aparece una nueva dirección IP:

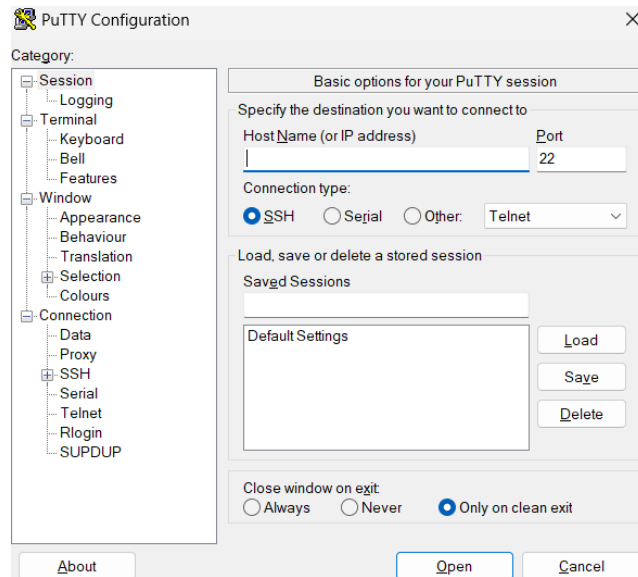
```
localhost:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:48:f9:73 brd ff:ff:ff:ff:ff:ff
    inet 172.16.0.237/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe48:f973/64 scope link
        valid_lft forever preferred_lft forever
localhost:~$
```

En este caso vemos que la nueva dirección IP es 172.16.0.237.

Una vez llevado a cabo toda esta configuración podremos utilizar las herramientas Putty y WinSCP. Estas herramientas las usaremos más adelante en el curso (cuando trabajemos con redes), pero de forma resumida diremos que nos van a proporcionar una terminal de comandos del *guest* en nuestro *host* (Putty) y una interfaz gráfica que nos permitirá pasar rápidamente archivos/movernos por el sistema de archivos de los dos sistemas desde nuestro sistema *host*.

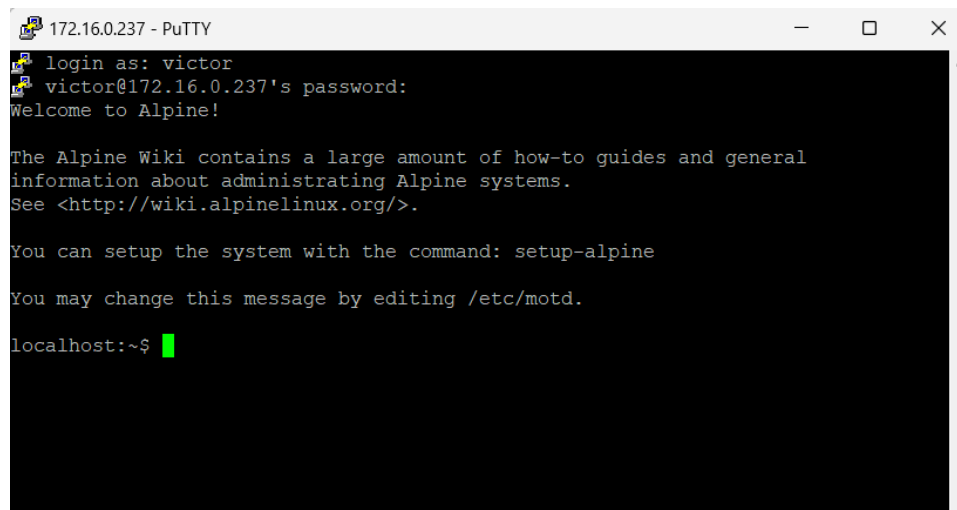
Para poder utilizar estas dos herramientas, nada más las abramos nos pedirá que digamos cuál es la dirección IP de la máquina con la que queremos establecer conexión.

En mi caso, al abrir el programa Putty aparece la siguiente interfaz:

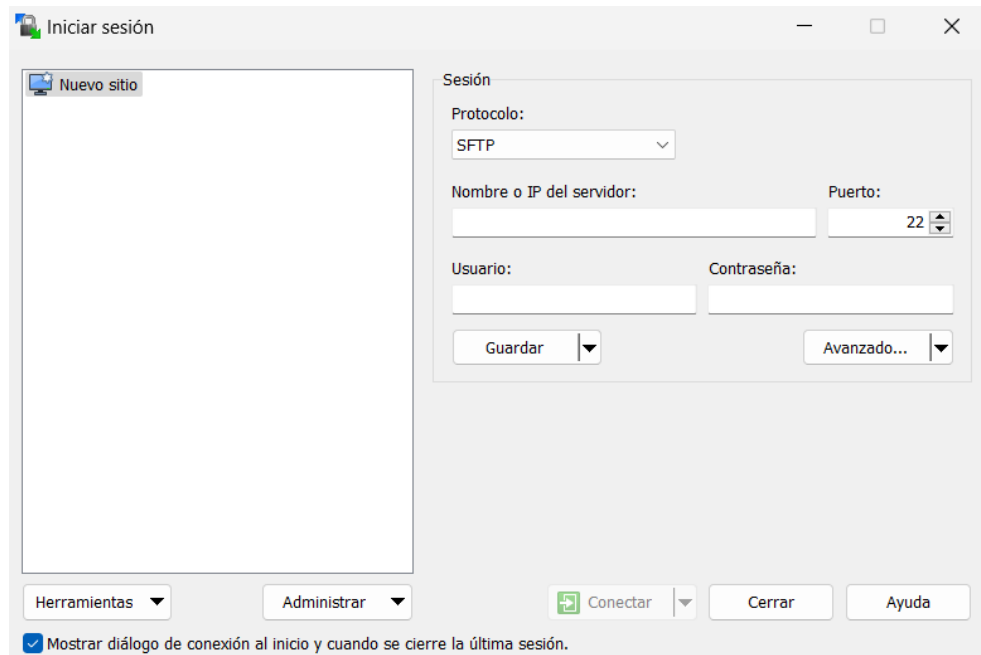


Lo único que deberemos hacer será introducir la IP de la máquina en cuestión, en mi caso 172.16.0.237.

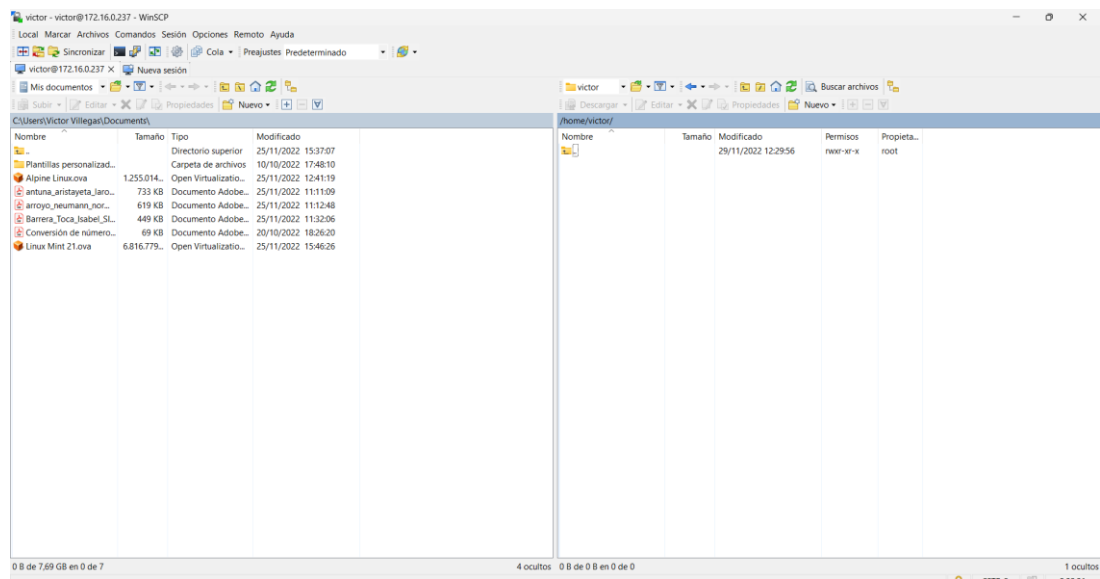
Una vez introducida la IP damos a *Open* y aceptaremos el aviso que nos salta. Finalmente, deberemos iniciar sesión en la máquina a la que nos hemos conectado.



WinSCP, por su parte se configura de una forma muy parecida a la que hemos realizado con Putty. Tendremos que introducir la IP de la máquina con la que queremos establecer conexión, así como las credenciales del usuario con el que accederemos al sistema.



Una vez hayamos realizado la conexión nos aparecerá una interfaz como la siguiente, donde a la izquierda tendremos el árbol de directorios de nuestro sistema (en el que está instalado WinSCP) y a la derecha la del sistema al que nos hemos conectado:



4. Imágenes y contenedores en Docker

Las imágenes son una serie de capas software que definen la estructura y funcionamiento que tendrá el contenedor que se cree a partir de ellas.

Si bien lo más sencillo en un primer momento es utilizar alguna de las imágenes ya disponibles en DockerHub, lo más habitual cuando nos familiarizamos con esta tecnología es crear nuestras propias imágenes personalizadas.

Pero vamos a empezar por el principio. Vamos a comenzar por utilizar alguna de las imágenes incluidas en el repositorio oficial de Docker.

El proceso comienza con la descarga de una imagen ya creada desde el repositorio principal de Docker (DockerHub) y la añade al registro de imágenes locales de nuestro sistema. A partir de este momento la imagen estará disponible para utilizarla como plantilla para cualquier contenedor que queramos lanzar basado en ella, generando una instancia única en cada uno de los contenedores.

Podremos descargar tantas imágenes como queramos para almacenarlas localmente, pero hay que recordar que las imágenes se van a poder modificar y que, además, podremos crear nosotros mismos nuestras propias imágenes a partir de otras ya existentes.

Una vez que tenemos alguna imagen disponible podemos lanzar un contenedor utilizando esa imagen como plantilla. Al lanzar la imagen como contenedor deberemos añadir algunas opciones al proceso de ejecución para indicar que sea interactivo o que utilice una terminal para poder interactuar con el contenedor generado. Es importante revisar tanto los contenedores como los procesos que tenemos en Docker ya que los procesos de Docker se corresponden con contenedores (como vimos al principio) y los contenedores finalizados siguen “estando” todavía en el sistema hasta que no los eliminemos. Parar un contenedor o salir de él no significa eliminarlo.

En cada contenedor podremos de esta forma ejecutar las acciones que sean necesarias, actualizar, instalar aplicaciones, etc. Pero sea como sea debemos recordar que estos cambios que realicemos serán volátiles y no se guardarán en la imagen de plantilla que estamos utilizando.

Para guardar estos cambios deberemos crear una nueva imagen basada en la anterior a la que le daremos un nombre nuevo y que se añadirá al registro local de imágenes de nuestro sistema.

Podemos almacenar tantas imágenes como queramos en nuestro registro local. Cada una de esas imágenes tiene un tamaño importante y aunque no suelen ser muy grandes en entornos más complejos la cantidad de

imágenes almacenadas puede ser enorme y el tamaño que ocupan en disco puede ser abrumador.

5. Comandos Docker

De momento vamos a aprender algunos comandos básicos de Docker que son:

\$ docker images → Muestra un listado de las imágenes guardadas localmente.

\$ docker image rm → Elimina una imagen guardada localmente.

\$ docker container ls -a → Muestra un listado con las características de los diversos contenedores que hemos creado.

\$ docker pull → Descarga una imagen desde el repositorio central.

\$ docker run → Permite ejecutar un comando en nuevo contenedor.

\$ docker ps -a → Equivalente a \$ docker container ls -a

\$ docker commit → Crea una nueva imagen a partir de los cambios realizados en un contenedor.

\$ docker info → Muestra información del sistema relacionada con Docker.

\$ docker rm | \$ docker container rm → Elimina uno o más contenedores

\$ docker stop | start | kill → Detiene, inicia y elimina contenedores.

Pero como siempre, existen una inmensidad de comandos, por lo que siempre que tengamos dudas:

\$ docker --help

\$ docker [comando] --help

Estos dos comandos serán nuestra ayuda para entender no solo los comandos que tenemos disponibles

6. Construcción de imágenes

Como hemos visto, los contenedores Docker se basan en una imagen que tomarán como referencia para prestar una determinada funcionalidad.

Estas imágenes base están almacenadas en DockerHub o bien en los registros locales que tengamos disponibles, pero en muchas ocasiones puede ser necesario crear imágenes personalizadas con una funcionalidad específica que no está disponible en los registros.

En otras ocasiones, el hecho de que existan en los registros imágenes con la funcionalidad que buscamos no nos garantiza ni su seguridad ni su adecuado funcionamiento o incluso su estado de actualización.
(IMPORTANTE)

En cualquiera de estos casos la creación de una nueva imagen con todas las configuraciones y funcionalidades que necesitamos va a ser algo imprescindible.

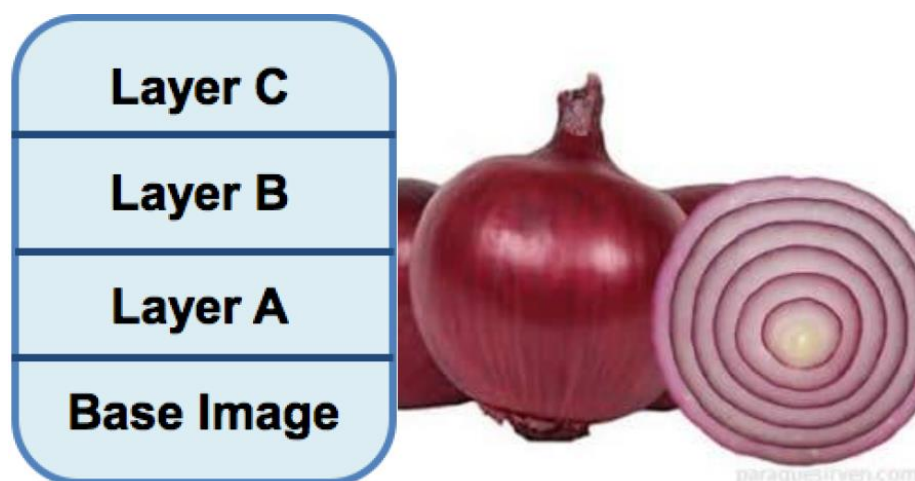
El proceso de creación de una imagen se puede realizar de dos formas diferentes.

Por un lado, podemos hacerlo partiendo de un contenedor que tengamos en ejecución y al que le vayamos añadiendo las configuraciones y funcionalidades en forma de aplicaciones o ejercicios que necesitamos (necesitaremos que el contenedor se haya creado sobre una imagen confiable) y por otro lado podremos crear la imagen a partir de un fichero en el que especificaremos todo lo que queremos incluir en la misma.

Hemos definimos las imágenes como modelos o plantillas que utilizarán los contenedores y que contienen tanto la configuración como las aplicaciones y funciones necesarias para prestar un servicio.

Las imágenes en Docker se construyen siguiendo un modelo de capas independiente que se van superponiendo unas encima de otras hasta alcanzar la imagen completa. Cada capa es independiente de las demás, pero tiene que estar relacionada con la anterior y la posterior.

Cada capa que forma parte de una misma imagen contendrá todo lo necesario para que tenga entidad en sí misma y el Daemon de Docker proporcionará un identificador único a cada capa de cada imagen.



Siguiendo este modelo deberemos partir de una imagen base sobre la que ir añadiendo las funcionalidades que necesitamos.

En este punto debemos hacer un inciso. Aunque Docker nos va a permitir añadir tantas capas con tantas funciones, servicios o aplicaciones

necesitemos en la misma imagen y en el mismo contenedor, debemos recordar que la idea detrás de Docker es la de una separación de cada funcionalidad en un contenedor independiente (microservicio), de modo que en nuestros proyectos de Docker trataremos identificar funcionalidades y generar una imagen diferente para cada una de ellas.

A pesar de todo, cualquier imagen que tengamos que crear va a requerir de varias capas diferentes y como veremos, el orden en el que definamos estas capas va a condicionar su funcionamiento y su rendimiento.

Siguiendo con el ejemplo anterior, la primera capa de nuestra imagen sería el sistema base, un sistema operativo sobre el que se añadirá el resto de funcionalidad, y las posteriores capas incluirán todo lo necesario para instalar las aplicaciones requeridas, configurarlas, ejecutarlas y servir los contenidos asociados a las mismas.

7. Construcción de imágenes (2). Dockerbuild y Dockerfile

La otra forma de crear imágenes es mediante un fichero de texto que contendrá todo lo necesario para que la imagen tenga funcionalidad.

Este fichero especial debe llevar el nombre `dockerfile.yaml` y puede crearse o editarse con cualquier editor de texto.

Si queremos ir un paso más allá podemos utilizar herramientas como Visual Studio Code que disponen de extensiones específicas para interpretar este tipo de archivos y que ayudan a la hora de crearlos.

De cualquiera de las formas en que vayamos a crear este tipo de archivos debemos tener en cuenta que tiene una estructura definida que debe respetarse para una correcta interpretación de este por parte del Daemon de Docker. Las principales opciones que se pueden añadir a un fichero `dockerfile` son las siguientes:

FROM: Define la imagen base.

MAINTAINER: Nombre e email del mantenedor de la imagen.

COPY: Copia un fichero o directorio a la imagen.

ADD: Copia ficheros desde urls. También tars, que descomprime.

RUN: Ejecuta un comando dentro del container.

CMD: Comando por defecto cuando ejecutamos un container. Se puede sobrescribir desde la CLI.

ENV: Variables de entorno.

EXPOSE: Define los puertos del servicio del contenedor. Se deberán añadir de forma explícita en la llamada desde la CLI.

VOLUME: Define los directorios de datos persistentes.

ENTRYPOINT: Comando a ejecutar de forma obligatoria al ejecutar un contenedor.

USER: Usuario para RUN, CMD y ENTRYPOINT.

WORKDIR: Directorio donde se ejecutan los comandos RUN, CMD, ENTRY-POINT, ADD y COPY.

Veámoslo con un ejemplo:

FROM ubuntu

ADD . /app

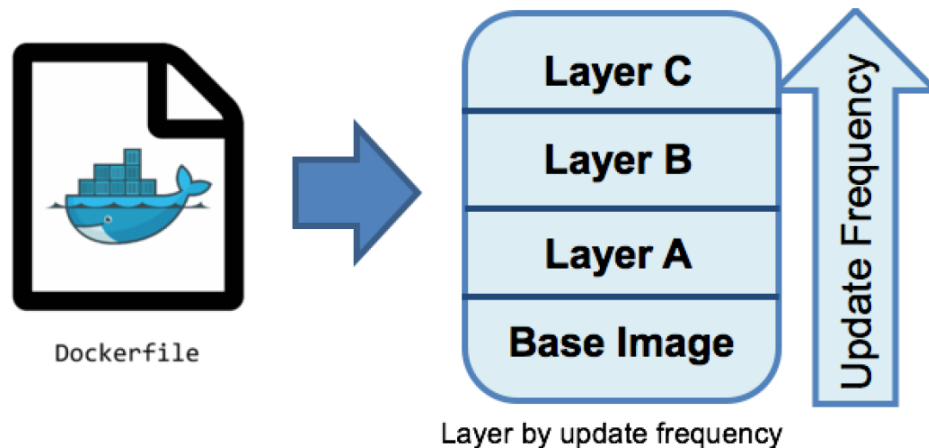
RUN apt-get update -y && apt-get upgrade -y

Una vez se haya creado el archivo con la configuración necesaria bastará construir la imagen utilizando como modelo el propio fichero. Será en este paso en el que el Daemon de Docker vaya generando todas las capas necesarias para la construcción completa de la imagen y una vez finalizado, la imagen quedará disponible a nivel local para su utilización en la creación de contenedores.

Las imágenes construidas de esta forma tienen las mismas características que las que podemos encontrar en el repositorio central de imágenes de DockerHub o en ellos registros locales. Es más, las imágenes así creadas se podrán subir a DockerHub o cualquier otro registro utilizando una cuenta de usuario en el servicio de registro.

Lo primero que debemos tener en cuenta a la hora de crear un Dockerfile es que cada contenedor, siguiendo con la filosofía de Docker, debe contener un único servicio. Esta regla de oro nos permitirá simplificar la creación de contenedores, pero nos obligará a prestar más atención al proceso de enlazado y lanzamiento de los mismos, lo que se conoce como orquestación de contenedores.

Partiendo de esta primera premisa de un contenedor un servicio, podemos pasar a revisar el funcionamiento de las capas en el proceso de creación de imágenes.



Como hemos visto en apartados anteriores, el proceso de creación de una imagen requiere de una pequeña planificación, ya que la estructuración por capas y el qué se pone en cada capa va a condicionar el rendimiento y la funcionalidad de la imagen y a posteriori de los contenedores que la instancien.

Las capas que se definen se van superponiendo una sobre otra partiendo siempre de una imagen base con una configuración mínima. A partir de aquí se irán añadiendo capas con programas, archivos, configuraciones o servicios que sumarán características y funcionalidad, pero también peso, tiempo de carga y actualización cada vez que se lance un contenedor basado en la imagen o que se tengan que hacer modificaciones sobre la propia imagen.

Cada una de las instrucciones que vayamos añadiendo al dockerfile del tipo FROM, RUN, COPY y ADD será una capa independiente. Habrá capas más pesadas como las que cargan el sistema operativo base o

aplicaciones más pesadas como una base de datos y otras más ligeras en las que simplemente se cargan variables de entorno de las aplicaciones o se copian archivos entre el anfitrión y la futura imagen.

En cualquier caso, todo suma, de modo que siempre que sea posible trataremos de simplificar al máximo la complejidad del archivo dockerfile tratando de optimizar la ejecución de las distintas instrucciones y de ordenarlas de cara a simplificar el proceso de modificación o creación de la imagen.

Debemos pensar en que los dockerfiles deben generar contenedores efímeros, entendiendo como efímeros que se puedan eliminar y volver a ejecutar con el mínimo de configuración o modificación de los mismos posible.

8. Automatización de contenedores: Docker-compose.

El uso de contenedores para implementar microservicios es cada vez más común. Las ventajas que aportan en cuanto a funcionamiento, despliegue y gestión son evidentes, pero los contenedores aislados no logran aportar toda la funcionalidad que las aplicaciones tradicionales aportan. Se requiere de una herramienta que permita gestionar varios contenedores como un todo y definir así todos los servicios necesarios para ejecutar una aplicación de una forma única.

Con Docker nos llevamos todo lo que necesitamos en nuestros contenedores. Ahora solo necesitamos poder juntar varios contenedores en una misma aplicación y poder lanzarlos y gestionarlos como si fueran uno solo. Y eso es lo que nos permite Docker-compose.

Docker-compose es una herramienta complementaria a Docker. Utiliza el concepto de stack o pila en la que agrupa diferentes servicios lanzados como contenedores (continuamos con la arquitectura de microservicios).

De esta forma docker-compose permite gestionar todos los microservicios o contenedores asociados a una misma aplicación y ejecutarlos desde la propia herramienta en forma de pila o stack que integra todos los elementos necesarios, desde los propios servicios, el almacenamiento, las redes de comunicaciones, etc.

Un solo comando permitirá lanzar todos los contenedores de una aplicación con todas sus configuraciones y parámetros de una forma tremendamente sencilla.

Para lograr esto Docker-compose utiliza un fichero de configuración denominado Docker-compose.yaml en el que de una forma similar a lo que hacíamos con dockerfile, preconfiguraremos todo lo necesario para poner en marcha nuestra aplicación multicontenedor.

Si todo el conjunto de contenedores desplegados desde un Docker-compose.yml se denomina pila o stack, cada uno de los contenedores individuales generados en este archivo se denomina servicio.

Dentro de un mismo Docker-compose.yml podremos tener tantos servicios definidos como necesitemos en nuestra aplicación. Al igual que en el dockerfile visto anteriormente, el Docker-compose deberá incluir toda la configuración individual de cada servicio particular y además debe permitir enlazar todos y cada uno de ellos.

En cualquiera de los casos el contenido de este fichero debe cumplir una serie de normas para que pueda ser interpretado por la herramienta Docker-compose y por el propio Daemon de Docker para lanzar los contenedores.

Vamos a ver un ejemplo en el que sólo se lanza un único servicio, pero que nos va a permitir ver las tripas de este fichero de configuración.

```
version: '3.3'
services:
  db:
    image: woahbase/alpine-mysql:x86_64
    restart: always
    environment:
      MYSQL_DATABASE: 'db'
      MYSQL_USER: 'user'
      MYSQL_PASSWORD: 'password'
      MYSQL_ROOT_PASSWORD: 'password'
    ports:
      - '10006:3306'
    expose:
      - '3306'
```

Como se puede apreciar el contenido del fichero está organizado. Lo primero que vamos a encontrar siempre es la versión. Los archivos de compose están versionados y cada versión está relacionada con la versión de la herramienta Docker-compose que la maneja y con la versión del Daemon de Docker que corre por debajo, por lo que cada versión diferente aporta unas herramientas y unas opciones de configuración que pueden variar ligeramente.

En cualquier caso, hay que destacar que las versiones tienen compatibilidad hacia atrás y que en caso de utilizar una instrucción no soportada por la versión o por el Daemon, éste nos los indicará en tiempo de ejecución y no lanzará la aplicación.

Tras la versión nos encontramos con la definición de los servicios. A este nivel nos encontraremos con cada uno de los servicios que va a gestionar este stack de Docker-compose. En este caso solo tenemos el servicio db.

Cada servicio que tenemos en este archivo puede tener el nombre que queramos, no hay una regla o una norma que indique qué nombre poner. Sin embargo, es necesario establecer unas pautas claras ya que todo el ecosistema Docker no vamos a poder tener dos contenedores o servicios con el mismo nombre, de modo que el propio Daemon establecerá nombres diferentes en caso de servicios que tengan el mismo nombre. Lo hará de la forma db_1 siendo db el nombre del servicio que ha encontrado duplicado.

Debemos tener en cuenta también que en un entorno de Docker nos podemos encontrar con múltiples aplicaciones que van a utilizar un servicio de bases de datos (db). Está claro que deberemos ser capaces de identificar el servicio asociado a cada aplicación de forma rápida y unívoca, asignándole un nombre identificativo claro, como puede ser db_wp, db_joomla, etc.

El siguiente elemento que nos encontramos es un viejo conocido: la imagen base. Al igual que en el dockerfile es importante establecer una imagen base sobre la que construir nuestro contenedor, y al igual también que en el dockerfile debemos ser especialmente cuidadosos a la hora de escoger la imagen base (oficial o de publicador verificado) que más se adapte a las características de nuestro proyecto.

Aquí van a regir también las mismas buenas prácticas que detallamos para las imágenes en el dockerfile, por lo que conviene revisarlas y tenerlas presentes antes de ponernos a crear nuestro Docker-compose.yaml.

Tras la imagen nos encontramos la opción restart. Restart hace referencia a la política de reinicio de este contenedor y es opcional. En este caso está definido como always que le indica que el contenedor debe reiniciarse siempre que se apague, sea cual sea el motivo. Otra opción que podemos encontrar habitualmente es unless-stopped que le indica al contenedor que debe reiniciarse siempre que se apague accidentalmente o por un fallo, pero no cuando se ha parado de forma manual o programada.

Lo siguiente que nos vamos a encontrar es la definición de las variables de entorno (environment) usadas por la aplicación. En este caso, al tratarse de un servidor MySQL, las variables hacen referencia a las necesidades de configuración del entorno MySQL, en este caso el nombre de la base de datos que se va a utilizar, el nombre del usuario que utilizará esa base de datos, la clase de ese usuario y la clave de administración general de MySQL para el usuario root.

A continuación, aparecen los puertos que utiliza la aplicación. En este caso se realiza un mapeo de puertos entre el contenedor y el anfitrión, de forma que se asigna el puerto 10006 del anfitrión al puerto 3306 que utiliza internamente en el contenedor el servicio MySQL.

Deberemos ser rigurosos en la asignación de puertos para el anfitrión. Dentro de cada contenedor podemos dejar que cada servicio corre en su puerto correspondiente, es decir podemos tener varios contenedores diferentes que corren un servicio de MySQL y que utilizan internamente el puerto por defecto de MySQL (3306) pero a nivel de anfitrión cada uno de esos contenedores deberán tener mapeado ese puerto interno con uno diferente del anfitrión ya que no puede haber dos redirecciones de puertos que utilizan un mismo puerto.

Finalmente nos aparece la opción `expose` que sirve para mostrar el puerto que sirve el contenedor a nivel interno de Docker, es decir, para otros servicios, pero no para el host. En este caso al no haber otros servicios dentro del `Docker-compose.yml` serviría para que otros servicios de otras aplicaciones pudieran enlazar con este.

9. Casos de usos comunes con docker-compose

Como hemos visto, docker-compose permite construir aplicaciones desde un archivo en el que se incluyen los distintos servicios y configuraciones necesarios para que la aplicación se pueda desplegar.

El uso más habitual es en aplicaciones multicontenedor donde existen varios servicios interrelacionados y dependientes entre ellos. De esta forma se facilita el despliegue de la aplicación completa mediante un único comando que permite arrancar todos los servicios contenidos en el fichero.

Sin embargo, docker-compose también puede utilizarse para aplicaciones con un solo servicio como vimos en el primer ejemplo. En este sentido docker-compose se convierte en un rival de peso para `dockerfile` pudiendo en muchas ocasiones quitarle protagonismo.

Pero será la combinación de estas dos poderosas herramientas, `dockerfile` y Docker-compose las que aporten un plus tanto para desarrolladores como administradores de sistemas.

La posibilidad de construir un servicio (build) basado en una imagen creada a través de un `dockerfile` desde un Docker-compose es algo que no debemos dejar pasar por alto. Mientras se realizan las pruebas y se va modificando la imagen que creamos para nuestra aplicación podemos utilizar una imagen finalizada y testada para otro de los servicios que forman nuestra aplicación. Permitiendo que distintos equipos de desarrollo y pruebas vayan interactuando con la aplicación en distintas fases de su evolución y de forma independiente.